Operating Systems Techniques for Reducing Processor Energy Consumption

by

Jacob Rubin Lorch

B.S. (Michigan State University) 1992
M.S. (University of California, Berkeley) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Alan Jay Smith, Chair
Professor Randy H. Katz
Professor Geoffrey Keppel

Fall 2001

Operating Systems Techniques for Reducing Processor Energy Consumption

Abstract

Operating Systems Techniques for Reducing Processor Energy Consumption

by

Jacob Rubin Lorch

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Alan Jay Smith, Chair

In the last decade, limiting computer energy consumption has become a pervasive goal in computer design, largely due to growing use of portable and embedded computers with limited battery capacities. This work concerns ways to reduce processor energy consumption, since the processor consumes much of a computer's energy. Our specific contributions are as follows.

First, we introduce our thesis that *operating systems should have a significant role in processor energy management.* The operating system knows what threads and applications are running, and can predict their future requirements based on their past usage and their user interaction. We motivate using software to control energy management decisions by describing how software has traditionally been applied to this regime.

Next, we describe operating system techniques for increasing processor sleep time. We suggest never running blocked processes, and delaying processes that execute without producing output or otherwise signaling useful activity. These techniques reduce CPU energy by 47–66%.

Next, we address ways to dynamically change a processor's speed and voltage. We suggest considering what tasks the system is working on and their performance needs, then using a speed schedule that just meets those needs. We show that the optimal schedule increases speed as a task progresses according to a formula dependent on the probability distribution of task CPU requirement. Such a schedule can reduce CPU energy consumption by 20.6% on average, with no effect on performance.

Next, we analyze real user workloads to evaluate ways to infer task information from observations of user interface events. We find that observable differences in such events have significant effects on CPU usage. Using such information in estimating the probability distribution of task CPU requirements can reduce energy consumption by a further 0.5–1.5%.

Finally, we implement our methods. We deal with I/O wait time, overlap of multiple simultaneous tasks, limited speed/voltage settings, limited timer granularity, and limited ability to modify an operating system. The resulting task-based scheduler implements our energy-saving methods with 1.2% background overhead. We find that our methods will be more effective on future processors capable of a wider range of speeds than modern processors.

Professor Alan Jay Smith
Dissertation Committee Chair

Dedicated to my parents, Steven and Harriet

Contents

List of Figures

List of Tables

Acknowledgments

There are many people who helped greatly in enabling this work. I would first like to thank my advisor, Alan Jay Smith, for accepting me as a student and helping me through the entire course of my Ph.D. work. He guided and advised me when it was necessary, and also allowed me great freedom to take the research in whatever direction I thought best. No matter what I chose to take on, he was very supportive.

I performed much of the early work for this dissertation at Apple Computer, where I was aided by several colleagues. As I began the work with little background about tracing and modeling Apple computers, I was greatly aided by individuals there. Marianne Hsiung guided the project and made sure I had the resources and contacts I needed. Dave Falkenburg, Jim Goche, and Phil Sohn taught me Apple-specific system programming tricks. Steve Sfarzo helped me find candidates for tracing Apple computers in normal usage. Finally, Helder Ramalho gave me information about techniques Apple uses for processor power management.

As processors with dynamic voltage scaling became a commercial reality rather than an abstract research phenomenon, I found people at Transmeta eager to provide me hardware and information about these processors. Marc Fleischmann introduced me to several people at Transmeta and arranged for me to borrow a prototype machine with a chip about to be released. Rowan Hamilton provided me with data and information about the chip and helped me set up experiments for measuring energy consumption. Both of them helped me see much about the reality of dynamic voltage scaling on modern machines.

People from AMD also provided me with hardware and information regarding their chips with dynamic voltage scaling. Here, Richard Russell and Fred Weber arranged to provide me with systems demonstrating AMD's chips and dynamic voltage scaling software. Dave Tobias answered many questions and provided a great deal of information about the chips.

Naturally, I had a lot of help and encouragement from colleagues at the University of California Berkeley. Most notable was my officemate, Jeff Rothman, from whom I learned a great deal. When I ran into problems with my research, he was always readily available and willing to discuss them and make me see them in a different way. Also, working with him, I went from being afraid of even opening a computer lest I touch (gasp) hardware to someone able to fix, upgrade, and build computers whenever needed. Windsor Hsu also helped me, especially with issues related to my tracer. I also gained some much-needed depth in my research outlook from collaboration with Drew Roselli, Radhika Malpani, and David Berger, who enabled me to venture outside my world of energy management and make contributions in other areas of computer science.

Several professors at the University of California at Berkeley also aided me in my research. Ronald Wolff, Geoffrey Keppel, and especially Michael Jordan helped me incorporate statistical modeling into my research repertoire, and made helpful suggestions when my research demanded statistical understanding beyond my limited abilities.

Finally, I would like to thank my parents, Steven and Harriet Lorch, and my girlfriend, Michelle Teague, for their patience and encouragement throughout the long process of getting my Ph.D. Well, maybe I could have done without quite so much encouragement from my mother. . .

Chapter I

Introduction

I.1 Motivation

In the past, computers were judged mainly on two criteria: price and performance. Lately, however, energy consumption has gained increasing importance, due to several factors. Portable computers, including notebooks and palm-sized devices, have gained popularity. These computers are limited in their operating time by the amount of energy in their batteries. And, unlike most properties of computers, battery capacity per unit weight has improved little in recent years and shows little indication of substantial improvement in coming years [Fuj97]. Even in nonportable computers, we have reached a point where designers must worry about the power consumption of computers due to the consequent heat dissipation, which can damage components and disturb nearby users. In addition, we are currently experiencing rising energy costs, contributing to a desire to keep energy consumption low.

Hardware designers can do many things to keep the energy consumption of devices low. One fruitful approach is to provide *low-power states* on these devices that consume less power at the cost of somehow reduced functionality. In this way, when the system does not require the full performance of the device, it can save power. However, the presence of such low-power states presents an interesting problem for the system designer. The system must somehow continuously decide what state is best suited to current requirements, and change the processor's state appropriately.

Our thesis is that it is useful for the operating system, instead of merely the hardware, to perform such processor energy management. It is in a better position than the hardware to understand the nature of the applications running, and therefore to effectively estimate future processor functionality requirements. Meanwhile, it is close enough to the hardware to be able to efficiently modify the processor state when necessary. In this dissertation, we will explore various ways in which the operating system can reduce processor energy consumption.

I.2 Processor states

Processor energy management involves switching between processor states of variable power. Modern processors have two main types of low-power states: sleep and reduced-voltage.

Essentially all modern processors, even ones not designed for mobile use, have sleep states. In a sleep state, the processor performs little or no work, and has consequently reduced power consumption. Often a processor will have multiple sleep states, with some having lower power consumption but requiring a greater delay to return to the normal state or having less functionality (such as not performing bus snooping). For example, Intel's Mobile Pentium III has seven states: Normal, Stop Grant, Auto Halt, Quick Start, HALT/Grant Snoop, Sleep, and Deep Sleep [Int01]. Deep Sleep consumes the least power but requires 30 μs to return to the Normal state; Auto Halt, in contrast, requires only 10 bus clocks to return to the Normal state. Using a low-power state when processing power is not needed can therefore substantially reduce energy consumption, albeit at the cost of some delay when the CPU must return to a higher-power state.

A small but growing number of processors, including Transmeta's Crusoe$^{\text{TM}}$ chips and AMD's Mobile K6-2 and Mobile Athlon 4 chips, have dynamic voltage scaling (DVS), the ability to dynamically change the CPU voltage level without rebooting. In a reduced-voltage state, the processor uses a lower supply voltage, thereby consuming less power and less energy. In general, CMOS circuits consume power proportional to V^2f where V is the voltage and f is the frequency. Energy consumption per cycle is power consumption divided by frequency, so energy consumption is proportional to V^2. In other words, reducing the voltage quadratically reduces the energy needed to perform the same number of cycles. However, a performance trade-off arises because running at a lower voltage increases gate settling times and thus necessitates running at a lower frequency. The maximum valid frequency for a given voltage is roughly linear in voltage; more accurately, it is proportional to $(V - V_{\text{th}})^2/V$ where V_{th} is the threshold voltage of the CMOS process. Due to this trade-off between performance and energy consumption, the decision about when to raise or lower the speed is a complex one requiring knowledge about CPU requirements both now and in the future. DVS algorithms attempt to predict such requirements and adjust speed and voltage accordingly.

I.3 Dissertation structure

This dissertation has seven chapters and two appendices. The first chapter is this introduction. The last chapter offers conclusions and describes avenues for future work. The five intermediate chapters address the following.

Chapter II describes how software, especially the operating system, can manage the energy consumption of computer components. We present a survey of software energy management techniques to demonstrate how software can be an effective complement to hardware in reducing energy consumption. This chapter also serves as useful background for the remaining chapters, as it describes processor characteristics and current software management techniques for processors.

Chapter III describes techniques we developed for taking better advantage of processor sleep modes than MacOS 7.5 had before. We show that turning off the CPU when all threads are idle is a better approach than turning it off after a certain period of user inactivity. We demonstrate how various modifications to the way the operating system handles process scheduling can make this technique even more effective.

Chapter IV introduces PACE, our method for improving existing dynamic voltage scaling algorithms. PACE works by replacing each speed schedule such an algorithm produces with a performance-equivalent schedule having lower expected energy consumption. Implementing PACE requires statistical modeling of distributions of tasks' CPU requirements, so we give methods and heuristics we developed for doing this modeling and incorporating the results into PACE's optimal formula. We show that PACE is extremely effective on simulated workloads, reducing processor energy consumption of DVS algorithms by an average of 20.6%.

Chapter V examines the workloads we observed in VTrace traces collected on users' machines over the course of several months each. We perform several analyses of these workloads to devise guidelines for the design of dynamic voltage scaling algorithms. For instance, we discover that user-interface events of different categories, such as pressing a letter key or pressing the enter key, trigger significantly different amounts of processing. Therefore, a dynamic voltage scaling algorithm can gain significant information about future processing needs by observing when user-interface events occur and to what category they belong.

Chapter VI tells how we implemented RightSpeed, a task-based dynamic voltage scaling system for Windows 2000 that incorporates the theories of Chapter IV and the suggestions of Chapter V. This expansion of the operating system allows applications to specify when tasks begin and end, and what their deadlines are, to guide appropriate dynamic voltage scaling decisions. RightSpeed can also automatically detect the characteristics of certain tasks triggered by user-interface events. We have implemented RightSpeed on two systems capable of dynamic voltage scaling, one containing a Transmeta Crusoe$^{\text{TM}}$ chip and the other containing an AMD Mobile Athlon 4 chip. We show that RightSpeed uses little background overhead, about 1.2%, and implements operations such as PACE calculation in only microseconds. We find that PACE is not effective at saving energy on these processors, but expect it to be more worthwhile in the future as processors with greater ranges of speeds become available.

Chapter II

Software Energy Management

II.1 Introduction

This chapter describes how software is used to manage the energy consumption of various computer components. It thus provides background on the general issue of software energy management. It also helps motivate the central issue of this thesis, that software should play a large role in energy management.

We classify the software issues created by power-saving hardware features into three categories: transition, load-change, and adaptation. The transition problem involves answering the question, "When should a component switch from one mode to another?" The load-change problem involves answering the question, "How can the functionality needed from a component be modified so that it can more often be put into low-power modes?" The adaptation problem involves answering the question, "How can software be modified to permit novel, power-saving uses of components?" Each of the software strategies we will consider addresses one or more of these problems.

Different components have different energy consumption and performance characteristics, so it is generally appropriate to have a separate energy management strategy for each such component. Thus in this chapter we will generally consider each component separately. For each component, first we will discuss its particular hardware characteristics, then we will discuss what transition, load-change, and adaptation solutions have been proposed

Category	**Description**
Transition	When should a component switch between modes?
Load-change	How can a component's functionality needs be modified so it can be put in low-power modes more often?
Adaptation	How can software permit novel, power-saving uses of components?

Table II.1: Categories of energy-related software problems

for that component. The components whose software power management issues are most significant are the secondary storage device, the processor, the wireless communication device, and the display, but we will also briefly discuss other components.

This chapter is organized as follows. Section II.2 discusses general issues in developing and evaluating solutions to the problems we have discussed. Sections II.3, II.4, II.5, and II.6 talk about specific problems and solutions involving the secondary storage device, the processor, the wireless communication device, and the display, respectively. Section II.7 considers other, miscellaneous, components. Section II.8 talks about strategies that deal with the system itself as a component to be power-managed. Finally, in Section II.9, we conclude.

II.2 General Issues

II.2.1 Strategy types

We call a strategy for determining when to switch from one component mode to another a *transition strategy.* Transition strategies require two sorts of information about a component: knowledge about its mode characteristics and information about its future functionality requirements. By mode characteristics we mean the advantages and disadvantages of each mode the component can be in, including how much power is saved by being in it, how much functionality is sacrificed by entering it, and how long it will take to return from it.

Mode characteristics are generally more easily obtained than future functionality requirements, so the most difficult part of a transition strategy is predicting future functionality requirements. Thus, transition strategies are sometimes called *prediction strategies.*

The most common, but not the only, prediction tactic is to assume that the longer a component has been inactive, the longer it will continue to be inactive. Combining this prediction method with knowledge about mode characteristics then leads to a period t such that whenever the component is inactive in a certain mode for longer than t, it should be placed in a lower-power mode. Such a period is called an *inactivity threshold*, and a strategy using one is called an inactivity threshold strategy.

We call a strategy for modifying the load on a component to increase its use of low-power modes a *load-change strategy.* Disk caching is an example, since it can reduce the load on a hard disk and thereby reduce its power consumption. Note that modifying component load does not always mean reducing it; sometimes merely reordering service requests can reduce power consumption. For instance, the hard disk will consume less power if one makes a disk request immediately before spinning the disk down than if one makes the request immediately after spinning it down.

We call a strategy for allowing components to be used in novel, power-saving ways an *adaptation strategy.* An example is modifying file layout on secondary storage so that magnetic disk can be replaced with lower-power flash memory.

II.2.2 Levels of energy management

Energy management can be done at several levels in the computer system hierarchy: the component level, the operating system level, the application level, and the user level. The end-to-end argument [SRC84] suggests that this management should be performed at the highest level possible, because lower levels have less information about the overall workload. However, certain types of strategy are inappropriate for the highest levels. Most strategies are inappropriate for the user, since the user lacks knowledge about power consumption of each component, is unable to make decisions within milliseconds or faster, and is generally unwilling to make frequent energy management decisions. Problems with the application level are that applications operate independently and that applications lack certain useful information about the state of the machine because of operating system abstraction. For these reasons, most energy management is best performed at the operating system level. The user typically just makes a few high-level decisions and applications typically just reduce their use of components.

One way to get the advantages of application-level management without most associated disadvantages is to use application-aware adaptation [IM93, NPS95]. In such a system, each application explicitly tells the operating system what its future needs are, and the operating system notifies each application whenever is a change in the state of the system relevant to energy management decisions. Thus, if an energy management strategy has to be implemented at the operating system level, it can still get information about the needs of an application from the definitive source: the application itself. Furthermore, if an energy management strategy is best implemented at the application level, it can be performed using machine state information normally confined to the operating system. Unfortunately, it is seldom the case that applications have the necessary knowledge or sophistication to take advantage of the ability to obtain or supply power-relevant information.

II.2.3 Strategy evaluation

When evaluating power management strategies, there are several points to remember. First, the effect of a strategy on the overall system power consumption is more important than its effect on the particular component it concerns. For example, a 50% reduction in modem power sounds impressive, but if the modem only accounts for 4% of total power consumption, this savings will only result in a 2% decrease in total power.

Second, it is important to use as the baseline the current strategy, rather than the worst possible strategy. For example, it would not be sufficient to simply know that a new strategy causes the disk motor to consume 19% of its maximum possible power. If the current strategy already caused it to be off 80% of the time, this would represent a small power reduction, but if the current strategy only turned it off 20% of the time, this would represent a significant power reduction.

Third, minimum energy consumption (and thus maximum battery lifetime in the case of portable computers) is not necessarily what users want—they want to maximize the amount of work they can accomplish with a given amount of energy, not simply the amount of time the computer can remain running on that amount of energy. For example, consider a strategy that halves the CPU speed and increases battery lifetime by 50%. If the sluggish response time makes papers take 10% longer to write, it is not reasonable to call the new strategy a 50% improvement just because the machine stays on 50% longer. The user can only write 36% more papers with one battery, so the strategy is really only a 36% improvement. Thus, to completely evaluate a new strategy, one must take into account not only how much power it saves, but also how much it extends or diminishes the time tasks take.

Fourth, when evaluating a strategy, it is important to consider and quantify its effect on components it does not directly manipulate. For example, a strategy that slows down the CPU may cause a task to take longer, thus causing the disk and backlight to be on longer and consume more energy.

Fifth, to be completely accurate in evaluations of battery lifetime on portable computers, one also has to consider that battery capacity is not constant. Battery capacity can vary depending on the rate of power consumption [Pow95] and on the way that that rate changes with time [ZR97]. Thus, it may be important to understand not only how much a strategy reduces power consumption, but also how it changes the function of power consumption versus time. Also, it means that computing battery lifetime is more difficult than just dividing a rated energy capacity by total power consumption.

In conclusion, there are four things one must determine about a component power management strategy to evaluate it: how much it reduces the power consumption of that component; what percentage of total system power, on average, is due to that component; how much it changes the power consumption of other components; and how it affects battery capacity through its changes in power consumption. The first, third, and fourth require simulation of the strategy; the second requires a power budget describing the average power consumption of each system component. In the next subsection, we will give some such budgets.

II.2.4 Power budget

Table II.2 shows examples of average power consumption for the components of some portable computers when power-saving techniques are used. This table shows measurements taken only when the computers were running off battery power, since we are most concerned with power management at such times; power management when a machine is

Component	**Hypo-thetical 386**	**Duo 230**	**Duo 270c**	**Duo 280c**	**Average**
Processor	4%	17%	9%	25%	14%
Hard disk	12%	9%	4%	8%	8%
Backlight	17%	25%	26%	25%	23%
Display	4%	4%	17%	10%	9%
Modem	n/a	1%	0%	5%	2%
FPU	1%	n/a	3%	n/a	2%
Video	26%	8%	10%	6%	13%
Memory	3%	1%	1%	1%	2%
Other	33%	35%	28%	22%	30%
Total	6 W	5 W	4 W	8 W	6 W

Table II.2: For various portable computers, percentage of total power used by each component when power-saving techniques are used [Lor95a, Mac91]

plugged in is less critical, may have different tradeoffs, and may experience different user behavior. Note that power supply inefficiency is not treated as a separate category, but rather as a "tax" on all power consumed by each component. So, for instance, if the power supply system is 80% efficient, then instead of attributing 20% of power consumption to the power supply we increase the effective power consumption of each component by 25%. The first machine is a hypothetical 386DXL-based computer [Mac91]. The next three examples describe measurements of Macintosh PowerBook Duo machines [Lor95a]. The Duo 230 has a supertwist monochrome display while the other Duos have active-matrix color displays.

The power budget of Table II.2 indicates the magnitude of possible power savings. For instance, since the hard disk consumes only 8% of total power on the Duo 280c given its current power-saving methods, better techniques for managing hard disk power could save at most 8% of total system power, increasing battery lifetime by at most 9%. With power management active, the main consumers of power include the backlight, processor, video system, and hard disk. Thus, these are the components for which further power-saving methods will be most important.

Note that these breakdowns are likely to change as time progresses [HDP$^+$95]. For instance, wireless communication devices are increasingly appearing in portable computers, adding about 1 W to total power consumption. Hardware improvements will decrease the power consumption of various other components, but this rate of decrease will be different for different components. In 1994, Douglis et al [DKM94b] observed that later models of portable computers seemed to spend a greater percentage of their power consumption on the hard disk than earlier models. Presumably, this was because later models had substantial relative savings in other components' power but not as much savings in hard disk power. These forecasts suggested that as time progressed, power-saving techniques might become more important for the display and hard disk, and less important for the processor. However, this turned out not to be the case, as in the intervening seven years processor power consumption has far outpaced disk power consumption as processor speeds escalated according to Moore's law. In the coming years, we may see a reversing of that trend, as users become more satisfied with moderate processor speeds and as processor manufacturers increasingly use lower processor supply voltages.

II.2.5 Battery technology

The importance of energy management in portable computers arises as much from limited battery capacity as from high power use. And, unfortunately, battery technology has been improving at only a modest pace in terms of increased capacity per unit weight and volume. The highest capacity battery technology currently used in portables is lithium ion, providing as much as 380 W-h/L and 135 W-h/kg [Fuj97]. This is an improvement over the roughly 260–330 W-h/L and 120 W-h/kg achievable from them in 1995 and the roughly 180 W-h/L achievable from them in 1991. Most impressive, though, is their improvement over earlier battery technologies, such as nickel-metal hydride with its 150 W-h/L and 50 W-h/kg in 1995 and nickel-cadmium with its 125 W-h/L and 50 W-h/kg in 1995 [Pow95]. Technologies in development, such as lithium polymer, lithium anode, zinc-manganese dioxide, and zinc-air, may lead to even higher battery capacities in the future [Pow95]. As an example of the battery capacity one can get today, a modern Dell Inspiron 4000 laptop comes with a 26.5 W-h lithium ion battery [Del01].

II.3 Secondary Storage

II.3.1 Hardware features

Secondary storage in modern computers generally consists of a magnetic disk supplemented by a small amount of DRAM used as a disk cache; this cache may be in the CPU main memory, the disk controller, or both. Such a cache improves the overall performance of secondary storage. It also reduces its power consumption by reducing the load on the hard disk, which consumes more power than the DRAM.

Most hard disks have five power modes; in order of decreasing power consumption, these are active, idle, standby, sleep, and off [HDP$^+$95]. In active mode, the disk is seeking, reading, or writing. In idle mode, the disk is not seeking, reading, or writing, but the motor is still spinning the platter. In standby mode, the motor is not spinning and the heads are parked, but the controller electronics are active. In sleep mode, the host interface is off except for some logic to sense a reset signal; thus, if there is a cache in the disk controller, its contents are lost. Transitions to active mode occur automatically when uncached data is accessed. Transitions to standby and sleep modes occur when explicit external directives are received; this is how software power-saving strategies influence hard disk power consumption.

Having the motor off, as in the standby mode, saves power. However, when it needs to be turned on again, it will take considerable time and energy to return to full speed. If this energy is greater than the savings from having the motor off, turning the motor off may actually increase energy consumption. Turning off the motor also has a performance impact, since the next disk request will be delayed until the motor returns to full speed. In addition, while the disk is returning to full speed, other components will typically continue consuming power, also increasing energy use. Going to sleep mode is an analogous operation, although one in which the savings in power, as well as the overhead required to return to the original state, are greater. Table II.3 quantifies some time and energy considerations for various hard disks.

A possible technology for secondary storage is an integrated circuit called flash memory [CDLM93, DKM$^+$94a, KNM95, MDK93, WZ94]. Like a hard disk, such memory is nonvolatile and can hold data without consuming energy. Furthermore, when reading or writing, it consumes only 0.15 to 0.47 W, far less than a hard disk. It has a read speed of

Hard disk	**Toshiba MK3017GAP**	**IBM Travel-star 48GH**	**Fujitsu MHL2300AT**	**Hitachi DK22AA-18**
Capacity	30 GB	48 GB	30 GB	18 GB
Idle power	0.7 W	0.9 W	0.85 W	0.8 W
Standby power	0.3 W	0.25 W	0.28 W	0.25 W
Sleep power	0.1 W	0.1 W	0.1 W	0.125 W
Spin-up time	4 sec	1.8 sec	5 sec	3 sec
Spin-up energy	10.8 J	9.0 J	22.5 J	13.5 J

Table II.3: Characteristics of various hard disks [Tos01, IBM01, Fuj01, Hit01]

about 85 ns per byte, similar to DRAM, but a write speed of about 4–10 μs per byte, about 10–100 times slower than hard disk. However, since flash memory has no seek time, its overall write performance is not that much worse than that of magnetic disk; in fact, for sufficiently small random writes, it can actually be faster. Flash memory is technically read-only, so before a region can be overwritten it must be electrically erased. Such erasure is done one full segment at a time, with each segment 0.5–128 KB in size and taking about 15 μs per byte to erase [WZ94]. A segment can only be erased 100,000 to 1,000,000 times in its lifetime before its performance degrades significantly, so the operating system must ensure that the pattern of erasures is reasonably uniform, with no single segment getting repeatedly erased. The current cost per megabyte of flash is $1–3 [She01], making it about 125–450 times more expensive than hard disk and about 8–24 times more expensive than DRAM. Flash memory offers great opportunities for secondary storage power savings if it can be substituted for the hard disk or used for caching. Before that, however, software must be designed to overcome the many limitations of flash memory, especially its poor write performance.

II.3.2 Transition strategies

Transition strategies for magnetic disks can be of three kinds: deciding when to go to sleep mode, deciding when to go to standby mode, and deciding when to turn off the disk completely. Most are of the first kind. We know of no studies of the second kind, for reasons we will discuss in the next paragraph. Strategies of the third kind also exist, but are generally simple inactivity threshold strategies that have not been experimentally scrutinized.

One reason for the lack of study of transition strategies for deciding when to enter standby mode is that this mode is a relatively new feature on disks. Another reason is that it may often be better to enter sleep mode than standby mode. Sleep mode consumes less power, and since the time it takes to go from sleep to idle mode is dominated by the spin-up time of the motor, this transition takes no longer than that from standby to idle mode. The main advantage to standby mode is that on-disk cache contents are preserved; this may or may not be significant, depending on the caching algorithm in the disk controller, and whether or not the main memory disk cache is a superset of the contents of the controller disk cache.

II.3.2.1 Fixed inactivity threshold

The most common transition strategy for going into sleep mode is to enter that mode after a fixed inactivity threshold. When hard disks allowing external control over the motor were first developed, their manufacturers suggested an inactivity threshold of 3–5 minutes. However, researchers soon discovered that power consumption could be minimized by using inactivity thresholds as low as 1–10 seconds; such low thresholds save roughly twice as much power as a 3–5 minute threshold [DKM94b, LKHA94].

The greater power savings from using a smaller inactivity threshold comes at a cost, however: perceived increased user delay. Spinning down the disk more often makes the user wait more often for the disk to spin up. The inactivity threshold yielding minimum disk power results in user delay of about 8–30 seconds per hour; some researchers believe this to be an unacceptable amount of delay [DKM94b], although in absolute terms, this amount is trivial. Another problem with short inactivity thresholds is that disks tend to last for only a limited number of start-stop cycles, and excessively frequent spin up-spin down cycles could cause premature disk failure. Thus, the best disk spin-down policy is not necessarily the one that minimizes power consumption, but the one that minimizes power consumption while keeping user delay and start-stop frequency at an acceptable level.

It is worth pointing out, although it should be obvious, that the time between disk accesses is not exponentially distributed; the expected time to the next disk access is generally an increasing function of the time since the last access. If interaccess times for disk reference were exponentially distributed, the correct strategy would use an inactivity threshold of either zero or infinity [Gre94].

II.3.2.2 Changing inactivity threshold

There are several arguments for dynamically changing the inactivity threshold, not necessarily consistent with each other. The first argument is that disk request interarrival times are drawn independently from some unknown stationary distribution. Thus, as time passes one can build up a better idea of this distribution, and from that deduce a good threshold. The second argument is that the interarrival time distribution is nonstationary, i.e., changing with time, so a strategy should always be adapting its threshold to the currently prevailing distribution. This distribution can be inferred from samples of the recent distribution and/or from factors on which this distribution depends. The third argument is that worst-case performance can be bounded by choosing thresholds randomly—any deterministic threshold can fall prey to a particularly nasty series of disk access patterns, but changing the threshold randomly eliminates this danger.

If disk interarrival times are independently drawn from some unknown stationary distribution, as the first argument states, then no matter what this distribution, there exists an inactivity threshold that incurs a cost no more than $e/(e-1)$ times that of the optimal off-line transition strategy [KMMO94]. One could find this threshold by keeping track of all interarrival times so that the distribution, and thus the ideal threshold, could be deduced.

One algorithm of that type, using constant space, builds up a picture of the past interarrival time distribution in the following indirect way [KLV95]. It maintains a set of possible thresholds, each with a value indicating how effective it would have been. At any point, the algorithm chooses as its threshold the one that would have performed the best. Incidentally, "best" does not simply mean having the least power consumption; the valuation might take into account the relative importance of power consumption and frequency of disk spin-downs specified by the user. This algorithm has been shown to perform well on real traces, beating many other practical algorithms.

Another strategy using a list of candidate thresholds is based on the second argument, that disk access patterns change with time [HLS96]. In this strategy, each candidate is initially assigned equal weight. After each disk access, candidates' weights are increased or decreased according to how well they would have performed relative to the optimal off-line strategy over the last interaccess period. At any point, the threshold chosen for actual use is the weighted average of all the candidates. Simulations show that this strategy works well on actual disk traces. The developers of this strategy only considered using it to minimize power consumption; however, it could easily be adapted to take frequency of spin-ups into account.

Another dynamic strategy based on the second argument tries to keep the frequency of annoying spin-ups relatively constant even though the interaccess time distribution is always changing [DKB95]. This strategy raises the threshold when it is causing too many spin-ups and lowers it when more spin-ups can be tolerated. Several variants of this strategy, which raise and lower the threshold in different ways, are possible. Simulation of these variants suggests that using an adaptive threshold instead of a fixed threshold can significantly decrease the number of annoying spin-ups experienced by a user while increasing energy consumption by only a small amount.

Note that all the dynamic strategies we have described that are based on the second argument make inferences about the current distribution of disk access interarrival times based on recent samples of this distribution. However, there are likely other factors on which this distribution depends and on which such inferences could be based, such as the current degree of multiprogramming or which applications are running. Additional research is needed to determine which of these factors can be used effectively in this way.

By the third argument, a strategy should make no assumptions about what the disk access pattern looks like, so that it can do well no matter when disk accesses occur. One such strategy chooses a new random threshold after every disk access according to the cumulative distribution function

$$\pi(t) = \frac{e^{t/c} - 1}{e - 1},$$

where c is the number of seconds it takes the running motor to consume the same amount of energy it takes to spin up the disk [KMMO94]. This strategy has been proven ideal among strategies having no knowledge of the arrival process. Note, however, that almost all transition strategies described in this chapter do purport to know something about the arrival process, and thus are capable of beating this strategy. In other words, although this strategy does have the best worst-case expected performance, it does not necessarily have the best typical-case performance.

II.3.2.3 Alternatives to an inactivity threshold

Some transition strategies have been developed that do not use an inactivity threshold explicitly [DKM94b]. One such strategy is to predict the actual time of the next disk access to determine when to spin down the disk. However, simulations of variants of this strategy show that they provide less savings than the best inactivity threshold strategy, except when disk caching is turned off. This may be because filtering a pattern of disk accesses through a disk cache makes it too patternless to predict. Another strategy is to predict the time of the next disk request so the disk can be spun up in time to satisfy that request. However, no techniques proposed for this have worked well in simulation, apparently because the penalty for wrong prediction by such strategies is high. Despite the shortcomings of the non-threshold-based transition strategies studied so far, some researchers remain hopeful about the feasibility of such strategies. Simulation of the optimal off-line strategy indicates that such strategies could save as much as 7–30% more energy than the best inactivity threshold method.

II.3.3 Load-change strategies

Another way to reduce the energy consumption of a hard disk is to modify its workload. Such modification is usually effected by changing the configuration or usage of the cache above it.

One study found that increasing cache size yields a large reduction in energy consumption when the cache is small, but much lower energy savings when the cache is large [LKHA94]. In that study, a 1 MB cache reduced energy consumption by 50% compared to no cache, but further increases in cache size had a small impact on energy consumption, presumably because cache hit ratio increases slowly with increased cache size [ZS97]. The study found a similar effect from changing the dirty block timeout period, the maximum time that cache contents are permitted to be inconsistent with disk contents. Increasing this timeout from zero to 30 seconds reduced disk energy consumption by about 50%, but further increases in the timeout delay had only small effects on energy consumption [LKHA94].

Another possible cache modification is to add file name and attribute caching. Simulation showed a moderate disk energy reduction of 17% resulting from an additional 50 KB of cache devoted to this purpose [LKHA94].

Prefetching, a strategy commonly used for performance improvement, should also be effective as an energy-saving load-change strategy. If the disk cache is filled with data that will likely be needed in the future before it is spun down, then more time should pass before it must again be spun up. This idea is similar to that of the Coda file system [SKM$^+$93], in which a mobile computer caches files from a file system while it is connected so that when disconnected it can operate independently of the file system.

Another approach to reducing disk activity is to design software that reduces paging activity. This can be accomplished by reducing working set sizes and by improving memory access locality. There are many things operating system and application designers can do to achieve these goals.

II.3.4 Adaptation strategies for flash memory as disk cache

Flash memory has two advantages and one disadvantage over DRAM as a disk cache. The advantages are nonvolatility and lower power consumption; the disadvantage is poorer write performance. Thus, flash memory might be effective as a second-level cache below the standard DRAM disk cache [MDK93]. At that level, most writes would be flushes from the first-level cache, and thus asynchronous. However, using memory with such different characteristics necessitates novel cache management strategies.

The main problem with using flash memory as a second-level cache is that data cannot be overwritten without erasing the entire segment containing it. One solution is to ensure there is always a segment with free space for writing; this is accomplished by periodically choosing a segment, flushing all its dirty blocks to disk, and erasing it [MDK93]. One segment choosing strategy is to choose the one least recently written; another is to choose the one least recently accessed. The former is simpler to implement and ensures no segment is cleaned more often than another, but the latter is likely to yield a lower read miss ratio. Unfortunately, neither may be very good in packing together blocks that are referenced together. One approach is to supplement the strategy with a copying garbage collector, such as that found in LFS [RO92], to choose recently used blocks of a segment about to be erased and write them into a segment that also contains recently used blocks and is not going to be erased.

Simulations have shown that using a second-level flash memory cache of size 1–40 MB can decrease secondary storage energy consumption by 20–40% and improve I/O response time by 30–70% [MDK93]. Thus, using appropriate cache management policies seem to allow a flash memory second-level cache to reduce energy consumption and still provide equal or better performance than a system using a traditional cache. The simulations would have been more persuasive, however, if they had compared the system with flash to one with a DRAM second-level cache rather than to one with no second-level cache.

II.3.5 Adaptation strategies for flash memory as disk

Flash memory is a low-power alternative to magnetic disk. However, the large differences between flash memory and magnetic disk suggest several changes to file system management. Since flash has no seek latency, there is no need to cluster related data on flash memory for the purpose of minimizing seek time [CDLM93]. Since flash is practically as fast as DRAM at reads, a disk cache is no longer important except to be used as a write buffer [CDLM93]. Such a write buffer would make writes to flash asynchronous, thus solving another problem of flash memory: poor write performance. In fact, if SRAM were used for this write buffer, its permanence would allow some writes to flash to be indefinitely delayed [DKM$^+$94a, WZ94]. Finally, unlike magnetic disk, flash memory requires explicit erasure before a segment can be overwritten, a slow operation that can wear out the medium and must operate on a segment at a time. One solution to these problems is to use a log-structured file system like LFS [KNM95, RO92], in which new data does not overwrite old data but is instead appended to a log. This allows erasures to be decoupled from writes and done asynchronously, thus minimizing their impact on performance. A flash file system also needs some way to ensure that no physical segment is cleaned especially often. One way to do this is to make sure that physical segments containing infrequently modified data and ones containing frequently modified data switch roles occasionally [WZ94].

Simulations of flash file systems using some of these ideas have found that they can reduce secondary storage power consumption by 60–90% while maintaining aggregate performance comparable to that of magnetic disk file systems [DKM+94a]. However, at high levels of utilization, the performance of file systems using asynchronous erasure can degrade significantly due to the overhead of that erasure.

II.3.6 Adaptation strategies for wireless network as disk

Another hardware approach to saving secondary storage energy is to use wireless connection to a plugged-in file server instead. Offloading storage has the advantage that the storage device can be big and power-hungry without increasing the weight or power consumption of the portable machine. Disadvantages include increased power consumption by the wireless communication system, increased use of limited wireless bandwidth, and higher latency for file system accesses. Adaptation strategies can help minimize the impact of the disadvantages but retain the advantages.

The general model for using wireless communication as secondary storage is to have a portable computer transmit data access requests to, and receive data from, a server. An improvement on this is to have the server make periodic broadcasts of especially popular data, so the portable computer will have to waste less power transmitting requests [IVB94]. A further improvement is to interleave index information in these broadcasts, so a portable computer can anticipate periods of no needed data and shut its receiver off during them.

Another model for using wireless communication for storage is proposed in extensions to the Coda scheme [SKM+93]. In this model, the portable computer storage system functions merely as a large cache for the server file system. When the portable computer is not wired to the file system server, it services cache misses using wireless communication. Because such communication is slow and bandwidth-consuming, the cache manager seeks to minimize its frequency by hoarding files that are anticipated to be needed during disconnection.

A third model for using wireless communication for storage, used by InfoPad, is to perform *all* processing on an unmoving server [BBB+94, BBB+95]. In this model, the portable "computer" is merely a terminal that transmits and receives low-level I/O information, so the energy consumption for general processing and storage is consumed by plugged-in servers instead of the mobile device. In this way, portable storage and CPU energy consumption are traded for high processing request latency, significant network bandwidth consumption, and additional energy consumption by the portable wireless device.

The limiting factor in all of these cases is network bandwidth; what is practical depends on the bandwidth between the local system and the data source. A packet radio connection at 28.8 Kb/s is very different than the type of multi-megabit per second system that could be implemented within a building.

II.3.7 Future hardware innovations

Researchers working on technological advances in hardware can also do much to aid software techniques in reducing power. To minimize the impact of decisions to spin down the hard disk, the energy and time consumed by a disk when spinning up should be reduced. Since disk power use drops roughly quadratically with rotation speed, it would also be useful to enable the disk to be put into low rotation speed modes, so that software could sacrifice some I/O performance to obtain reduced disk power consumption. An additional benefit of reduced rotation speed would be a reduction in spin-up times. To make it easier for software to achieve good performance with flash memory, its design should emphasize fast writing and erasing, as well as the ability to erase at the same time that data is being read or written. Increasing the number of erasures possible in the lifetime of a segment would simplify the management of flash memory.

II.4 Processor

II.4.1 Hardware features

Processors designed for low-power computers have many power-saving features. One power-saving feature is the ability to slow down the clock. Another is the ability to selectively shut off functional units, such as the floating-point unit; this ability is generally not externally controllable. Such a unit is usually turned off by stopping the clock propagated to it. Finally, there is the ability to shut down processor operation altogether so that it consumes little or no energy. When this last ability is used, the processor typically returns to full power when the next interrupt occurs.

In general, slowing down the processor clock without changing the voltage is not useful. Power consumption P is essentially proportional to the clock frequency f, the switching capacitance C, and the square of the voltage V ($P \propto CV^2 f$), but the time t it takes the processor to complete a task is inversely proportional to the clock frequency ($t \propto 1/f$). Since the energy E it takes the processor to complete a task is the product of its power consumption and the time it spends ($E = Pt$), this energy consumption is invariant with clock speed. Thus, reducing the clock speed lengthens processor response time without reducing the amount of energy the processor consumes during that time. In fact, slowing the clock speed will usually increase total energy consumption by extending the time other components need to remain powered. However, if the voltage can be decreased whenever the clock speed is reduced, then energy consumption, which is proportional to the square of the voltage ($E \propto CV^2$), would usually be reduced by slowing the clock.

Dynamic voltage scaling (DVS) is the ability of a processor to dynamically change its supply voltage to reduce its energy consumption. Since this necessitates a reduction in clock frequency whenever voltage is reduced, DVS also involves dynamic changing of clock frequency. In general, the speed used is roughly proportional to the voltage, so energy consumption is proportional to the square of the frequency ($E \propto f^2$) [WWDS94]. More precisely, frequency should be set equal to $k(V - V_{\mathrm{th}})^2/V$ where k is some constant [CSB92]. This yields the following relationship between energy and frequency: $E \propto \left(V_{th} + \frac{f}{2k} + \sqrt{\frac{V_{th} f}{k} + \left(\frac{f}{2k}\right)^2} \right)^2$.

Turning off a processor has little downside; no excess energy is expended turning the processor back on, the time until it comes back on is barely noticeable, and the state of the processor is unchanged from it turning off and on, unless it has a volatile cache [GDE+94]. On the other hand, there is a clear disadvantage to reducing the clock rate: tasks take longer. There may also be a slight delay while the processor changes clock speed.

Reducing the power consumption of the processor saves more than just the energy of the processor itself. When the processor is doing less work, or doing work less quickly, there is less activity for other components of the computer, such as memory and the bus. For example, when the processor on the Macintosh Duo 270c is off, not only is the 1.15 W of the processor saved, but also an additional 1.23 W from other components [Lor95a]. Thus, reducing the power consumption of the processor can have a greater effect on overall power savings than it might seem from merely examining the percentage of total power attributable to the processor.

II.4.2 Transition strategies for turning the processor off

When the side effects of turning the processor off and on are insignificant, the optimal off-line transition strategy is to turn it off whenever the processor will not be needed until the next interrupt occurs. With a well-designed operating system, this can be deduced from the current status of all processes. Thus, whenever any process is running or ready to run, the processor should not be turned off; when all processes are blocked, the processor should be turned off [LS96, SCB96, SU93]. Examples of operating systems using this strategy are Windows [Con92, Pie93] and UNIX.

MacOS, however, uses a different strategy, perhaps because its strategy was designed when processors *did* suffer side effects from turning off. It uses an inactivity threshold, as is commonly used for hard disk power management. The processor is shut off when there have been no disk accesses in the last fifteen seconds and no sound chip accesses, changes to the cursor, displaying of the watch cursor, events posted, key presses, or mouse movements in the last two seconds. The savings achievable from this strategy vary greatly with workload [Lor95b, Lor95a].

II.4.3 Load-change strategies when the CPU can turn off

Given a transition strategy that turns off the processor when it is not performing any tasks, the goal of a load-change strategy is simply to limit the energy needed to perform tasks. This suggests three approaches: reducing the time tasks take, using lower-power instructions, and reducing the number of unnecessary tasks. Below we present several load-change strategies that use different subsets of these approaches.

One technique, which uses the first two approaches, is to use more efficient operating system code [SML94]. However, if overall system performance has not been a sufficient reason for system designers and implementors to write good code, energy efficiency considerations are unlikely to make much difference.

Another technique, which uses the same approaches, is to use energy-aware compilers, i.e., compilers that consider the energy efficiency of generated code [TMWL96]. Traditional compiler techniques have application as load-change strategies in that they reduce the amount of time a processor takes to complete a task. Another way for the compiler to decrease energy consumption is to carefully choose which instructions to use, since some instructions consume more power than others. However, preliminary studies indicate that the primary gain from code generation is in decreasing the number of instructions executed, not in choosing between different equally long code sequences [TMWL96]. There may be some special cases, such as generating entirely integer versus mixed integer and floating point code, where the effects are significant, but we believe that these are the exception.

Another load-change technique uses the third approach, performing fewer unnecessary tasks [LS96]. In some cases, when an application is idle, it will "busy-wait" for an event instead of blocking. Then, the standard transition strategy will not turn off the processor even though it is not doing any useful work. One way to solve this problem is to force an application to block for a certain period whenever it satisfies certain conditions that indicate it is likely to be busy-waiting and not performing any useful activity. We will discuss our strategy for this further in Chapter III. We will describe simulations of such a strategy using traces of machines running on battery power; these simulations show that the strategy would allow the processor to be off, on average, 66% of the time, compared to 47% when no measures were taken to forcibly block applications.

II.4.4 Transition strategies for dynamically changing CPU speed

As explained before, slowing the clock is useless if voltage is kept constant. Therefore, when we discuss strategies to take advantage of slowing the processor clock, we are assuming that slowing the clock is accompanied by reducing the voltage. The voltage can be reduced when the clock speed is reduced because under those conditions the longer gate settling times resulting from lower voltage become acceptable.

Previous calculations have shown that the lowest energy consumption comes at the lowest possible speed. However, performance is also reduced by reducing the clock speed, so any strategy to slow the clock must achieve its energy savings at the expense of performance. Furthermore, reducing processor performance may cause an increase in the energy consumption of other components, since they may need to remain on longer. Thus, it is important to make an appropriate trade-off between energy reduction and performance.

Weiser et al. [WWDS94] and Chan et al. chan:1995 described the first strategies for making this trade-off. They designed these strategies to achieve two general goals. The first goal is to not delay the completion time of any task by more than several milliseconds. This ensures that interactive response times do not lengthen noticeably, and ensures that other components do not get turned off noticeably later. The second goal is to adjust CPU speed gradually. This is desirable because voltage scaling causes the minimum voltage V permissible at a clock speed f to be roughly linear in f. Thus, the number of clock cycles executed in an interval I is proportional to $\int_I f(t)\,dt$, while the energy consumed during that interval is roughly proportional to $\int_I [f(t) + C]^3\,dt$. Given these equations, it can be mathematically demonstrated that the most energy-efficient way to execute a certain number of cycles within a certain interval is to keep clock speed constant throughout the interval.

One strategy for adjusting CPU speed seeks to achieve these goals in the following way [WWDS94]. Time is divided into 10–50 ms intervals. At the beginning of each interval, processor utilization during the previous interval is determined. If utilization was high, CPU speed is slightly raised; if it was low, CPU speed is slightly lowered. If, however, the processor is falling significantly behind in its work, CPU speed is raised to the maximum allowable. Simulations of this strategy show 50% energy savings when the voltage, normally constrained to be 5 V, can be reduced to 3.3 V, and 70% savings when it can be reduced to 2.2 V. Interestingly, the strategy shows worse results when the voltage can be reduced to 1 V, seemingly because the availability of the low voltage causes the strategy to generate extreme variation in the voltage level over time. Obviously, a strategy should be designed so that it never yields worse results when the range over which parameters can be varied increases.

Another strategy also divides time into 10–50 ms intervals [CGW95]. At the beginning of each interval, it predicts the number of CPU busy cycles that will occur during that interval, and sets the CPU speed just high enough to accomplish all this work. There are actually many variants of this strategy, each using a different prediction technique. In simulations, the most successful variants were one that always predicted the same amount of work would be introduced each interval, one that assumed the graph of CPU work introduced versus interval number would be volatile with narrow peaks, and one that made its prediction based on a weighted average of long-term and short-term CPU utilization but ignored any left-over work from the previous interval. The success of these variants suggests that it is important for such a strategy to balance performance and energy considerations by taking into account both short-term and long-term processor utilization in its predictions. Too much consideration for short-term utilization increases speed variance and thus energy consumption. Too much consideration for long-term utilization makes the processor fall behind during especially busy periods and thus decreases performance.

Several authors, including Pering et al. [PBB98] and Grunwald et al. [GLF$^+$00], have shown that Weiser et al. and Chan et al.'s algorithms are impractical because they require knowledge of the future. However, they have proposed practical versions of these algorithms. Prediction methods they suggest include:

- **Past.** Predict the upcoming interval's utilization will be the same as the last interval's utilization.
- **Aged-a.** Predict the upcoming utilization will be the average of all past ones. More recent ones are more relevant, so weight the kth most recent by a^k, where $a \leq 1$ is a constant.
- **LongShort.** Predict the upcoming utilization will be the average of the 12 most recent ones. Weight the three most recent of these three times more than the other nine.
- **Flat-u.** Always predict the upcoming utilization will be u, where $u \leq 1$ is a constant.

Speed-setting methods they suggest include:

- **Weiser-style.** If the utilization prediction x is high ($> 70\%$), increase the speed by 20% of the maximum speed. If the utilization prediction is low ($< 50\%$), decrease the speed by $60 - x\%$ of the maximum speed.
- **Peg.** If the utilization prediction is high ($> 98\%$), set the speed to its maximum. If the utilization prediction is low ($< 93\%$), decrease the speed to its minimum positive value.

- **Chan-style.** Set the speed for the upcoming interval just high enough to complete the predicted work. In other words, multiply the maximum speed by the utilization to get the speed.

However, dividing time into intervals and using those boundaries as deadlines is somewhat arbitrary. For example, if a task arrives near the end of an interval, it does not really have to complete by the end of that interval. Furthermore, without deadlines, there is no particular reason to complete any given task by a certain time; it is best to simply measure the average number of non-idle cycles per second and run the CPU at that speed. (Transmeta's LongRun^{TM} system does something like this [Kla00].) Pering et al., recognizing this, suggested considering deadlines when evaluating DVS algorithms [PBB98]. To do so, they suggest considering a task that completes before its deadline to effectively complete at its deadline.

Grunwald et al. [GLF$^+$00] considered deadlines when they compared several of the algorithms described above (as well as others not listed here) by implementing them on a real system. They decided that although none of them are very good, Past/Peg is the best: it never misses any deadlines for the workload they considered, yet still saves a small but significant amount of energy.

There are several theoretical results that suggest approaches to scheduling CPU speed based on task deadlines. One important result is that if a set of tasks has feasible deadlines, scheduling them in increasing deadline order will always make all the deadlines [LL73]. Another useful result, described by Błażewicz et al. [BEP$^+$96, pp. 346–350], is that when the rate of consumption of some resource is a convex function of CPU speed, an ideal schedule will run each task at a constant speed. Yao et al. [YDS95] observe that with DVS, power consumption is a convex function of CPU speed. They show how to compute an optimal speed-setting policy by constructing an earliest-deadline-first schedule, and then choosing the minimal possible speed for each task that will still make the deadlines.

However, one can only compute such optimal schedules if the tasks' CPU requirements are known in advance, and task requirements in most systems are unpredictable random variables; see, e.g., [SBB72]. For this reason, most research on scheduling for DVS has focused on heuristics for estimating CPU requirements and attempting to keep CPU speed as constant as possible. In Chapter IV, we will describe our approaches for dynamic voltage scaling to meet deadlines.

II.4.5 Load-change strategies when functional units can turn off

Typically, if functional units can be turned off, the chip internals turn them off automatically when unused. This makes software transition strategies unnecessary and impossible to implement, but makes load-change strategies tenable. One such load-change strategy is to use a compiler that clusters together several uses of a pipelined functional unit so that the unit is on for less time. Another is, when compiling, to preferentially generate instructions using functional units that do not get power-managed. However, no research has been done yet on such load-change strategies. It is unclear whether the power savings to be obtained from these strategies would be significant.

II.4.6 Future hardware innovations

There are several things that researchers working on technological advances in hardware can do to increase the usefulness of software strategies for processor energy reduction. Perhaps the most important is designing the motherboard so that reduction in the energy consumption of the processor yields a consequent large reduction in the energy consumption of other components. In other words, motherboard components should have states with low power consumption and negligible transition side effects that are automatically entered when the processor is not presenting them with work. Voltage scaling is not widely available, and needs to be made more so. Once this happens, the software strategies that anticipate this technology can be put to good use. Finally, if hardware designers can keep the time and energy required to make transitions between clock speeds low, the savings from clock speed transition strategies will be even greater.

II.5 Wireless communication devices

II.5.1 Hardware features

Wireless communication devices are appearing with increasing frequency on portable computers. These devices can be used for participating in a local or wide area wireless network, or for interacting with disconnected peripherals like a printer or mouse. They typically have five operating modes; in order of decreasing power consumption, these are transmit, receive, idle, sleep, and off [HDP$^+$95]. In transmit mode, the device is transmitting data; in receive mode, the device is receiving data; in idle mode, it is doing neither, but the transceiver is still powered and ready to receive or transmit; in sleep mode, the transceiver circuitry is powered down, except sometimes for a small amount of circuitry listening for incoming transmissions. Typically, the power consumption of idle mode is not significantly less than that of receive mode [SK97], so going to idle mode is not very useful. Transitions between idle and sleep mode typically take some time. For example, HP's HSDL-1001 infrared transceiver takes about 10 μs to enter sleep mode and about 40 μs to wake from it [Hew96], AT&T's WaveLAN PCMCIA card and IBM's infrared wireless LAN card each take 100 ms to wake from sleep mode, and Metricom's Ricochet wireless modem takes 5 seconds to wake from sleep mode [SK97].

Some devices provide the ability to dynamically modify their transmission power. Reducing transmission power decreases the power consumption of the transmit mode; it also has the advantage of reducing the interference noise level for neighboring devices, leading to a reduction in their bit error rates and enabling higher cell capacity. The disadvantage, however, is that when a device reduces its transmission power, it decreases the signal to noise ratio of its transmissions, thus increasing its bit error rate.

Wireless device power consumption depends strongly on the distance to the receiver. For instance, the wide-area ARDIS system, in which each base station covers a large area, requires transmit power of about 40 W, but the wide-area Metricom system, which uses many base stations each serving a small area, requires mobile unit transmit power of only about 1 W [Cox95]. Local-area networks also tend to provide short transmission distances, allowing low power dissipation. For instance, the WaveLAN PCMCIA card, meant for use in such networks, consumes only about 1.875 W in transmit mode [Luc96b]. Even smaller distances, such as within an office or home, yield even smaller power requirements. For instance, the HSDL-1001 infrared transceiver consumes only 55 mW in transmit mode [Hew96], and the CT-2 specification used for cordless telephones requires less than 10 mW for transmission [Cox95].

II.5.2 Transition strategies for entering sleep mode

Transition strategy issues for wireless communication devices entering sleep mode are quite similar to those for hard disks, so the solutions presented for hard disks are generally applicable to them. However, some features of wireless communication suggest two changes to the standard inactivity threshold methods used with hard disks. First, because a wireless communication device does not have the large mechanical component a hard disk has, the time and energy required to put it in and out of sleep mode are generally much smaller. Further, the user is unlikely to know when the unit is off or on unless some sort of monitor is installed, so users should be unaware of start-up times unless they are unduly long. These factors suggest a more aggressive energy management strategy such as using a much shorter inactivity threshold. Second, it may be necessary to have wireless devices periodically exit sleep mode for a short period of time to make contact with a server, so that the server does not decide the unit is off or out of range and delete state information related to the connection [Luc96a].

Experimental simulation has been used to evaluate the effect of some transition strategies [SK97]. One simulation, which assumed a Ricochet modem was used only for the retrieval of electronic mail, considered a strategy that put the modem to sleep whenever no mail was being retrieved, and woke it up after a certain period of time to check for new mail. It showed that using a period of about four minutes would reduce the energy consumption of the wireless device by about 20%, and only cause mail to be, on average, two minutes old when it was seen by the user. Another simulation assumed a WaveLAN PCMCIA card was used only for Web browsing, and considered the strategy of putting the wireless device to sleep whenever a certain inactivity threshold passed with no outstanding HTTP transactions. It showed that using a very small inactivity threshold reduced power consumption of the device by 67% without noticeably increasing the perceived latency of document retrieval.

II.5.3 Load-change strategies when sleep mode is used

One way to increase the amount of time a wireless device can spend sleeping is simply to reduce network usage altogether. There are many strategies for doing this, some examples of which are as follows. One strategy is to compress TCP/IP headers; this can reduce their size by an order of magnitude, thus reducing the wireless communication activity of a mobile client [DENP96]. Another strategy is to reduce the data transmission rate or stop data transmission altogether when the channel is bad, i.e., when the probability of a dropped packet is high, so that less transmission time is wasted sending packets that will be dropped [ZR97]. Of course, if data transmission is ceased altogether, probing packets must be sent occasionally so that the unit can determine when the channel becomes good again. Yet another strategy is to have servers [NPS95] or proxies [FGBA96] use information about client machine characteristics and data semantics to provide mobile clients with versions of that data with reduced fidelity and smaller size; this reduces the amount of energy mobile clients must expend to receive the data. For example, a data server might convert a color picture to a black-and-white version before sending it to a mobile client. A fourth strategy is to design applications that avoid unnecessary communication, especially in the expensive transmit direction.

Another way is to use a medium access protocol that dictates in advance when each wireless device may receive data. This allows each device to sleep when it is certain that no data will arrive for it. For example, the 802.11 LAN standard has access points that buffer data sent to wireless devices and that periodically broadcast a beacon message indicating which mobile units have buffered data. Thus, each wireless device only has to be listening when it expects a beacon message, and if it discovers from that message that no data is available for it, it may sleep until the next beacon message [BD96]. Of course, strategies such as this either require a global (broadcast) clock or closely synchronized local clocks. Similar protocols, designed to increase battery life in one-way paging systems, include the Post Office Code Standardization Advisory Group (POCSAG) protocol and Motorola's FLEX protocol [MS95]. A different type of power-conserving protocol, which does not require buffering access points and is thus peer-to-peer, is LPMAC [MSGN+96]. LPMAC divides time into fixed-length intervals, and elects one terminal in the network the network coordinator (NC). The NC broadcasts a traffic schedule at the beginning of each interval that dictates when each unit may transmit or receive data during that interval. Each interval ends with a short contention period during which any unit may send requests for network time to the NC. Thus, each mobile unit only needs be awake during the broadcast of the traffic schedule, and may sleep until the next such broadcast if the schedule indicates that no one will be sending data to it. This protocol does not require intermediate buffering because data is buffered at the sending unit until the NC gives it permission to send.

II.5.4 Transition strategies for changing transmission power

Many approaches to dynamically changing the transmission power in wireless networks have been proposed. However, few of them were designed with consideration for the battery lifetime of mobile units, being meant solely to achieve goals like guaranteeing limits on signal to noise ratio, balancing received power levels, or maximizing cell capacities [RB96]. Here we will focus on those strategies that at least address the battery lifetime issue. Of course, a strategy cannot consider battery lifetime alone, but must balance the need for high battery lifetime with the need to provide reasonable cell capacity and quality of service.

A transition strategy to decide when to change power should should take into account the consequences of reducing transmission power: increased battery lifetime, lower bit error rate for neighbors (enabling higher cell capacities), and higher bit error rate for one's own transmissions. Such a strategy can be local, meaning that it accounts only for the needs of the wireless device on which it is running, or global, meaning that it accounts for the needs of other devices on the network. Global strategies seem most appropriate considering that the decisions one wireless unit makes about its transmission power and link bandwidth affect the other wireless units. Local strategies have the advantage of being simpler to implement, especially in a heterogeneous environment where communication between units in an attempt to cooperate is difficult. Global strategies, however, need not require global communication; estimates of the effect of power changes on interference with other units may be sufficient.

One suggested local transition strategy is to choose transmission power at each moment based on the current quality of service required and the current interference level observed, using a function analytically selected to optimize battery lifetime. Simulations show that such an approach can yield significant energy savings and no reduction in quality of service compared to other schemes that do not consider battery lifetime, such as one that attempts to always maintain a certain signal to noise ratio. The amount of energy savings obtained decreases with increasing quality of service required, since more required activity means less opportunity to reduce transmission power and save energy [RB96].

The developers of that strategy also considered a global variant of it, in which each mobile unit also considers the interference levels it has observed in the past when making its decisions about transmission power [RB96]. In this scheme, the only communication between mobile units is indirect, via the noise they generate for each other with their transmissions. Using such indirect communication makes implementation simpler, since it requires no protocol for explicit communication of control information. However, it does not allow the units to make intelligent distributed decisions, such as to use time-division multiple access, i.e., to take turns transmitting, so as to minimize interference and maximize use of cell capacity. Nevertheless, simulations indicate that even without explicit communication the strategy seems to achieve reasonable global behavior. One reason for this may be that when devices act independently to reduce their power levels when interference makes transmission unworthwhile, the machines will tend somewhat to take turns in transmitting. This is similar to back-off strategies in Ethernet.

Other global strategies can be imagined that use explicit communication of such things as quality of service needs and transmission power schedules. Such explicit communication would allow mobile units to coordinate their transmission power in an attempt to optimize overall battery lifetime given reasonable overall goals for quality of service and cell capacity.

II.5.5 Load-change strategies when transmission power can change

When a wireless device transmits with reduced power, its bit error rate increases. Load-change strategies can be used to mitigate the effect of this increased bit error rate and thus enable the use of less transmission power. One way to do this is to use more error correction code bits, although this has the side effect of reducing effective data bandwidth by consuming extra bandwidth for the additional code bits. Another way is to request more link bandwidth, to counter the effects of increased error correction and more frequent retransmission of dropped packets.

So, a strategy for modifying transmission power can be made more effective by simultaneously effecting changes in the amount of error correction and link bandwidth used. For instance, suppose the needed quality of service can be attained by transmitting with power P_1, using error correction method E_1, and consuming bandwidth B_1, or by transmitting with power P_2, using error correction method E_2, and consuming bandwidth B_2. Then, the strategy can choose among these two combined options based on how they will influence battery lifetime and cell capacity.

Some researchers have suggested a global strategy that allows mobile units in a network to optimize overall system utility by coordinating which units will be transmitting when, what transmission power each unit will use, and how much error correction each unit will use [LB96]. They considered only bit error rate and bandwidth seen by each user application in determining system utility, but their model could be adapted to take battery lifetime into account as well. Their system uses explicit communication, but uses a hierarchically distributed algorithm to reduce the complexity, control message bandwidth, and time required to perform the optimization.

II.6 Display and Backlight

II.6.1 Hardware features

The display and backlight have few energy-saving features. This is unfortunate, since they consume a great deal of power in their maximum-power states; for instance, on the Duo 280c, the display consumes a maximum of 0.75 W and the backlight consumes a maximum of 3.40 W [Lor95a]. The backlight can have its power reduced by reducing the brightness level or by turning it off, since its power consumption is roughly proportional to the luminance delivered [HDP$^+$95]. The display power consumption can be reduced by turning the display off. It can also be reduced slightly by switching from color to monochrome or by reducing the update frequency. Reducing the update frequency reduces the range of colors or shades of gray for each pixel, since such shading is done by electrically selecting each pixel for a particular fraction of its duty cycle. Generally, the only disadvantage of these low-power modes is reduced readability. However, in the case of switches among update frequencies and switches between color and monochrome, the transitions can also cause annoying flashes.

II.6.2 Transition strategies

Essentially the only transition strategy currently used to take advantage of these low-power features is to turn off or down the backlight and display after a certain period of time has passed with no user input. The reasoning behind this strategy is that if the user has not performed any input recently, then it is likely he or she is no longer looking at the screen, and thus the reduced readability of a low-power mode is acceptable for the immediate future. To lessen the effect of a wrong guess about such inactivity on the part of the user, some machines do not shut the backlight off immediately but rather make it progressively dimmer. In this way, if the user is actually still looking at the screen, he or she gets a chance to indicate his or her presence before the entire screen becomes unreadable. One study found that thanks to the use of low-power states, the backlights on some machines consumed only 32–67% of maximum possible energy while running on battery power [Lor95a].

A possible modification of this standard strategy is to automatically readjust the inactivity threshold to make it a better predictor of user inactivity. For example, if the user hits a key just as the backlight begins to dim, such a strategy might increase the inactivity threshold on the assumption that its current value is too short.

Another possible transition strategy is to switch the display to monochrome when color is not being displayed, or to switch it to a lower update frequency when the items displayed do not require a high update frequency. The operating system might even switch to a lower-power display mode when those parts of the screen making use of the current display mode are not visually important to the user. For example, if the only color used on the screen were in a window belonging to an application not recently used, the operating system might switch the display to monochrome. The use of such features would be most acceptable if such transitions could be made unobtrusive, e.g., without a flash, and perhaps even with progressive fading.

Other transition strategies become feasible when additional hardware is present on the machine. For example, if a device can detect when the user is looking at the screen, the system can turn off the display and backlight at all other times. Such a device might consist of a light emitter and receiver on the machine and a reflector on the forehead of the (unusually docile) user. Or, it might be similar to those used by some video cameras that focus by watching where the user is looking. If a sensing device can determine the ambient light level, the system can dim the backlight when ambient light is sufficiently bright to see by [Soh95].

II.6.3 Load-change strategies

Currently, there are no formal load-change strategies for reducing the energy consumption of the display or backlight. However, it has been suggested that using a light virtual desktop pattern rather than a dark one can reduce the load on the backlight. This happens because lighter colors make the screen seem brighter and thus encourage users to use lower default backlight levels [Kut95]. Furthermore, since most LCD's are "normally white," i.e., their pixels are white when unselected and dark when selected [Wer94], the display of light colors consumes marginally less power than the display of dark colors. A similar strategy would be for the operating system or an application to decrease the resolution of a screen image by only illuminating a certain fraction of its pixels.

II.6.4 Future hardware innovations

Researchers working on technological advances in display and backlight hardware have many opportunities to make software power management of these components more effective. Switching to low-power modes could be made unobtrusive. If an ambient light sensor were available, the operating system could automatically reduce the backlight level when ambient light brightened. Finally, as for all other components mentioned in this chapter, designers of display and backlight hardware should seek to include as many low-power modes as possible that provide reduced but reasonable levels of functionality.

II.7 Other Components

II.7.1 Main memory

Main memory is generally implemented using DRAM with three modes: active, standby, and off. In active mode, the chip is reading or writing; in standby, it is neither reading nor writing but is maintaining data by periodically refreshing it. As an example of the power consumption in these states, 256 Mb of SDRAM memory from Micron consumes about 6.6 mW in standby mode, 198 mW in active mode with all banks idle, and 545 mW in active mode while being accessed [Mic01]. The only transition strategy currently used to reduce memory energy makes use of the off state: when it is determined that the entire system will be idle for a significant period of time, all of main memory is saved to disk and the memory system is turned off. The memory contents are restored when the system is no longer idle. When memory is saved in this manner, the machine state is said to be *suspended*; restoring memory is called *resuming*. Load-change strategies for saving memory power have been discussed before in the context of load-change strategies for saving processor power: they included using energy-aware compilers and using compact and efficient operating system code. Such strategies reduce the load on the memory system by making application and system code more compact and efficient, thus permitting greater use of the standby state. They may also convince the user to purchase a machine with less main memory, thus reducing the energy consumption of the memory system.

In future machines, memory may be divided into banks, with each bank able to turn on and off independently. Such capability broadens the ability of the operating system to manage memory energy. At times when the memory working set could fit in a small amount of memory, unused memory could be turned off. If the contents of a bank of memory were not expected to be used for a long time, they could even be saved to disk. Note that the expected period of idle time would have to be large to make up for the significant energy and time consumed in saving and restoring such memory. A related approach would be to compress, using standard data compression methods, the contents of memory, and turn off the unused banks; memory contents could be uncompressed either as needed or when activity resumed.

II.7.2 Modem

A modem can be transmitting, receiving, idle, or off. Some modems provide another state with power consumption between idle and off, called wake-on-ring; in this state, the modem consumes just enough power to detect an incoming call. MacOS uses no power saving strategies for the modem, relying on the user or an application to turn the modem on and off explicitly [Lor95a]. Presumably, this is because the operating system has no idea when data will arrive at the modem, and needs to make sure the modem is enabled whenever such data arrives so that it is not lost. A better strategy would be to have the operating system, or even the modem itself, switch the modem to the off or wake-on-ring state whenever there is no active connection to the modem.

II.7.3 Sound system

The sound system is another miscellaneous consumer of energy that can be active, idle, or off. MacOS uses an inactivity timer to decide when to turn the sound card off [Lor95a]. Another possibility is to turn the sound card off whenever a sound request from an application that triggered it turning on is completed; this has the disadvantage of increasing sound emission latency when one sound closely follows another.

II.8 Overall Strategies

It is possible to energy manage the entire computer as if it were a single component. When the computer is unneeded now and probably for some time, the operating system may put the entire system in a low-power state. Just how low-power a state depends on how long the system is expected to be idle since, in general, the lower the power of the state, the greater the time to return the system to full functionality.

Transition strategies for entering low-power system states generally use inactivity thresholds. If the user and all processes have been idle for some set period of time, the next lower system state is entered. APM 1.1 [IM93], a standard for energy management in computers, allows this decision-making process to be enhanced in at least two ways. First, the user may be allowed to make an explicit request to switch to a lower-power system state.

Second, certain applications may be consulted before making a transition to a lower-power system state, so they can reject the request or make internal preparations for entering such a state.

Several low-power system states can be devised, and some examples of these are defined in APM 1.1. In the APM standby state, most devices are in a low-power state but can return to their full-power states quickly. For example, the disk is spun down and the backlight is off. In the APM suspend state, all devices are in very low-power states and take a relatively long time to return to functionality. For instance, the contents of memory are saved to disk and main memory is turned off. In the APM off state, the entire machine is off; in particular, all memory contents are lost and a possibly long boot period is needed to return to functionality.

Note that the sequence with which low power states are entered is significant. For example, if the memory is copied to the disk before the disk is spun down, then the machine, or at least the memory, can be shut down without spinning up the disk to establish a checkpoint.

There are both a disadvantage and an advantage to energy managing the system as a whole instead of separately managing each component. The disadvantage is that it requires all components' energy management be synchronized. Thus, if one component is still active, some other inactive component may not get turned off. Also, if it takes microseconds to determine the idleness of one component but seconds to determine the idleness of another, the first component will not be energy-managed as efficiently as possible. The advantage of treating the system as a single component is simplicity. It is simpler for the operating system to make a single prediction about the viability of entering a system state than to make separate predictions for each component state. It is simpler for an application to give hints to the operating system about when state transitions are reasonable, and to accept and reject requests by the operating system to make such transitions. Also, it is simpler for the user, if he or she is called upon to make energy management decisions, to understand and handle a few system state transitions than to understand and handle an array of individual component transitions. For these reasons, an operating system will typically do both component-level and system-level energy management. For example, APM 1.1 has a system state called enabled, in which individual component energy management is performed. After extended periods of idleness, when most components can be managed uniformly, different low-power system states can be entered.

II.9 Conclusions

Computer hardware components often have low-power modes. These hardware modes raise software issues of three types: transition, load-change, and adaptation. Several solutions to these issues have been implemented in real portable computers, others have been suggested by researchers, and many others have not yet been developed. Generally, each solution targets the energy consumption of one component.

The disk system has been the focus of many software solutions. Currently, the main hardware power-saving feature is the ability to turn the motor off by entering sleep mode. The main existing software solutions consist of entering sleep mode after a fixed period of inactivity and caching disk requests to reduce the frequency with which the disk must be spun up. Other technologies may improve the energy consumption of the storage system further, but present new challenges in file system management. These technologies include flash memory, which can function either as a secondary storage cache or a secondary storage unit, and wireless communication, which can make remote disks appear local to a portable computer.

New low-power modes for the CPU also present software challenges. Currently, the main hardware energy-saving feature is the ability to turn the CPU off, and the main existing software solution is to turn the CPU off when all processes are blocked. Other software strategies include using energy-aware compilers, using compact and efficient operating system code, and forcing processes to block when they appear to be busy-waiting. An energy-saving feature that is increasingly appearing in portable computers is the ability to reduce the processor voltage by simultaneously reducing the clock speed. Initial proposed solutions that take advantage of this feature were interval-based, attempting to complete all processor work by the end of each interval. However, there are effective deadline-based solutions as well.

The wireless communication device is appearing with increasing frequency in portable computers, and is thus an important recent focus of software energy management research. Commonly, the device is put in sleep mode when no data needs to be transmitted or received. To make this happen often, various techniques can be used to reduce the amount of time a device needs to be transmitting or receiving, including the adoption of protocols that let devices know in advance when they can be assured of no incoming data. Some new devices have the ability to dynamically change their transmission power; given this, a device needs a strategy to continually decide what power level is appropriate given quality of service requirements, interference level, and needs of neighboring units.

The display unit, including the backlight, typically consumes more power than any other component, so energy management is especially important for it. Power-saving modes available include dimming the backlight, turning the display and/or backlight off, switching from color to monochrome, and reducing the update frequency. Current system strategies only take advantage of the former two abilities, dimming the backlight and eventually turning off the display unit after a fixed period of inactivity. Other software strategies can be envisioned, especially if future hardware makes transitions to other low-power modes less obtrusive.

Other components for which software power management is possible include main memory, the modem, and the sound system. It is also possible to power manage the entire system as if it were a single component, bringing all components simultaneously to a low-power state when general inactivity is detected. Such system-level power management is simple to implement and allows simple communication with applications and users about power management; however, it should not completely supplant individual component power management because it requires synchronization of all components' power management.

To conclude, there are a few things we believe developers of future solutions to computer energy reduction should keep in mind.

1. A hardware feature is rarely a complete solution to an energy consumption problem, since software modification is generally needed to make best use of it.

2. Energy consumption can be reduced not only by reducing the power consumption of components, but also by introducing lower-power, lower-functionality modes for those components and permitting external control over transitions between those modes.

3. Standard operating system elements may need to be redesigned when dealing with low-

power components that have different performance characteristics than the components they replace.

4. On a portable computer, the main goal of a component energy management strategy is to increase the amount of work the entire system can perform on one battery charge; thus, evaluation of such a strategy requires knowledge of how much energy *each* component consumes.

5. Evaluation of a power management strategy should take into account not only how much energy it saves, but also whether the trade-off it makes between energy savings and performance is desirable for users.

6. Seemingly independent energy management strategies can interact.

Chapter III

Improving CPU Sleep Mode Use

III.1 Introduction

In earlier work not presented in this dissertation, we analyzed the power consumption of various Macintosh PowerBook computers in typical use by a number of engineering users [Lor95a]. We found that, depending on the machine and user, up to 18–34% of total power was attributable to components whose power consumption could be reduced by power management of the processor, i.e., the CPU and logic that could be turned off when the CPU was inactive. This high percentage, combined with our intuition that software power management could be significantly improved for the processor, led us to conclude that the most important target for further research in software power management was the processor.

Many modern microprocessors have low-power states, in which they consume little or no power. To take advantage of such low-power states, the operating system needs to direct the processor to turn off (or down) when it is predicted that the consequent savings in power will be worth the time and energy overhead of turning off and restarting. In this way, the goal of processor power management strategies is similar to that of hard disks [DKM94b, LKHA94]. Some strategies for making these predictions are described by Srivastava et al. [SCB96]. Unlike disks, however, the delay and energy cost for a modern microprocessor to enter and return from a low-power mode are typically low. For instance, Intel's Mobile Pentium III requires only ten system clocks to return from the Auto Halt state to the Normal state [Int01]. Even back in 1996, the AT&T Hobbit and certain versions of the MC68030 and MC68040 used static logic so that most of their state could be retained when the clock is shut down [SCB96]; also, the PowerPC 603 could exit the low-power Doze mode in about ten system clocks [GDE+94].

Because of the short delay and low energy cost for entering and leaving a low-power state, the optimal CPU power management strategy is trivial: turn off the CPU whenever there is no useful work to do. An opportunity for such a strategy is described by Srivastava et al. [SCB96], who point out that the process scheduling of modern window-based operating systems is event-driven, i.e., that the responsibility of processes in such systems is to process events such as mouse clicks when they occur and then to block until another such event is ready. In this type of environment, the most appropriate strategy is to shut off the processor when all processes are blocked, and to turn the processor back on when an external event occurs. An essentially equivalent version of this strategy, namely to establish a virtual lowest-priority process whose job is to turn off the processor when it runs, is recommended by Suessmith and Paap [SI94] for the PowerPC 603, and by Suzuki and Uno [SU93] in a 1993 patent. Such a virtual lowest-priority process has in the past been called the "idle loop," and in mainframes typically lighted a bulb on the console.

III.1.1 Why it isn't trivial

We refer to the strategy of turning off the processor when no process is available to run the *basic strategy*. Unfortunately, in Apple's MacOS 7.5, processes can run or be scheduled to run even when they have no useful work to do. This feature is partially by design, since in a single-user system there is less need for the operating system to act as an arbiter of resource use [RDH+80]. Partially, it is because the OS was not written with power management in mind. Partially, it is because MacOS 7.5, like other personal computer operating systems (e.g., those from Microsoft), is based on code originally developed for 8- and 16-bit non-portable machines, for which development time and code compactness were far more important goals than clean design or faithfulness to OS design principles as described in textbooks.

There are two main problems with the management of processor time by computers running MacOS 7.5, one having to do with the system and one having to do with applications. The first problem is that the operating system will sometimes schedule a process even though it has no work to do. We were first made aware of this phenomenon when we studied traces of MacOS process scheduling calls, and found that often a process would be scheduled to run before the conditions the process had established as necessary for it to be ready were fulfilled. It seems that often, when there are no ready processes, the OS picks one to run anyway, usually the process associated with the active window. The second problem is that programmers writing applications generally assume that when their application is running in the foreground, it is justified in taking as much processing time as it wants. First, a process will often request processor time even when it has nothing to do. We discovered this problem in MacOS 7.5 when we discovered periods of as long as ten minutes during which a process never did anything, yet never blocked; we describe later what we mean by "never did anything." Second, when a process decides to block, it often requests a shorter sleep period than necessary. Solutions to both these problems seem to be necessary to obtain the most savings from the basic strategy.

For this reason, we have developed additional techniques for process management. Our technique for dealing with the first problem is to simply make the operating system never schedule a process when it has requested to be blocked; we call this the *simple scheduling technique*. Dealing with the second problem is more difficult, since the determination of when a process is actually doing something useful is difficult. One technique we suggest is to use a heuristic to decide when a process is making unnecessary requests for processor time and to forcibly block any such process. Another technique we suggest is that all sleep times requested by processes be multiplied by a constant factor, chosen by the user or operating system, to ensure that a reasonable trade-off between energy savings and performance is obtained. We call these latter two techniques the *greediness technique* and *sleep extension technique*, respectively. We will show how using these techniques can improve the effectiveness of the basic strategy, allowing it to far surpass the effectiveness of the MacOS 7.5 inactivity-timer based strategy. Each of these is described in more detail below.

In this chapter, we evaluate these different strategies, over a variety of parameter values, using trace-driven simulation. These simulations enable us to compare these algorithms to the MacOS 7.5 strategy, and to optimize their parameters. A comparison between two strategies is based on two consequences of each strategy: how much processor power it saves and how much it decreases observed performance.

The chapter is structured as follows. In Section III.2, we give background on processor power consumption and operating systems, including a description of the processor power management strategies used in MacOS 7.5 and other contemporary operating systems. In Section III.3, we describe our suggested process management techniques for improving the basic strategy: (a) do not schedule blocked processes, (b) extend the time processes spend sleeping, and (c) detect processes that appear to be busy-waiting and forcibly block them. In Section III.4, we describe the methodology we used to evaluate these strategies: the evaluation criteria, the tools we used for the trace-driven simulation, and the nature of the traces we collected. In Section III.5, we present the results of our simulations, showing that our techniques save 47–66% of processor energy. In Section III.6, we discuss the meaning and consequences of these results. Finally, Section III.7 concludes the chapter.

III.2 Background

III.2.1 Power consumption breakdown of portables

When discussing the effectiveness of techniques for reducing the power consumption of a component, the most important number to know is the percentage of total power consumption attributable to that component. Furthermore, it is not good to know this percentage when all components are in their high-power state, since this is not a realistic description of the typical state of a laptop. We must determine the percent of total power consumption attributable to that component when power-saving features are in use for all other components; these power-saving features generally increase the percentage by reflecting reduced use of other components.

We must also be careful about percentage due to a component when the power state of that component influences the power consumption of other components. For instance, in a previous study we found that shutting down the processor in Apple laptops reduces power by more than the power consumption of the processor itself, indicating that some components on the motherboard have their power reduced when the processor power is reduced. So, when we speak of percentage of total energy consumption due to the processor, we include energy consumption of other components that can be reduced by reducing the time the processor is on.

A previous study [Lor95a] showed that, for the workloads studied, the further power consumption that could be saved by making more use of the low-power state of the processor on Apple laptops ranged from 18–34% for the machines studied. Note that these figures take into account power savings attained in real working environments for the processor and all other components. Li et al. estimate the percent of power consumption attributable to the processor of a "typical" computer to be 12% [LKHA94].

III.2.2 The Macintosh operating system

The Macintosh operating system uses cooperative multitasking to allocate processing time among all running applications and device drivers. This means that for multiple applications to run concurrently, each application must, periodically, voluntarily give up control of the system so that another application can run. If an application decides to give up control while it still has work to do, it can request that control be returned to it after a certain period of time has passed. Otherwise, control will be returned to it whenever an event such as a mouse click is available for it to process.

However, there are two problems with this rosy picture of the operating system. First, the voluntary nature of this protocol invites abuse on the part of applications, and many applications accept this invitation by requesting control returned to them sooner than they need that control. Second, for various low-level reasons, the operating system sometimes violates the application-level interface by returning control to applications much sooner than they need control returned to them. Years of this practice may have caused applications to err in the opposite direction, asking for too much time before control is returned to them, in the knowledge that the operating system will probably return control earlier.

Because of these problems, the amount of processing time requested by applications does not necessarily reflect their actual processing needs. Therefore, the MacOS 7.5 operating system does not make use of such information provided by applications in deciding when it is appropriate to turn off the processor. Instead, it uses an inactivity timer. The strategy it uses, which we will call the *current strategy*, initiates processor power reduction whenever no activity has occurred in the last two seconds and no I/O activity has occurred in the last 15 seconds. Power reduction is halted whenever activity is once again detected. *Activity* is defined here and in later contexts as any user input, any I/O device read or write, any change in the appearance of the cursor, or any time spent with the cursor as a watch. The reason for the classification of these latter two as activity is that MacOS human interface guidelines specify that a process that is actively computing must indicate this to the user by having the cursor appear as a watch or by frequently changing the appearance of the cursor, e.g., by making a "color wheel" spin.

Another reason the MacOS 7.5 operating system uses user inactivity is that when it was designed, Macintosh computers had high overhead associated with turning off and on the processor, making the basic strategy less applicable. In older processors, for example, the contents of on-chip caches were lost when the processor was powered down. Another reason is that, as we have described before and will see later, the effectiveness of the basic strategy is not very different from that of the inactivity timer based strategy, given the MacOS 7.5 method of process time management.

We feel that, despite the limitations of the information about application processing time needs, this information is far more valuable in determining when to turn the processor off than user inactivity levels. Thus, we suggest a heuristic for converting the imperfect information about processing needs into more useful information, and making use of this instead to decide when to turn off the processor.

III.2.3 Other operating systems

The Windows 3.1 operating system also uses cooperative multitasking, and its mechanism for permitting this is similar to that of the Macintosh operating system. However, some subtle differences allow the system greater confidence in assessing the true processing needs of the applications. First, the standard interface for giving up control does not allow the application to specify a time after which control should be returned. The default behavior is that control will not be returned until an event is ready; if control is needed earlier than that, it is up to the application to explicitly arrange for a timer event to occur when it is needed. This extra programming effort seems to be sufficient for ensuring that applications seldom have control returned to them before it is needed. Second, the fact that control is returned without an event occurring only when timers go off eliminates the problem of the operating system returning control too soon.

Thus, the processor power saving technique used in Windows 3.1 is to have the system idle loop, that part of the system that gets executed when all applications have given up control and there are no events waiting for them, send a message to the power management system indicating that the CPU is idle. In this way, the power management system can immediately turn off the CPU or at least reduce its power consumption by reducing its clock rate. We call this strategy, which turns off the CPU whenever all processes are blocked, the *basic strategy*.

We feel that the fine-grain control over processor-off times this affords yields great power savings compared to the long delays associated with waiting for the sluggish user to stop being active.

III.3 Strategy Improvement Techniques

In this section, we describe three techniques for improving processor energy management in MacOS 7.5 when the basic strategy is used. These techniques should allow the basic strategy to equal and even surpass the current strategy in terms of power savings by changing operating system behavior.

III.3.1 The simple scheduling technique

The simple scheduling technique is to not schedule a process until the condition under which it has indicated it will be ready to run has been met. In MacOS 7.5, this condition is always explicitly indicated by the process, and is always of the form, "any of the event types $e_1, e_2, \ldots$ has occurred, or a period of time t has passed since the process last yielded control of the processor." (In modern Windows operating systems, this is indicated by a call to the common API function `MsgWaitForMultipleObjects`.) The period of time for which the process is willing to wait in the absence of events before being scheduled is referred to as the *sleep period*.

Note that, in some other operating systems, such as UNIX or Microsoft Windows, the simple scheduling technique is not needed, since it is the default behavior of the operating system.

III.3.2 The sleep extension technique

Using only the simple scheduling technique described above means that a process is given control of the processor whenever it wants it (unless the CPU is otherwise busy). For example, if it asks to be unblocked every 1 second, it is unblocked every 1 second, even if all it wants to do is blink the cursor, a common occurrence. Since MacOS 7.5 is not a real time system, a real time sleep period does not actually have to be honored. In fact, in the MacOS 7.5 power management strategy, with power management enabled, the cursor may blink much more slowly than it would without power management. If this kind of behavior is acceptable, it is possible to increase the effectiveness of the simple scheduling technique by using what we call the *sleep extension technique*. This technique specifies a *sleep multiplier*, a number greater than one by which all sleep periods are multiplied, thus eliminating some fraction of the process run intervals. We envision that the sleep multiplier can be set, either by the user or by the operating system, so as to maximize energy savings, given a certain level of performance desired. We note that sleep extension may negatively impact performance, or even functionality, since not all delays will be as inconsequential as a cursor which blinks less frequently.

III.3.3 The greediness technique

The *greediness technique* is, in overview, to identify and block processes that are not doing useful work. First, we will describe the technique in general terms, and then we will indicate the details of its implementation for MacOS 7.5.

The technique is based on the following model of the appropriate way a process should operate in an event-driven environment. A process, upon receiving an event, should process that event, blocking when and only when it has finished that processing. Once blocked, it should be scheduled again when and only when another event is ready to be processed; an exception is that the process may want to be scheduled periodically to perform periodic tasks, such as blinking the cursor or checking whether it is time to do backups. We say that a process is acting greedily when it fails to block even after it has finished processing an event. This can occur when a process busy-waits in some manner, e.g., it loops on "check for event." When we determine a process is acting greedily, we will forcibly block that process for a set period of time.

MacOS 7.5 uses cooperative multitasking, meaning that once a process gets control of the processor, it retains that control until it chooses to yield control. For this reason, application writers are strongly encouraged to have their processes yield control periodically, even when they still have work to do. Processes indicate that they still have work to do by specifying a sleep period of zero, thereby failing to block. We call the period of time between when a process gains control of the processor and when it yields control a quantum.

Part of our technique is a heuristic to determine when a process is acting greedily. We say that a process is acting greedily when it specifies a sleep period of zero even though it seems not to be actively computing. We consider a process to start actively computing when it receives an event or shows some signs of "activity," as defined below. We estimate that a process is no longer actively computing if it explicitly blocks, or if it yields control several times in a row without receiving an event or showing signs of activity. The exact number of control-yield times, which we call the *greediness threshold*, is a parameter of the technique; we expect it to be set so as to maximize energy savings, given a desired level of performance. We say that a process shows no sign of activity if it performs no I/O device read or write, does not have the sound chip on, does not change the appearance of the cursor, and does not have the cursor appear as a watch. The absence of activity as we have so defined it implies that either the CPU is idle, the process running is busy-waiting in some manner, or the process running is violating the MacOS human interface guidelines that we mentioned earlier.

The greediness technique works as follows. When the OS determines that a process is acting greedily as defined above, it blocks it for a fixed period called the *forced sleep period.* The forced sleep period is a parameter to be optimized, with the following tradeoff: a short sleep period saves insufficient power, while a long sleep period may, in the case that our heuristic fails, block a process that is actually doing something useful.

III.4 Methodology

III.4.1 Evaluation of strategies

Evaluation of a strategy requires measuring two consequences of that strategy: processor energy savings and performance impact. *Processor energy savings* is easy to deduce from a simulation, since it is essentially the percent decrease in the time the processor spends in the high-power state. In contrast, *performance impact*, by which we mean the percent increase in workload runtime as a result of using a power-saving strategy, is difficult to measure. This performance penalty stems from the fact that a power saving strategy will sometimes cause the processor not to run when it would otherwise be performing useful work. Such work will wind up having to be scheduled later, making the workload take longer to complete. Without detailed knowledge of the purpose of instruction sequences, it is difficult for a tracer to accurately determine what work is useful and what is not, so our measure will necessarily be inexact.

We have decided to use the same heuristic used in the greediness technique to determine when the processor is doing useful work. In other words, we will call a quantum *useful* if, during that quantum, there is any I/O device read or write, the sound chip is on, there is any change to the cursor, or the cursor appears as a watch. It might be objected that using the same heuristic in the evaluation of a strategy as is used by that strategy is invalid. However, remember that a strategy does not have prior knowledge of when a quantum will be useful, whereas the evaluation system does. Thus, we are evaluating the accuracy of our guess that a quantum will be useful or useless.

We must also account for the time not spent inside application code in the original trace. We divide this time into time spent switching processes in and out (context switching), time the OS spent doing useful work, and OS idle time. The OS is considered to be doing useful work whenever it shows signs of activity that would cause a process quantum to be labeled useful. Such useful work is scheduled in the simulations immediately after the quantum that it originally followed is scheduled, on the assumption that most significant OS work is necessitated by the actions of the process that just ran. Idle time is identified whenever no process is running and the operating system is not doing useful work for a continuous period over 16 ms. We chose this threshold for two reasons. First, it is the smallest time unit used for process scheduling by MacOS 7.5, so we expect any decision to idle to result in at least this much idleness. Second, 16 ms is much greater than the modal (most common) value of interprocess time, indicating that it is far longer than should ever be needed to merely switch between processes. Finally, context switch time is assumed to occur any time a process switches in but did not just switch out; context switches are considered to take 0.681 ms, the observed difference between the average interprocess time when no context switch occurs and the average interprocess time when one does occur.

In the simulations, all time spent processor cycling is multiplied by 0.976, to model the fact that about 2.4% of total time is spent servicing VBL interrupts, during which time no processor cycling is possible.

III.4.2 Tools

We used three tools to measure the effectiveness of our processor energy management strategies. The first, IdleTracer, collects traces of relevant events from real environments. The second, ItmSim, uses these traces to drive simulations of the current strategy, i.e., the strategy of using an inactivity threshold, to determine how much time would have been spent processor cycling if that method had been used. The third, AsmSim, works analogously for the basic strategy, allowing us to simulate the basic strategy by itself as well as augmented with the techniques we recommend.

III.4.2.1 IdleTracer

IdleTracer collects traces of events related to processor and user activity in real environments. Thus, each log file created by IdleTracer contains a list of everything relevant that happened during the period measured, along with when those things happened. IdleTracer only collects data while the machine it is tracing is running on battery power, since that is when processor cycling is most important, and we want our analysis to reflect the appropriate workload. While it is tracing, IdleTracer turns processor power management off, so that the trace obtained is independent of the processor cycling technique currently used and thus can be used to simulate any strategy. IdleTracer makes use of the *SETC* [Soh94] module, a set of routines for tracing and counting system events. The types of events IdleTracer collects are as follows:

- Tracing begins or ends
- Machine goes to or wakes from sleep
- Application begins or ends
- Sound chip is turned on or off
- Cursor changes, either to a watch cursor or to a non-watch cursor
- An event is posted
- Mouse starts or stops moving
- Disk is read or written, either at a file level or a driver level
- Control is returned to an application
- Control is given up by an application

III.4.3 ItmSim

The types of events recorded by IdleTracer include those events used by the current inactivity threshold method for processor cycling to determine when activity precluding processor cycling is occurring. Thus, it is straightforward to use IdleTracer traces to drive simulations of that technique of processor cycling. The program that does this is ItmSim, so named because it simulates the **i**nactivity **t**hreshold **m**ethod.

The simulator works as follows. It determines, from the traces, when the processor would turn off due to the inactivity threshold. When, later in the simulation, the processor comes back on due to some activity, any quanta in the original trace that preceded that activity but have not yet been scheduled are divided into two categories: useful and non-useful. Useful quanta are immediately scheduled, delaying the rest of the trace execution and thus contributing to the performance impact measure. Non-useful quanta are discarded and never scheduled. Any useful OS time associated with these quanta is also immediately scheduled, contributing to the performance impact measure.

One wrinkle in simulating the current strategy is that during periods that the processor is supposed to be off, MacOS 7.5 will occasionally turn the processor on for long enough to schedule a process quantum. This is done to give processes a chance to demonstrate some activity and put an end to processor power management, in case the processor was shut off too soon. The details of how process quanta are scheduled while the processor is supposed to be off is proprietary and thus is not described here; however, *ItmSim* does attempt to simulate this aspect of the strategy. To give an idea of the consequences of this proprietary modification, our simulations showed that for the aggregate workload we studied, it decreased the performance impact measure from 1.93% to 1.84%, at the expense of decreasing processor off time from 29.77% to 28.79%. This particular proprietary modification, therefore, has only a trivial effect on the power savings.

ItmSim has several parameters that the user can modify to alter its behavior and model slightly different situations or techniques. One parameter allows the typical inactivity threshold to be modified from its default of two seconds. Another allows the I/O inactivity threshold to be modified from its default of 15 seconds. Finally, the factor of 0.976 that takes into account time spent processing VBL tasks can be changed with a parameter.

III.4.4 AsmSim

The third tool, AsmSim, uses traces produced by IdleTracer to drive a simulation of using the simple scheduling technique on a hypothetical machine. It can simulate using zero or more of our two other suggested techniques: sleep extension and greediness. The parameters for these techniques may be varied at will in the simulations. When, in the simulation, an event becomes ready for a process, all quanta of that process preceding the receipt of the ready event that have not yet been scheduled will be treated as described above, i.e., all useful quanta will be run immediately (before the power-up event), all useless quanta will be discarded, and any useful OS time associated with such quanta will also be run immediately. Even for periodic processes, we schedule quanta in the order in which they occurred. For example, if after its quantum i a process originally slept for 1 second but is actually awoken after 4 seconds, then at that point we schedule quantum $i + 1$, not some later quantum. Note that this approach may cause inaccuracies in the simulation, since

User number	1	2	3	4	5	6
Machine	Duo 280c	Duo 230	Duo 280c	Duo 280c	Duo 280c	Duo 280c
MacOS version	7.5	7.5.1	7.5	7.5	7.5.1	7.5
RAM size	12 MB	12 MB	12 MB	12 MB	12 MB	12 MB
Hard disk size	320 MB	160 MB	320 MB	320 MB	320 MB	240 MB
Trace length (hr:min:sec)	2:48:34	3:01:21	9:09:00	5:26:41	4:52:55	4:14:52

Table III.1: Information about the six users traced

the process might in reality check how long it has been since it last went to sleep, and act differently seeing that 4 seconds have passed than it did when only 1 second had passed. We expect and hope that such dependence of process action on time is rare enough that this does not introduce significant errors into the results of our simulations.

After simulating events according to these conditions, AsmSim outputs how long the trace would have taken, and how much of that time would have been spent processor cycling, were the simulated strategy used. Note that it may take longer to execute the entire trace under this technique than it originally took; AsmSim reports the percent increase in total time compared to the length of the original trace.

III.4.5 Traces

We collected traces from six users, each an engineer at Apple Computer, Inc. (We distributed IdleTracer to many users, but only six of them actually used it and returned traces.) Table III.1 indicates data about the traces obtained from each user and the machines on which those traces were collected. Most results we present will concern the aggregate workload, i.e., the trace composed of the concatenation of all six of these traces.

III.5 Results

In this section, we refer to the *C*urrent MacOS 7.5 strategy as strategy C and the *B*asic strategy as strategy B. We append the letter I to indicate use of the s*I*mple schedule technique, append the letter G to indicate use of the *G*reediness technique, and append

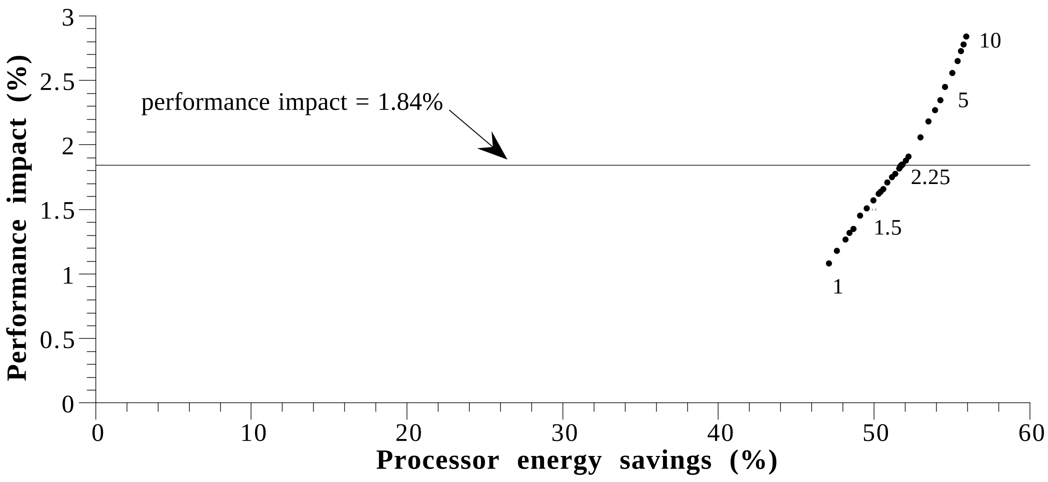

Figure III.1: Performance impact measure versus processor energy savings for strategy BIS with various sleep multipliers. Certain points are labeled with the sleep multipliers to which they correspond.

the letter S for the *S*leep extension technique. Note that we never simulate the greediness technique or sleep extension technique without the simple scheduling technique, since they are designed as supplements to the simple scheduling technique.

III.5.1 Per-strategy results

The first thing we shall do is determine the *optimal* energy savings attainable. An optimum strategy would schedule only time that was spent doing useful work, and would entirely omit non-useful time; its performance impact would be zero, since it would have foreknowledge of when useful work would occur and arrange to have the processor on when it happens. Simulation indicates that such a strategy would yield an energy savings of 82.33%; thus, this is an absolute ceiling on what can be obtained by any realizable strategy. This is a remarkably high figure—what it says is that the processor is doing useful computation during only 17.67% of the 29.56 hours of the trace; the rest of the time is busy-waiting by a user process or idling.

The second simulation results concern strategy C. We find from simulation that strategy C yields an energy savings of 28.79% along with a performance impact measure of 1.84%. In other words, it causes the processor to consume only 71.21% of the energy it would without a power-saving strategy, but increases overall workload completion time by 1.84%. The strategy increases processor energy consumption by 303% compared with the optimal strategy, since it only recovers 35% of the real idle time. Note also that since only 17.67% of the CPU time is actually useful, the performance impact of 1.84% means that we have misclassified 10% of the useful CPU time, and have had to run that work in a delayed manner. Thus, the actual real time delay perceived by the user may not be 1.84%, but may be closer to 10%, since the user waits for a reply only during periods of real, useful, work.

The next simulation results concern strategy B, which turns off the processor when and only when there was idling in the original trace. Strategy B has an energy savings of 31.98% and a performance impact of 0%. Thus, we see that the basic strategy without any new process management techniques saves slightly more energy than the current strategy, and has no impact on performance. However, it causes the processor to consume 285% more energy than under the optimal strategy, since it only recovers 39% of real idle time.

The next simulation results concern strategy BI. Strategy BI has an energy savings of 47.10% and a performance impact of 1.08%. Thus, we see that strategy BI decreases processor energy consumption by 26% and decreases workload completion time by 0.7% compared to strategy C. Compared to the optimal strategy, it causes the processor to consume 199% more energy, since it only recovers 57% of real idle time.

The next simulation results concern strategy BIS. Figure III.1 shows the performance versus energy savings graph for variations of this strategy using sleep multipliers between 1 and 10. We see that the point at which this strategy has performance impact of 1.84%, equal to that of strategy C, corresponds to a sleep multiplier of 2.25 and a processor energy savings of 51.72%. Thus, we see that, comparing strategies BIS and C on equal performance grounds, strategy BIS decreases processor energy consumption by 32%. Increasing the sleep multiplier to 10 saves 55.93% of the CPU energy, with a performance impact of 2.84%. Note, however, that the performance impact measure does not tell the whole story in this case. Generally, a real time delay is used by some process that wakes up, checks something, and if certain conditions are met, does something. A very large real time delay in the wakeup period may mean that certain checks are not made in a timely manner; we have ignored that issue here. In practice, sleep extension factors over some level, perhaps 3

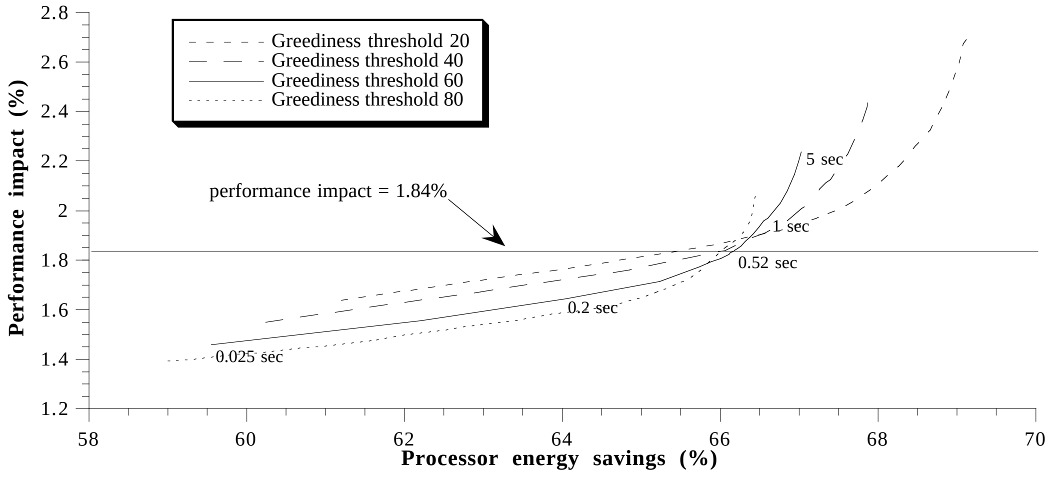

Figure III.2: Performance impact measure versus processor energy savings for strategy BIG with various greediness thresholds and forced sleep periods. Points on the greediness threshold 60 curve are labeled with the forced sleep periods to which they correspond. The reader is cautioned that nonzero origins are used in this figure to save space and yet have sufficient resolution to enable its key features to be discerned.

to 5, may not be desirable.

The next simulation results concern strategy BIG. Figure III.2 shows the performance versus energy savings graph for variations of this strategy using greediness thresholds between 20 and 80 and forced sleep periods between 0.025 seconds and 10 seconds. We find, through extensive exploration of the parameter space, that the parameter settings giving the best energy savings at the 1.84% performance impact level are a greediness threshold of 61 and a forced sleep period of 0.52 seconds. These parameters yield an energy savings of 66.18%. Thus, we see that, comparing strategies BIG and C on equal performance grounds, strategy BIG reduces processor energy consumption by 53%. Compared to the optimal strategy, it increases processor energy consumption by 91%, since it only saves 80% of real idle time.

The next results we present concern strategy BIGS. Figure III.3 shows that, in the realm we are interested in, a performance impact of 1.84%, increasing the sleep multiplier always produces worse results than changing the greediness threshold and forced sleep period. The energy savings attainable by increasing the sleep multiplier can be attained at a lower

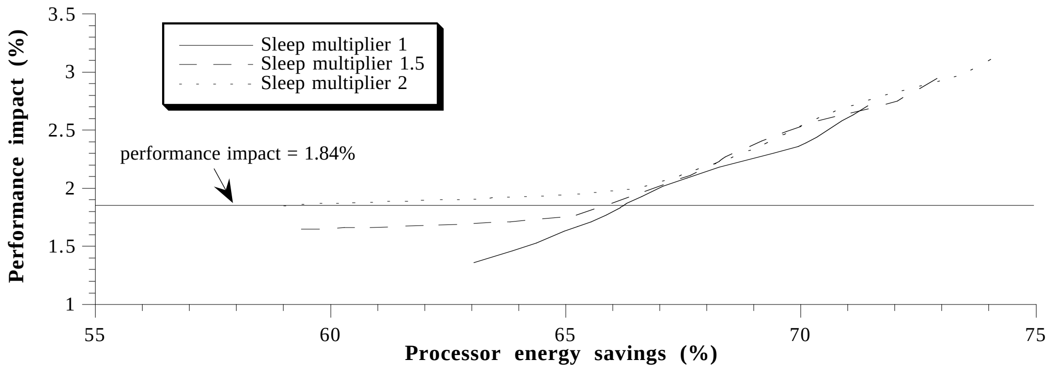

Figure III.3: Performance impact measure versus processor energy savings for strategy BIGS with various sleep multipliers, various greediness thresholds, and a forced sleep period of 0.52 sec. The reader is cautioned that nonzero origins are used in this figure to save space and yet have sufficient resolution to enable its key features to be discerned.

performance cost by instead decreasing the greediness threshold or by increasing the forced sleep period. Thus, the best BIGS strategy is the BIG strategy, which does not make any use of the sleep extension technique. The figure suggests that if we could tolerate a greater performance impact, such as 2.7%, this would no longer be the case, and the best energy savings for BIGS would be attained at a sleep multiplier above one. We conclude that for some values of performance impact, it is useful to combine the greediness technique and sleep extension technique, but for a performance impact of 1.84% it is useless to use the sleep extension technique if the greediness technique is in use.

A summary of all the findings about the above strategies can be seen in Table III.2, as well as the columns of Figure III.5 corresponding to users 1–6.

III.5.2 Sensitivity to parameter values

An important issue is the extent to which the parameters we chose are specific to the workload studied, and whether they would be optimal or equally effective for some other workload. Furthermore, it is unclear how effective the user or operating system could be at dynamically tuning these parameters in the best way to achieve optimal energy savings at a given level of performance. Thus, it is important to observe the sensitivity of the results we obtained to the particular values of the parameters we chose.

Strategy	Processor power savings	Performance impact
Optimal	82.33%	0.00%
C	28.79%	1.84%
B	31.89%	0.00%
BI	47.10%	1.08%
BIS	51.72%	1.84%
BIG	66.18%	1.84%

Table III.2: Simulation results for each strategy on the aggregate workload. Strategy BIS achieves the same performance impact as strategy C by using a sleep multiplier of 2.25; strategy BIG achieves this performance impact by using a greediness threshold of 61 and a forced sleep period of 0.52 sec.

The graphs we showed that demonstrate the relationship between performance, energy savings, and parameter values also demonstrate the reasonably low sensitivity of the results to the parameter values. For instance, varying the forced sleep period threshold in Figure III.2 across a wide range of values only causes the consequent energy savings to vary between 59–67%. Varying the greediness threshold in Figure III.4 across another wide range of values only causes the consequent energy savings to vary in the range 63–71%. Finally, varying the sleep multiplier across a wide range, as in Figure III.1, only causes the consequent energy savings to vary in the range 47–56%.

Another way to gauge the sensitivity of the results to the parameters is to evaluate the effectiveness of the techniques on each of the six workloads corresponding to the users studied. To show the effect of using parameters tuned to an aggregate workload on individual users, Figure III.5 shows the processor energy savings that would have been attained by each of the users given the strategies we have discussed. We see from this figure that strategy BIG is always superior to strategy C, and that strategy BIS is superior to strategy C for all users except user 2. And, even in this case, the fault seems to lie with the basic strategy and simple scheduling technique rather than the sleep multiplier parameter, since user 2 is also the only user for which the savings from C are much greater than those from strategies B and BI. These figures suggest that even parameters not tuned for a specific workload still yield strategies that in general save more processor energy than the current strategy. It is also interesting to note that there is a clear ordering between strategies BI, BIS, and BIG:

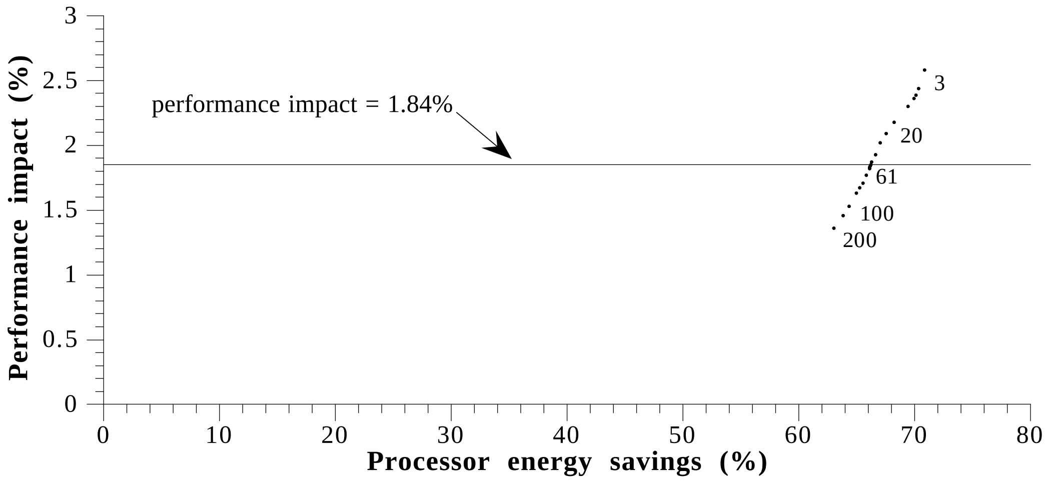

Figure III.4: Performance impact measure versus processor energy savings for strategy BIG with forced sleep period of 0.52 seconds and various greediness thresholds. Certain points are labeled with the greediness thresholds to which they correspond.

for each user, strategy BIG saved more energy than strategy BIS, which saved more energy than strategy BI.

We were curious why strategy C is so much superior to strategy B for user 2, so we inspected the simulation results for that user carefully. We found that the reason strategy C does so much better than strategy B is that in that trace, the application Finder (discussed further later) frequently yields control requesting a sleep time of zero but then performs no activity when it gets control again; indeed, there is a contiguous section of the trace lasting over an hour (a third of the trace) during which Finder has this behavior. When this happens, the basic strategy, strictly obeying the request of Finder to never block, never gets a chance to cycle the processor, while on the other hand the current strategy notices that no activity is occurring and turns off the processor anyway. The sleep extension technique does not alleviate this problem, since multiplying the sleep request of zero by any factor still makes it zero. However, the greediness technique is able to overcome this problem, since this is exactly the problem for which it was designed. Consequently, strategy BIG beats strategy C for user 2, even though strategies B, BI, and BIS do not.

Yet another way to see that the basic strategy with the new techniques is effective

Figure III.5: Processor energy savings for each strategy and each user. Strategy BIS uses a sleep multiplier of 2.25, while strategy BIG uses a greediness threshold of 61 and a forced sleep period of 0.52 seconds.

even without tuning the parameters is to pick somewhat arbitrary parameters and note that the energy savings are still superior to that of strategy C. For example, the parameter settings we envisioned before running any of these simulations, a greediness threshold of 5, a forced sleep period of 0.25 seconds, and a sleep multiplier of 1.5, would yield a respectable energy savings of 71.70% and a performance impact of 2.61%, which, compared to MacOS 7.5, trades off a 60% decrease in processor energy consumption for a 0.8% increase in workload completion time. Even a conservative set of parameters, namely a greediness threshold of 100, a forced sleep period of 0.10 seconds, and a sleep multiplier of 1, yields a processor energy savings of 62.87% with a performance impact of only 1.48%, decreasing processor energy consumption by 48% and reducing workload completion time by 0.4% compared to strategy C.

Strategy	Useful time (hours)			Useful quanta
	Process	Scheduler	OS	
C	**0.27**; 3.73	**0.23**; 0.63	**0.04**; 0.33	**88,055**; 473,075
B	**0.00**; 4.00	**0.00**; 0.86	**0.00**; 0.37	**0**; 561,130
BI	**0.20**; 3.80	**0.02**; 0.84	**0.10**; 0.27	**20,027**; 541,103
BIS	**0.37**; 3.63	**0.04**; 0.82	**0.12**; 0.25	**29,132**; 531,998
BIG	**0.33**; 3.67	**0.04**; 0.82	**0.18**; 0.19	**32,130**; 529,000

Strategy	Nonuseful time (hours)			Nonuseful quanta
	Process	Scheduler	OS	
C	2.38; *1.59*	8.31; *2.61*	5.14; *4.31*	6,303,597; *1,682,648*
B	3.96; *0.00*	10.92; *0.00*	0.00; *9.45*	7,986,245; *0*
BI	2.97; *1.00*	7.45; *3.47*	0.00; *9.45*	5,965,396; *2,020,849*
BIS	2.81; *1.15*	6.23; *4.69*	0.00; *9.45*	5,412,942; *2,573,303*
BIG	0.87; *3.09*	3.90; *7.02*	0.00; *9.45*	2,420,825; *5,565,420*

Table III.3: A breakdown, for each strategy, of what happens to the time and quanta originally spent running processes, the scheduler, and the operating system. Time and quanta that are delayed and thus contribute to performance impact are shown in **bold**, time and quanta that are run on time are shown in the standard font, and time and quanta that are never run and thus are subsumed by cycling are shown in *italics*.

III.5.3 Additional explanation of results

We have seen that the greediness technique by itself can be quite effective even within a broad range of parameter values. This suggests that processes often act greedily. In fact, using the ideal parameters of strategy BIG, 49 of 63 applications were found to act greedily at some point. The percent of an application's quanta during which it was determined to be acting greedily varied widely from one application to another, from 0.0009% for Express Modem to 93.7% for eWorld. It is especially amusing that Finder, the user desktop interface application written at Apple itself, seems to lie routinely about its processing time needs, having been determined to be acting greedily for 64.7% of its quanta. This supports our suggestion that designers of operating systems and applications for single-user systems are not generally concerned with rigorous management of processor time.

To examine where the savings and performance impact of a strategy are coming from, it is useful to observe what happens to different classes of time in the original trace when that strategy is used. Some time in the original trace is spent running processes, some is spent running the scheduler, and the rest is spent running other OS code. Each of these classes of time can be broken down into useful and nonuseful time. Useful time will always be scheduled in the simulation, but some of it will be delayed if the technique simulated decides to cycle instead of running it. This delayed time is what contributes to performance impact. Some nonuseful time will be scheduled by the technique, while the rest gets skipped over and contributes to cycling time. This skipped time is what contributes to energy savings. Table III.3 shows this breakdown in time and number of quanta for each strategy.

First, let us compare the sources of the energy savings for each strategy; this involves looking at the italic figures in Table III.3. Strategy C spends 4.31 hours cycling instead of idling in the OS, 1.95 hours cycling instead of performing nonuseful process quanta, and 2.61 hours cycling instead of switching nonuseful process quanta in and out. It is interesting to note that more time is saved from not having to switch processes in and out than from actually not running them. This is probably because the technique attempts to cycle the processor when processes are not doing anything, and when a process is not doing anything it should be doing little besides getting switched in and out. Strategy B never cycles instead of running or scheduling process quanta, but it makes up for this by always cycling when the OS would otherwise be idle, 9.45 hours. Strategies BI and BIS cycle even more because they can cycle instead of running or scheduling process quanta. Indeed, even if we did not consider idle time spent cycling, strategies BI and BIS still save more nonuseful process and scheduler time than strategy C. Interestingly, the increased savings stem from reducing the amount of time spent *scheduling* nonuseful process quanta, not from reducing the time spent *running* nonuseful process quanta. In fact, each of strategies BI and BIS spends longer running nonuseful process quanta than strategy C. The reason these strategies spend less time scheduling nonuseful process quanta is that, as shown in the table, the strategies schedule fewer nonuseful process quanta, and fewer quanta to schedule means less time spent scheduling quanta. So we see that the savings from strategies BI and BIS stem not from giving processes less time when they are not doing useful work, but from switching them in and out less often when they are not doing useful work, thus saving time associated with process switching. On the other hand, strategy BIG is superior to strategy C in its conversion of all types of nonuseful time. So, the savings from strategy BIG stems both from giving processes control less often when they are not busy and from giving them less time to run

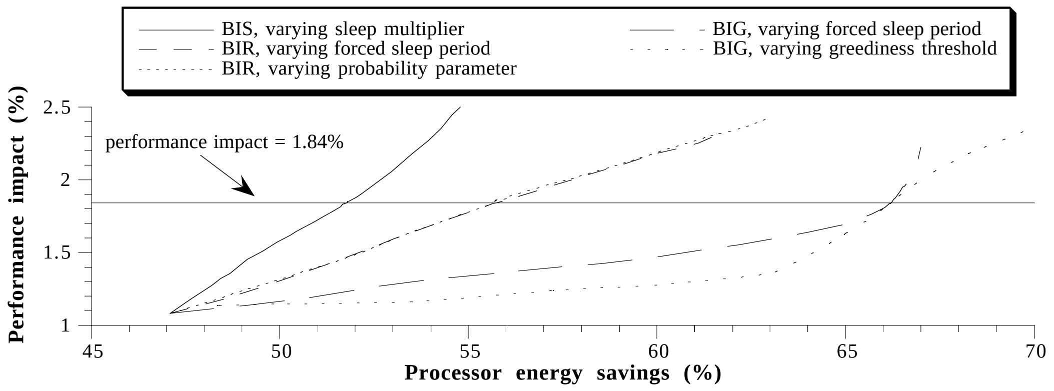

Figure III.6: Performance impact measure versus processor energy savings for various strategies with various parameter values varied. The reader is cautioned that nonzero origins are used in this figure to save space and yet have sufficient resolution to enable its key features to be discerned.

useless tasks.

Table III.3 also allows us to compare the sources of the performance impact in each strategy; this involves looking at the bold numbers. We see that for strategy C, about half of the performance impact is due to delaying the running of useful process quanta and half is due to delaying the switching of those quanta. On the other hand, strategy BI delays about the same amount of useful process run time, but far less time spent switching process quanta. This is explained by the fact that it delays a much lower absolute number of useful process quanta. The low amount of delayed scheduler time in strategy BI extends to BIG and BIS, allowing them to have a greater amount of delayed process quantum and OS time while still maintaining the same performance impact as strategy C. Our main observation, then, is that strategies BI, BIG, and BIS delay far fewer useful quanta than strategy C even though by design the latter two delay the same amount of total useful time.

III.5.4 Further analysis of the greediness technique

Any technique that forces a process to sleep when it would otherwise not do so will almost certainly decrease the energy consumed by the processor and increase the performance impact. It does the former because it causes some useless process quanta to be put off so long that they never get scheduled, and it does the latter because it causes some useful process quanta to be delayed. Thus, it is no surprise that the greediness technique was able to improve the energy savings of strategy BI, since it was allowed to increase the performance impact from 1.08% to 1.84%. However, the real test of such a strategy is how much energy savings it achieves per unit performance impact; a strategy is most effective if the heuristic it uses to determine when a process is forced to sleep is a good predictor of the future non-usefulness of the quanta of that process. So far, we have determined that the greediness technique is a good technique for decreasing the energy consumption of the processor, but we do not know if this is because it incorporates a good predictor of future process usefulness or because any such technique would have done well.

To evaluate the effectiveness of the predictor implicit in the greediness technique, we compare it to a similar technique that replaces this predictor with a random predictor. The *random greediness technique* has two parameters: a probability parameter $0 \leq p < 1$ and a forced sleep period f. Our random strategy is that each time a process requests a sleep time of zero, with probability p we force it to sleep for a period f instead. We use the letter R in a strategy name to denote the use of this technique.

We performed experiments with strategy BIR, attempting to maximize power savings while achieving a 1.84% performance impact by varying the probability parameter and the forced sleep period. We found that the best values of our experimental parameters (determined by running ad hoc experiments) were a forced sleep period of 0.3 seconds and a probability parameter of 0.023; with these values the energy savings is 55.6%. Since strategy BIG saves 66.2% of processor energy, strategy BIR causes the processor to consume 31% more energy than strategy BIG. It seems, then, that the indicator we use in the greediness technique is a better-than-random predictor of future process inactivity.

We can better gauge the effectiveness of strategy BIG by examining more than just its effect at a performance impact level of 1.84%, i.e., by looking at how its energy savings varies over a range of performance impact values. Figure III.6 shows how performance impact varies as a function of processor energy savings for strategies BIS, BIR, and BIG. The reciprocal of the slope of such a curve is a measure of how much energy is saved per unit performance impact at that point; we call this the *efficiency* of the strategy at that point. The figure shows that, at their most conservative parameter values, strategy BIG is more efficient than strategy BIR, which in turn is more efficient than strategy BIS. Thus, increasing sleep periods of zero is more efficient than multiplying sleep periods by a factor, and using a greediness heuristic to decide when to increase sleep periods of zero is more efficient than increasing them randomly. Another interesting point about the figure is that although strategies BIS and BIR retain essentially the same efficiency throughout the parameter ranges shown, strategy BIG shows greater efficiency at the lower performance impact levels than at the higher ones. An explanation for the decreasing efficiency of strategy BIG with decreasing greediness threshold is that the less evidence the strategy requires before it decides that a process is doing non-useful things, the less accurate its predictions about usefulness will be. On the other hand, strategies BIS and BIR, which do not attempt to predict anything, do not change in efficiency over different parameter values.

III.5.5 Significance of results

In an experimental study, different subjects are given different treatments and then response variables are measured. The experimenter then looks for effects, by which we mean differences in response variable values that are correlated with differences in treatments. These differences can have two causes: true effects of the treatments and random fluctuation in the values of the response variable. Thus, statistical tests are usually performed to determine whether the differences are *significant*, i.e., unlikely to have occurred as a result of mere random fluctuation, in which case a true effect can be said to exist. Usually, a test for significance is done at the $p < 0.05$ level, meaning that a difference is considered significant if it would have occurred with probability less than 5% in the absence of any true effect.

Our study is an experimental study, with users or workloads serving as subjects, energy-saving strategies serving as treatments, and energy savings and performance impact serving as response variables. We have found effects, differences in the energy savings arising from different strategies, but we have not yet commented about whether these differences are significant. For instance, we have shown that, for our subjects, strategy BIG saves more energy than strategy C, but we have not shown that this difference in energy savings is unlikely to have been due to chance factors. To remedy this, we performed a statistical analysis of the results.

We used the ANOVA technique described in [KZ89] for a within-subjects design, checking the results with the statistically equivalent paired observation technique described in [Jai91]. The comparisons we performed were those we considered the most meaningful comparisons between pairs of strategies, namely differences in energy savings between strategy C and each other strategy, between B and BI, between BI and BIS, between BI and BIG, and between BIS and BIG. We found significant differences between C and BIG, between B and BI, between BI and BIS, and between BI and BIG. In other words, we found that our best composite strategy, strategy BIG, was significantly better than the current MacOS 7.5 strategy; that each of the sleep extension technique and the greediness technique significantly improved the basic strategy with the simple scheduling technique; and that the simple scheduling technique significantly improved the basic strategy. Note that we did not find a significant difference between strategies BIG and BIS. We note again that our failure to find a significant difference does not mean that it does not exist, since it may merely reflect the small number of subjects.

III.6 Discussion

In our simulations, we found the following. The basic strategy by itself saves barely more energy than the current MacOS 7.5 strategy, although it does have less performance impact. Adding the simple scheduling technique has a significant effect on the energy savings of the basic strategy, allowing it to reduce energy consumption by 26% and to reduce workload completion time by 0.7% compared to the current MacOS 7.5 strategy. Further techniques for managing processor time also have significant effects on energy savings, allowing the basic strategy to even further surpass the current MacOS 7.5 strategy in energy savings for the same level of performance impact. Using the sleep extension technique allows 32% less processor energy consumption, and using the greediness technique allows 53% less processor energy consumption. In all cases, the absolute level of performance impact is very low. Note, however, that at best we get about 66% absolute energy savings, compared to the approximately 82% that would be available with optimal power reduction.

Our simulations suggest that although both the greediness technique and the sleep extension technique are helpful in increasing the savings attainable from the basic strategy with the simple scheduling technique, the greediness technique does this job better than the simple scheduling technique. Furthermore, when attempting to achieve the same performance impact the current MacOS 7.5 provides, it is not worthwhile to use the sleep extension technique if one is already using the greediness technique. The reason for this is that the greediness technique is more efficient than the sleep extension technique, making the cost in performance impact of an increase in energy savings greater for the sleep extension technique than for the greediness technique. This difference in efficiency seems to have two causes. The first cause, suggested by the fact that strategy BIR is more efficient than strategy BIS, is that increasing sleep periods of zero is a better approach than multiplying sleep periods by a constant factor. This is probably because a process is more likely to block less often than necessary than to block for shorter periods than necessary; our observations of the user 2 trace demonstrate how frequent the former occurrence is. The second cause, suggested by the fact that strategy BIG is more efficient than strategy BIR, is that the greediness technique uses predictions about the future usefulness of process time while strategy BIS does not incorporate any such prediction. Besides these differences between the two techniques, another reason to not use the sleep extension technique is that it is quite possible that extending real-time sleep times will produce undesirable effects in terms of system performance or functionality.

Analysis of the sensitivity of the techniques to changes in parameter settings and workloads suggests that ideal parameter setting is not necessary to the consistent functioning of the techniques. Thus, we expect the techniques to be successful with even naive parameter-setting by the operating system or user. However, one anomalous case, user 2, points out the vulnerability of the basic strategy to poor application behavior, suggesting that the greediness technique or something like it should always accompany the basic strategy.

To illustrate how percent time in low-power mode would translate into overall power savings, let us consider an example based on estimates of power consumption from previous work [Lor95a]. In this example, based on the Duo 280c, turning off the processor saves 3.74 W, while the components that remain on consume, on average, 5.65 W, given current power management techniques. From these figures, the current strategy, with processor energy savings of 28.79% and performance impact of 1.84%, would make average total power consumption

$$5.65 \text{ W} + \left(\frac{1 - 0.2879}{1 + 0.0184}\right)(3.74 \text{ W}) = 8.27 \text{ W}.$$

Note that we are assuming that during extra time spent with the processor on, other components are, on average, at their average power levels given current power management techniques. The BIG strategy, with processor energy savings of 66.18% and the same performance impact, would make average total power consumption

$$5.65 \text{ W} + \left(\frac{1 - 0.6618}{1 + 0.0184}\right)(3.74 \text{ W}) = 6.89 \text{ W}.$$

thus achieving a 16.7% savings in total power and yielding a 20.0% increase in battery lifetime relative to strategy C, assuming that battery lifetime is inversely proportional to average power drain. The BIS strategy, with its processor energy savings of 51.72% and same performance impact, would make average total power consumption 7.42 W, yielding a power savings of 10.3% and battery lifetime increase of 11.5% compared to strategy C. Note that all these figures are only applicable to the workload studied here, i.e., the 30 hours of traces obtained from these six users. It is unclear how much more or less effective these methods would be for some other workload; for instance, a previous study with a different workload [Lor95a] found that 48.1% of processor power was saved by the current MacOS 7.5 strategy. Even within the workload studied here, we have seen that the savings from each strategy varies greatly from one user to another.

We believe that even though our simulations were performed on MacOS 7.5 traces, our techniques are generally applicable to other single-user operating systems. For example, Microsoft Windows 3.1, Windows 95, Windows 98, Windows NT, and Windows 2000 essentially all use the basic strategy with the simple scheduling technique. However, they make no attempt to police processes that make unfair processor time requests; in fact, one recognized problem with its power management is that if a single process requests events using `PeekMessage` rather than `GetMessage` or `WaitMessage`, then processor power management cannot take place [Con92, Pie93]. This problem is exactly the sort that the greediness technique was designed to alleviate, so we may find that, as we demonstrated for MacOS 7.5, the effectiveness of the basic strategy is greatly improved by the use of such a technique.

III.7 Conclusions

Reducing the power consumption of computer systems is becoming an important factor in their design, especially in the case of portable computers. An important component for power management is the processor, as it accounts for a large percentage of total power. Obviously, the CPU can be turned off when it is inactive; in some operating systems, however, such as MacOS 7.5, the CPU is frequently running even when there isn't any "useful" work to do. We have suggested several heuristic techniques to decrease power use, such as (a) never running a process that is still blocked and waiting on an event, (b) delaying processes that execute without producing output or otherwise signaling useful activity, and (c) delaying the frequency of periodic processes, most of which seem to wake up, look around, find nothing interesting to do, and go back to sleep. To the extent that similar phenomena occur in other operating systems, these techniques should apply to them also.

We have used trace-driven simulation to evaluate these techniques on the basis of processor energy saved and increase in processing time. We found that the best techniques are those labeled (a) and (b) in the last paragraph, with (c) producing somewhat disappointing results. We found that our techniques save 47–66% of processor energy, thus decreasing processor energy consumption by 26–53% compared to the simple strategy employed by MacOS 7.5. Taking into consideration the fraction of system power used by the CPU, we estimate that implementing the best subset of our techniques will increase battery lifetime for Macintosh portables by around 20%. These savings do imply a small loss in effective system performance (due to the fact that the CPU may be inactive even though there is real work to do); in all cases, this loss is less than 2%.

Chapter IV

PACE: A Method for Improving DVS Algorithms

IV.1 Introduction

In Section II.4.1, we described the hardware features available in modern processors for energy management. A relatively recent energy-saving technology we described is dynamic voltage scaling (DVS), which allows software to dynamically alter the voltage of the processor, at the cost of reduced processor operating frequency when low voltages are used. Various chip makers, including Transmeta, AMD, and Intel, have recently announced and sold processors with this feature.

In Section II.4.4, we described current DVS algorithms and their limitations. We pointed out, using arguments from Pering et al. [PBB98], that interval-based algorithms are generally inferior to deadline-based algorithms. In this chapter, we will show how approaches we developed can improve any currently recommended deadline-based algorithm, including those recommended by Pering et al. [PBB98] and Grunwald et al. [GLF$^+$00].

The main reason we can improve current algorithms is that those algorithms consume more energy than necessary due to an invalid assumption. They assume an algorithm should keep speed as constant as possible. This assumption is valid when task CPU requirements are completely known; however, in reality the system almost never knows exactly how many cycles an arriving task requires. We will show that given such an uncertain task work requirement, a constant speed is *not* optimal. Surprisingly, expected energy consumption is minimized by gradually *increasing* speed as the task progresses, i.e., as we discover the task requires more work. So, we call our approach for improving algorithms in this way PACE: Processor Acceleration to Conserve Energy.

We will show mathematically that an optimal schedule exists for increasing speed in this way. However, there are two problems with implementing this schedule directly. First, the schedule describes speed as a continuous function of time, but a practical implementation may not be able to change CPU speed continuously. Second, the schedule depends on knowledge of the probability distribution of task work requirements.

To solve the first problem, we must estimate the probability distribution of task work from the work requirements of previous, similar tasks. We describe and compare various methods for this to find some general, practical methods that achieve good results for a variety of real workloads. To solve the second problem, we must approximate the scheduling function with a schedule that changes speed a limited number of times. We present and test heuristics for this as well.

Using trace-driven simulations of real workloads, we compare the resulting algorithms to previously proposed algorithms. We show that our algorithms consume significantly less energy while achieving identical performance. Without PACE, existing algorithms use DVS to reduce CPU energy consumption by 11–94% with an average of 54.3%. With the best version of PACE, the savings increase to 36–96% with an average of 65.4%. The overall effect is that PACE reduces the CPU energy consumption of existing algorithms by 1.4–49.5% with an average of 20.6%. We also demonstrate that our algorithms are practical and efficient.

PACE is not a complete DVS algorithm by itself; it is a method for improving such an algorithm. It leaves some characteristics of that algorithm unchanged, such as which deadlines it makes and how it schedules tasks that have missed their deadlines. Thus, different algorithms will still be different after PACE modifies them. We will compare these algorithms to show which ones work best when modified by PACE. Note that PACE itself does not provide a uniquely desirable means of selecting the probability for meeting the deadline for each task; the other algorithms we will discuss make that decision. We explain this issue further in Section IV.5.2.

The rest of this chapter is structured as follows. Section IV.2 introduces useful terminology for the remainder of the chapter and presents our model of the problem of dynamic voltage scaling of tasks with deadlines. This analytical model describes a task as a sequence of operations with a known deadline, and describes a dynamic voltage scaling algorithm as an algorithm that produces a speed schedule to use for the duration of such a task. Section IV.3 describes the theory behind PACE, our main approach for improving existing dynamic voltage scaling algorithms. This theory demonstrates that any algorithm can be improved by modifying the schedules it produces to account for the probability distribution of the present task's work requirements. Section IV.4 describes our approach to practically implementing PACE. This approach creates a piecewise constant, rather than continuous, speed schedule, and estimates the probability distribution of task work rather than assuming it is known a priori. Section IV.5 describes how algorithms differ even after modification by PACE and how we can choose between them. Section IV.6 describes the workloads we use for analyzing the performance and energy consumption of algorithms. Section IV.7 presents and discusses results of simulating our suggested algorithm improvements. Here, we see that PACE reduces the energy consumption of existing DVS algorithms by up to 49.5% with an average of 20.6%. Section IV.8 notes how PACE could be used for tasks with hard deadlines as well as soft deadlines. Finally, Section IV.9 concludes.

Although terms defined in this chapter are explained when first presented, the reader may find it helpful to refer to Tables IV.1 and IV.2 occasionally. These tables summarize terms used in this chapter, giving their definitions and their abbreviations.

IV.2 Model

IV.2.1 Deadlines

Software must be careful when it trades performance for energy with DVS, for two reasons. First, a user wants the performance for which he paid. Second, other computer components, such as the disk, display, and backlight, also consume power. If they stay on longer because the CPU runs more slowly, the overall effect can be worse performance and

Term (Abbreviation)	**Definition**
Work requirement / work (W)	The number of CPU cycles a task requires.
Completion time	The number of seconds a task takes to complete.
Deadline (D)	The number of seconds a task has to complete. Generally, a deadline will be *soft*, meaning some tasks may miss their deadlines. The key property of a deadline is that as long as a task completes by its deadline, its actual completion time does not matter.
Effective completion time	The completion time of a task, or its deadline, whichever is greater. This measure reflects the fact that as long as a task completes by its deadline, its actual completion time does not matter.
Delay	The number of seconds a task takes beyond its deadline.
Average delay (AvgDelay)	The average delay of all tasks in a workload.
Excess	The number of cycles of work a task still has left to do after its deadline has passed.
Maximum speed ($s_{\max}$)	The maximum speed, in cycles per second, at which the CPU can run (at maximum voltage).
Minimum speed ($s_{\min}$)	The minimum positive speed, in cycles per second, at which the CPU can run. In other words, the speed it uses at minimum voltage.
Possible deadline	A deadline that could be made if the task were run at maximum speed. (A deadline is possible if and only if $W \leq s_{\max}D$.)
Fraction of deadlines made (FDM)	The fraction of tasks in a workload that make their deadlines.
Fraction of possible deadlines made (FPDM)	The fraction of tasks with possible deadlines in a workload that make their deadlines.
Cumulative distribution function (CDF or F)	A function describing the probability a task will require various amounts of work. $F(w)$ is the probability that the task will require no more than w cycles.
Tail distribution function (F^c)	One minus the cumulative distribution function. $F^c(w)$ is the probability that the task will require more than w cycles.
Megacycle (Mc)	1,000,000 CPU cycles.
Speed schedule (f or s)	A function that describes how the CPU speed will vary as a task runs. $f(t)$ is the speed to use after the task has run for t seconds. $s(w)$ is the speed to use after the task has completed w cycles of work.
Pre-deadline cycles (PDC)	The number of cycles the CPU can complete by the deadline according to some speed schedule. For example, if the speed schedule calls for the speed to always be 300 MHz, and the deadline is 50 ms, then PDC = 15 Mc. Note: even if the task only requires 8 Mc of work, PDC is still 15 Mc, since the schedule *could have* completed 15 Mc by the deadline.

Table IV.1: Terms used in this chapter, along with their abbreviations and definitions.

Term (Abbreviation)	**Definition**
Performance equivalent	Guaranteed to yield the same effective completion time, no matter what the task's work requirements.
Parametric method	A method for estimating a probability distribution from a sample by assuming the distribution belongs to some family of distributions (e.g., normal) and estimating the parameters of that distribution (e.g., the mean).
Nonparametric method	A method for estimating a probability distribution from a sample without assuming any given distribution type. Such a method thus lets the data "speak for themselves."
Kernel density estimation	A nonparametric method that builds up a probability distribution by adding up lots of little distributions, each centered on one of the sample points.
Bandwidth (h)	The width of each little distribution in kernel density estimation.

Table IV.2: More terms used in this chapter, along with their abbreviations and definitions.

increased energy consumption. Thus, a voltage scaling algorithm should generally reduce the voltage only when it will not noticeably affect performance.

A natural way to express this goal is to consider the computer's activity to consist of a set of tasks, each of which has a soft deadline. (We call the deadline *soft* because the task should have a high probability of completing within this amount of time, but this probability does not have to be 1.) For example, user interface studies have shown that user think time is unaffected by response time as long as response time is under 50–100 ms [Shn98], so it is reasonable to consider the deadline for handling a user interface event to be 50 ms. As another example, multimedia programs that operate on real-time streams or that have limited buffering need to complete processing a frame in time equal to the inverse of the display rate. When goals can be codified this way, the job of a dynamic voltage scaling algorithm is to run the CPU just fast enough to meet the deadline with a high probability.

The key property of a deadline is that as long as a task completes by its deadline, its actual completion time does not matter. This means that if we run the task more slowly, but it still completes by its deadline, performance is unaffected. The primary goal of this chapter is to improve voltage scaling algorithms so that their performance remains the same but their energy consumption goes down.

IV.2.2 Processor

We model the processor as follows. The processor can attain any speed between some minimum speed $s_{\min}$ cycles/sec and some maximum speed $s_{\max}$ cycles/sec, inclusive. At each such speed, there is a minimum voltage at which it can run, and we assume it uses that voltage at that speed. This causes the power consumption of the processor to vary with speed.

We assume there exists a function P, defined over the domain $s \geq 0$, such that the power consumption at speed s is $\mathsf{P}(s)$ joules per second. We denote the energy consumption function by E: $\mathsf{E}(s) = \mathsf{P}(s)/s$, defined over the domain $s > 0$. Thus, $\mathsf{E}(s)$ gives the joules consumed per cycle at speed s.

We assume there exists an $\Omega > 0$ such that $\mathsf{P}''(s) > \Omega$ for all $s_{\min} \leq s \leq s_{\max}$. We assume that $\mathsf{E}'(s) > 0$ for all $s_{\min} \leq s \leq s_{\max}$. Essentially, these mean that power consumption is a concave-up function of speed, and energy consumption is an increasing function of speed.

IV.2.3 DVS Algorithms

Now, we describe our model of DVS algorithms. A *DVS algorithm* is one that decides how quickly to run a task as that task progresses. This task has some *work requirement* (W) that is the number of CPU cycles it takes to complete. We will sometimes refer to this simply as the task's *work*. The task has some *deadline* (D): the number of seconds in which the algorithm should try to complete the task. The number of seconds the task actually takes, given the algorithm's CPU speed choices, is its *completion time*. Its *effective completion time* is the maximum of its completion time and its deadline; this reflects the fact that if a task completes by its deadline, it may as well have completed at its deadline.

If the task does not make its deadline, that means it still has work to do. We call the number of cycles still left to do upon reaching the deadline the *excess*. Its *delay* is the number of seconds it takes beyond its deadline to complete this excess work, i.e., its effective completion time minus its deadline.

When a task arrives, an algorithm must decide on the CPU speed to use in completing it. In general, the algorithm may choose to vary the CPU speed as the task progresses; for

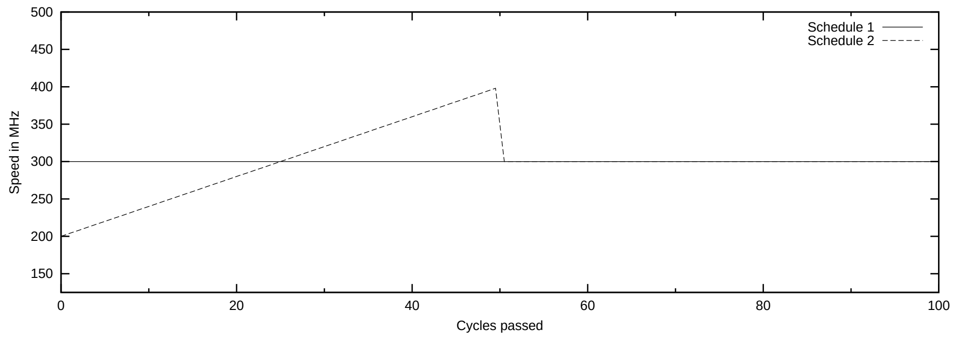

Figure IV.1: This graph shows two performance equivalent speed schedules with deadline 50 ms. Their pre-deadline cycles are equal (15 Mc) and their post-deadline parts are identical.

instance, it might choose to use 300 MHz for the first 10 ms then 400 MHz for any remaining time. Thus, the algorithm is actually choosing the speed as a function of time. We call this function the *speed schedule*, and denote it by f: $f(t)$ is the speed, in cycles per second, that the algorithm will run the CPU after the task has run for t seconds.

A practical algorithm cannot know the task's work requirement until the task completes. Since it gains no information about the task's work as the task progresses, it could, in theory, compute the entire schedule when the task arrives. For example, a practical implementation of an algorithm may not compute $f(8)$ until 8 seconds actually pass, but we consider that $f(8)$ is nevertheless defined as soon as the task arrives. Indeed, even if the task completes after 5 seconds, $f(8)$ is still defined: it is the speed the algorithm *would have* used if the task had taken longer than 8 seconds.

We can think of a speed schedule as consisting of two parts, the *pre-deadline part* and the *post-deadline part*. The former is the part of f that describes what happens before the task reaches its deadline (when $t \leq D$), and the latter describes what happens after the task misses its deadline (when $t > D$). A speed schedule has a certain number of *pre-deadline cycles* (PDC), the number of cycles it can perform before the deadline. Note that $\mathsf{PDC} = \int_0^D f(t)\,dt$. The pre-deadline part determines this value, since $\mathsf{PDC} = \int_0^D f(t)\,dt$. This value is important because the task will miss its deadline if and only if its work requirement exceeds the schedule's pre-deadline cycles, i.e., if $W > \mathsf{PDC}$.

We say that two speed schedules are *performance equivalent* if, no matter what a task's work requirement, it will have the same effective completion time under both schedules. We call two algorithms performance equivalent if they always have performance equivalent speed schedules. We make the following important observation:

> ***If two speed schedules have equal pre-deadline cycles and identical post-deadline parts, then they are performance equivalent.***

Figure IV.1 illustrates two such schedules.

The above observation is true for the following reasons. First, if a task's work is no greater than the PDC the schedules share, then both schedules complete the task by the deadline, and both yield an effective completion time of D. Second, if a task's work is greater than the PDC, then both schedules leave the task the same excess to do after the deadline: $W - \mathsf{PDC}$. Since the schedules have identical post-deadline parts, and both have the same excess to do in that part, both will complete the task at the same time.

This is the key to the PACE approach. PACE modifies algorithms without changing their pre-deadline cycles or their post-deadline parts, so it keeps performance the same. However, by strategically choosing the speed schedule for the pre-deadline part, it can make the expected energy consumption lower than the original algorithm.

It is often useful to consider the schedule as describing speed as a function of work completed, instead of speed as a function of time. So, we will sometimes use the function $\mathsf{s}(w)$ to describe this schedule, where $\mathsf{s}(w)$ gives the speed to use after the task has completed w cycles of work. $f(t)$ and $\mathsf{s}(w)$ are simply different expressions of the same function; it is straightforward to convert a function from one style to the other.

IV.2.4 Names for existing algorithms

We will often need to refer to existing DVS algorithms, so we need a naming scheme to describe them. As Chan et al. [CGW95] describe, each existing algorithm can be described as a *prediction method*, which estimates the upcoming time interval's CPU utilization, combined with a *speed-setting method*, which computes the speed for an interval based on the prediction. Suggested prediction methods include:

- **Past.** This method assumes the upcoming interval's utilization will be the same as the last interval's utilization.

- **Aged-a.** This method averages all past intervals' utilizations to predict the upcoming interval's utilization. More recent intervals are considered more reliable than older intervals. So, the kth most recent interval's utilization is weighted by a^k in the average, where $a \leq 1$ is some constant.

- **LongShort.** This method averages the 12 most recent intervals' utilizations to predict the upcoming interval's utilization. The three most recent of these are weighted 3 times more than the other nine.

- **Flat-u.** This method always makes the same prediction for the upcoming interval's utilization. It predicts it to be some constant u, chosen based on policy.

Suggested speed-setting methods include:

- **Weiser-style.** If the upcoming interval's utilization is predicted to be high (more than 70%), this method increases the speed by 20% of the maximum speed. If the upcoming interval's utilization is predicted to be low (less than 50%), this method decreases the speed by $60 - x\%$ of the maximum speed, where x is the upcoming interval's predicted utilization as a percentage.

- **Peg.** If the upcoming interval's utilization is predicted to be high (more than 98%), this method sets the speed to its maximum. If the upcoming interval's utilization is predicted to be low (less than 93%), this method decreases the speed to its minimum positive value. The 93% and 98% figures were determined empirically in [GLF$^+$00].

- **Chan-style.** This method sets the speed for the upcoming interval just high enough to complete the predicted work. In other words, it multiplies the maximum speed by the utilization to get the speed for the upcoming interval.

We name each algorithm after those methods. Four algorithms have been shown to be particularly effective, so we will be particularly interested in them. These four are:

- **Past/Weiser-style.** This is a practical version of the algorithm Weiser et al. proposed [WWDS94].

- **LongShort/Chan-style.** This is a practical version of one of the best algorithms Chan et al. proposed [CGW95].

- **Flat/Chan-style.** This is a practical version of another of the best algorithms Chan et al. proposed [CGW95]. It runs the CPU at a constant speed, so it is similar to Transmeta's LongRun$^{\text{TM}}$ in steady state. We choose the speed so that the FPDM is at least 98% (99% for the Low-Level workload).

- **Past/Peg.** This is the algorithm Grunwald et al. favored [GLF+00].

IV.3 PACE

IV.3.1 Improving the Pre-deadline Part

Suppose some algorithm produces a speed schedule s. We would like to improve this by producing a performance equivalent speed schedule s_{new} with lower expected energy consumption. To do this, we will update the schedule so that it performs the same number of cycles by the deadline, but by running at different speeds before the deadline.

The traditional basis for dynamic speed scheduling techniques has been keeping speed constant, so it may seem that the optimal schedule would be to run at a constant speed. However, surprisingly, the ideal speed schedule is actually a changing speed, one that increases speed as the task progresses. An intuitive explanation is that if the task work requirement is unknown, it may be high or low. It is worthwhile to run slowly at first, because the task may require little work and thus end before we get to the point where we increase the speed and thus the power consumption. Since this may not convince the reader, we provide a simple example. Suppose a task with deadline 50 ms takes 5 Mc (megacycles) 75% of the time and 10 Mc 25% of the time. Suppose further that CPU power is given by $\mathsf{P}(s) = 5 \times 10^{-26} s^3$. The ideal constant speed is 200 MHz, the slowest speed that will always meet the deadline; this consumes

$$(25\text{ ms})(200 \times 10^6)^3(5 \times 10^{-26})\text{ W} + (25\%)(25\text{ ms})(200 \times 10^6)^3(5 \times 10^{-26})\text{ W} = 12.5\text{ mJ}$$

on average. An alternate, variable speed schedule is 163 MHz for the first 30.675 ms, then

259 MHz for any remaining time; this consumes

$$(30.675 \text{ ms})(163 \times 10^6)^3(5 \times 10^{-26}) \text{ W} \\ + (25\%)(19.325 \text{ ms})(259 \times 10^6)^3(5 \times 10^{-26}) \text{ W} = 10.84 \text{ mJ}$$

on average, an energy savings of 13.3%.

We thus see that the optimal speed schedule depends on the probability distribution of the task's work requirement. We denote the cumulative distribution function (CDF) of this work by F: $F(w)$ is the probability the task requires no more than w cycles of work. The tail distribution function is denoted F^c: $F^c(w) = 1 - F(w)$. The qth *quantile* of this distribution is the value w such that $F(w) = q$.

We are trying to minimize the expected energy consumption of the pre-deadline part of the algorithm, while keeping the pre-deadline cycles the same. The expected energy consumption is

$$\int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}(w)]\,dw,$$

by the following reasoning. Consider the dw cycles of work after the first w; if dw is small, the speed over this period is approximately constant at $\mathsf{s}(w)$, and the energy consumption per cycle over this period is approximately constant at $\mathsf{E}[\mathsf{s}(w)]$. The amount of work done is dw, so the energy consumption is $\mathsf{E}[\mathsf{s}(w)]\,dw$. The probability that this work actually ever gets done is $F^c(w)$.

We call a schedule s *valid* if it has the same pre-deadline cycles as the original algorithm and if it stays within the allowed CPU speed range. We call a valid schedule s_{opt} *optimal* if no other valid schedule has lower expected pre-deadline energy. We seek such an optimal valid schedule.

One can construct an optimal valid schedule as follows. Define the function

$$\mathsf{S}(p) = \begin{cases} s_{\min} & \text{if } ps_{\min}^2\mathsf{E}'(s_{\min}) > 1 \\ s_{\max} & \text{if } ps_{\max}^2\mathsf{E}'(s_{\max}) < 1 \\ s \in [s_{\min}, s_{\max}] \text{ such that } p\,s^2\mathsf{E}'(s) = 1 & \text{otherwise.} \end{cases}$$

Next, find a $K > 0$ such that

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{S}[F^c(w)/K]}\, dw = D.$$

Finally, construct the speed schedule as $\mathsf{s}(w) = \mathsf{S}[F^c(w)/K]$ for $0 \le w \le \mathsf{PDC}$.

In other words, one can form an optimal schedule by making the pre-deadline part of $\mathsf{s}(w)$ equal to $\Theta^{-1}[K/F^c(w)]$ bounded to between $s_{\min}$ and $s_{\max}$, where

$$\Theta(s) = s^2\mathsf{E}'(s).$$

One chooses this K such that the schedule achieves the proper number of pre-deadline cycles. We will see that Θ^{-1} is an increasing function; therefore, since $F^c(w)$ decreases as w increases, this schedule speeds up the CPU as the task progresses, as suggested earlier.

If power consumption is proportional to the cube of the speed, as the typical dynamic voltage scaling model suggests, then $\mathsf{P}(s) = Cs^3$ for some constant C. This means $\mathsf{E}(s) = Cs^2$, $\mathsf{E}'(s) = 2Cs$, and $\Theta(s) = 2Cs^3$. Thus, the optimal schedule uses $\mathsf{s}_{\text{opt}}(w)$ proportional to $[F^c(w)]^{-1/3}$, the inverse cube root of the tail distribution function. The optimal constant of proportionality is simply the one that ensures the schedule performs the required number of pre-deadline cycles.

To illustrate the use of this formula, we will show how to derive the optimal formula for the example shown earlier: a task with deadline 50 ms that takes 5 Mc 75% of the time and 10 Mc 25% of the time. First, observe that

$$F^c(w) = \begin{cases} 1 & \text{if } w \le 5 \text{ Mc} \\ 0.25 & \text{if } 5 \text{ Mc} < w \le 10 \text{ Mc} \\ 0 & \text{if } w > 10 \text{ Mc.} \end{cases}$$

So, we want the speed for the first 5 Mc to be proportional to $1^{-1/3} = 1$ and the speed for the next 5 Mc to be proportional to $0.25^{-1/3} = 1.587$. To make the deadline, the constant of proportionality c must satisfy

$$D = 0.05 \text{ sec} = \frac{5 \text{ Mc}}{c} + \frac{5 \text{ Mc}}{1.587c}.$$

The solution to this equation is $c = 163$ MHz, meaning we should use a speed of 163 MHz for the first 5 Mc, then a speed of $163 \cdot 1.587 = 259$ MHz for the next 5 Mc. It takes 30.675 ms to run 5 Mc at 163 MHz, so our schedule is 163 MHz for the first 30.675 ms, then 259 MHz for the remaining 19.325 ms.

Given any scheduling algorithm, it is worthwhile to replace its pre-deadline part with the optimal formula. In this way, we reduce the expected energy consumption without affecting performance. We call this the PACE approach.

IV.3.2 Proof

We now prove that the construction described in the previous section always produces an optimal valid speed schedule. We begin with some useful notation and lemmas.

Notation. We define $\langle x_1, x_2 \rangle$ to be the closed interval with endpoints x_1 and x_2, even if $x_2 < x_1$. In other words, it is $[x_1, x_2]$ if $x_1 < x_2$ and is $[x_2, x_1]$ otherwise.

Improvement Lemma. If $\sum_{i=1}^{n} y_i \leq \sum_{i=1}^{n} z_i$; $y_i < z_i \Rightarrow \Psi_i'(z_i) \leq c$; $y_i > z_i \Rightarrow \Psi_i'(z_i) \geq c$; and, for all $x \in \langle y_i, z_i \rangle$, $\Psi_i'(x) \leq 0$ and $\Psi_i''(x) \geq 0$; then $\sum_{i=1}^{n} \Psi_i(y_i) \geq \sum_{i=1}^{n} \Psi_i(z_i)$.

Proof. Observe that since $\Psi_i'(x) \leq 0$ for all $x \in \langle y_i, z_i \rangle$, that each such $\Psi_i(x)$ is nonincreasing for $x \in \langle y_i, z_i \rangle$. Since $\Psi_i''(x) \geq 0$ for all $x \in \langle y_i, z_i \rangle$, each such $\Psi_i'(x)$ is nondecreasing for $x \in \langle y_i, z_i \rangle$.

First, note that if $y_i \leq z_i$ for all i, then $\Psi_i(y_i) \geq \Psi_i(z_i)$ for all i, and the proof is trivial. So, we assume $y_j > z_j$ for some $j \in \{1, 2, \ldots, n\}$. Since $y_j > z_j$, we cannot have $y_i \geq z_i$ for all i, because in this case we would have $\sum_{i=1}^{n} y_i > \sum_{i=1}^{n} z_i$, contradicting an assumption of the lemma. So, $\exists k \in \{1, 2, \ldots, n\}$ such that $y_k < z_k$.

We perform this proof by induction on $n \geq 2$, since it is trivial for $n < 2$.

To start the induction, we prove the lemma for $n = 2$. Without loss of generality, we assume $j = 2$ and $k = 1$, i.e., that $y_1 < z_1$ and $y_2 > z_2$.

Since $y_1 < z_1$, we have $\Psi_1'(y_1) \leq \Psi_1'(z_1) \leq c$. Since $y_2 > z_2$, we have $\Psi_2'(y_2) \geq \Psi_2'(z_2) \geq c$. Together, we have $\Psi_2'(y_2) \geq \Psi_2'(z_2) \geq c \geq \Psi_1'(z_1) \geq \Psi_1'(y_1)$.

Define $\delta = y_2 - z_2$, so $\delta > 0$. Note that $\delta \leq z_1 - y_1$, since assuming otherwise means $y_2 - z_2 > z_1 - y_1$, which means $\sum_{i=1}^2 y_i > \sum_{i=1}^2 z_i$, contradicting an assumption of the lemma.

Now, define

$$Z(u) = \Psi_1(y_1 + u) + \Psi_2(y_2 - u).$$

$Z(u)$ is well-defined over this interval because $y_1 \leq y_1 + u \leq y_1 + \delta \leq z_1$ and $y_2 \geq y_2 - u \geq y_2 - \delta = z_2$ there, so $y_1 + u \in \langle y_1, z_1 \rangle$ and $y_2 - u \in \langle y_2, z_2 \rangle$. The derivative of Z over this interval is

$$\frac{dZ}{du} = \Psi_1'(y_1 + u) - \Psi_2'(y_2 - u).$$

Now, for all $0 \leq u \leq \delta$,

$$\Psi_1'(y_1 + u) \leq \Psi_1'(y_1 + \delta) \leq \Psi_1'(z_1) \leq c \leq \Psi_2'(z_2) = \Psi_2'(y_2 - \delta) \leq \Psi_2'(y_2 - u),$$

so $\frac{dZ}{du} \leq 0$. This means that $Z(u)$ is nonincreasing over $0 \leq u \leq \delta$. Thus, $Z(0) \geq Z(\delta)$, or, in other words,

$$\Psi_1(y_1) + \Psi_2(y_2) \geq \Psi_1(y_1 + \delta) + \Psi_2(y_2 - \delta) = \Psi_1(y_1 + \delta) + \Psi_2(z_2) \geq \Psi_1(z_1) + \Psi_2(z_2).$$

This concludes the proof for $n = 2$.

Now, supposing the hypothesis holds for all n such that $2 \leq n < N$, we show it holds for $n = N$. Without loss of generality, $N - 1$ and N form the assumed pair $\{j, k\}$ (not necessarily in that order) with $|z_N - y_N| \leq |z_{N-1} - y_{N-1}|$.

- If $j = N$ and $k = N - 1$, then $y_N > z_N$ and $y_{N-1} < z_{N-1}$, so we have $y_N - z_N \leq z_{N-1} - y_{N-1}$. So, $y_{N-1} \leq z_{N-1} + z_N - y_N < z_{N-1}$. So, $\Psi_{N-1}'(y_{N-1}) \leq \Psi_{N-1}'(z_{N-1} + z_N - y_N) \leq \Psi_{N-1}'(z_{N-1}) \leq c$.
- If $j = N - 1$ and $k = N$, then $y_{N-1} > z_{N-1}$ and $y_N < z_N$, so we have $z_N - y_N \leq$

$y_{N-1} - z_{N-1}$. So, $y_{N-1} \geq z_{N-1} + z_N - y_N > z_{N-1}$. So, $\Psi'_{N-1}(y_{N-1}) \geq \Psi'_{N-1}(z_{N-1} + z_N - y_N) \geq \Psi'_{N-1}(z_{N-1}) \geq c$.

Thus, we have either

$$\begin{gathered} y_{N-1} \leq z_{N-1} + z_N - y_N < z_{N-1} \quad \text{and} \quad \Psi'_{N-1}(z_{N-1} + z_N - y_N) \leq \Psi'_{N-1}(z_{N-1}) \leq c \\ \text{or} \\ y_{N-1} \geq z_{N-1} + z_N - y_N > z_{N-1} \quad \text{and} \quad \Psi'_{N-1}(z_{N-1} + z_N - y_N) \geq \Psi'_{N-1}(z_{N-1}) \geq c. \end{gathered} \tag{IV.3.1}$$

Define

$$\bar{z}_i = \begin{cases} z_i & \text{if } i \in \{1, 2, \ldots, N-2\} \\ z_{N-1} + z_N - y_N & \text{if } i = N-1. \end{cases}$$

Then, we have

$$\sum_{i=1}^{N-1} \bar{z}_i = \sum_{i=1}^{N-1} z_i + z_N - y_N = \sum_{i=1}^{N} z_i - y_N \geq \sum_{i=1}^{N} y_i - y_N = \sum_{i=1}^{N-1} y_i.$$

Since $\bar{z}_i = z_i$ for all $i \in \{1, 2, \ldots, N-2\}$, the following hold:

- for all $i \in \{1, 2, \ldots, N-2\}$ and all $x \in \langle y_i, z_i \rangle$, $\Psi'_i(x) \leq 0$ and $\Psi''_i(x) \geq 0$;
- for all $i \in \{1, 2, \ldots, N-2\}$ such that $y_i < \bar{z}_i$, $\Psi'_i(\bar{z}_i) \leq c$; and
- for all $i \in \{1, 2, \ldots, N-2\}$ such that $y_i > \bar{z}_i$, $\Psi'_i(\bar{z}_i) \geq c$.

We now show these three things hold for $i = N-1$. First, for all $x \in \langle y_{N-1}, \bar{z}_{N-1} \rangle = \langle y_{N-1}, z_{N-1} + z_N - y_N \rangle$, by (IV.3.1) we have $x \in \langle y_{N-1}, z_{N-1} \rangle$, so $\Psi'_{N-1}(x) \leq 0$ and $\Psi''_{N-1}(x) \geq 0$. Second, if $y_{N-1} < \bar{z}_{N-1}$, then by (IV.3.1) we have $\Psi'_{N-1}(\bar{z}_{N-1}) \leq c$. Third, if $y_{N-1} > \bar{z}_{N-1}$, then by (IV.3.1) we have $\Psi'_{N-1}(\bar{z}_{N-1}) \geq c$.

By the inductive hypothesis, the lemma holds for $n = N-1$. Thus, we can apply it to the sequences $\{\Psi_1, \Psi_2, \ldots, \Psi_{N-1}\}$, $\{y_1, y_2, \ldots, y_{N-1}\}$, and $\{\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_{N-1}\}$ to get $\sum_{i=1}^{N-1} \Psi_i(y_i) \geq \sum_{i=1}^{N-1} \Psi_i(\bar{z}_i)$. In other words,

$$\sum_{i=1}^{N-1} \Psi_i(y_i) \geq \Psi_{N-1}(z_{N-1} + z_N - y_N) + \sum_{i=1}^{N-2} \Psi_i(z_i). \tag{IV.3.2}$$

Now, define $\dot{\Psi}_1 = \Psi_{N-1}$; $\dot{\Psi}_2 = \Psi_N$; $\dot{y}_1 = z_{N-1} + z_N - y_N$; $\dot{y}_2 = y_N$; $\dot{z}_1 = z_{N-1}$; and $\dot{z}_2 = z_N$. Then,

$$\sum_{i=1}^{2} \dot{y}_i = \sum_{i=1}^{2} \dot{z}_i.$$

Also, by (IV.3.1), we have $\langle \dot{y}_1, \dot{z}_1 \rangle \subseteq \langle y_{N-1}, z_{N-1} \rangle$. Thus, for all $i \in \{1, 2\}$ and all $x \in \langle \dot{y}_i, \dot{z}_i \rangle$, $\dot{\Psi}_i''(x) > 0$ and $\dot{\Psi}_i'(x) \leq 0$. From (IV.3.1), we also see that

- for all $i \in \{1, 2\}$ such that $\dot{y}_i < \dot{z}_i$, $\quad \dot{\Psi}_i'(\dot{z}_i) \leq c$; and
- for all $i \in \{1, 2\}$ such that $\dot{y}_i > \dot{z}_i$, $\quad \dot{\Psi}_i'(\dot{z}_i) \geq c$.

By the inductive hypothesis, the lemma holds for $n = 2$, so we can apply it to the sequences $\{\dot{\Psi}_1, \dot{\Psi}_2\}$, $\{\dot{y}_1, \dot{y}_2\}$, and $\{\dot{z}_1, \dot{z}_2\}$ to get $\sum_{i=1}^{2} \dot{\Psi}_i(\dot{y}_i) \geq \dot{\Psi}_i(\dot{z}_i)$. In other words,

$$\Psi_{N-1}(z_{N-1} + z_N - y_N) + \Psi_N(y_N) \geq \Psi_{N-1}(z_{N-1}) + \Psi_N(z_N).$$

Combining this with (IV.3.2), we get

$$\sum_{i=1}^{N-1} \Psi_i(y_i) \geq \Psi_{N-1}(z_{N-1}) + \Psi_N(z_N) - \Psi_N(y_N) + \sum_{i=1}^{N-2} \Psi_i(z_i),$$

which directly leads to $\sum_{i=1}^{N} \Psi_i(y_1) \geq \sum_{i=1}^{N} \Psi_i(z_i)$. This completes the induction step, and thus the proof. $\Box$

Continuous Improvement Lemma. If

- $a \leq b$;
- $\int_a^b g(w)\, dw \leq \int_a^b h(w)\, dw$;
- for all $w \in [a, b]$ and all $x \in \langle g(w), h(w) \rangle$, $\quad \psi_w'(x) \leq 0$ and $\psi_w''(x) \geq 0$;
- for all $w \in [a, b]$ such that $g(w) < h(w)$, $\quad \psi_w'[h(w)] \leq c$;
- for all $w \in [a, b]$ such that $g(w) > h(w)$, $\quad \psi_w'[h(w)] \geq c$; and
- the integrals $\int_a^b \psi_w[g(w)]\, dw$ and $\int_a^b \psi_w[h(w)]\, dw$ exist;

then $\int_a^b \psi_w[g(w)]\,dw \geq \int_a^b \psi_w[h(w)]\,dw$.

Proof. If $a = b$, then $\int_a^b \psi_w[g(w)]\,dw = 0 = \int_a^b \psi_w[h(w)]\,dw$, so the result trivially holds. Therefore, from this point on, we assume $a < b$.

Since $\psi'_w(x) \leq 0$ for $x \in \langle g(w), h(w)\rangle$, $\psi_w(x)$ must be nonincreasing for $x \in \langle g(w), h(w)\rangle$. Since $\psi''_w(x) \geq 0$ for $x \in \langle g(w), h(w)\rangle$, $\psi'_w(x)$ must be nondecreasing for $x \in \langle g(w), h(w)\rangle$.

If $c \geq 0$, then consider any $w \in [a, b]$.

- If $g(w) \leq h(w)$, then $\psi_w[g(w)] \geq \psi_w[h(w)]$ since $\psi_w(x)$ is nonincreasing over $[g(w), h(w)]$.
- If $g(w) > h(w)$, then $\psi'_w[h(w)] \geq c \geq 0$, and since $\psi'_w[h(w)] \leq 0$ we must have $\psi'_w[h(w)] = 0$. Now, $\psi'_w(x)$ is nondecreasing and nonpositive for $x \in [h(w), g(w)]$, so we must have $\psi'_w(x) = 0$ there. Thus, $\psi_w(x)$ must be constant over $[h(w), g(w)]$, meaning $\psi_w[g(w)] = \psi_w[h(w)]$.

So, we see that in either case $\psi_w[g(w)] \geq \psi_w[h(w)]$. Thus, $\int_a^b \psi_w[g(w)]\,dw \geq \int_a^b \psi_w[h(w)]\,dw$, and the proof is solved. As we have proved the case $c \geq 0$, from now on we assume $c < 0$.

Define γ by

$$\gamma = \int_a^b \left(\psi_w[h(w)] - \psi_w[g(w)]\right)\,dw.$$

This quantity exists, since both of the integrals in this expression are assumed to exist. Suppose, for the purpose of proof by contradiction, that $\gamma > 0$. By the definition of the integral, this means that there exists some $\delta_1 > 0$ such that

$$v_0 = a; \quad v_n = b; \quad \text{and, } \forall i \in \{1, 2, \ldots, n\}, \quad 0 < v_i - v_{i-1} < \delta_1 \text{ and } v_{i-1} \leq w_i \leq v_i \Rightarrow \left|\gamma - \sum_{i=1}^{n} \left(\psi_{w_i}[h(w_i)] - \psi_{w_i}[g(w_i)]\right)(v_i - v_{i-1})\right| < \gamma/2. \quad \text{(IV.3.3)}$$

Define $\epsilon = -\gamma/2c$. Since $c < 0$, we have $\epsilon > 0$. By the definition of the integral, there exists some $\delta_2 > 0$ such that

$$v_0 = a; \quad v_n = b; \quad \text{and, } \forall i \in \{1, 2, \ldots, n\}, \quad 0 < v_i - v_{i-1} < \delta_2 \text{ and } v_{i-1} \le w_i \le v_i \Rightarrow \left| \int_a^b [h(w) - g(w)]\, dw - \sum_{i=1}^n [h(w_i) - g(w_i)](v_i - v_{i-1}) \right| < \epsilon. \quad \text{(IV.3.4)}$$

Let $\delta = \min\{\delta_1, \delta_2\}$. Note that $\delta > 0$. Consider any sequences $\{v_0, v_1, \ldots, v_n\}$ and $\{w_1, w_2, \ldots, w_n\}$ such that $v_0 = a, \quad v_n = b$, and, for all $i \in \{1, 2, \ldots, n\}, \quad 0 < v_i - v_{i-1} < \delta$ and $v_{i-1} \le w_i \le v_i$.

Then, for $i \in \{1, 2, \ldots, n\}$, define $y_i = g(w_i)(v_i - v_{i-1})$ and $z_i = h(w_i)(v_i - v_{i-1})$. Also, for $i \in \{1, 2, \ldots, n\}$, define

$$\Psi_i(x) = (v_i - v_{i-1})\psi_{w_i}\left(\frac{x}{v_i - v_{i-1}}\right).$$

This is well-defined for $x \in \langle y_i, z_i \rangle$, since $\psi_{w_i}(t)$ is defined for $t \in \langle g(w_i), h(w_i) \rangle$. Finally, define $y_{n+1} = 0, \quad z_{n+1} = \epsilon$, and, for all $x > 0$, $\Psi_{n+1}(x) = cx$. This way,

$$\begin{aligned} \sum_{i=1}^{n+1}(z_i - y_i) &= (\epsilon - 0) + \sum_{i=1}^{n}[h(w_i) - g(w_i)](v_i - v_{i-1}) \\ &> \epsilon + \int_a^b [h(w) - g(w)]\, dw - \epsilon && \text{by (IV.3.4)} \\ &\ge 0. && \text{since } \int_a^b g(w)\, dw \le \int_a^b h(w)\, dw \end{aligned}$$

We thus see that $\sum_{i=1}^{n+1} y_i \le \sum_{i=1}^{n+1} z_i$. For all $i \in \{1, 2, \ldots, n\}$ and all $x \in \langle y_i, z_i \rangle$,

$$\Psi_i'(x) = \psi_{w_i}'\left(\frac{x}{v_i - v_{i-1}}\right) \qquad \text{and} \qquad \Psi_i''(x) = \left(\frac{1}{v_i - v_{i-1}}\right)\psi_{w_i}''\left(\frac{x}{v_i - v_{i-1}}\right),$$

so $\Psi_i'(x) \le 0$ and $\Psi_i''(x) \ge 0$. For $i = n+1$ and all $x \in \langle y_{n+1}, z_{n+1} \rangle$, $\Psi_i'(x) = c$ and $\Psi_i''(x) = 0$, so $\Psi_i'(x) \le 0$ and $\Psi_i''(x) \ge 0$ (since $c < 0$).

For all $i \in \{1, 2, \ldots, n\}$ such that $y_i < z_i$, we have $g(w_i) < h(w_i)$, so $\psi_{w_i}'[h(w_i)] \le c$, giving $\Psi_i'(z_i) \le c$. For all $i \in \{1, 2, \ldots, n\}$ such that $y_i > z_i$, we have $g(w_i) > h(w_i)$, so $\psi_{w_i}'[h(w_i)] \ge c$, giving $\Psi_i'(z_i) \ge c$. For $i = n+1$, we have $y_i < z_i$ and also $\Psi_i'(z_i) = c$.

We can thus apply the improvement lemma to the sequences $\{\Psi_1, \Psi_2, \ldots, \Psi_{n+1}\}$, $\{y_1, y_2, \ldots, y_{n+1}\}$, and $\{z_1, z_2, \ldots, z_{n+1}\}$, yielding $\sum_{i=1}^{n+1} \Psi_i(y_i) \geq \sum_{i=1}^{n+1} \Psi_i(z_i)$. In other words,

$$0c + \sum_{i=1}^{n} \psi_{w_i}[g(w_i)](v_i - v_{i-1}) \geq \epsilon c + \sum_{i=1}^{n} \psi_{w_i}[h(w_i)](v_i - v_{i-1}).$$

This means that

$$\sum_{i=1}^{n} \left(\psi_{w_i}[h(w_i)] - \psi_{w_i}[g(w_i)]\right)(v_i - v_{i-1}) \leq -\epsilon c = \gamma/2,$$

contradicting (IV.3.3). So, our supposition must be false, and we must have $\gamma \leq 0$. In other words, $\int_a^b \left(\psi_w[h(w)] - \psi_w[g(w)]\right) dw \leq 0$, which leads directly to the desired result, $\int_a^b \psi_w[g(w)]\, dw \geq \int_a^b \psi_w[h(w)]\, dw$. $\square$

Properties of S Lemma. If

- $s_{\max} > s_{\min} > 0$;
- $\Omega > 0$;
- for all $s \in [s_{\min}, s_{\max}]$, $\mathsf{P}''(s) > \Omega$, $\mathsf{P}(s) = s\mathsf{E}(s)$, and $\mathsf{E}'(s) > 0$;

then we can define

$$\mathsf{S}(p) = \begin{cases} s_{\min} & \text{if } p s_{\min}^2 \mathsf{E}'(s_{\min}) > 1 \\ s_{\max} & \text{if } p s_{\max}^2 \mathsf{E}'(s_{\max}) < 1 \\ s \in [s_{\min}, s_{\max}] \text{ such that } p\, s^2 \mathsf{E}'(s) = 1 & \text{otherwise.} \end{cases}$$

Also, S is continuous, nonincreasing, bounded below by $s_{\min}$, and bounded above by $s_{\max}$. Furthermore, for all p_1 and p_2,

$$\left|\frac{1}{\mathsf{S}(p_2)} - \frac{1}{\mathsf{S}(p_1)}\right| \leq \frac{s_{\max}^4 [\mathsf{E}'(s_{\max})]^2 |p_2 - p_1|}{s_{\min}^3 \Omega}.$$

Proof. Define $\Theta(s) = s^2\mathsf{E}'(s)$. This way, another way to write $\mathsf{S}(p)$ is

$$\mathsf{S}(p) = \begin{cases} s_{\min} & \text{if } p > 1/\Theta(s_{\min}) \\ s_{\max} & \text{if } p < 1/\Theta(s_{\max}) \\ \Theta^{-1}(1/p) & \text{if } p \in [1/\Theta(s_{\max}), 1/\Theta(s_{\min})]. \end{cases}$$

Now, $\Theta'(s) = 2s\mathsf{E}'(s) + s^2\mathsf{E}''(s)$. Observing that

$$\begin{aligned} \mathsf{P}(s) &= s\mathsf{E}(s) \\ \mathsf{P}'(s) &= \mathsf{E}(s) + s\mathsf{E}'(s) \\ \mathsf{P}''(s) &= 2\mathsf{E}'(s) + s\mathsf{E}''(s), \end{aligned}$$

we find $\Theta'(s) = s\mathsf{P}''(s)$. Since $\mathsf{P}''(s) > 0$ for $s \in [s_{\min}, s_{\max}]$, we have $\Theta'(s) > 0$ there. We conclude that, for $s \in [s_{\min}, s_{\max}]$, $\Theta(s)$ is continuous, differentiable, and strictly increasing.

The only possible case where $\mathsf{S}(p)$ may not be defined is its third case, in which $\Theta(s_{\min}) \le 1/p \le \Theta(s_{\max})$. Since $\Theta(s)$ is continuous and strictly increasing for $s \in [s_{\min}, s_{\max}]$, there must be exactly one $s \in [s_{\min}, s_{\max}]$ such that $\Theta(s) = 1/p$, i.e., such that $ps^2E'(s) = 1$. Therefore, $\mathsf{S}(p)$ is well-defined.

Now, $\Theta(s)$ is continuous and strictly increasing for $s \in [s_{\min}, s_{\max}]$, so its inverse is continuous and strictly increasing; since $\mathsf{S}(p) = \Theta^{-1}(1/p)$ for $p \in [1/\Theta(s_{\max}), 1/\Theta(s_{\min})]$, $\mathsf{S}(p)$ is continuous and strictly decreasing there. Since S is also constant over the domains $(-\infty, 1/\Theta(s_{\max})]$ and $[1/\Theta(s_{\min}), \infty)$, it is continuous and nonincreasing everywhere. By its definition, it is bounded below by $s_{\min}$ and bounded above by $s_{\max}$.

Now, suppose we have p_1 and p_2; we want to show that

$$\left| \frac{1}{\mathsf{S}(p_2)} - \frac{1}{\mathsf{S}(p_1)} \right| \le \frac{s_{\max}^4 [\mathsf{E}'(s_{\max})]^2 |p_2 - p_1|}{s_{\min}^3 \Omega}.$$

Without loss of generality, since S is nonincreasing, we can assume $p_1 \le p_2$ and just show that

$$\frac{1}{\mathsf{S}(p_2)} - \frac{1}{\mathsf{S}(p_1)} \le \frac{[\Theta(s_{\max})]^2 (p_2 - p_1)}{s_{\min}^3 \Omega}.$$

If $p_2 < 1/\Theta(s_{\max})$, the proof is simple, since $p_1 \leq p_2$ implies $\mathsf{S}(p_1) = \mathsf{S}(p_2) = s_{\max}$. If $p_1 > 1/\Theta(s_{\min})$, the proof is also simple, since $p_1 \leq p_2$ implies $\mathsf{S}(p_2) = \mathsf{S}(p_1) = s_{\min}$. So, from now on we assume $p_1 \leq 1/\Theta(s_{\min})$ and $p_2 \geq 1/\Theta(s_{\max})$.

Let $p_3 = \max\{p_1, 1/\Theta(s_{\max})\}$ and $p_4 = \min\{p_2, 1/\Theta(s_{\min})\}$. Since the following four inequalities hold:

$$p_1 \leq p_2, \qquad 1/\Theta(s_{\max}) \leq p_2, \qquad p_1 \leq 1/\Theta(s_{\min}), \qquad \text{and} \qquad 1/\Theta(s_{\max}) \leq 1/\Theta(s_{\min}),$$

we see that $p_3 \leq p_4$ no matter which of the selections is used for p_3 or p_4. So, $1/\Theta(s_{\max}) \leq p_3 \leq p_4 \leq 1/\Theta(s_{\min})$. Also, note that $p_3 \geq p_1$ and $p_4 \leq p_2$.

Now, if $p_1 < 1/\Theta(s_{\max})$, then $\mathsf{S}(p_1) = s_{\max} = \mathsf{S}[1/\Theta(s_{\max})] = \mathsf{S}(p_3)$; if $p_1 \geq 1/\Theta(s_{\max})$, then $p_1 = p_3$, so we have $\mathsf{S}(p_1) = \mathsf{S}(p_3)$ in this case as well. Similarly, if $p_2 > 1/\Theta(s_{\min})$, then $\mathsf{S}(p_2) = s_{\min} = \mathsf{S}[1/\Theta(s_{\min})] = \mathsf{S}(p_4)$; if $p_2 \leq \Theta(s_{\min})$, then $p_2 = p_4$, so we have $\mathsf{S}(p_2) = \mathsf{S}(p_4)$ in this case as well. We conclude that $\mathsf{S}(p_1) = \mathsf{S}(p_3)$ and $\mathsf{S}(p_2) = \mathsf{S}(p_4)$.

Given these facts, we find

$$\begin{aligned}
\frac{1}{\mathsf{S}(p_2)} - \frac{1}{\mathsf{S}(p_1)} &= \frac{1}{\mathsf{S}(p_4)} - \frac{1}{\mathsf{S}(p_3)} \\
&= \int_{p_3}^{p_4} \left(\frac{1}{\mathsf{S}}\right)'(p)\,dp \\
&= \int_{p_3}^{p_4} -[\mathsf{S}(p)]^{-2}\mathsf{S}'(p)\,dp \\
&= \int_{p_3}^{p_4} -[\mathsf{S}(p)]^{-2}(-p^{-2})[\Theta^{-1}]'(1/p)\,dp \\
&= \int_{p_3}^{p_4} \frac{[p\mathsf{S}(p)]^{-2}}{\Theta'[\mathsf{S}(p)]}\,dp \\
&= \int_{p_3}^{p_4} \frac{[p\mathsf{S}(p)]^{-2}}{\mathsf{S}(p)\mathsf{P}''[\mathsf{S}(p)]}\,dp \\
&= \int_{p_3}^{p_4} \frac{p^{-2}}{[\mathsf{S}(p)]^3\mathsf{P}''[\mathsf{S}(p)]}\,dp \\
&\le \int_{p_3}^{p_4} \frac{[\Theta(s_{\max})]^2}{s_{\min}^3\Omega}\,dp \\
&= \frac{[\Theta(s_{\max})]^2(p_4-p_3)}{s_{\min}^3\Omega} \\
&\le \frac{[\Theta(s_{\max})]^2(p_2-p_1)}{s_{\min}^3\Omega}.
\end{aligned}$$

This completes the proof. □

Optimal Schedule Theorem. If

- $s_{\max} > s_{\min} > 0$;
- $\Omega > 0$;
- for all $s \in [s_{\min}, s_{\max}]$, $\mathsf{P}''(s) > \Omega$, $\mathsf{P}(s) = s\mathsf{E}(s)$, and $\mathsf{E}'(s) > 0$;
- F^c is a tail distribution function, i.e., it is a nonincreasing function bounded below by 0 and above by 1;
- $D > 0$;
- $F^c(s_{\min}D) > 0$;

- PDC is the pre-deadline cycle count for some algorithm's speed schedule, i.e., $s_{\min}D \leq \mathsf{PDC} \leq s_{\max}D$;

- we define an *equivalent schedule* as any function s with exactly as many pre-deadline cycles as the original algorithm's speed schedule, i.e., one such that

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\,dw = D;$$

- we define a *valid schedule* as any function s such that $s_{\min} \leq \mathsf{s}(w) \leq s_{\max}$ for all $w \geq 0$; and

- we define an *optimal equivalent schedule* as a valid equivalent schedule s_{opt} such that

$$\int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}_{\text{opt}}(w)]\,dw \leq \int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}(w)]\,dw$$

  for any valid schedule $\mathsf{s}(w)$ such that

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\,dw \leq D;$$

then we can define

$$\mathsf{S}(p) = \begin{cases} s_{\min} & \text{if } ps_{\min}^2\mathsf{E}'(s_{\min}) > 1 \\ s_{\max} & \text{if } ps_{\max}^2\mathsf{E}'(s_{\max}) < 1 \\ s \in [s_{\min}, s_{\max}] \text{ such that } p\,s^2\mathsf{E}'(s) = 1 & \text{otherwise.} \end{cases}$$

Also, there exists a $K > 0$ such that

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{S}[F^c(w)/K]}\,dw = D.$$

Furthermore, given such a K, $\mathsf{s}_{\text{opt}}(w) = \mathsf{S}[F^c(w)/K]$ is an optimal valid equivalent schedule.

Proof. S is well-defined by the Properties of S Lemma.

Define

$$Z(\sigma) = \int_0^{\mathsf{PDC}} \frac{1}{\mathsf{S}[\sigma F^c(w)]}\, dw$$

for $\sigma > 0$. This is well-defined, since $\frac{1}{\mathsf{S}[\sigma F^c(w)]}$ is nonnegative and bounded above by $1/s_{\min}$ over the interval of integration.

Define Θ as in the proof of the Properties of S Lemma: $\Theta(s) = s^2 E'(s)$.

Suppose $\sigma_1 > 0$ and $\epsilon > 0$. Let

$$\delta = \min\left\{\frac{\sigma_1}{2}, \frac{\epsilon\,\sigma_1 \Theta(s_{\min}) s_{\min}^3 \Omega}{4[\Theta(s_{\max})]^2 \cdot \mathsf{PDC}}\right\}.$$

Note that $\delta > 0$. Now, suppose $|\sigma_2 - \sigma_1| < \delta$, and consider any $x \geq 0$. If $x \geq 2/\sigma_1\Theta(s_{\min})$, we have $\sigma_2 x > 1/\Theta(s_{\min})$ since $\sigma_2 > \sigma_1/2$, which leads to $\mathsf{S}(\sigma_2 x) = s_{\min}$, which leads to

$$\left|\frac{1}{\mathsf{S}(\sigma_2 x)} - \frac{1}{\mathsf{S}(\sigma_1 x)}\right| = \left|\frac{1}{s_{\min}} - \frac{1}{s_{\min}}\right| = 0.$$

If $x < 2/\sigma_1\Theta(s_{\min})$, then by the Properties of S Lemma we have

$$\left|\frac{1}{\mathsf{S}(\sigma_2 x)} - \frac{1}{\mathsf{S}(\sigma_1 x)}\right| \leq \frac{[\Theta(s_{\max})]^2}{s_{\min}^3 \Omega}|\sigma_2 x - \sigma_1 x| \leq \frac{2\delta[\Theta(s_{\max})]^2}{\sigma_1 \Theta(s_{\min}) s_{\min}^3 \Omega} \leq \frac{\epsilon/2}{\mathsf{PDC}}.$$

Thus,

$$|\sigma_2 - \sigma_1| < \delta \Rightarrow \left|\frac{1}{\mathsf{S}(\sigma_2 x)} - \frac{1}{\mathsf{S}(\sigma_1 x)}\right| \leq \frac{\epsilon/2}{\mathsf{PDC}}$$

for all $x \geq 0$. So, since $F^c(w) \geq 0$ for all w,

$$\begin{aligned}|Z(\sigma_2) - Z(\sigma_1)| &= \left|\int_0^{\mathsf{PDC}} \left(\frac{1}{\mathsf{S}[\sigma_2 F^c(w)]} - \frac{1}{\mathsf{S}[\sigma_1 F^c(w)]}\right) dw\right| \\ &\leq \int_0^{\mathsf{PDC}} \left|\frac{1}{\mathsf{S}[\sigma_2 F^c(w)]} - \frac{1}{\mathsf{S}[\sigma_1 F^c(w)]}\right| dw \leq \int_0^{\mathsf{PDC}} \frac{\epsilon/2}{\mathsf{PDC}}\, dw = \epsilon/2 < \epsilon.\end{aligned}$$

We thus see that for any $\sigma_1 > 0$ and $\epsilon > 0$, there exists a $\delta > 0$ such that $|\sigma_2 - \sigma_1| < \delta \Rightarrow |Z(\sigma_2) - Z(\sigma_1)| < \epsilon$. We conclude that $Z(\sigma)$ is continuous for $\sigma > 0$.

Now, $F^c(w) \leq 1$ for all w, so $F^c(w)/\Theta(s_{\max}) \leq 1/\Theta(s_{\max})$ and thus

$\mathsf{S}[F^c(w)/\Theta(s_{\max})] = s_{\max}$. Thus,

$$Z[1/\Theta(s_{\max})] = \int_0^{\mathsf{PDC}} \frac{1}{\mathsf{S}[F^c(w)/\Theta(s_{\max})]}\,dw = \int_0^{\mathsf{PDC}} \frac{1}{s_{\max}}\,dw = \frac{\mathsf{PDC}}{s_{\max}} \le D.$$

Next, observe that $F^c(w) \ge F^c(s_{\min}D)$ for all $w \le s_{\min}D$ since F^c is a tail distribution function. By assumption, $F^c(s_{\min}D) > 0$, so we have $F^c(w)/\Theta(s_{\min})F^c(s_{\min}D) \ge 1/\Theta(s_{\min})$ and thus $\mathsf{S}[F^c(w)/\Theta(s_{\min})F^c(s_{\min}D)] = s_{\min}$ for all $w \le s_{\min}D$. Thus,

$$\begin{aligned} Z\left[\frac{1}{\Theta(s_{\min})F^c(s_{\min}D)}\right] &= \int_0^{\mathsf{PDC}} \frac{1}{\mathsf{S}[F^c(w)/\Theta(s_{\min})F^c(s_{\min}D)]}\,dw \\ &\ge \int_0^{s_{\min}D} \frac{1}{\mathsf{S}[F^c(w)/\Theta(s_{\min})F^c(s_{\min}D)]}\,dw = \int_0^{s_{\min}D} \frac{1}{s_{\min}}\,dw = D. \end{aligned}$$

Since

$$Z[1/\Theta(s_{\max})] \le D,$$
$$Z\left[\frac{1}{\Theta(s_{\min})F^c(s_{\min}D)}\right] \ge D,$$

and $Z(\sigma)$ is continuous for $\sigma > 0$, there must exist some $\sigma_0 > 0$ such that $Z(\sigma_0) = D$. Then, if we set $K = 1/\sigma_0$, we get

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{S}[F^c(w)/K]}\,dw = D.$$

This completes the existence part of the proof.

Now, suppose $K > 0$ satisfies

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{S}[F^c(w)/K]}\,dw = D.$$

We need to show that $\mathsf{s}_{\text{opt}}(w) = \mathsf{S}[F^c(w)/K]$ is an optimal equivalent schedule. Clearly, it is an equivalent schedule. The Properties of S Lemma shows that the range of S is $[s_{\min}, s_{\max}]$, so clearly $s_{\min} \le \mathsf{s}_{\text{opt}}(w) \le s_{\max}$ for all $0 \le w \le \mathsf{PDC}$. Thus, s_{opt} is also a valid schedule.

Consider any valid schedule s such that

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\,dw \le D.$$

Define $g(w) = \frac{1}{\mathsf{s}(w)}$ and $h(w) = \frac{1}{\mathsf{s}_{\text{opt}}(w)}$ for all $0 \le w \le \mathsf{PDC}$. Note that $g(w)$ and $h(w)$ are in $[1/s_{\max}, 1/s_{\min}]$, so $\langle g(w), h(w)\rangle \subseteq [1/s_{\max}, 1/s_{\min}]$. Also, let $a = 0$, $b = \mathsf{PDC}$, and $c = -K$. For each $0 \le w \le \mathsf{PDC}$ and each $x \in \langle g(w), h(w)\rangle$, define $\psi_w(x) = F^c(w)\mathsf{E}(1/x)$. Note that $\psi_w(x)$ is well-defined since $\langle g(w), h(w)\rangle \subseteq [1/s_{\max}, 1/s_{\min}]$.

Clearly, $a \le b$. Also,

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\, dw \le D = \int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}_{\text{opt}}(w)}\, dw,$$

so $\int_a^b g(u)\, du \le \int_a^b h(u)\, du$.

Also, for all $w \in [a, b]$ and all $x \in \langle g(w), h(w)\rangle$,

$$\psi'_w(x) = -F^c(w)\frac{1}{x^2}\mathsf{E}'(1/x) = -F^c(w)\Theta(1/x). \tag{IV.3.5}$$

Clearly, Θ is positive for $s \in [s_{\min}, s_{\max}]$, so we have $\psi'_w(x) \le 0$. Also,

$$\psi''_w(x) = F^c(w)x^{-2}\Theta'(1/x).$$

Recall from the proof of the Properties of S Lemma that $\Theta'(s)$ is positive for $s \in [s_{\min}, s_{\max}]$; thus, we have $\psi''_w(x) \ge 0$.

Using the expression for $\psi'_w(x)$ in (IV.3.5), we see that

$$\psi'_w[g(w)] = -F^c(w)\Theta[\mathsf{s}(w)] \qquad \text{and} \qquad \psi'_w[h(w)] = -F^c(w)\Theta[\mathsf{s}_{\text{opt}}(w)].$$

So, for any w, one of the following three cases applies:

- If $F^c(w) > K/\Theta(s_{\min})$, then $\mathsf{s}_{\text{opt}}(w) = \mathsf{S}[F^c(w)/K] = s_{\min}$, so $\mathsf{s}(w) \ge \mathsf{s}_{\text{opt}}(w)$ and thus $g(w) \le h(w)$. Also,

$$\psi'_w[h(w)] = -F^c(w)\Theta[\mathsf{s}_{\text{opt}}(w)] = -F^c(w)\Theta(s_{\min}) < -K = c.$$

- If $F^c(w) < K\Theta(s_{\max})$, then $\mathsf{s}_{\text{opt}}(w) = \mathsf{S}[F^c(w)/K] = s_{\max}$, so $\mathsf{s}(w) \le \mathsf{s}_{\text{opt}}(w)$ and thus

$g(w) \geq h(w)$. Also,

$$\psi'_w[h(w)] = -F^c(w)\Theta[\mathsf{s}_{\text{opt}}(w)] = -F^c(w)\Theta(s_{\max}) > -K = c.$$

- Otherwise, $1/\Theta(s_{\max}) \leq F^c(w)/K \leq 1/\Theta(s_{\min})$, so $\Theta(\mathsf{S}[F^c(w)]/K) = K/F^c(w)$. This means that $\Theta[\mathsf{s}_{\text{opt}}(w)] = K/F^c(w)$, so

$$\psi'_w[h(w)] = -F^c(w)\Theta[\mathsf{s}_{\text{opt}}(w)] = -\frac{KF^c(w)}{F^c(w)} = -K = c.$$

In any case, we see that whenever $g(w) < h(w)$ we have $\psi'_w[h(w)] \leq c$ and whenever $g(w) > h(w)$ we have $\psi'_w[h(w)] \geq c$.

Finally, observe that each $\psi_w(x) = F^c(w)\mathsf{E}(1/x)$ is nonnegative and bounded above by $\mathsf{E}(s_{\max})$. Thus, each of the integrals

$$\int_0^{\mathsf{PDC}} \psi_w[g(w)]\,dw \quad \text{and} \quad \int_0^{\mathsf{PDC}} \psi_w[h(w)]\,dw$$

exists.

We now have enough information to apply the Continuous Improvement Lemma using the functions ψ, g, and h. This gives $\int_0^{\mathsf{PDC}} \psi_w[g(w)]\,dw \geq \int_0^{\mathsf{PDC}} \psi_w[h(w)]\,dw$, which is equivalent to

$$\int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}(w)]\,dw \geq \int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}_{\text{opt}}(w)]\,dw.$$

We have thus shown all that is needed to prove that $\mathsf{s}_{\text{opt}}(w)$ is an optimal schedule.
□

IV.4 Practically Implementing PACE

In this section, we discuss how we apply the theory of PACE to a practical implementation. First, we will show how to create a piecewise constant, rather than continuous, speed schedule. Second, we will show how to estimate the probability distribution of task work requirements, on which the PACE formula depends.

IV.4.1 Piecewise constant speed schedules

The optimal formula gives a continuous speed schedule, which may be unreasonable if software must notify the CPU each time it wants to change the speed. In practice, we should probably change speed for a task no more than some reasonable number of times n. So, we want a schedule with a limited number of *transition points*, points where the speed may change. We denote the jth transition point by w_j. Note that we are using as transition points values of w where $\mathsf{s}(w)$ changes, not points in time where $\mathsf{f}(t)$ changes. The latter is more natural, but the former makes optimization easier.

IV.4.1.1 Constructing a piecewise constant speed schedule

Suppose we are given fixed transition points $w_0, w_1, w_2, \ldots, w_n$ such that $w_0 = 0$ and $w_n = \mathsf{PDC}$. (We will address the issue of choosing a "good" sequence of transition points in Section IV.4.1.2.) We say a function s is *properly-divided* if it is constant over each interval $(w_{i-1}, w_i]$, i.e., if, for all $i \in \{1, 2, \ldots, n\}$ and any $w_{i-1} < w \leq w_i$, $\mathsf{s}(w) = \mathsf{s}(w_i)$. We say a function s_{opt} is an *optimal properly-divided schedule* if it is properly-divided, valid, and equivalent and if, for any properly-divided, valid function s with at least PDC pre-deadline cycles,

$$\int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}(w)]\, dw \geq \int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}_{\text{opt}}(w)]\, dw.$$

One can construct an optimal properly-divided schedule as follows. For all $i \in \{1, 2, \ldots, n\}$, define

$$H_i = \frac{\int_{w_{i-1}}^{w_i} F^c(w)\, dw}{w_i - w_{i-1}}.$$

Next, find a $K > 0$ such that

$$\sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{S}(H_i/K)} = D.$$

Then, use the speed schedule

$$\mathsf{s}_{\text{opt}}(w) = \mathsf{S}(H_i/K) \text{ for } w_{i-1} < w \leq w_i.$$

Essentially, this uses a speed for each interval as if the tail distribution function were constant over that interval, set equal to the average of the true tail distribution function over that interval. As before, one chooses the constant of proportionality K so that the schedule achieves the proper number of pre-deadline cycles.

If power consumption is proportional to the cube of the speed, as the typical dynamic voltage scaling model suggests, then the optimal schedule uses $\mathsf{s}_{\text{opt}}(w)$ proportional to $[H_i]^{-1/3}$, where H_i is the average value of $F^c(w)$ over the constant-speed interval in which w occurs. The optimal constant of proportionality is simply the one that ensures the schedule performs the required number of pre-deadline cycles.

We now prove this construction works.

Optimal Properly-Divided Schedule Theorem. If, for all $i \in \{1, 2, \ldots, n\}$, we define

$$H_i = \frac{\int_{w_{i-1}}^{w_i} F^c(w)\, dw}{w_i - w_{i-1}},$$

then there exists a $K > 0$ such that

$$\sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{S}(H_i/K)} = D.$$

Furthermore, given such a K, if we define $\mathsf{s}_{\text{opt}}(w) = \mathsf{S}(H_i/K)$ for all $w_{i-1} < w \le w_i$, then s_{opt} is an optimal properly-divided schedule.

Proof. Define

$$Z(\sigma) = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{S}(H_i/\sigma)}$$

for $\sigma > 0$. Since $Z(\sigma)$ is formed by adding and dividing functions that are continuous over $\sigma > 0$, it is continuous over $\sigma > 0$.

Now, observe that for all $i \in \{1, 2, \ldots, n\}$,

$$H_i = \frac{\int_{w_{i-1}}^{w_i} F^c(w)\, dw}{w_i - w_{i-1}} \le \frac{\int_{w_{i-1}}^{w_i} dw}{w_i - w_{i-1}} = \frac{w_i - w_{i-1}}{w_i - w_{i-1}} = 1.$$

So,

$$Z[\Theta(s_{\max})] = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{S}[H_i/\Theta(s_{\max})]} = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{s_{\max}} = \frac{\mathsf{PDC}}{s_{\max}} \leq D.$$

Let $\mathcal{J} = \{1, 2, \ldots, n\} \cap \{i : w_i \geq s_{\min} D\}$. Now, $w_n = \mathsf{PDC} \geq s_{\min} D$, so $\mathcal{J}$ contains at least the element n. Thus, we can define $j = \min(\mathcal{J})$. Then, $w_{j-1} < s_{\min} D \leq w_j$, and from this we get

$$H_j = \frac{\int_{w_{j-1}}^{w_j} F^c(w)\, dw}{w_j - w_{j-1}} \geq \frac{\int_{w_{j-1}}^{s_{\min} D} F^c(w)\, dw}{w_j - w_{j-1}} \geq \frac{(s_{\min} D - w_{j-1}) F^c(s_{\min} D)}{w_j - w_{j-1}} > 0.$$

Now, observe that for all $i \in \{1, 2, \ldots, n-1\}, \quad H_{i+1} \leq H_i$, by the following reasoning:

$$\begin{aligned} H_{i+1} = \frac{\int_{w_i}^{w_{i+1}} F^c(w)\, dw}{w_{i+1} - w_i} &\leq \frac{\int_{w_i}^{w_{i+1}} F^c(w_i) dw}{w_{i+1} - w_i} = F^c(w_i) \\ &= \frac{\int_{w_{i-1}}^{w_i} F^c(w_i)\, dw}{w_i - w_{i-1}} \leq \frac{\int_{w_{i-1}}^{w_i} F^c(w)\, dw}{w_i - w_{i-1}} = H_i. \end{aligned}$$

Thus,

$$\begin{aligned} Z[H_j \Theta(s_{\min})] = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{S}[H_i / H_j \Theta(s_{\min})]} &\geq \sum_{i=1}^{j} \frac{w_i - w_{i-1}}{\mathsf{S}[H_i / H_j \Theta(s_{\min})} \\ &= \sum_{i=1}^{j} \frac{w_i - w_{i-1}}{s_{\min}} = \frac{w_j}{s_{\min}} \geq \frac{s_{\min} D}{s_{\min}} = D. \end{aligned}$$

We thus see that $Z[\Theta(s_{\max})] \leq D$ and $Z[H_j \Theta(s_{\min})] \geq D$. Since $Z(\sigma)$ is continuous over $\sigma > 0$, and $\Theta(s_{\max})$ and $H_j \Theta(s_{\min})$ are both positive, there must exist some $K > 0$ such that $Z(K) = D$. This means

$$\sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{S}[H_i/K]}\, dw = D,$$

as desired. So, we have proved the existence part of the lemma.

Now, suppose we have such an $K > 0$, and define $\mathsf{s}_{\text{opt}}(w) = \mathsf{S}(H_i/K)$ for all $w_{i-1} < w \leq w_i$. We already showed that the range of S is $[s_{\min}, s_{\max}]$, so obviously s_{opt} is valid. Since

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\,dw = \sum_{i=1}^{n} \int_{w_{i-1}}^{w_i} \frac{1}{\mathsf{S}(H_i/K)}\,dw = D,$$

we see s_{opt} is equivalent. Also, it is clear from inspection that s_{opt} is properly-divided.

Consider any properly-divided, valid schedule s such that

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\,dw \le D.$$

For each $i \in \{1, 2, \ldots, n\}$, define

$$y_i = \frac{w_i - w_{i-1}}{\mathsf{s}(w_i)} \text{ and } z_i = \frac{w_i - w_{i-1}}{\mathsf{s}_{\text{opt}}(w_i)}.$$

Also, define

$$\Psi_i(x) = H_i(w_i - w_{i-1})\mathsf{E}\left[\frac{w_i - w_{i-1}}{x}\right]$$

for all $x \in \langle y_i, z_i \rangle$. Clearly, this is well-defined, since $\langle \mathsf{s}(w_i), \mathsf{s}_{\text{opt}}(w_i) \rangle \subseteq [s_{\min}, s_{\max}]$. Finally, let $c = -K$.

Since

$$\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\,dw \le D,$$

we have

$$\begin{aligned}\int_0^{\mathsf{PDC}} \frac{1}{\mathsf{s}(w)}\,dw = \sum_{i=1}^{n} \int_{w_{i-1}}^{w_i} \frac{1}{\mathsf{s}(w)}\,dw &= \sum_{i=1}^{n} \int_{w_{i-1}}^{w_i} \frac{1}{\mathsf{s}(w_i)}\,dw \\ &= \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{s}(w_i)} \le D = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\mathsf{s}_{\text{opt}}(w_i)},\end{aligned}$$

or, in other words, $\sum_{i=1}^{n} y_i \le \sum_{i=1}^{n} z_i$.

Now, observe that for all $x \in \langle y_i, z_i \rangle$,

$$\Psi_i'(x) = -\frac{H_i(w_i - w_{i-1})^2}{x^2}\,\mathsf{E}'\left[\frac{w_i - w_{i-1}}{x}\right] = -H_i\Theta[(w_i - w_{i-1})/x]$$

and

$$\Psi_i''(x) = \frac{H_i(w_i - w_{i-1})}{x^2}\,\Theta'\left[\frac{w_i - w_{i-1}}{x}\right].$$

Thus, $\Psi_i'(x) \le 0$ and $\Psi_i''(x) \le 0$. Also, note that $\Psi_i'(z_i) = -H_i\Theta[\mathsf{s}_{\text{opt}}(w_i)]$.

Now, consider any $i \in \{1, 2, \ldots, n\}$.

- If $H_i \ge K/\Theta(s_{\min})$, then $\mathsf{s}_{\text{opt}}(w_i) = s_{\min}$. This means $\mathsf{s}(w_i) \ge \mathsf{s}_{\text{opt}}(w_i)$ and $\Psi_i'(z_i) = -H_i\Theta(s_{\min}) \le -K = c$. So, $y_i \le z_i$ and $\Psi_i'(z_i) \le c$.
- If $H_i \le K/\Theta(s_{\max})$, then $\mathsf{s}_{\text{opt}}(w_i) = s_{\max}$. This means $\mathsf{s}(w_i) \le \mathsf{s}_{\text{opt}}(w_i)$ and $\Psi_i'(z_i) = -H_i\Theta(s_{\max}) \ge -K = c$. So, $y_i \ge z_i$ and $\Psi_i'(z_i) \ge c$.
- Otherwise, $K/\Theta(s_{\max}) < H_i < K/\Theta(s_{\max})$, so $\mathsf{s}_{\text{opt}}(w_i) = \Theta^{-1}(K/H_i)$. This means $\Psi_i'(z_i) = -H_i(K/H_i) = -K = c$.

In any case, we see that whenever $y_i < z_i$ we have $\Psi_i'(z_i) \le c$, and whenever $y_i > z_i$ we have $\Psi_i'(z_i) \ge c$.

We now know enough to apply the improvement lemma using the sequences $\{\Psi_1, \Psi_2, \ldots, \Psi_n\}$, $\{y_1, y_2, \ldots, y_n\}$, and $\{z_1, z_2, \ldots, z_n\}$. This gives $\sum_{i=1}^n \Psi_i(y_i) \ge \sum_{i=1}^n \Psi_i(z_i)$. In other words,

$$\sum_{i=1}^n H_i(w_i - w_{i-1})\mathsf{E}[\mathsf{s}(w_i)] \ge \sum_{i=1}^n H_i(w_i - w_{i-1})\mathsf{E}[\mathsf{s}_{\text{opt}}(w_i)].$$

Substituting the definition of H_i, we get

$$\sum_{i=1}^n \int_{w_{i-1}}^{w_i} F^c(w)\mathsf{E}[\mathsf{s}(w_i)]\,dw \ge \sum_{i=1}^n \int_{w_{i-1}}^{w_i} F^c(w)\mathsf{E}[\mathsf{s}_{\text{opt}}(w_i)]\,dw.$$

Since s and s_{opt} are properly-divided, we can rewrite this as

$$\int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}(w)]\,dw \ge \int_0^{\mathsf{PDC}} F^c(w)\mathsf{E}[\mathsf{s}_{\text{opt}}(w)]\,dw.$$

This completes the demonstration that s_{opt} is an optimal properly-divided schedule. □

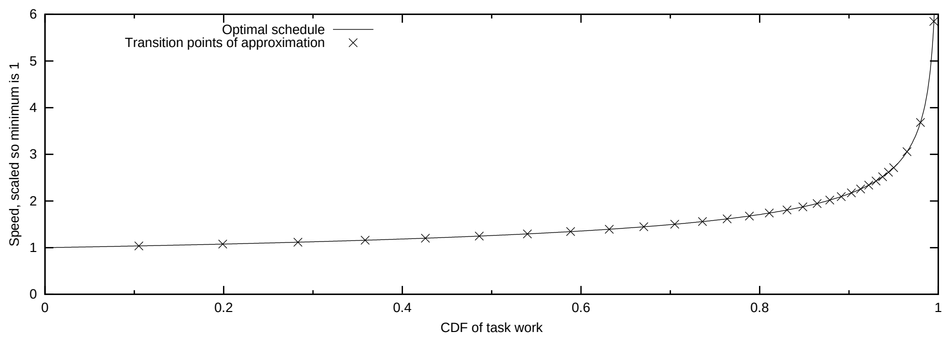

Figure IV.2: This graph shows optimal speed (scaled so that the minimum speed is 1 unit) as a function of the work distribution CDF reached. For example, the optimal schedule uses speed 1.5 after having completed an amount of work equal to the 0.7 quantile of the task work distribution; we graph this as $(0.7, 1.5)$. We choose the quantiles for transition points as follows: we space CDF values less than the knee so that consecutive speeds vary by a constant factor; we use uniform spacing on those few CDF values that are above the knee. Showing the transition points as points along the optimal curve illustrates that, for all but the last few transitions, the speed does not change much between these points. Note that the particular values for this graph assume that power consumption is proportional to speed cubed.

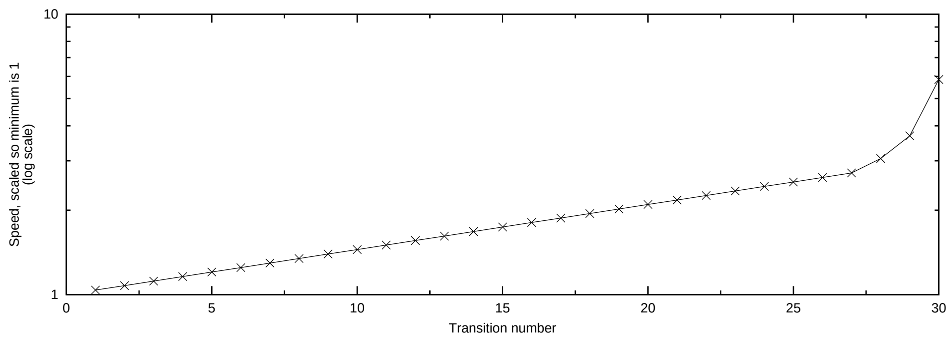

Figure IV.3: This graph shows optimal speed (scaled so that the minimum speed is 1 unit) as a function of the transition number reached. For instance, transition number 11 occurs at the 0.7 quantile, where the optimal speed would be 1.5; we graph this as the point $(11, 1.5)$. This graph uses a log scale for the y axis to illustrate that, for all but the last few transitions, the graph is linear, indicating that the optimal speed varies by no more than a constant factor over the period between two consecutive transition points. Note that the particular values for this graph assume that power consumption is proportional to speed cubed.

IV.4.1.2 Choosing transition points for a piecewise constant speed schedule

To construct a piecewise constant schedule, we need to choose a "good" sequence of N transition points. We want the optimal schedule to vary little between any two consecutive transition points, so that keeping the speed constant between those points approximates the optimal schedule. We proceed as follows. For each integer j, define $q_j = 1 - c^{-j}$ for some constant c. Then, F^c at the q_jth quantile of F equals c^{-j}. If we use these quantiles as transition points, then $K/F^c(w)$ never varies by more than a factor of c between any two consecutive transition points. Thus, the optimal speed $\Theta^{-1}[K/F^c(w)]$ should not vary much between any two consecutive transition points. In the particular case were power is proportional to speed cubed, and thus the optimal speed is proportional to $[F^c(w)]^{-1/3}$, the optimal speed never varies by more than a factor of $c^{1/3}$ between any two consecutive transition points.

A problem with this is that as the sequence $\{q_j\}$ increases, the q_j values get close together, and it wastes our limited supply of speeds to use them. Thus, we terminate this sequence near $q_j = 0.95$ and pick further values of q_j so that they uniformly partition the remaining range. More precisely, we pick some J near N and some Q near 0.95. (We will address later what actual values work well.) We set $q_J = Q$, then compute c by solving the equation $Q = 1 - c^{-J}$. For each $1 \leq j \leq J$, we set $q_j = 1 - c^{-j}$; for each $j > J$, we set $q_j = Q + (j - J)\frac{0.995-Q}{N-J}$. Figures IV.2 and IV.3 illustrate how this works.

IV.4.1.3 Implementing a speed schedule

To implement a piecewise constant speed schedule, software must be able to interrupt the task at predetermined intervals to change the CPU speed. If the CPU can be programmed to cause an interrupt at a given cycle count, the algorithm can use this feature. If the system has a high-frequency hardware timer, the algorithm can use that. Another method is to use soft timers, an operating system facility suggested by Aron et al. [AD99] that lets events be scheduled for the next time one can be performed cheaply, such as when a system call begins or a hardware interrupt occurs. This could only work if these events occur sufficiently frequently. A better way to implement speed schedules would be to implement them in hardware. For instance, the processor could accept as input not just a speed at which to run but a full schedule. Even better would be for the processor to implement the algorithm itself, to let it set its own schedule. However, this kind of specialized functionality in the processor seems too much to expect.

IV.4.2 Sampling methods

Implementing PACE requires some way to estimate the probability distribution of the current task's work requirement. Usually, an application will not provide information about this distribution, so we must model the distribution by sampling the work requirements of similar recent tasks. We consider the following sampling methods.

- **Future.** This method uses as its sample the entire set of tasks in the workload, including future ones. Naturally, this sampling method is impractical, as it uses future information.
- **All.** This method uses as its sample all past tasks.
- **Recent-k.** This method uses as its sample the k most recent tasks.
- **LongShort-k.** This method uses as its sample the k most recent tasks, with the most recent $k/4$ of them weighted 3 times more than the others. This method is inspired by Chan et al.'s methods [CGW95].
- **Aged-a.** This method uses as its sample all past tasks, with the kth most recent having weight a^k. $a \leq 1$ is some constant.

Each of these methods will produce a weighted sample, which we will use to estimate the distribution. We denote the values in this sample by $X_1, X_2, \ldots, X_n$, and denote their weights by $\omega_1, \omega_2, \ldots, \omega_n$. (Future, All, and Recent-k produce samples in which all weights are 1.) Define $\omega = \sum_{i=1}^{n} \omega_i$. Then, the sample mean and variance are

$$\hat{\mu} = \frac{1}{\omega} \sum_{i=1}^{n} \omega_i X_i \quad \text{and} \quad \hat{\sigma}^2 = \left(\frac{n}{n-1} \right) \left[\frac{1}{\omega} \sum_{i=1}^{n} \omega_i X_i^2 - \hat{\mu}^2 \right].$$

Note that all we need to compute these values are the weighted sum, the weighted sum of squares, and the count. For each of our sampling methods, there exists a simple algorithm to update these three quantities in $O(1)$ time whenever a new sample value arrives. Thus, we can recompute the sample mean and variance in $O(1)$ time whenever a task completes.

If tasks can be classified into types in such a way that tasks of the same type have similar work requirements, then we can keep separate samples for each type. When a task arrives, we can better estimate its distribution by using only the sample of tasks of the same type. One way to classify tasks into types is by what application they belong to and by what user interface event triggered them. For instance, we can keep one sample of Microsoft Word tasks triggered by letter keypresses, another sample of Microsoft Excel tasks triggered by releasing the left mouse button, etc.

IV.4.3 Distribution estimation methods

The next step in implementing PACE is to derive the task work distribution from a sample. Note that there are various equivalent ways to express this distribution: as a cumulative distribution function F, as a tail distribution function F^c, or as a set of quantiles. There are two general ways to estimate the distribution from a sample: parametric and nonparametric. Parametric methods assume the distribution belongs to a certain family of distributions (e.g., normal distributions) and estimates the parameters that fully specify a member of that family (e.g., the mean and standard deviation of a normal distribution). Nonparametric methods make no such assumption, letting the sample "speak for itself" in describing the entire distribution.

Normal. The first method we consider is the parametric method assuming a normal distribution. This assumption may seem unwarranted, especially since work cannot be negative but the normal distribution can. However, for our limited purposes, the normal distribution may be a reasonable approximation to task work distributions, and it has the advantage of being easy to model. The normal distribution has only two parameters: the mean μ and the standard deviation σ, whose unbiased estimators are $\hat{\mu}$ and $\hat{\sigma}$. (The maximum likelihood estimator for $\hat{\sigma}$ leaves out the $n/(n-1)$, but we have found it does slightly worse for our purposes.) Furthermore, since the normal distribution $N(\mu, \sigma)$ is a simple linear transformation of the unit normal distribution $N(0, 1)$, one can easily compute quantiles and CDF values using lookup tables.

Gamma. The second method we consider is the parametric method assuming a gamma distribution. This distribution is commonly used to model service times [Jai91, p. 490], and we will show later that it works well. The gamma distribution has range $x \geq 0$. It has two parameters: the shape α and the scale β. The probability density function is

$$p(x) = \frac{(x/\beta)^{\alpha-1}e^{-x/\beta}}{\beta\,\Gamma(\alpha)}.$$

Reasonable estimators for the model parameters are $\hat{\alpha} = \hat{\mu}^2/\hat{\sigma}^2$ and $\hat{\beta} = \hat{\sigma}^2/\hat{\mu}$ [Jai91]. Maximum likelihood estimators also exist, but we do not use them, since (a) they cannot be computed precisely or easily, and (b) we have found that they generally do not work as well for our purposes.

We can approximate quantiles of the gamma distribution using the Wilson-Hilferty approximation, described by Johnson and Kotz [JK70, p. 176]. It estimates a quantile using

$$\alpha\beta\left(\frac{U_q}{3\sqrt{\alpha}} + 1 - \frac{1}{9\alpha}\right)^3$$

where U_q is the relevant quantile of the normal distribution. When needed, we can compute CDF values using algorithms such as those given by Press et al. [Pre92], but we avoid them when possible since they can be computationally expensive.

Computing the average value of the gamma CDF over an interval is computationally expensive, so we approximate it by the average of the two CDF's at the endpoints. This relies on the fact that the gamma CDF is roughly linear over sufficiently short intervals. Unfortunately, when the gamma distribution has a standard deviation that is small relative to the mean, the length of the interval between the schedule start and the first transition point can be large. This is because the schedule start is at work 0 corresponding to quantile 0, and the first transition point is only a few standard deviations below the mean. We avoid this problem by always using the 0.001th quantile as our first transition point. This point may still be some distance from 0, but since the CDF never varies outside of $[0.999, 1]$ over the interval between it and 0, we can closely approximate the average CDF over this interval using 1.

Kernel density estimation. The nonparametric method we consider is kernel density estimation, a popular nonparametric method [Sil86]. This method builds up a distribution by adding up several little distributions, each centered on one of the sample points. The *kernel function*, K, determines the shape of these little distributions. The *bandwidth*, h, determines the width of each little distribution. The result is to estimate the probability density function (PDF) at x to be

$$\hat{p}(x) = \frac{1}{\omega} \sum_{i=1}^{n} \frac{\omega_i}{h} \, K\left(\frac{x - X_i}{h}\right).$$

Silverman [Sil86, pp. 42–43] points out that most kernels perform comparably, so one should choose a kernel based primarily on its ease of implementation. We have chosen the triangular kernel: $K(t) = \max\{1 - |t|, 0\}$.

The theoretical optimal bandwidth is

$$h_{\text{opt}} = \left(\int t^2 K(t)\, dt\right)^{-\frac{2}{5}} \left(\int K(t)^2\, dt\right)^{\frac{1}{5}} \left(\int p''(x)^2\, dx\right)^{-\frac{1}{5}} n^{-\frac{1}{5}}$$

where p'' is the second derivative of the true probability density. For the triangular kernel, $\int t^2 K(t)\, dt = \frac{1}{6}$ and $\int K(t)^2\, dt = \frac{2}{3}$. However, $\int p''(x)^2\, dx$ is impossible to compute since the true probability density is obviously unknown. Fortunately, our estimate of it does not have to be exact, since it will only influence the degree of smoothing in the distribution. Generally, one assumes a normal distribution with parameters $\hat{\mu}$ and $\hat{\sigma}$, making the estimate $\frac{3}{8\sqrt{\pi}}\hat{\sigma}^{-5}$. Assuming a gamma distribution makes the estimation far more complex, and we have found this complexity not to be worthwhile.

We can compute the CDF by observing the following about $\hat{p}(x)$ and $\hat{p}'(x)$. (By $\hat{p}'(x)$ here, we mean the derivative of the estimated PDF from the right, since the general derivative does not always exist.) For $x < -h$, both are 0. As x increases, $\hat{p}(x)$ experiences no jump discontinuities, but $\hat{p}'(x)$ does. At each point $x = X_i - h$, it jumps up by $\omega_i/h^2\omega$. At each point $x = X_i$, it jumps down by $\omega_i/2h^2\omega$. And, at each point $x = X_i + h$, it jumps up by $\omega_i/h^2\omega$. Other than these three cases, it does not change. So, we sort these $3n$ inflection points and then iterate through them to determine the CDF, PDF, and $\hat{p}'(x)$ at each of them.

Note that the range of the kernel density estimate may extend below 0. We use reflection [Sil86, pp. 29–31] to avoid this. This method adds to the sample the set of values $\{-X_i\}$, each weighted ω_i, making the sample size $2n$. It then computes the probability density $\hat{p}_{\mathrm{adj}}(x)$ using this adjusted sample, and sets $\hat{p}(x) = 2\hat{p}_{\mathrm{adj}}(x)$ for $x \geq 0$, $\hat{p}(x) = 0$ otherwise. Because we are using the triangular kernel, we can accomplish all this with just two simple changes to the algorithm from the previous paragraph. First, if $X_i < h$, we use the inflection point $h - X_i$ instead of $X_i - h$. Second, instead of setting $\hat{p}(0) = 0$, we set

$$\hat{p}(0) = \sum_{i:X_i<h} \frac{2(h - X_i)\omega_i}{h^2\omega}.$$

IV.4.4 Efficiently implementing PACE

We now describe how to efficiently compute piecewise constant PACE schedules in software. We assume that power consumption is proportional to speed cubed in this implementation.

Some information about PACE schedules is common to all such schedules and can thus be precomputed. We can precompute the CDF values at which we will perform speed transitions. Knowing these, we can estimate the average value of $F^c(w)$ over each interval between such transitions by averaging the $F^c(w)$ values at the ends of that interval. From these, we can compute the values of $H_i^{-1/3}$ for each interval. This gives us relative speeds for each interval; i.e., we know there is some constant c such that the PACE schedule uses for interval i the speed $cH_i^{-1/3}$, bounded to between $s_{\min}$ and $s_{\max}$. Thus, all we need to compute for a given task work distribution is when the intervals begin and end and what value of c gives the desired PDC.

In outline, the steps are as follows.

1. For each transition point CDF, compute the quantile of the task work distribution from the sample. This gives us the amount of work that is to be done during each interval of the piecewise constant schedule. This step is fairly complicated when using kernel density estimation; see IV.4.3 for a description of this step.

2. Remove any work from the schedule that is beyond the PDC. We only will consider the pre-deadline part of the algorithm for computing the PACE schedule.

3. Determine how much time each interval would take running at its relative speed $H_i^{-1/3}$ by simply dividing the work in that interval by that relative speed. For now, assume that there are no bounds on valid speeds. Add all this time together, giving the total time it would take to perform a number of cycles equal to PDC.

4. Divide the total time by the deadline, indicating how much faster we would have to go to make the deadline. This is our initial estimate of c.

5. Now, we consider whether c is too high or too low given that the actual speed we have to use for each interval is not $cH_i^{-1/3}$ but actually $cH_i^{-1/3}$ bounded to between $s_{\min}$ and $s_{\max}$. If bounding the speed will only lower the speed (as when the end of the schedule must be bounded but the start of the schedule need not), then c is too low; it must be higher to make up for bounding reducing the speed. If bounding the speed will only raise the speed (as when the end of the schedule need not be bounded but the start of the schedule must be bounded), then c is too high; it must be lower to make up for bounding raising the speed. If bounding the speed sometimes lowers and sometimes raises the speed, then we will have to actually compute the amount of time it takes to perform the schedule to determine if c is too high or too low.

6. If c is too low, then we can fix the speed at $s_{\min}$ for all those intervals at the beginning of the schedule that with the current value of c already must be bounded up to $s_{\min}$. We can then recompute c by dividing the total time for the unfixed intervals by the deadline reduced by the time for the fixed intervals. We then return to step 5.

7. If c is too high, then we can fix the speed at $s_{\max}$ for all those intervals at the beginning of the schedule that with the current value of c already must be bounded down to $s_{\max}$. We can then recompute c by dividing the total time for the unfixed intervals by the deadline reduced by the time for the fixed intervals. We then return to step 5.

8. Once we reach the point when c is just right, we can compute the speed for each interval as $cH_i^{-1/3}$ bounded to between $s_{\min}$ and $s_{\max}$. By dividing the cycles of work for each interval by its speed, we can determine how long each interval takes.

Note that there is a simple limit to the number of times we loop back to step 5: each time we loop back we fix at least one interval to be either $s_{\min}$ or $s_{\max}$, so we can never loop more times than there are intervals. In practice, we expect to loop far fewer times than this, on average.

IV.4.5 Accounting for transition times

On real systems capable of dynamic voltage scaling, making a transition between two speed and voltage levels takes a nonzero amount of time and energy. According to Burd et al. [BB00], the transition time and energy is roughly proportional to the voltage differential between the two levels. Therefore, if voltage levels V_1, V_2, and V_3 satisfy $V_1 < V_2 < V_3$, it should consume roughly the same amount of time and energy to transition from V_1 to V_2 and then from V_2 to V_3 as it does to transition directly from V_1 to V_3. Indeed, on Transmeta CPU's, the CPU accomplishes multilevel voltage switches by passing through each intermediate voltage level [Ham01]. Therefore, if $\mathsf{TransitionTime}_{a,b}$ and $\mathsf{TransitionEnergy}_{a,b}$ represent the time and energy required by a transition from speed a to speed b, we can consider the time and energy a schedule spends on transitions to be

$$\mathsf{TransitionTime}_{s_{\min},s(W)} + \mathsf{TransitionTime}_{s(W),s_{\min}}$$

and

$$\mathsf{TransitionEnergy}_{s_{\min},s(W)} + \mathsf{TransitionEnergy}_{s(W),s_{\min}},$$

respectively.

To take the transition time into account, one should consider the deadline to be effectively reduced by the transition time between $s_{\min}$ and the speed at the end of the pre-deadline part of the schedule. In this way, the deadline can still be made despite the time spent on transitions. Taking the energy into account in the choice of an optimal schedule is a difficult problem, and not likely one worth solving considering the small magnitude of such energy consumption.

IV.5 Choosing a Base Algorithm

When PACE modifies an algorithm, it leaves two aspects of that base algorithm intact: what PDC it uses for each task, and what post-deadline schedule it uses for each task. Thus, different base algorithms will still have different performance even after both are improved with PACE. In this section, we discuss how to choose among base algorithms. We also discuss how to efficiently implement the parts of those base algorithms that PACE does not modify.

IV.5.1 Choosing a post-deadline part

First we consider what the base algorithm for post-deadline scheduling should be. Unlike in the previous section, we will not be creating a performance equivalent algorithm, so we will need a performance metric. Since the post-deadline part has no influence on the fraction of deadlines made, we use the average delay. Our goal is to create an algorithm that consumes the least possible energy for a given average delay.

Let $\mathsf{TotalExcess}$ be the total excess of all tasks in the workload. Note that this is determined by the pre-deadline part; we cannot change it in the post-deadline part. To achieve a certain average delay $\mathsf{AvgDelay}$, we have to perform these $\mathsf{TotalExcess}$ cycles in total time equal to $n \cdot \mathsf{AvgDelay}$ where n is the number of tasks. As Weiser et al. [WWDS94] point out, the way to do this with minimum energy is to run at constant speed equal to $\mathsf{TotalExcess}/(n{\cdot}\mathsf{AvgDelay})$. Another way to look at this is that if we use a fixed, constant speed after the deadline, we are assured that the energy consumption we achieve is the minimum possible for the achieved average delay. Therefore, we propose that a scheduling algorithm pick a fixed speed to use for all its post-deadline schedules. Many existing algorithms already do this, either because they always use a fixed speed or because they increase speed as average recent utilization increases and thus achieve the maximum CPU speed by the time a task reaches its deadline.

A complication is that, usually, other components like the backlight will be running and consuming power, and delay past the deadline will generally cause these components to consume more energy. If other components have total power consumption P_{other}, then running at speed s in the post-deadline part yields system power consumption $\mathsf{P}(s) + P_{\text{other}}$.

Note, however, that this quantity has the same second derivative with respect to s as $\mathsf{P}(s)$. Therefore, it is also concave-up, so it still holds that the way to run for a given total amount of time with minimal energy is to use a constant speed.

It may seem odd that the optimal pre-deadline schedule varies speed as a function of time but the optimal post-deadline schedule uses a constant speed. The reason for this is that our goal for a pre-deadline schedule is different from our goal for a post-deadline schedule. For a pre-deadline schedule, we attempt to achieve a reasonable effective completion time for each task. Thus, it does not matter how long each task takes as long as it completes by its deadline. For a post-deadline schedule, we attempt to achieve a reasonable total delay for all tasks. Thus, all the delay incurred by all tasks is relevant and considered. For a pre-deadline schedule, time taken beyond the deadline is more important than time taken before the deadline, so as we approach the deadline the importance of high speed increases; for a post-deadline schedule, all time is equally important, so there is never a need for higher or lower speed.

We must now determine what fixed speed to use in the post-deadline part. We believe that it is best to always use the maximum available speed, for two reasons. First, as stated before, running at lower speed to save processor energy comes at the cost of higher energy consumption for other components. Second, the user purchased the computer with the given maximum speed to achieve that level of performance. When running past the deadline, the user presumably notices differences in performance from running at different speeds, and therefore generally desires the maximum available speed. In summary, the maximum available speed is appropriate because it minimizes delay, generally at some energy cost, but not at substantial energy cost considering that other components' power consumption would mitigate the effect of lower speeds.

It may be that some situations warrant a lower fixed speed in the post-deadline part than the maximum available speed. As evidence, consider the fact that laptop computers using Intel's SpeedStep$^{\text{TM}}$ technology permanently lower the speed when the user is running off of battery power. This suggests that in limited-energy scenarios, the user may be content with a lower maximum speed than the theoretical maximum speed of the processor. However, even in this case, there is a minimum speed below which it is pointless to run in the post-deadline part. If the total power consumption of other components besides the CPU is P_{other}, then energy consumption per cycle at speed s is $\mathsf{E}(s) + P_{\text{other}}/s$. This quantity can have a negative derivative for some values s, indicating that one can get better energy consumption *and* better performance by increasing the speed. Therefore, there is no reason to use a speed for which the derivative of $\mathsf{E}(s) + P_{\text{other}}/s$ is negative.

The total energy consumption per cycle at speed s is given by

$$\mathsf{E}_{\text{sys}}(s) = \frac{\mathsf{P}(s) + P_{\text{other}}}{s}.$$

Now,

$$\begin{aligned}\mathsf{E}'_{\text{sys}}(s) &= s^{-1}[\mathsf{P}'(s)] - s^{-2}[\mathsf{P}(s) + P_{\text{other}}] \\ &= \frac{s\mathsf{P}'(s) - \mathsf{P}(s) - P_{\text{other}}}{s^2} \\ &= \frac{Q(s) - P_{\text{other}}}{s^2}\end{aligned}$$

where $Q(s) = s\mathsf{P}'(s) - \mathsf{P}(s)$. Now, $Q'(s) = s\mathsf{P}''(s) + \mathsf{P}'(s) - \mathsf{P}'(s) = s\mathsf{P}''(s)$, so $Q'(s) > 0$ for all $s \in [s_{\min}, s_{\max}]$. Thus, $Q(s)$ is continuous and strictly increasing over this range. So, exactly one of the following holds:

- $Q(s) < P_{\text{other}}$ for all $s \in [s_{\min}, s_{\max}]$.
- $Q(s) > P_{\text{other}}$ for all $s \in [s_{\min}, s_{\max}]$.
- $Q(s)$ attains both a value no less than P_{other} and a value no greater than P_{other} for $s \in [s_{\min}, s_{\max}]$. Since it's continuous, it must attain the value P_{other} for at least one such s. Since it's strictly increasing, it cannot attain the value P_{other} for more than one such s. So, $Q(s) = P_{\text{other}}$ for exactly one $s \in [s_{\min}, s_{\max}]$.

Thus, exactly one of the following must hold:

- $\mathsf{E}'_{\text{sys}}(s) < 0$ for all $s \in [s_{\min}, s_{\max}]$, so E_{sys} is minimized at $s = s_{\max}$.
- $\mathsf{E}'_{\text{sys}}(s) > 0$ for all $s \in [s_{\min}, s_{\max}]$, so E_{sys} is minimized at $s = s_{\min}$.
- There is exactly one point $s \in [s_{\min}, s_{\max}]$ for which $\mathsf{E}'_{\text{sys}}(s) = 0$; thus, E_{sys} is minimized there. This is the point that satisfies $s\mathsf{P}'(s) - \mathsf{P}(s) = P_{\text{other}}$.

We therefore conclude that the post-deadline schedule speed that minimizes system energy consumption is that unique value $s \in [s_{\min}, s_{\max}]$ at which $s\mathsf{P}'(s) - \mathsf{P}(s)$ is closest to P_{other}. Speeds lower than this value are pointless to use in a post-deadline schedule since they consume more energy and have worse performance. Speeds higher than this value can be worthwhile as they have better performance to go along with using more energy than the minimum.

As a special case, note that if CPU power consumption is proportional to the speed cubed, i.e., if $\mathsf{P}(s) = as^3$, then $\mathsf{P}'(s) = 3as^2$, and the minimum energy occurs at that speed at which $s\mathsf{P}'(s) - \mathsf{P}(s) = 2as^3 = 2\mathsf{P}(s)$ is closest to P_{other}. So, we would like to run the CPU at a speed at which its power will be at least half that of the other power-consuming components. In other words, we want the CPU to consume at least one-third the total system power, since at lower relative power consumption both performance and energy savings are worse. Note that this rule of thumb only applies when CPU power consumption is proportional to the cube of the speed.

Another approach is to choose a target average delay, predict the average excess, and choose a speed that is the ratio of these two. However, in practice, we have found this approach to be impractical, since two factors make predicting average excess difficult. First, excess should be nonzero only rarely, since an algorithm will attempt to complete most tasks by the deadline, so samples of excess will tend to be small until many tasks have occurred. Second, the distribution of excess depends strongly on the tail of the task work distribution, and such tails tend to be hard to model.

IV.5.2 Choosing PDC for each task

Here we consider how to optimally compute PDC. In other words, given some target fraction of deadlines to make, TFDM, we would like to compute the optimal PDC for each task. This constraint is interesting because it is a single constraint on all tasks in the workload rather than one constraint per task. Therefore, we have great freedom in choosing the PDC values; for instance, we might decrease the PDC of one task, thereby increasing its probability of missing its deadline, and make up for that by increasing the PDC of another task, thereby decreasing its probability of missing its deadline.

We can describe the optimization problem mathematically as follows. Suppose there are n tasks. If we denote the distribution of task i by F_i, we want to choose a PDC_i for each task i to minimize the expected energy consumption, which is proportional to

$$\sum_{i=0}^{n}\left[\int_{0}^{\mathsf{PDC}_i} F_i^c(w)\mathsf{E}(\mathsf{S}[F^c(w)/K_{F_i,\mathsf{PDC}_i}])\,dw \;+\int_{\mathsf{PDC}_i}^{\infty} F_i^c(w)s_{\max}^2\,dw\right],$$

subject to the constraint that we must expect to make a fraction TFDM of the deadlines:

$$\sum_{i=0}^{n} F_i(\mathsf{PDC}_i) = n\cdot\mathsf{TFDM}.$$

Here, we use K_{F_i,PDC_i} to denote the PACE scaling factor K that is dependent on F_i and on PDC_i.

Unfortunately, we cannot solve this optimization problem, for two reasons. First, the complex dependence of K on PDC_i makes optimizing this quantity intractable. Second, even if we had an analytical solution, it would still depend on all of the work distributions simultaneously. Therefore, we would need to plug in a model of the distribution of distributions, i.e., a model of the nonstationarity of the work distribution, and we know of no reasonable way to model this.

These problems also make it impossible to analytically determine how well a given algorithm for computing PDC will work from a standpoint of energy consumption versus performance. Therefore, we must rely on empirical, rather than analytic, methods to compare such algorithms. We consider several methods for computing PDC, and present results comparing them in Section IV.7.5.3. Most of these algorithms are simply previously published DVS algorithms; in other words, we compute the PDC for each task by computing the PDC of the schedule that the previously published algorithm would generate.

One interesting distinction between these existing algorithms is that for some, such as Flat/Chan-style, PDC is independent of the current task work distribution, while for others, such as LongShort/Chan-style, PDC it is not. (Flat/Chan-style uses a constant speed, so its PDC is the same for all tasks regardless of the current work distribution: PDC is always the constant speed times the deadline. LongShort/Chan-style uses a speed proportional to recent utilization, so its PDC is higher when recent tasks have been long.) The former type will tend to miss the deadlines of the longest tasks in the workload, so we call them *global*. The latter type will tend to miss the deadlines of tasks whose work requirements are local maxima, so we call these *local* algorithms. When the distribution is nonstationary, as frequently occurs, global approaches will tend to miss a different set of tasks' deadlines than local ones. Our model, unfortunately, does not allow us to analytically determine whether global approaches have better energy consumption than local ones, or even whether one global approach has better energy consumption than another. Therefore, we rely on empirical data to compare them.

It might seem that if the distribution were stationary, the best algorithm would be to keep PDC constant. However, although this is true for many distributions, there are some distributions for which this does not hold. For example, suppose a certain type of task usually takes a short amount of time but on rare occasions takes much longer: its work distribution has an 0.96 quantile of 10 Mc, an 0.97 quantile of 24 Mc, and an 0.98 quantile of 25 Mc. If we want to make 97% of deadlines, we could use a constant PDC of 24 Mc. However, a better approach in this case is to use 10 Mc half the time and 25 Mc the other half; this makes the same number of deadlines, but allows the CPU to run much more slowly half the time.

Because we do not know how to determine an optimal PDC for each task, we must use heuristics. Generally, for those heuristics, we use various previously published algorithms. We determine, for each task, what schedule such an algorithm would use for it, compute what the PDC of this schedule is, and use that PDC for that task.

A problem with using a previously published algorithm to choose PDC values in this way is that it does not give predictable performance, i.e., there is no way to choose parameters to make a given fraction of deadlines. To solve this, we have developed the following new algorithm for computing PDC. Suppose the target fraction of deadlines we want to make is TFDM. We then always set PDC to be the TFDM-th quantile of the task work distribution. This way, we expect to make each deadline with probability TFDM. Normally we will actually want to achieve some target fraction of *possible* deadlines made TFPDM, so we instead set PDC to be the $[\mathsf{TFPDM} \cdot F(s_{\max}D)]$-th quantile of the distribution. Note that this algorithm bases its choice on the current task distribution, and is thus a local algorithm.

IV.5.3 Computing PDC efficiently

We have shown how to improve an existing scheduling algorithm by changing its pre-deadline and post-deadline schedules. Thus, the only remaining influence the existing scheduling algorithm has on the final schedule is its choice of pre-deadline cycles (PDC). However, considering that this is now the only function of the existing scheduling algorithm, it is probably wasteful to simulate the entire algorithm just to figure out the PDC for each task. So, in this section, we consider more efficient algorithms to compute the PDC for each task.

In general, Past/Peg begins a task running at the minimum speed $s_{\min}$, then 10 ms later pegs the speed at the maximum $s_{\max}$. So, the pre-deadline cycles for this algorithm are $s_{\min}(10\text{ ms}) + s_{\max}(D - 10\text{ ms})$. An important special case is when the previous task took so long that the utilization of its last interval was more than 70%. In this case, Past/Peg still has the speed pegged at maximum when the current task begins, so it winds up running the task at the maximum speed the whole time; thus, the pre-deadline cycles are $s_{\max}D$. So, a reasonable approximation to Past/Peg is to set

$$\mathsf{PDC} = \begin{cases} s_{\min}(10\text{ ms})+ & \text{if } W_{\text{prev}} \le s_{\min}(10\text{ ms})+ \\ \quad s_{\max}(D - 10\text{ ms}) & \quad s_{\max}(D - 20\text{ ms})+ \\ & \quad 0.7 s_{\max}(10\text{ ms}) \\ s_{\max}D & \text{otherwise,} \end{cases}$$

where W_{prev} is the previous task's work requirement. We can extend this derivation to interval lengths other than 10 ms in the obvious manner.

Flat-0.6/Chan-style always runs each task at speed $0.6s_{\max}$, so its pre-deadline cycles are always $0.6s_{\max}D$. We can easily extend this derivation to versions of Flat that assume a utilization other than 0.6.

LongShort/Chan-style begins a task by running fast enough to complete its expected work by the deadline. Because this work estimate is weighted heavily toward recent values, it is usually approximately equal to the last task's work requirement, W_{prev}. As the task progresses, utilization goes up, and the algorithm ramps up speed toward the maximum $s_{\max}$.

If the initial speed were used for the entire task, the pre-deadline cycles would obviously be W_{prev}; if the latter speed were used for the entire task, the pre-deadline cycles would be $s_{\max}D$. Depending on the rate at which speed is ramped up, which depends on the interval length used, the pre-deadline cycles actually achieved is somewhere between W_{prev} and $s_{\max}D$. We have found that we can reasonably approximate the pre-deadline cycles by using a weighted average of W_{prev} and $s_{\max}D$; the appropriate weighting varies with interval length. We will show that for an interval length of 10 ms, a reasonable approximation to PDC is $0.55W_{\text{prev}} + 0.45s_{\max}D$.

IV.6 Workloads

We evaluate these algorithms using six workloads. We derived several of these workloads from Windows NT and Windows 2000 traces obtained using VTrace, the tracer described in Appendix A. This tracer collects timestamped records describing events related to processes, threads, messages, disk operations, network operations, the keyboard, the mouse, and the cursor. From this data, we deduce how much work is done due to a user interface event as follows: we assume that a thread is working on such an event from the time it receives the message describing that event until the time it either performs a wait for a new event or requests and receives a message describing a different event. Furthermore, if the thread sends a message to another thread while working on such an event, we assume that any work done due to that sent message is actually done due to the original event. (For example, while processing a key-down message, a thread may post a character-pressed message to its message queue. Obviously, any work done to process the character-pressed message is really work done to process the key-down message.) Similarly, if a thread working on an event signals another thread, we assume that the signaled thread begins working on the same event.

To reduce the amount of data VTrace must collect and upload, it only collects the full set of events it can for sessions lasting 90 minutes at a time, after which it pauses for 2 hours. So, when we discuss a trace longer than 90 minutes, we mean only the 40% of the time that VTrace actually traced its full set of events. We have collected data on several machines for a couple of years. However, our analyses for this chapter do not require huge workloads, so we limit the workloads to a few months each. This way, we can efficiently analyze the data all at once using a reasonable amount of disk space.

Each workload is defined by a class of events, such as letter keypresses in Microsoft Word. The workload consists of the series of tasks triggered by all such events that occurred during the tracing period. Each such task is described by the amount of CPU processing in the task triggered by that event. In other words, each task of each workload is roughly of the same type; by separating different task types into different workloads, we model the effect of keeping separate samples for different task types, as described in Section IV.4.2. A full machine workload would consist of many of these kinds of workloads, interleaved. Since our approach operates independently on each different task type, we can correctly simulate it by considering each task type in isolation. Since task work distributions should be different for different task types, the approach of separating the task types and estimating work requirements based on task type should yield higher energy savings than using the combined work distribution.

We discard any task that blocked on any I/O, e.g., to a disk or network device. We do this because when a task blocks for I/O, it should use a different algorithm that takes I/O time into account, and such algorithms are beyond the scope of this chapter. Also, I/O generally occurs in only a small fraction of the tasks, so leaving them out should not significantly influence the results.

For our simulations, we assume the minimum speed 100 MHz, the maximum speed is 500 MHz. This maximum CPU speed is comparable to the (various different) speeds of the systems traced. We assume that power consumption is proportional to the cube of the speed, with peak power consumption of 3 W at 500 MHz.

IV.6.1 Word processor typing

One of the most common activities for laptop users is typing in a word processor, and Microsoft Word is the most common word processor. Therefore, our first workload uses simple letter keystrokes in Microsoft Word as its class of events. We derived this workload from traces. VTrace collected these traces on a 450 MHz Pentium III computer with 128 MB of memory running Windows NT used by the author of this dissertation, a computer science graduate student. The workload is from 3.4 months of traces. This workload is interactive, so we use a 50 ms deadline for each task.

IV.6.2 Groupware

Groupware, i.e., software that enables and enhances communication with others, is already an important application on the desktop, and will become more important in portable computers as users demand greater wireless networking functionality. Thus, we include a workload using a common groupware product, Novell's GroupWise. This workload uses releases of the left mouse button as its class of events. We derived this workload from traces. VTrace collected these traces on a 350 MHz Pentium II with 64 MB of memory running Windows NT 4.0 used by a crime laboratory director in the Michigan State Police. The workload is from 6.5 months of traces. This workload is interactive, so we use a 50 ms deadline for each task.

IV.6.3 Spreadsheet

Spreadsheets are another common application on portable computers. So, our next workload uses a common spreadsheet application, Microsoft Excel. The workload uses releases of the left mouse button as its class of events. We derived this workload from traces. VTrace collected these traces on a 500 MHz Pentium III with 96 MB of memory running Windows NT 4.0 and used by the chief technical officer of a computing-related company. The workload is from 3 months of traces, during which the user used Excel many times. This workload is interactive, so we use a 50 ms deadline for each task.

IV.6.4 Video playback

Multimedia applications are becoming more common on portable computers [FS99]. Therefore, we include a movie player as one of our workloads. We use the MPEG player included with the Berkeley MPEG Tools developed by the Berkeley Multimedia Research Center (BMRC) [PSR93]. Since they provide full source code for their tool, we were easily able to instrument it to measure and output the CPU time taken for each frame. Thus, each task of the workload represents the processing of one frame.

Title	Description	Frames
Genoa	Demo of Genoa products	2,592
Jet	Flying in varying terrain	1,085
Earth	Rotating Earth model	720
Red's Nightmare	Bicycle's nightmare	1,210
Dünne Gitter	Illustrations of integration	2,753
IICM	Flying in Mandelbrot set	810
Gromit	Gromit wakes up	331

Table IV.3: Animations used in the MPEG workloads

The animations we use are all works taken from the BMRC FTP site where we got the MPEG decoder. Table IV.3 gives names and descriptions for these videos. One workload, which we call MPEG-One, consists only of the Red's Nightmare animation. The other workload, which we call MPEG-Many, consists of all seven video clips, one played after the other. We made the measurements of CPU time on a 450 MHz Pentium III with 128 MB of memory running RedHat Linux 6.1. Assuming a typical rate of 25 frames per second, we assign a deadline of 40 ms to each frame.

IV.6.5 Low-level workload

A hardware manufacturer, such as Transmeta or its partners, may want to implement a scheduling algorithm entirely in hardware without instrumenting the operating system. Such an algorithm could only use information it could collect at the hardware level. So, we devised a workload representing such a scenario.

We derive this workload from VTrace, but in a different manner than the others. We define a task as the time between a keypress and the next time either the CPU becomes idle or there is another keypress. For the keypress time, we use the actual time the keyboard device was invoked, not the time the keypress message was delivered to an application. To determine when the CPU is idle, we use the time that the idle thread is running. (In battery-powered systems, the idle thread typically halts the CPU, so hardware can deduce when this thread is active.) If any disk operations are ongoing when the thread goes idle, we throw out any keypress currently being worked on, for two reasons. First, we do not know if the I/O is part of the processing of the keypress, so we cannot determine whether the processing is over or simply waiting for I/O. Second, as stated before, we are not including tasks that perform I/O in our workloads since we are only considering algorithms for dealing with tasks that perform no I/O.

The workload comes from a trace of one 90-minute session, chosen because it had a lot of keypresses that took a reasonably long period of time, on average. VTrace collected this trace on a 400 MHz Pentium II with 128 MB of memory running Windows NT 4.0 used by a Michigan State Police captain primarily for groupware and office suite applications. This workload is interactive, so we use a 50 ms deadline for each task.

IV.6.6 Statistical characteristics

Table IV.4 lists the statistical characteristics of the six workloads' task work requirements. Figure IV.4 shows the overall cumulative distribution function for each workload's task work requirements. The number of events in some workloads may appear small considering the trace is several months long. This is due to two factors: the user typically did not use each application every day during those months, and the workload only consists of a particular event type which may occur infrequently during normal application use. The standard deviations are moderate because all tasks in each workload are the same general type.

IV.7 Results

In this section, we present results of experiments that illustrate how effective various techniques for dynamic voltage scaling are. First, we consider how PACE can best model distributions of task work requirements; this involves choosing a sampling method and a distribution estimation method. Next, we consider how PACE should choose its transition points when approximating the optimal schedule with a piecewise constant one. We also examine how effective our approximations are at producing a practical schedule that consumes almost as little energy as an optimal schedule would produce. Next, we examine how effective PACE is at reducing the energy consumption of existing algorithms. Then we

Workload	Word	Excel	GroupWise
Count	6849	400	7314
Mean	5615	2381	2149
Sample standard deviation	3046	7312	5154
Coefficient of variation	0.54257	3.0711	2.3984
Coefficient of skewness	2.3134	6.6743	13.649
Second moment	40806685	59007956	31182242
Second central moment	9279558	53338595	26563561
Third moment	398742280797	3004266775542	2050211083196
Third central moment	65409116062	2609763481916	1869020832810
Workload	**Low-Level**	**MPEG-Many**	**MPEG-One**
Count	2930	9112	1164
Mean	1869	6111	8869
Sample standard deviation	4110	3921	1849
Coefficient of variation	2.1994	0.64166	0.20844
Coefficient of skewness	12.882	0.58112	4.3209
Second moment	20380636	52722803	82064984
Second central moment	16888263	15375359	3414207
Third moment	995703913629	545168913082	815647723947
Third central moment	894495597345	35040888975	27294202425

Table IV.4: This table shows the statistical characteristics of the workloads' task work requirements. These work requirements are in Kc. The coefficient of skewness is the third central moment divided by the cube of the sample standard deviation.

evaluate various techniques for computing pre-deadline cycles. Finally, we examine the time and energy overhead involved in implementing PACE.

IV.7.1 Modeling task work distributions

In this section, we determine how best to practically estimate the probability distribution of task work requirements. We want to know which methods are general enough to work well for a variety of workloads, so we evaluate them all using simulations with our six different workloads. To determine how effective a method is at describing the distribution of task work requirements, we use a pragmatic approach: we use the method to implement PACE and simulate how much energy consumption results. We consider a method better if it produces lower pre-deadline energy consumption. No other metric is relevant, since PACE

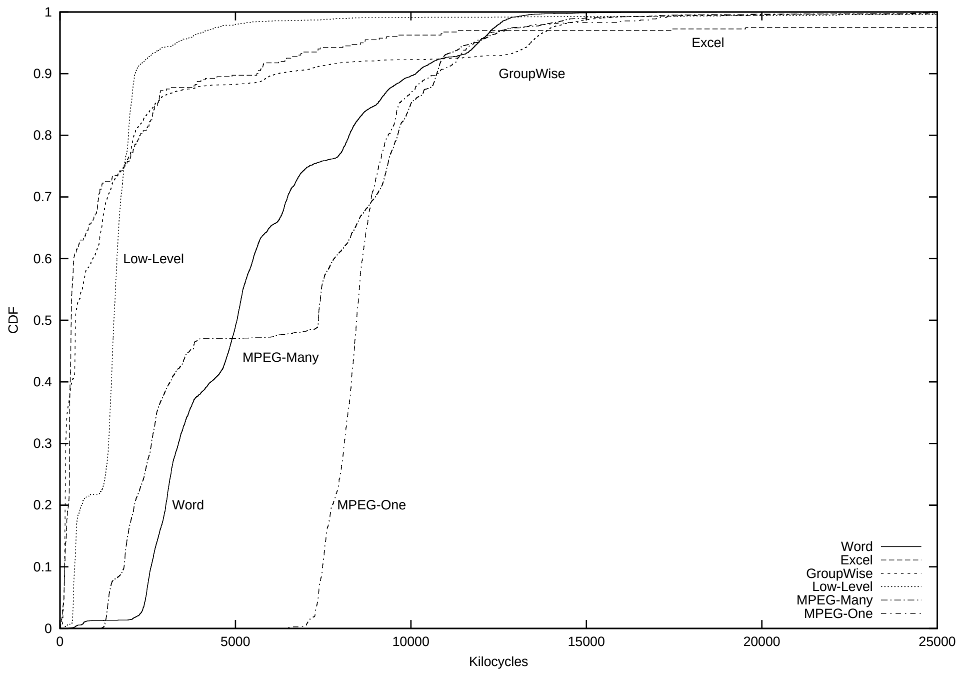

Figure IV.4: Overall CDF's of each workload's task work requirements

does not change any other metric.

Throughout this section, for our simulations, we assume the PDC is fixed at the value that assures at least 98% of possible deadlines get made. (For the Low-Level workload, we use 99%, because 98% of the tasks are so short that their deadlines can be achieved with just the minimum speed.)

IV.7.1.1 Which sampling method to use

We now compare the various sampling methods to determine the best ones to use with PACE. For the purpose of this comparison, we assume that we use kernel density estimation to estimate the distribution.

First, we consider what sample size to use for the sampling methods that use only recent data, Recent-k and LongShort-k. Figures IV.5 and IV.6 show the outcome of using different sample sizes for different workloads. It is difficult to pick an "ideal" sample size,

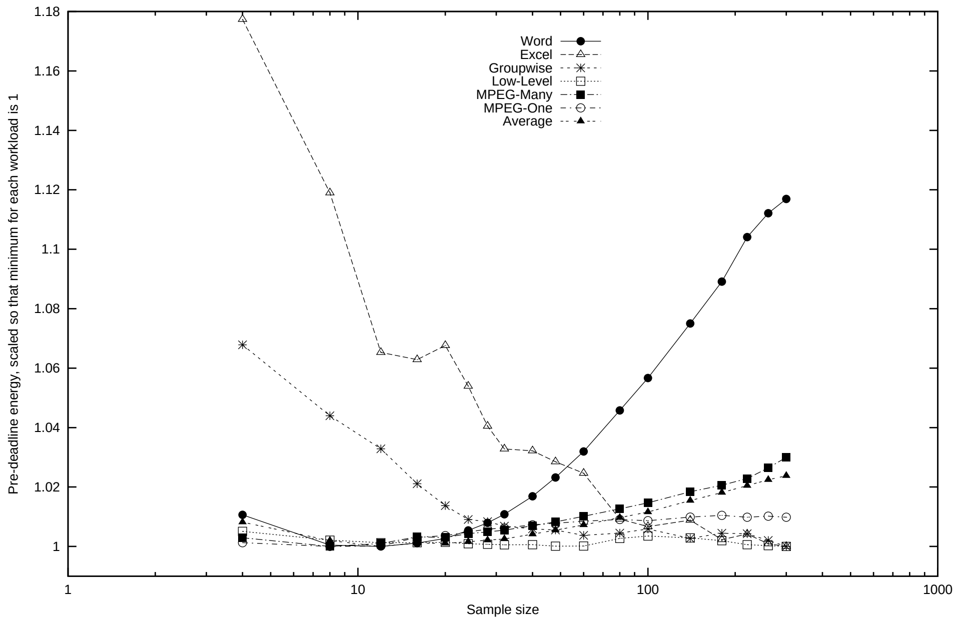

Figure IV.5: A comparison of the effect of various sample sizes k on energy consumption when PACE uses the Recent-k sampling method

because some workloads do best with high sample sizes, while others do best with low sample sizes. Presumably, the ones that do better with high sample sizes are the ones with more stationary distributions, i.e., the ones whose distributions change the least with time. Since higher sample sizes require more memory and, for some distribution estimation methods, more processing time, we feel a reasonable compromise is a sample size of 28. For all workloads except Excel, this sample size produces energy consumption within 0.8% of the ideal for Recent-k and within 1.6% of the ideal for LongShort-k. Excel, presumably because it is highly stationary, can take advantage of higher sample sizes, but these sample sizes produce worse results in most of the other, less stationary workloads.

We now consider what aging factor a to use for the Aged-a sampling method. Figure IV.7 shows the outcome of using different aging factors for different workloads. Just as with sample sizes, some workloads do best with high aging factors while others do best with low aging factors. Not surprisingly, the workloads that do best with high aging factors are the same ones that do best with high sample sizes. This is expected, because a high aging

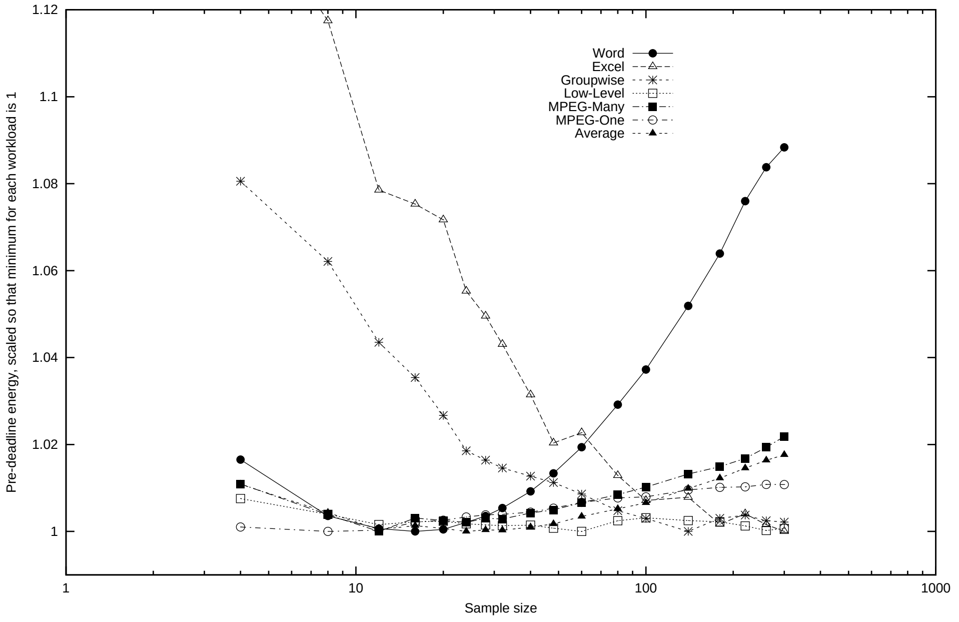

Figure IV.6: A comparison of the effect of various sample sizes k on energy consumption when PACE uses the LongShort-k sampling method

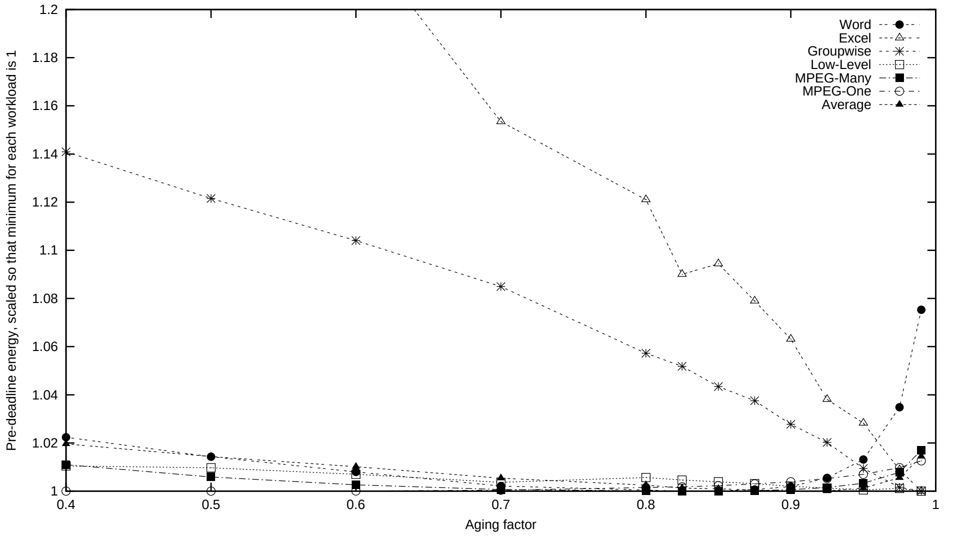

Figure IV.7: A comparison of the effect of various aging factors a on energy consumption when PACE uses the Aged-a sampling method

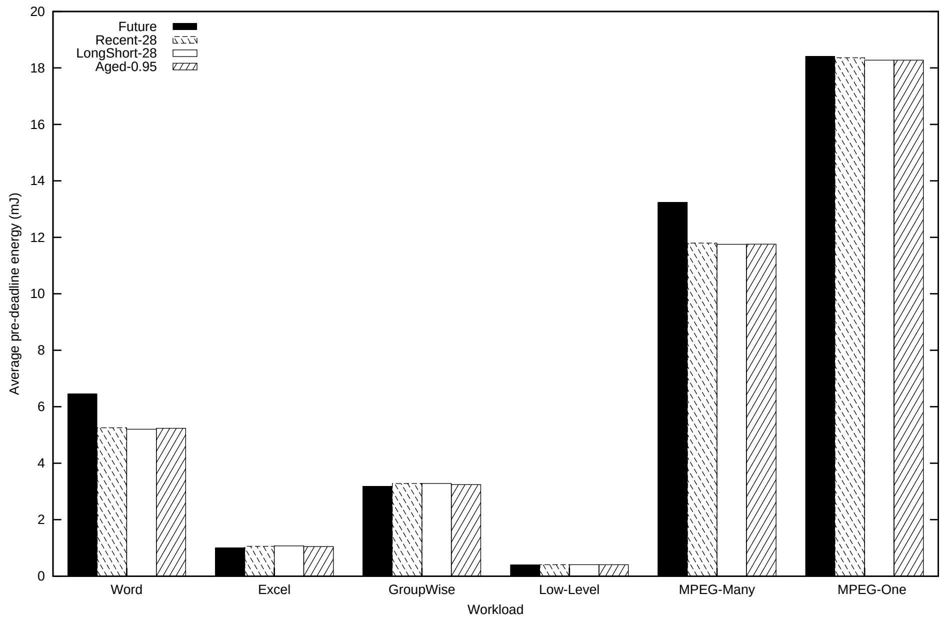

Figure IV.8: A comparison of the effect of various PACE sampling methods on energy consumption

factor makes sample values age more slowly, so that PACE effectively uses more old values. We feel a reasonable aging factor is 0.95. With this aging factor, each workload besides Excel has energy within 1.3% of what it would be using the best aging factor for that workload, and Excel has energy within 2.8% of the best possible.

Next, we determine which sampling method works best. Figure IV.8 compares the Future, Recent-28, LongShort-28, and Aged-0.95 sampling methods for the six workloads. We do not evaluate the All sampling method, since it is equivalent to Recent-∞, and we have already shown that the Recent method works quite badly for some workloads with large sample sizes. The first thing we observe is that the Future method sometimes uses less energy than the other methods, but sometimes uses more. This indicates that even if complete information about the full distribution of task work is available, it can still be better to use recent information to predict the distribution of the next task work. Presumably, this is because these distributions are non-stationary, so recent information is a better predictor than global information. This is fortunate, since the Future method requires knowledge of the future and will thus usually be impossible to implement. The next thing to observe is that the remaining three methods have virtually identical energy consumption. Generally, Recent-28 consumes the most, LongShort-28 the next most, and Aged-0.95 the least. However, the difference between any two is never more than 2.2%. We conclude that, all things being equal, it is better to use the Aged-0.95 sampling method. However, if one of the three methods is substantially easier to implement, one should consider using the simplest implementation, since this sacrifices little energy.

In conclusion, many workloads change their characteristics with time, so using all past information often leads to worse predictions than using only recent information. Unfortunately, workloads change at different rates, so it is difficult to decide how to weight old information. A reasonable compromise seems to be to only use about 28 recent values, or to use all values but reduce the weight of each value by a factor of 0.95 each time a new one arrives. Of all the methods we propose, aging seems to produce the best results, but other methods work reasonably and may be good choices if they are easier to implement.

IV.7.1.2 Which distribution estimation method to use

The kernel density estimation method can model any kind of distribution, but it is complex to implement. So, we now investigate how effectively we can model task work distribution with simpler parametric models, the normal and the gamma distribution.

To test whether a model truly fits a set of data, one can use that model to estimate the CDF at each data point, and test whether the set of CDF's is distributed uniformly over the interval $(0, 1)$. For this uniformity test, Rayner and Best [RB89] recommend Neyman's Ψ_4^2 test. Applying this test to any of our workloads, using any of our sampling methods, the test reveals an extremely small probability that the data fit either the normal or gamma model. Fortunately, the key issue is not the accuracy with which we can approximate the distribution of the task work. The key issue is the extent to which a statistically unacceptable model of this distribution produces a suboptimal solution to the energy minimization problem.[1]

[1] When we are estimating the distribution to approximate the optimal bandwidth for use in kernel density estimation, we are more concerned with the lack of applicability of the model. Fortunately, in this case, the only consequence of an improper result is an inappropriate level of smoothing, and not a systematic bias in the CDF estimate.

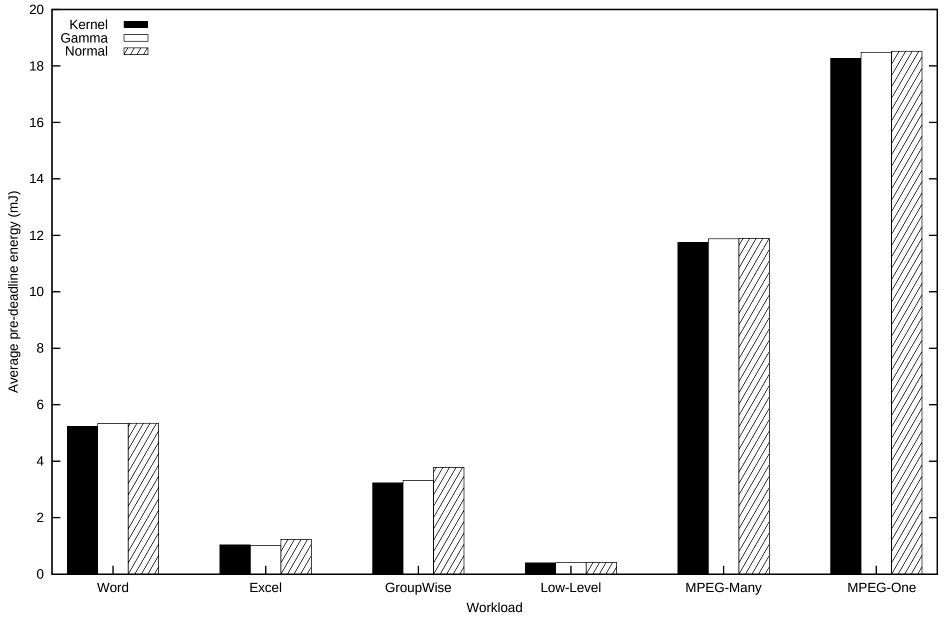

Figure IV.9: A comparison of the effect of various PACE distribution estimation methods on energy consumption

Therefore, the more important question to ask is how effectively PACE can use each model to approximate the optimal schedule. We thus simulate using each model along with the Aged-0.95 sampling method for each workload. We use the same PDC values that we did in the last section. Figure IV.9 shows the results of these simulations. For almost all workloads, the kernel density model is best, followed by the gamma model, followed by the normal model. In all cases, the gamma model consumes no more than 2.3% more energy than the kernel density model. We conclude that, all things being equal, one should use the kernel density estimation method. However, if this method is too complex to implement, the gamma model can achieve respectably similar results.

In conclusion, no workload truly fits the gamma or normal model. However, the gamma model describes most workloads closely enough to achieve results not appreciably worse than the kernel density estimation method. This latter method is general enough to describe any workload no matter what its distribution, but may be complex to implement.

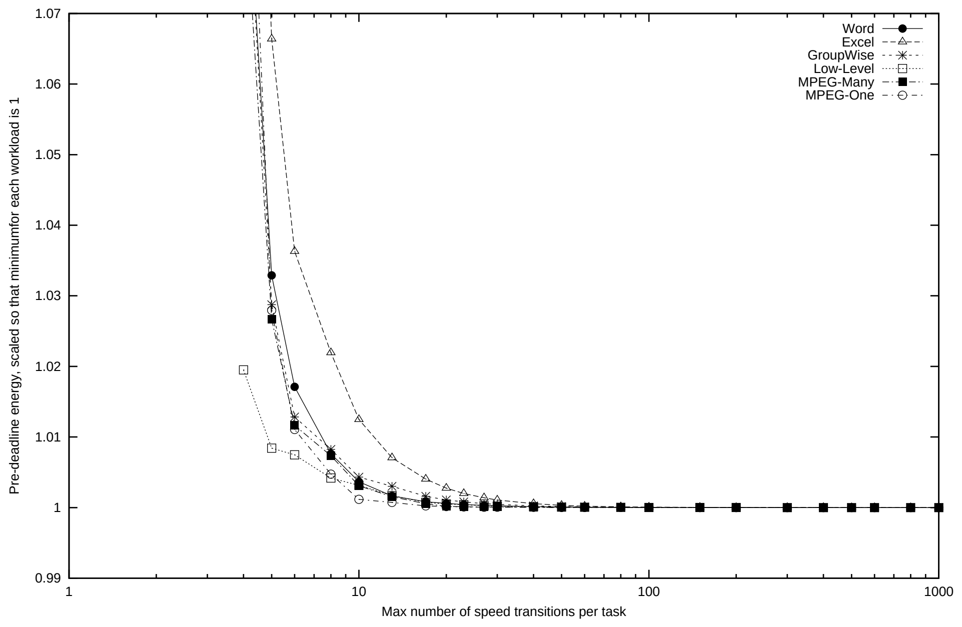

Figure IV.10: A comparison of the effect on energy consumption of using different numbers of speed transitions to approximate the continuous schedule

IV.7.2 Piecewise constant approximations to PACE's optimal formula

In Section IV.4.1, we discussed how PACE can approximate the optimal, continuous schedule using a piecewise constant schedule with a limited number of transitions. In this section, we determine empirically the best ways to choose the speed transition points for the schedule.

Figure IV.10 shows the effect of using different numbers of transitions. We see that the principle of diminishing returns applies: increasing the number of transitions becomes less and less worthwhile as the number of transitions increases. Using 10 transitions yields energy consumption always within 1.2% of the minimum. Using 20 transitions reduces the maximum penalty to 0.27%, and using 30 transitions reduces it to 0.1%. All the results in this chapter use a maximum number of transitions of 30. However, even if a practical implementation requires no more than 10 transitions be used for each task, this should not be a problem for PACE; such a practical consideration would increase energy consumption by no more than 1.2%.

We also discussed how to choose N transition points. The first step is to choose some J near N and some Q near 0.95. Simulations show that energy consumption is generally insensitive to the choices of J and Q. As long as one picks reasonably, i.e., as long as Q is somewhere between 0.85 and 0.99 and as long as $N - J$ is between 3 and 9, the difference between the best and worst outcomes for any workload is always less than 0.6%. For this chapter, we always use $J = N - 3$ and $Q = 0.95$.

IV.7.3 Theoretical effect of approximations

Although we presented a theoretical optimal formula for PACE, our focus in implementing it has been on practical approximations to this formula. To evaluate the effect of these approximations on energy consumption, it is useful to know the theoretical optimal schedule. For real workloads, this is impossible, since we cannot know the underlying distribution of each task's work requirements. (We can know the distribution of all the tasks in the workload, but as we showed, workloads can exhibit non-stationarity, so we cannot know the distribution for any given individual task.) However, we can use a synthetic workload to evaluate the effect of these approximations on energy consumption. Our synthetic workload uses task work requirements distributed according to the gamma distribution with $\alpha = 25$ and $\beta = 0.2$ Mc.

For the optimal realizable algorithm, the average task pre-deadline energy is 2.1016 mJ. When the algorithm must produce a piecewise constant speed schedule using only 30 transitions, energy goes up 0.025% to 2.1022 mJ. When the algorithm does not know the model parameters a priori and must infer them from past tasks, energy goes up 0.026% to 2.1027 mJ. When the algorithm must infer model parameters mainly from recent tasks, using the Aged-0.95 sampling method, energy goes up another 0.72% to 2.1179 mJ. Altogether, the practical requirements of using piecewise constant speed schedules and inferring distributions from limited recent information raises energy consumption by just 0.77%.

IV.7.4 Improving existing algorithms with PACE

In Section IV.3, we described how PACE can replace the pre-deadline part of an existing scheduling algorithm with a schedule that has lower expected energy consumption. In this section, we simulate this as follows. First, we simulate an existing algorithm. Then, we modify that algorithm so that it uses PACE to recompute the pre-deadline part of its schedule. Since the two algorithms are performance equivalent, we compare them solely on the basis of pre-deadline energy consumption; all other metrics are always identical.

For these simulations, we use the four standard interval-based algorithms described in Section IV.2.4, each with an interval length of 10 ms.

Figure IV.11 shows the effect of using PACE to modify these existing algorithms. We evaluate the effect of two versions of PACE, both using the Aged-0.95 sampling method: one uses the gamma model, which is easier to implement, and one uses the kernel density estimation method, which produces better results. Both versions of PACE reduce the CPU energy consumption of every workload and every existing algorithm. PACE using a gamma model reduces the CPU energy consumption of existing algorithms by 2.4–49.0% with an average reduction of 20.3%. PACE using the kernel density estimation method reduces the CPU energy consumption of existing algorithms by 1.4–49.5% with an average reduction of 20.6%. The 1.4% value is lower than the 2.4% value because Excel, the workload that gains the least benefit from PACE, happens also to be the only workload for which the gamma model sometimes outperforms the kernel density estimation method. Excel gains less benefit from PACE than other workloads because it consumes a lot of post-deadline energy, and PACE has no effect on post-deadline schedules. Interestingly, the existing algorithm most improved with PACE is Past/Peg, the one favored by the most recent comparison of existing algorithms [GLF$^+$00]. Past/Peg was favored in that work because it misses fewer deadlines than other algorithms; unfortunately, this requires higher energy consumption, as Figure IV.11 shows.

Another way to examine the results is to consider them relative to how much energy would be consumed in the absence of DVS. Without PACE, existing algorithms use DVS to reduce CPU energy consumption by 10.7–94.1% with an average of 54.3%. With PACE using a gamma model, the CPU energy savings increase to 35.9–95.5% with an average of 65.2%.

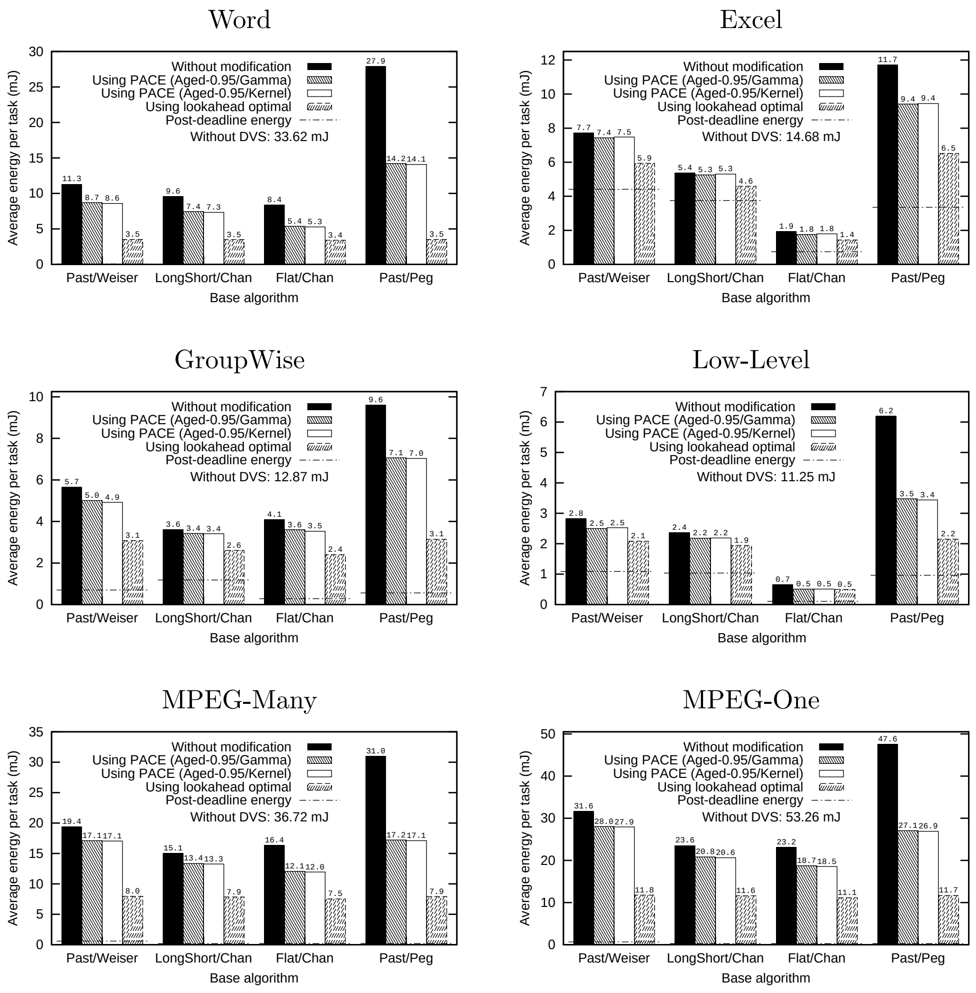

Figure IV.11: These graphs show the effect of modifying existing algorithms with PACE to get performance equivalent, but lower-energy, algorithms. Results are shown for each workload. Since PACE only modifies the pre-deadline part of the algorithm, it keeps the post-deadline energy the same; this energy is shown with horizontal lines. The Lookahead Optimal results are obtained by giving PACE perfect knowledge of the current task time; these figures are not attainable by a realizable algorithm, but serve as a lower bound on what can be attained by a performance equivalent algorithm. To give perspective on the effectiveness of dynamic voltage scaling at reducing energy consumption, each workload is labeled with the average energy consumption per task if DVS is not used at all.

With PACE using the kernel density method, the CPU energy savings increase to 35.6–95.5% with an average of 65.4%. Thus, on average, if a CPU consumes 100 J without DVS, existing DVS algorithms allow it to only consume 46 J; PACE reduces that figure even further to 35 J. Given these figures, if the CPU accounted for 33% of total energy consumption in a portable computer without DVS [LS98], existing DVS algorithms would increase its battery lifetime by about 22%; with PACE, the battery lifetime improvement would be about 28%.

Figure IV.11 also shows the results that could be obtained by the lookahead optimal strategy, a strategy that can see into the future and know exactly what the next task's work requirement is. These results are not attainable in practice on these workloads, but they provide a lower bound on the results that can be attained. We see that the PACE-modified algorithms consume significantly more energy than the lookahead optimal results, illustrating that knowledge of the distribution of task work is much less useful than knowledge of actual task work. In certain real-time environments, a system might have knowledge of actual task work, and using that data would substantially reduce energy consumption.

In conclusion, PACE is not just theoretically useful, but is a practical means to achieve substantial energy savings without affecting performance. It works on a variety of workloads, and can improve a variety of existing algorithms. The high energy savings is all the more exciting because PACE by definition has absolutely no effect on performance.

IV.7.5 Computing pre-deadline cycles

IV.7.5.1 Approximations to existing algorithms

In Section IV.5.3 we presented some efficient methods for approximating the PDC values produced by various existing algorithms. In this section, we evaluate how close those approximations are. To do this comparison, we compute PDC values using both the existing algorithm and the approximation. For each set of PDC values, we use PACE to compute a good pre-deadline schedule, and use a constant speed ($s_{\max}$) for the post-deadline schedule. We then compare the resulting metrics: fraction of possible deadlines made, average energy consumption, and average delay.

Table IV.5 shows the results of these simulations. Note that results are not shown for the method that approximates the Flat-0.6/Chan-style algorithm, because we know this ap-

Workload	Algorithm	FPDM	Total energy	AvgDelay
Word	Past/Peg	0%	0%	0%
	LongShort/Chan	0.01%	−1.39%	−4.15%
Excel	Past/Peg	0%	0%	0%
	LongShort/Chan	0.82%	0.34%	−5.03%
GroupWise	Past/Peg	0%	−0.03%	0.14%
	LongShort/Chan	−0.17%	−4.6%	−29.05%
Low-Level	Past/Peg	0%	0.16%	0%
	LongShort/Chan	0.03%	0.19%	−3.68%
MPEG-Many	Past/Peg	0.01%	0.12%	−0.84%
	LongShort/Chan	−0.02%	0.17%	2.79%
MPEG-One	Past/Peg	0%	0.25%	0%
	LongShort/Chan	−0.17%	0.21%	2.11%

Table IV.5: This table shows the resulting differences in various metrics when we approximate the PDC values of the Past/Peg and LongShort/Chan-style existing algorithms. For example, the approximate Past/Peg algorithm consumes 0.03% less energy than the actual Past/Peg algorithm on the GroupWise workload, so we report the corresponding approximation error as −0.03%.

proximation to be exact. We see that our approximations for Past/Peg and LongShort/Chan-style are almost always within 5% of the existing algorithms on all metrics, with one exception. That exception is GroupWise, where our approximation produced an average delay over 29% less than the existing algorithm. Since this produces much better energy savings and delay than the original algorithm, while only decreasing FPDM by 0.17%, this should not be a problem.

IV.7.5.2 Targetting FPDM

Another approach we suggested for setting PDC was to set it to a certain quantile of the task work distribution, to achieve a certain target fraction of possible deadlines made TFPDM. Figure IV.12 shows the result of using the Aged-0.95 sampling method and kernel distribution estimation method to estimate this quantile and use it as PDC. We see that increasing the target TFPDM increases FPDM achieved for all workloads. However, there is often substantial error between the target FPDM and the actual FPDM. There are a few reasons for this. First, the model does not work very well at describing the tail of the

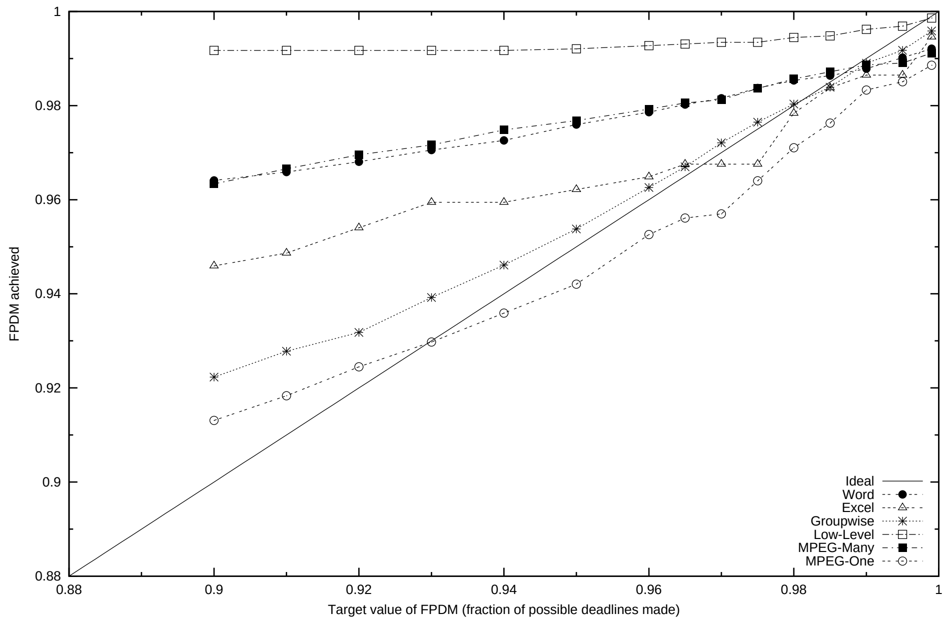

Figure IV.12: This graph shows the FPDM achieved when various target levels of FPDM_T are sought using the Aged-0.95/Kernel method.

distribution, so computed quantiles are inaccurate. Second, for some workloads, some values of FPDM are simply too low to be achievable. For example, 94.3% of tasks in Excel are shorter than $s_{\min}D$, so FPDM cannot be less than 0.943. The other example is Low-Level, for which 93.4% of tasks are shorter than $s_{\min}D$.

We conclude that we cannot use this method to achieve a certain target of fraction of possible deadlines made. At best, it can be used to roughly tune this fraction to some desired value.

IV.7.5.3 Which PDC computation method uses the least energy?

In Section IV.5.2, we discussed why we need to use empirical rather than analytical methods to compare the performance of different algorithms for computing PDC. In this section, we perform such empirical tests. Figure IV.13 plots the average task energy consumption as a function of fraction of possible deadlines made for each of the workloads; Figure IV.14 is similar, but plots energy consumption versus average delay instead. The

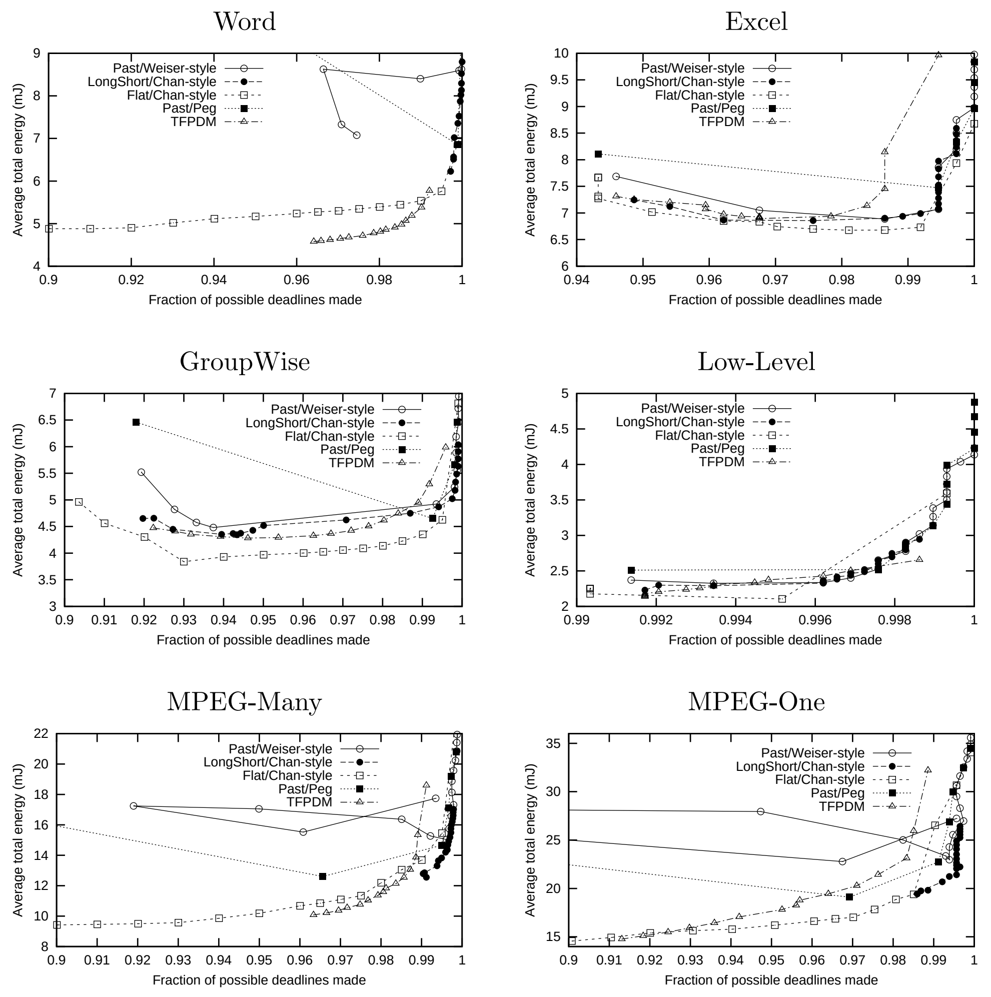

Figure IV.13: These graphs compare existing algorithms when the pre-deadline part is adjusted by PACE (Aged-0.95/Kernel) and the post-deadline part is changed to always use the maximum CPU speed. We generate each line by choosing an existing algorithm and simulating it for various parameter values. Each point reflects the energy consumed and the fraction of deadlines made for one such simulation, so each line graphs how well the algorithm trades off deadline completions for energy.

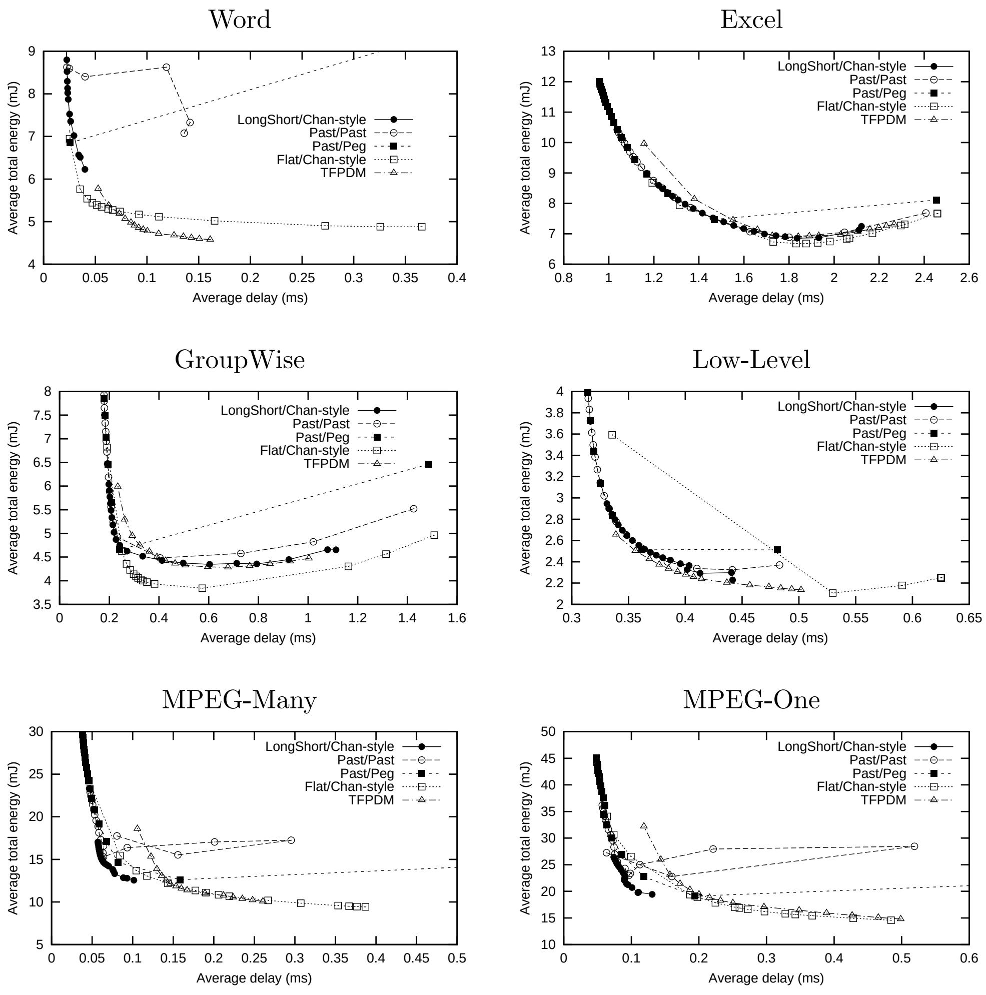

Figure IV.14: These graphs compare existing algorithms when the pre-deadline part is adjusted by PACE (Aged-0.95/Kernel) and the post-deadline part is changed to always use the maximum CPU speed. We generate each line by choosing an existing algorithm and simulating it for various parameter values. Each point reflects the average delay and the average total energy for one such simulation, so each line graphs how well the algorithm trades off delay for energy.

LongShort/Chan-style, Past/Past, and Past/Peg lines show the effect of varying interval length; the Flat/Chan-style line shows the effect of varying the constant predicted utilization (and thus the constant PDC); the TFPDM line shows the effect of varying TFPDM. We see that Flat/Chan-style, the only global strategy we considered, most commonly gives the lowest energy consumption for a given fraction of possible deadlines made or for a given average delay. This suggests that global strategies tend to do better than local ones. Among the local algorithms, LongShort/Chan-style does best, achieving reasonable energy savings for a given fraction of possible deadlines made or a given average delay.

IV.7.6 Overhead analysis

Although PACE reduces the energy consumption of algorithms, it also increases their complexity. Thus, it makes the CPU spend more time, and thus more energy, computing speed schedules. We can evaluate this overhead by simulating how much time and energy PACE-modified algorithms would consume to compute schedules.

The two PACE methods we simulate this way are Aged-0.95/Gamma and Recent-28/Kernel. The former pairs the computationally efficient gamma model with the preferred Aged-0.95 sampling method. The latter uses the more effective but less computationally efficient kernel density estimation method. To mitigate the computational complexity, we use it with the Recent-28 sampling method, which produces unweighted samples. We use a fixed PDC of $0.6MD$, as would Flat-0.6/Chan-style. We coded these algorithms in C in a couple of hours, making use of some obvious optimizations but by no means using every optimization possible. We use a maximum of 20 transitions per schedule, since we showed earlier that this gives nearly ideal results.

The resultant times are shown in Table IV.6. This table shows, for each workload and each algorithm, the average time per task to compute a speed schedule on a 450 MHz Pentium III with 128 MB of memory running RedHat Linux 6.2. The table also shows how much energy the simulated CPU would consume to perform this computation, as a percentage of the energy consumed to perform the tasks of the workload. We see from this table that we can implement these algorithms with minimal overhead. The Aged-0.95/Gamma algorithm is more efficient than the Recent-28/Kernel algorithm, but even the Recent-28/Kernel

	Word	**Excel**	**Group-Wise**	**Low-Level**	**MPEG-Many**	**MPEG-One**
Aged-0.95/ Gamma	31 μs (0.05%)	27 μs (0.08%)	30 μs (0.10%)	35 μs (0.25%)	29 μs (0.06%)	28 μs (0.04%)
Recent-28/ Kernel	68 μs (0.11%)	70 μs (0.20%)	77 μs (0.25%)	73 μs (0.51%)	63 μs (0.14%)	62 μs (0.08%)

Table IV.6: This table shows the average time a 450 MHz Pentium III takes per task to execute variants of the PACE algorithm. The numbers in parentheses indicate the energy overhead: how much energy the simulated CPU would consume to perform the PACE speed schedule computations, as a percentage of the energy it would consume just to execute the workload tasks.

algorithm imposes overhead of at most 77 μs per task and at most a 0.51% increase in energy consumption. The time overhead is small in all cases, and considering that the computation can be done at the end of each task in anticipation of the next task, it should not in general delay the completion of any task.

IV.8 Hard deadlines

Although we initially framed the dynamic voltage scaling problem in terms of soft deadlines, there are many workloads, especially in embedded systems, for which hard deadlines are more appropriate [PS01]. Fortunately, PACE is useful in these situations as well. Hard real-time systems generally specify each task as a certain worst-case number of cycles that must complete by a certain time. We can accommodate such hard deadlines by making the PDC equal to the worst-case number of cycles, ensuring that the hard deadline is made. Energy can still be saved by modeling the distribution of task work and increasing speed as the task progresses in accordance with PACE. When tasks require less than their worst-case requirement, they will consume less energy because PACE starts at a slow speed and increases it as time progresses. It is permissible, even in a real-time system, to begin running the task at a slower speed than the worst-case requirement divided by the deadline, since if the task turns out to be a long one, we can always increase the speed later to ensure it meets its deadline.

IV.9 Conclusions

The main focus of this chapter has been PACE, an approach to reducing the energy consumption of dynamic voltage scaling algorithms without affecting their performance. We showed that it is possible to change how an algorithm schedules tasks in a way that has no effect on performance but can reduce energy consumption. Furthermore, we developed an optimal formula for scheduling tasks with minimal energy consumption.

An important prerequisite for using the formula is estimating the distribution of a task's work requirement from recent data on similar tasks. We presented several methods that work well for a variety of workloads. The best we found is to use an aged sample as input to a nonparametric kernel distribution estimation method. Estimating the distribution with a gamma model works almost as well, and in many cases can be more practical. Practically implementing PACE also involves choosing a limited number of speed transitions and efficiently simulating the algorithms we seek to improve. We found heuristics for these that yield reasonable approximations and are practical and quick to implement.

Simulations using real workloads showed that PACE can substantially reduce CPU energy consumption without affecting performance. Without PACE, existing algorithms use DVS to reduce CPU energy consumption by 11–94% with an average of 54.3%. With the best version of PACE, the savings increase to 36–96% with an average of 65.4%. The overall effect is that PACE reduces the CPU energy consumption of existing algorithms by 1.4–49.5% with an average of 20.6%.

Besides PACE, we made other suggestions for changing scaling algorithms. We recommend that algorithms use a constant speed for all tasks once they have passed their deadlines; we believe the best speed for this purpose is the maximum CPU speed, largely because of the power consumption of other components. Furthermore, among the existing algorithms we considered, the best one to use along with PACE appears to be Flat/Chan-style. This algorithm always plans to complete the same number of cycles by each deadline.

We therefore recommend the following recipe for constructing a dynamic voltage scaling algorithm. For each task type, pick a reasonable deadline (e.g., 50 ms for interactive tasks), a reasonable number of cycles to always complete by the deadline (probably 40–60% of the maximum possible), and a reasonable speed to always use after the deadline has passed (probably the maximum CPU speed). Whenever a task completes, determine how many cycles it used, add this value to the sample of similar tasks' work requirements, then estimate the distribution of the next similar task using the new sample. For the sample, either only use recent values, or weight values as they age. Estimate the distribution using the kernel density estimation method, or the gamma model if the kernel density estimation method is impractical. When a task arrives, run it according to a PACE schedule that reflects the probability distribution for that type of task.

Chapter V

Windows Workload Characterization

V.1 Introduction

Implementing a deadline-based dynamic voltage scaling algorithm requires an understanding of the types of deadline-based tasks that occur in a typical workload. Thus, in this chapter, we will characterize the workloads we observed on Windows NT/2000 desktop computers. We obtained data about these workloads by having VTrace collect traces on several users' machines. These users did not act specially while being traced; they merely performed their standard daily activities over the course of months while the tracer happened to be running on their machines. Analyzing these traces provided many insights relevant to our search for a reasonable dynamic voltage scaling algorithm.

There are two general ways that a dynamic voltage scaling algorithm can determine the deadlines of computer tasks. One is to provide an interface that allows each application to specify when tasks begin and end and what their deadlines are. Another is to infer such information from incomplete information culled from observation of the application. The former technique provides detailed, useful information, but will only work with the cooperation of application programmers. The latter technique is more challenging and more generally applicable, so we focus on schemes for inferring deadlines in this chapter.

One way to infer tasks' deadlines is to observe when user interface events arrive and what those events are. We can consider a task to begin when a user interface event arrives. We can also infer the deadline for that task from user interface studies. For example, a keystroke or mouse click should generally be completed within 50–100 ms, since studies have shown that user think time is unaffected by response time as long as response time is under this threshold [Shn98]. As another example, processing related to a mouse movement should be completed within 25–75 ms [ES00].

When designing a DVS algorithm to take advantage of such information about user-interface task boundaries, it is also useful to know the distribution of such tasks' processing requirements, for several reasons. First, this distribution determines how many more deadlines will be missed if the CPU speed is reduced to save energy. Second, in Chapter IV, we saw that the optimal algorithm for dynamic voltage scaling depends on the distribution of the task work requirement. Third, this distribution determines how much energy will be saved by various dynamic voltage scaling algorithms.

Several other characteristics of workloads are important when designing a dynamic voltage scaling algorithm. For instance, if some categories of user interface events trigger substantially different amounts of processing than other categories of user interface events, then dynamic voltage scaling algorithms should likely consider such categories of events separately when inferring distributions of task work requirements. Therefore, in this chapter, we will analyze several characteristics of the workloads we collected to make suggestions about the design of dynamic voltage scaling algorithms.

This chapter is structured as follows. Section V.2 describes the methodology for our analyses, including how we process the traces to infer when tasks begin and end. Section V.3 describes the users whose machines we traced to obtain the analyses for this chapter. Section V.4 analyzes the workloads in many different ways to shed light on the effectiveness of various dynamic voltage scaling techniques. Here, we present a new heuristic for inferring when user interface tasks complete and show that it is more efficient and produces nearly the same results as a more complex heuristic. We also show that tasks triggered by user interface events that differ from each other in various ways have significantly different CPU requirements. This leads to the conclusion that PACE should separate information about tasks triggered by different types of events in order to better predict the probability distributions of their CPU requirements. Section V.5 gives the results of experiments with different DVS algorithms to clarify and validate various conclusions from our analyses. Finally, Section V.6 concludes.

V.2 Methodology

V.2.1 Tasks

We derived our workloads from Windows NT and Windows 2000 traces obtained using VTrace, the tracer described in Appendix A. This tracer collects timestamped records describing events related to processes, threads, messages, disk operations, network operations, the keyboard, the mouse, and the cursor.

To evaluate the distribution of tasks presented to computers during the traced activity, we need some way of defining and determining when such a task begins and when it ends. To do this, we first define an *event* as an occurrence that can trigger a thread to begin working on a task. The possible events are:

1. a thread receives a message,
2. a thread successfully completes a wait on a waitable object,
3. a thread is notified of an incoming network packet destined for it,
4. a thread starts,
5. a thread times out after an unsuccessful wait, and
6. a thread begins an asynchronous procedure call (i.e., a function call made asynchronously to the thread and invoked via a software interrupt).

If a thread is already running or ready to run when VTrace begins a session of full logging, we do not have enough information about its past to know what it is working on. We consider such work to be triggered by VTrace starting a session of logging, to represent the fact that we do not know the true trigger.

We now consider the problem of associating trigger events with the time threads spend running. Normally, we consider time a thread spends running to be triggered by the most recent event that thread performed. However, if that event was directly caused by a thread while it was working on some task triggered by a different event, we consider that different event to be the true trigger. For example, if a thread is working on a user interface message event, and it signals an object some other thread is waiting for, then whatever work that other thread performs when it completes its wait on that object will be considered triggered by the user interface message event. As an exception, if a thread causes a time-delayed event, for example by requesting that a timer message be delivered at some time in the future, then the thread that receives the time-delayed event considers the time-delayed event to be the trigger, rather than the event that the thread causing the time-delayed event was working on. The reason for this is that if a process is willing to allow time to pass before the work continues, we assume this work is not critical to the performance of the current task and is instead associated with the future time-delayed event.

Identifying when a thread is notified of a packet arrival is not always straightforward. Occasionally, a thread will explicitly perform a receive operation on a socket, and it is easy to see when that operation completes successfully. However, the more common case is for the thread to wait on a waitable object that is signaled when a network packet arrives. As VTrace cannot connect the packet with the object, we must use some heuristic approach to infer the connection in this case. The approach we use is as follows. Whenever the system is notified of an incoming packet on a connection, we note what process performed the most recent send on that connection. The next time a thread of that process successfully completes a wait on an unidentified object, we assume it is notified of that packet. We devised this heuristic from observation in several traces of this general pattern between network arrivals and completions of associated waits.

Note that these techniques for identifying when tasks begin, what triggered them, and how long they last, are imperfect heuristics. These heuristics are necessary to make reasonable inferences about tasks without unnecessarily intrusive tracing techniques. It is possible that we will miss the true trigger event for a task, and it is also possible that we will improperly infer the continuation of a task by the act of a thread sending a message or signaling an object that another task receives. For example, VTrace makes no distinction between different types of waitable object, as it traces these objects at too low a level to infer their meanings. A lock is a waitable object, so if one thread releases a lock and a second thread then acquires that lock, it will appear that the first thread is signaling the second thread, and that the second thread, upon receiving this signal, is continuing the work of the first thread. But, it may be that the second thread is working on a completely unrelated task that simply requires access to an object protected by the same lock. Also, it may be that threads communicate by writing data into shared memory, and we would not trace such communication. Nevertheless, the most common method for threads passing on work in Windows is sending messages, and our methods should work well in this common case.

V.2.2 User interface events

When we speak of user interface events, we mean events of one of the following three types: key press/release, mouse movement, and mouse click. User interface events can be further divided into *categories*. For instance, "spacebar press" and "number-key release" are categories of key press/release, and "left mouse down" and "right mouse up" are categories of mouse click. We define a *user interface task* as a task triggered by a user interface event.

V.2.3 Applications

We define an application as a set of processes with the same name, ignoring the extension. Thus, when we speak of the iexplore application, we mean all processes with a name of iexplore.exe, iexplore.bat, or the like.

V.2.4 Aggregation

For much of the data we will present, we will give averages across all users. Since the users were traced for different periods of time, and for other reasons have different amounts of activity, weighting each user according to his activity would cause aggregations to mostly reflect the users with the most activity. Thus, for averages, we will always scale all users' results to the same level of activity. For example, if there were only two users, one with 4,000,000 keystrokes taking an average of 5 ms to process and one with 1,000,000 keystrokes taking an average of 4 ms to process, we would report the average keystroke processing time

User #	1	2	3	4	5	6	7	8
Trace duration	8 months	7 months	4 months	15 months	3 months	19 months	2 months	9 months
Time full tracing on	435.8 hours	504.3 hours	83.0 hours	212.2 hours	134.9 hours	202.6 hours	106.9 hours	215.1 hours
Compressed trace size	21 GB	18 GB	17 GB	17 GB	16 GB	12 GB	15 GB	18 GB
CPU speed	450 MHz	300 MHz	500 MHz	200 MHz	500 MHz	400 MHz	433 MHz	350 MHz
CPU type	Pentium III	Pentium II	Pentium III	Pentium Pro	Pentium III	Pentium II	Celeron	Pentium II
Memory size	128 MB	unreported	96 MB	128 MB	256 MB	128 MB	256 MB	64 MB
Windows version and latest service pack (SP)	NT 4.0 SP 6	NT 4.0 SP 4	NT 4.0	NT 4.0 SP 3	2000 SP 1	NT 4.0 SP 4	2000 SP 1	NT 4.0 SP 4

Table V.1: Trace information for all users

as 4.5 ms, not 4.8 ms. Note that in reality different users do have different activity levels, and thus should indeed be weighted differently, but developing such a weighting is beyond the scope of this work.

V.3 Users

We used traces from several users in the analyses in this chapter. Table V.1 contains data about these users' machines as well as summary information about the workloads traced from these users. Recall that VTrace only collects the full set of events it can for sessions lasting 90 minutes at a time, after which it pauses for 2 hours. Also, it stops collecting the full set of events when the user is idle for 10 minutes, or when the user chooses to temporarily turn off tracing. In this chapter, we will only use data collected while full tracing was on.

The reported characteristics of the users are as follows.

- User #1 is a computer science graduate student. He uses his machine primarily as an X server, but also for mail, web browsing, software development, and office applications.
- User #2 is a computer science graduate student. He uses his machine primarily for mail, web browsing, software development, and office applications.
- User #3 is the chief technical officer of a computing-related company. He uses his machine for system administration, office applications, web browsing, and mail.
- User #4 is a system administrator at a university. He uses his machine for day-to-day networking and system administration tasks including use of an X server, e-mail client, web browser, and Windows NT system administration tools.
- User #5 did not report his profession; from the applications used, it appears he uses this machine largely for recreational purposes.
- User #6 is a Michigan State Police captain. He uses his machine primarily for groupware and office suite applications.
- User #7 is a software developer living in Korea. He uses his machine primarily for software development, web browsing, and mail.
- User #8 is a crime laboratory director in the Michigan State Police. He uses the machine primarily for groupware and office suite applications.

V.4 Analyses

In this section, we analyze the trace characteristics to gain insight about appropriate dynamic voltage scaling techniques.

V.4.1 Idle time

An important consideration for DVS algorithms is how often the CPU is idle, since more idle time means more potential ability to slow down the processor and still complete all work at a lower power consumption. We find that for user #1, the CPU is idle 92.6% of the time; user #2, 87.4%; user #3, 71.4%; user #4, 81.5%; user #5, 80.8%; user #6, 90.6%; user #7, 44.7%; and user #8, 85.3%. We see that for most traces, the CPU spends most of the time idle. The average is 79.3%, or 84.2% if we do not include the outlying result for user #7. We will see later that the reason for user #7's low idle time is because his machine acts as a server. The presence of substantial idle time in the traces allows the possibility of achieving substantial savings by slowing the processor and reducing its voltage.

From this point forward, we use the term *CPU time* to mean time the CPU spent not running the idle process.

V.4.2 Applications

In our analyses, we may find different behavior by different applications, yielding two effects. First, it may be more meaningful to characterize the behavior of applications by themselves than of the system in aggregate. Second, we may recommend DVS algorithms operate separately and/or differently on different applications. For these reasons, if we find that a small number of applications account for a large percentage of CPU time, then we can characterize a workload and the performance of a DVS algorithm by focusing on only these few applications.

Table V.2 shows application use by the eight users. For more detailed information, see Section B.2. We find that the months-long traces exhibit a large number of distinct applications, ranging from 141–311; note that this includes many non-obvious processes such as background daemons. However, of the total time the CPU runs, almost all time (90.6–98.6%) is spent running the most frequently run 25 of those applications. Thus, we can understand a great deal about the processor load by considering just those applications. We also observe that in all cases, over half the total CPU time is taken by the top five applications. Thus, savings for just a small number of applications can have substantial effect on the CPU usage of the entire system. When we look at the character of the top five applications for each user, we find that most have substantial user interface components. We will consider later how much of the activity of these applications is triggered directly by user interface events.

We observe that for all users, a web browser appears in the top six CPU time users,

User #	# of apps	% of CPU time due to top 1 app	% of CPU time due to top 5 apps	% of CPU time due to top 10 apps	% of CPU time due to top 25 apps
1	289	27.4%	56.9%	77.7%	93.0%
2	302	27.3%	52.6%	70.0%	90.6%
3	141	41.1%	80.6%	92.2%	98.6%
4	209	33.1%	76.0%	86.4%	95.7%
5	244	19.5%	60.6%	77.7%	92.9%
6	180	20.8%	53.5%	74.0%	97.4%
7	311	58.5%	74.4%	82.7%	94.6%
8	156	38.6%	75.7%	90.7%	97.9%
Avg	229	33.3%	66.3%	81.4%	95.1%

User #	Top eight applications, in order of decreasing CPU usage
1	netscape, acrord32, msdev, system, exceed, ssh, explorer, symbolsx
2	iexplore, ss3dfo, starcraft, acrord32, winword, psp, java, system
3	aim, netscape, realplay, ntvdm, blackd, faxmain, psp, system
4	netscape, realplay, msdtc, outlook, ssh, system, explorer, exceed
5	dwrcc, iexplore, realplay, explorer, system, winmgmt, savenow, rtvscan
6	grpwise, ntvdm, findfast, eudora, mcshield, netscape, realplay, system
7	v3webnt, winmgmt, iexplore, realplay, explorer, inetinfo, ndmonnt, system
8	netscape, grpwise, realplay, acrord32, ntvdm, explorer, system, mcshield

Table V.2: Users' usage of distinct applications

and for most of those users, it appears in the top two. It seems that browsers are either extremely frequently used, or consume a substantial amount of CPU time, or both. We will consider later whether browsers tend to use more CPU time to satisfy a single user interface request than other applications.

V.4.3 Trigger events

The next question we consider is how much time is spent on different trigger event types. As our focus in this chapter is on inferring tasks from user interface messages, it is especially important to know how much of the total system work is due to such messages. Table V.3 shows how much CPU time is due to the various types of trigger event. Note that we divide message triggers into three separate categories: user interface messages, i.e., key

User #	User interface message	Timer message	Other message	Timer object	Other object	Packet	Thread start	Session start	Timeout	APC
1	29.5%	29.2%	3.6%	0.0%	23.4%	0.0%	2.9%	2.1%	2.7%	6.7%
2	43.7%	27.2%	3.9%	0.0%	16.4%	0.1%	2.6%	0.9%	4.1%	1.2%
3	7.3%	68.1%	2.5%	0.0%	6.4%	0.0%	0.2%	1.3%	11.6%	2.6%
4	22.9%	28.0%	3.9%	0.0%	21.6%	0.0%	0.4%	2.2%	20.4%	0.7%
5	10.9%	23.4%	4.9%	0.1%	29.8%	0.0%	6.1%	16.8%	7.9%	0.0%
6	17.7%	46.9%	2.3%	0.0%	22.7%	0.1%	1.0%	5.3%	3.8%	0.3%
7	5.6%	6.8%	1.5%	0.0%	39.0%	0.0%	0.2%	43.0%	3.9%	0.0%
8	24.5%	51.1%	1.4%	0.0%	16.0%	0.1%	0.6%	1.7%	4.0%	0.6%
Avg	20.3%	35.1%	3.0%	0.0%	21.9%	0.0%	1.8%	9.2%	7.3%	1.5%

Table V.3: % of CPU time triggered by each event type

press/release, mouse click, and mouse movement; timer messages; and other messages. A timer message is a special type of message that is received only after some specified delay after it is posted. To see this information broken down by application, see Section B.3.

User interface messages trigger tasks with presumed deadlines, as users require quick response time to the requests they make. We find that the fraction of overall total CPU time due to user interface events differs substantially from one user to another. It ranges from 5.6%–43.7%, with an average of 20.3%. This is a curiously low percentage, as most Windows applications are user interface applications, and most apparent work done by such applications is in direct response to user interface events. We thus now explore how the CPU spends the remaining time.

A substantial fraction of total CPU time, especially for users with little time spent on user interface events, is due to timer messages. This is activity scheduled to occur at some point in the future, either once or repeatedly every n milliseconds. For example, a process may blink the cursor periodically, choose to periodically poll for messages or interesting events, schedule a web page reload or redirect at some point in the future, or play an audio or video frame at a particular time. Some of this activity has important implied deadlines, such as media playback, and some does not, such as blinking the cursor. The average percentage of

CPU time spent working on such timer messages is 35.1%.

User #7 uses a substantial amount of time (43.0%) on tasks that were already running when VTrace began a session of logging. This suggests a great deal of time on long-running processes, and looking at the particular applications responsible, we see they tend to be server processes. Therefore, we see that for this user, the low percentage of time for user interface events is due to the machine acting as a server. In retrospect, this also explains the low fraction of overall idle time observed for this user. Such server workloads are not characteristic of what we would expect on modern laptops, so we would expect CPU time due to user interface events, as well as idle time, to be higher on a portable computer than in this user's trace.

The largest remaining component of CPU time, accounting for 21.9% of it on average, is time triggered by threads completing a wait on a non-timer waitable object that VTrace could not identify as having been signaled by some thread. In other words, it is time spent working on tasks whose purpose VTrace could not determine. Such a waitable object may be signaled by the system, for example to indicate the insertion of a CD or to notify a thread that a certain file's contents were modified. Another possibility is that a thread triggered this event in some implicit way that VTrace could not detect. For example, a thread can wait on a queue object, and another thread posting an entry to the queue would implicitly signal the queue object to wake the waiting thread. In Section V.4.5, we will consider an alternate approach to tracking the duration of events that can account for such implicit signaling of events.

In conclusion, our methods show that only about 20.3% of CPU time is spent responding to user interface events. This figure is low partly due to limitations of our methodology, which cannot identify the cause of 21.9% of CPU time. It is also low partly due to server activity in traces that laptops typically would not have. Another large component of CPU time is time spent working on timer events, which can also be a source of tasks with deadlines.

User #	UI events	Key press/release		Mouse move		Mouse click	
1	11,456,238	51.6%	*46.3%*	47.1%	*28.4%*	1.3%	*25.4%*
2	15,075,773	10.6%	*17.1%*	85.4%	*44.5%*	4.0%	*38.3%*
3	2,523,477	5.2%	*6.1%*	92.8%	*61.2%*	2.0%	*32.6%*
4	7,090,460	18.8%	*22.3%*	78.4%	*39.6%*	2.7%	*38.1%*
5	3,527,579	6.7%	*9.8%*	91.1%	*53.6%*	2.1%	*36.6%*
6	4,627,439	11.9%	*14.0%*	85.9%	*50.0%*	2.2%	*36.0%*
7	2,059,678	7.7%	*19.6%*	88.8%	*45.3%*	3.5%	*35.1%*
8	8,976,067	3.0%	*4.4%*	94.8%	*41.0%*	2.2%	*54.6%*
Avg	6,917,089	14.4%	*17.5%*	83.0%	*45.5%*	2.5%	*37.1%*

Table V.4: This table shows, for each user, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

V.4.4 User interface event types

In this section, we consider the breakdown of user interface messages among the different types of such messages. The three main types of user interface message are key press/release, mouse move, and mouse click. Table V.4 shows what percentage of user interface messages are of the three different types, and what percentage of CPU time spent working on such messages is due to the three types. To see this information broken down by application, see Section B.4.

For all users, mouse clicks account for a small percentage of all user interface events, ranging from 1.3–4.0% with an average of 2.5%. Nevertheless, they always account for a much larger percentage of total CPU time spent on user interface messages, ranging from 25.4–54.6% with an average of 37.1%. Thus, we see that mouse clicks, though relatively rare, consume a proportionally large amount of CPU time compared to key press/release and mouse movement events. Mouse movements show the opposite pattern: they account for a large percentage of all user interface events, ranging from 47.1–94.8% of all events with an average of 83.0%, but account for a relatively small percentage of CPU time spent on user interface events, ranging from only 28.4–61.2% of all time with an average of 45.5%. The last category is key press/release events, which account for 3.0–51.6% of all events with an average of 14.4%, and 4.4–46.3% of all CPU time with an average of 17.5%. So, we see that mouse movements tend to be the most frequent events while mouse clicks are the least frequent events, but mouse movements take the least time per event while mouse clicks take the most.

Different users can have different patterns of user interface usage for the same application. For instance, of the users who have winword among their top 25 applications, the percentage of user interface messages that are key press/release range from 4.7–66.0%, while the percentage of messages that are mouse clicks range from 1.0–4.5%. So, it appears that some people use the mouse more than others when using Microsoft Word.

V.4.5 Inferring task completion

Determining when a task triggered by a user interface message has completed is difficult for an operating system, since it has no access to the source code of the application or otherwise to the algorithm it is using. In Section V.2, we explained our methodology for this. This is similar to the *task set* approach recommended by Flautner et al's [FRM01], which works as follows. When the GUI controller sends a message indicating a user interface request occurred, the task is said to begin. The GUI controller and the receiving thread form the initial *task set* for this request. Whenever a member of the task set communicates with another thread, that other thread is added to the task set. They consider the request to have completed when, for each task in the task set, the following conditions hold:

- it is not executing;
- data it has written have been consumed;
- it was not preempted the last time it ran, i.e., it is blocked; and
- it is not blocked on I/O.

Unfortunately, this approach is quite complex for use in an on-line algorithm. It requires modifying or interposing many system calls to keep track of thread communications and how they block; these modifications can create high system overhead. Furthermore, this approach can require keeping track of an unbounded amount of information, since the location of the data written by a thread that has not yet been consumed and the set of signals and messages sent by a thread that have not yet been received may be large. Finally, this approach can never be truly complete, since there are ways for threads to communicate with each other, e.g., by writing to shared memory, that cannot be efficiently tracked. Even if the system detected that a thread wrote to shared memory, it would have no way of knowing whether to wait for another thread to read that memory before the task should be considered complete.

We propose a simplified approach, as follows. A user interface request begins when received by an application. It is considered complete when one of the following becomes true:

- the idle thread is running and no I/O is ongoing; or
- the application receives another user interface request

This greatly simplifies the implementation, at the cost of occasionally misidentifying a task as complete when it is not. It is easy to determine when the idle thread is running and no I/O is ongoing, and when another user interface request occurs. In particular, this approach does not require any tracking of communication between threads. One possible flaw is that this approach considers as part of a task all processing done by other unrelated threads in the system, since the task is not considered complete until *all* threads in the system are blocked. However, this is actually a boon, since it aids in the automatic inclusion of other thread time in the estimate of task work requirements, as it causes the task end time to take into account work performed by other threads.

To evaluate the potential inaccuracy in this method, we have determined what percentage of user interface requests are still being worked on past the point when the system goes idle with no I/O ongoing, and what percentage of these user interface requests are still being worked on when the next user interface request occurs. The results of this experiment are in Table V.5. More detailed results are in Section B.5.

We find that for most users, the fraction of requests whose processing continues past either boundary is low. However, user #1 shows a rather large fraction of such requests, 20.8%. Breaking the results down by application, we see that this variation is due to a single application, exceed, which has 47.0% of its requests go beyond the next boundary. Ignoring exceed, user #1 shows a much lower fraction of requests extending beyond either boundary. The other user using exceed is user #4, and for that user exceed has 38.6% of its requests

User #	# of requests	# of requests continuing past...		
		system idle	next request	either boundary
1	11,456,238	1,252,592 (10.9%)	2,357,960 (20.6%)	2,382,664 (20.8%)
2	15,238,862	419,769 (2.8%)	467,337 (3.1%)	474,104 (3.1%)
3	2,523,477	8,466 (0.3%)	17,382 (0.7%)	21,011 (0.8%)
4	7,090,460	146,716 (2.1%)	404,498 (5.7%)	415,569 (5.9%)
5	3,527,579	77,351 (2.2%)	112,057 (3.2%)	116,340 (3.3%)
6	4,617,042	64,141 (1.4%)	87,421 (1.9%)	91,990 (2.0%)
7	2,033,964	53,628 (2.6%)	106,988 (5.3%)	108,703 (5.3%)
8	8,976,067	44,511 (0.5%)	71,472 (0.8%)	80,617 (0.9%)
Without exceed or java...				
1	6,577,941	41,075 (0.6%)	82,951 (1.3%)	89,728 (1.4%)
2	14,881,519	206,858 (1.4%)	231,114 (1.6%)	236,484 (1.6%)
4	6,276,461	47,532 (0.8%)	93,376 (1.5%)	101,683 (1.6%)
Without considering object signaling to pass on event processing responsibility...				
1	11,456,238	21,644 (0.2%)	67,220 (0.6%)	70,755 (0.6%)
2	15,238,862	108,905 (0.7%)	124,764 (0.8%)	127,354 (0.8%)
4	7,090,460	26,512 (0.4%)	71,255 (1.0%)	76,364 (1.1%)

Table V.5: For each user, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

go beyond either boundary. The only other application with a significant number of tasks extending beyond the next boundary was java, which, for user #2, has 66.5% of its tasks extend beyond the next boundary.

Thus, we see that there are a few applications for which there are a substantial number of user interface tasks that go beyond the next idle time and the next user interface task, but most other applications do not show this behavior. We hypothesize that this is because the outlier applications use objects to signal threads to perform unrelated work; for example, it may use one or more locks, and our experiment sees the release of such a lock and the subsequent acquire of that lock as a continuation of the same task when it is not. To determine if object signaling may be the cause, we performed the experiment without considering object signals to continue the work of the same task. Results for user #1, user #2, and user #4 are also shown in Table V.5. We see that here the applications have a more normal, low number of tasks that continue past the next idle and user interface event arrival time, suggesting it is misjudgment of object signaling that is the cause of those applications' high percentage of user interface tasks continuing past these natural task boundaries.

Ignoring these two applications, we find that the percentage of tasks continuing past system idle is quite low, no more than 2.6%. We find that the percentage of tasks continuing past the next user interface task is also quite low, no more than 5.3%. Furthermore, there is a substantial amount of overlap between tasks that persist beyond the next system idle time and those that persist beyond the next user interface task. So, the percentage of tasks continuing past either is no more than 5.3%. The reason for the overlap is straightforward: tasks that take a long time are more likely to both persist past system idle and past the next user interface task arrival. It is likely that many of these tasks only appear to take a long time due to a failure of our base heuristic to detect the true time the task ends, so if our heuristic were better we would see even fewer tasks extending beyond either boundary.

Thus, our simplified approach to identifying when tasks end will likely operate well in practice for most applications. However, for some small set of applications it may produce unintended wrong results, considering tasks to be complete when they are not yet complete.

V.4.6 I/O

An important concern when applying dynamic voltage scaling algorithms is how often tasks will require I/O. This is because I/O power and time do not scale as the CPU voltage and speed are scaled, so a different algorithm must be used when I/O occurs. Table V.6 shows what percentage of user interface tasks require I/O of two kinds: disk and network. For breakdowns of this information by application, see Section B.6.

We find that the fraction of all user interface tasks requiring I/O ranges from 0.3–3.5%, with an average of 1.3%. In other words, for all users at least 96.6% of all user interface tasks were completed with neither kind of I/O, and on average 98.7% of them were. Restricting consideration to just the disk, the percent of user interface tasks requiring disk I/O ranged from only 0.2–1.0%, with an average of 0.5%. This may be the most relevant figure, since the percentage of user interface tasks requiring network I/O should be even smaller for laptop computers running on battery power due to their typically low network connectivity in this scenario. On the other hand, even though fewer user interface tasks will

User	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
1	0.3%	*5.0%*	**5.2%**	0.2%	*1.1%*	**1.2%**	8.3%	*9.3%*	**15.5%**
2	0.7%	*5.4%*	**5.9%**	0.2%	*0.4%*	**0.6%**	5.0%	*2.7%*	**6.8%**
3	0.7%	*0.7%*	**1.4%**	0.3%	*0.2%*	**0.4%**	12.5%	*2.9%*	**14.5%**
4	0.1%	*5.4%*	**5.5%**	0.1%	*1.1%*	**1.2%**	6.1%	*10.0%*	**14.6%**
5	0.2%	*0.1%*	**0.3%**	0.1%	*0.1%*	**0.2%**	3.1%	*2.7%*	**5.1%**
6	0.4%	*0.2%*	**0.6%**	0.2%	*0.1%*	**0.3%**	10.0%	*5.1%*	**13.3%**
7	2.6%	*0.4%*	**2.7%**	0.5%	*0.1%*	**0.5%**	10.0%	*1.5%*	**10.5%**
8	0.8%	*2.5%*	**3.1%**	0.3%	*0.1%*	**0.4%**	17.2%	*7.9%*	**18.1%**
Avg	0.7%	*2.5%*	**3.1%**	0.2%	*0.4%*	**0.6%**	9.0%	*5.3%*	**12.3%**

User	All user interface events		
	disk	*network*	**either**
1	0.3%	*3.2%*	**3.5%**
2	0.5%	*1.1%*	**1.4%**
3	0.5%	*0.3%*	**0.7%**
4	0.3%	*2.2%*	**2.4%**
5	0.2%	*0.2%*	**0.3%**
6	0.5%	*0.2%*	**0.6%**
7	1.0%	*0.2%*	**1.0%**
8	0.7%	*0.3%*	**0.8%**
Avg	0.5%	*1.0%*	**1.3%**

Table V.6: For each user, and for various types of user interface events, the percentage of those events that require waiting for disk I/O and/or the network

access the network when operating wirelessly, each network access will tend to take more time since wireless networks are generally slower than wired ones.

Part of the reason for the infrequent I/O is that mouse movements are the most common user interface events and they seldom require I/O. If we examine mouse click events by themselves, they tend to involve I/O far more often. The percent of mouse click events triggering tasks involving I/O ranges from 5.1–18.1%, with an average of 12.3%. For this reason, dynamic voltage scaling algorithms planning schedules for mouse click events may need to provide extra slack for I/O time whose speed and power consumption do not scale with clock speed. When we examine only keystroke operations, the rate of I/O is a modest 0.3–5.9% with an average of 3.1%, or only 0.1–2.6% with an average of 0.7% if we ignore network I/O.

We have also found that different applications have different fractions of their events causing I/O, so some have more significant I/O demands than others for these events. For example, for user #4, ssh requires network I/O for 29.1% of all its key press/release events, and for user #2, starcraft requires network I/O for 20.5% of its key press/release events. Thus, it may be worthwhile for a dynamic voltage scaling algorithm to keep track of which applications require substantial I/O and treat them specially.

V.4.7 Mouse clicks

In later analyses, we will be analyzing to what extent treating different categories of user interface event differently can aid DVS algorithms in modeling and characterizing the work requirements of user interface tasks. Thus, it is important to know what the different categories of user interface event are. It is also useful to know which such categories account for the most user interface events and CPU time, since these are the best candidates for separate consideration by a DVS algorithm. A DVS algorithm that uses too many categories, each of which occurs relatively infrequently, will not have sufficiently large sample sizes in each category to model the work distributions for those categories.

In this subsection, we consider what categories of mouse click events typically occur, and how many tasks and how much work each category of mouse click event triggers. Table V.7 shows this information. For breakdowns of this information by application, see Section B.7.

Note that the number of messages posted indicating the left mouse button was pressed is not the same as the number of messages posted indicating the left mouse button was released. This is a well-known peculiarity of Windows. One cause (among many) of this is that a double-click can cause two mouse-down events and only one mouse-up event.

We see that by far the largest percentage of mouse click events is due to pressing and releasing the left mouse button. Left mouse button presses account for 48.1–55.1% of the click events with an average of 50.1% and 27.4–59.2% of the click time with an average of 42.9%, while left mouse button releases account for 37.5–43.0% of the click events with an average of 41.1% and 34.7–63.5% of the click time with an average of 47.5%. Together, left

User	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category...						
1	49.1%	41.6%	3.6%	2.9%	1.8%	0.9%
2	48.7%	43.0%	3.1%	2.4%	2.5%	0.4%
3	50.3%	40.7%	3.2%	4.2%	1.6%	0.0%
4	53.3%	42.7%	1.5%	1.2%	0.7%	0.6%
5	46.1%	43.0%	4.4%	2.8%	2.8%	0.9%
6	50.4%	37.5%	9.6%	1.3%	1.3%	0.0%
7	48.1%	47.2%	1.8%	1.5%	1.4%	0.0%
8	55.1%	33.0%	7.3%	2.5%	2.1%	0.0%
Avg	50.1%	41.1%	4.3%	2.4%	1.8%	0.4%
Percent of CPU time due to each category...						
1	45.1%	43.5%	4.5%	4.3%	2.5%	0.2%
2	33.7%	57.2%	2.9%	2.4%	3.6%	0.1%
3	59.2%	34.7%	2.3%	2.2%	1.6%	0.0%
4	54.1%	40.7%	1.9%	1.1%	2.2%	0.0%
5	27.4%	52.8%	9.0%	0.6%	10.0%	0.1%
6	48.6%	40.2%	10.0%	0.3%	0.8%	0.0%
7	28.9%	63.5%	3.0%	0.5%	4.0%	0.0%
8	46.1%	47.0%	3.7%	2.3%	0.8%	0.0%
Avg	42.9%	47.5%	4.7%	1.7%	3.2%	0.1%

Table V.7: For each user, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events.

mouse button presses and releases account for 87.9–96.0% of all click events with an average of 91.2% and 80.2–94.8% of all click time with an average of 90.4%. By far less numerous are left button double-click, right button press, and right button release events. The remaining four categories of mouse click (right button double-click, middle button press, middle button release, and middle button double-click) account for vanishingly few of the click events and little of the click processing time: 0.0–0.9% of all click events with an average of 0.4% and 0.0–0.2% of all click time with an average of 0.1%.

We can use this information to guide the design of a dynamic voltage scaling algorithm that distinguishes user interface tasks by the particular event that generated them. We see that we can account for a large proportion of mouse click events just by tracking left mouse button presses and releases. If we also include left button double-clicks, right button presses, and right button releases, we can account for nearly all such events. A DVS

Key description	Key press or release		Key press		Key release	
Letter	37.10%	*29.39%*	17.79%	*19.15%*	19.31%	*10.25%*
Modifier(s) alone	20.42%	*9.93%*	17.70%	*6.99%*	2.72%	*2.94%*
Arrows	8.72%	*11.09%*	6.09%	*9.72%*	2.63%	*1.37%*
Spacebar	7.20%	*7.37%*	3.84%	*5.24%*	3.37%	*2.13%*
Eastern-language-char	3.66%	*5.09%*	1.62%	*0.87%*	2.04%	*4.22%*
Backspace	3.65%	*3.39%*	2.07%	*2.45%*	1.58%	*0.95%*
Ctrl-letter	3.56%	*2.69%*	2.65%	*2.38%*	0.91%	*0.30%*
Punctuation	2.80%	*2.84%*	1.46%	*1.88%*	1.34%	*0.96%*
Number	2.67%	*2.24%*	1.51%	*1.44%*	1.16%	*0.81%*
Enter	2.27%	*9.49%*	1.16%	*5.21%*	1.11%	*4.28%*
Shift-letter	2.14%	*1.87%*	1.20%	*1.38%*	0.94%	*0.49%*
Del	1.26%	*3.85%*	0.76%	*2.87%*	0.50%	*0.98%*
Modified-arrow	1.25%	*0.94%*	0.77%	*0.76%*	0.48%	*0.18%*
Tab	0.92%	*2.81%*	0.39%	*1.24%*	0.53%	*1.57%*
Page up/down	0.88%	*2.68%*	0.51%	*2.39%*	0.36%	*0.29%*
Home/End	0.51%	*0.42%*	0.25%	*0.32%*	0.26%	*0.11%*
F-keys	0.13%	*2.24%*	0.07%	*1.04%*	0.06%	*1.21%*
Escape	0.11%	*0.20%*	0.05%	*0.15%*	0.06%	*0.05%*
Modified Eastern-char	0.05%	*0.05%*	0.02%	*0.01%*	0.03%	*0.04%*
Alt-letter	0.03%	*0.13%*	0.02%	*0.13%*	0.01%	*0.00%*
Ctrl-spacebar	0.03%	*0.01%*	0.01%	*0.01%*	0.01%	*0.00%*
Ctrl-F-keys	0.00%	*0.01%*	0.00%	*0.01%*	0.00%	*0.00%*
Others	0.65%	*1.26%*	0.30%	*0.84%*	0.35%	*0.42%*

Table V.8: For all users, what percentage of events (in normal type) and CPU time (in italics) from key events are due to the various categories of key events. User workloads are scaled so that each user has the same number of key press/release events and CPU time.

algorithm that aggregates mouse click events this way will have large sample sizes for most categories, enabling good estimation of task work requirement distributions.

V.4.8 Key presses and releases

In the last subsection, we explained why it is important to know the relative frequencies of various categories of mouse clicks, and how much CPU time is triggered by the different categories. For the same reasons, we would like to know this information for the various categories of keystroke events. Table V.8 shows this information. For breakdowns of this information by application, see Section B.8.

Note that key press messages are more frequent than key releases. This can occur because pressing and holding a key causes multiple key press messages but only one release message. Occasionally, key release messages of some category are more frequent than key press messages of the same category. This is because a different set of modifiers may be down when the key is pressed than when it is released. So, the key press may be placed in a different category than the corresponding key release. For instance, if the user types Ctrl-c then releases the Ctrl key before the c, then it will register as a ctrl-modified letter press and an unmodified letter release.

We consider a small set of 24 key categories: letter, shift-letter, ctrl-letter, spacebar, number, shift-number, miscellaneous punctuation, backspace, tab, enter, escape, home, end, del, arrows, F-keys, page up/down, ctrl-F-keys, ctrl-spacebar, alt-letter, modified-arrow (e.g., ctrl-arrow), Eastern-language special character, modified Eastern-language special character, and modifiers alone. We find that these cover almost all key press events: for all users, the remaining key categories account for 0.14–1.13% of keystroke events with an average of 0.65% and 0.41–1.94% of key press/release time with an average of 1.26%.

Key categories occurring frequently for all users are letter, modifiers alone, arrow, spacebar, and backspace. For some users, other key categories occur with significant frequency. For instance, user #1 uses emacs so much (both in emacs and via the X server) that he uses the ctrl-letter category more often than the normal letter category, and user #7 lives in Korea, so he uses Eastern language characters while the others do not. Furthermore, there is a great deal of variation from one user to another in the frequency of use of different key categories. For instance, 13.06% of user #6's key events use the spacebar, but only 1.96% of user #7's key events use the spacebar.

Considering all users in aggregate, with all users' traces weighted to have the same amount of key events and key CPU time, the top ten most frequently occurring key categories are letter, modifiers alone, arrow, spacebar, Eastern-language character, backspace, ctrl-letter, punctuation, number, and enter. Overall, they account for 92.05% of all key events but only 83.52% of CPU time spent on key events. This is because some key categories, namely del, tab, page up/down, and F-keys, account for relatively few key press/release events, but account for a relatively high percentage of time when they do. This is presumably because keys like these tend to trigger more complex activity than simply pressing letters and numbers and the like. If we include these additional four key categories, we account for 3.19% more events but 11.58% more time, so that we cover a total of 95.24% of events and 95.10% of time. Thus, a DVS algorithm need only separately consider these 14 key categories to account for most key press/release events and the CPU time they account for. Using such a small number of categories, each of which occurs relatively frequently, should yield reasonable sample sizes in each category for the estimation of task work distributions.

V.4.9 Significance of user interface event category differences

In this section, we consider whether different user interface event categories have significantly different processing requirements. To the extent they do, it is relevant to have different scheduling policies for different event categories. In addition, if user interface event categories are significantly different, it makes sense to consider them separately when making inferences about task work distributions. For example, if pressing the spacebar yields tasks of significantly different lengths than pressing enter, it may be best not to combine observed lengths for spacebar-triggered tasks with observed lengths for enter-triggered tasks.

In this subsection, we will only consider the significance of differences between mean task work requirements. We will consider broader differences in the distribution of task work in Section V.4.11. It may be that two event categories have similar means but significantly different distributions; if so, we will not detect those differences in this subsection. It may also be that two means are significantly different from a statistical standpoint, but the difference is small enough to be practically meaningless; such information will not be revealed in this subsection but must wait until the next. Also, in this subsection we will only consider tasks that consist only of CPU time, since as stated earlier tasks requiring I/O must be treated differently by dynamic voltage scaling algorithms.

Given a pair of event categories, we determine whether they have significantly different mean task lengths using a t-test [Jai91]. The input to such a test is the set of task lengths for each event category. The test determines how much variance each set has, and from this the likelihood that the difference in the two mean task lengths is not due to just this variance. It can then determine whether the mean task lengths are different within some specified confidence level. We consider three confidence levels, 90%, 95%, and 99%. When reporting such results, we will signify pairs that are different with 99% confidence by using a 'D', pairs that are different with 95% (but not 99%) confidence by using a 'd', pairs that are different with 90% (but not 95%) confidence by using a '.', and pairs that are not different with 90% confidence by using a blank space.

Note that an inability to detect a significant difference within a reasonable confidence level does not indicate that such a significant difference does not exist. It merely means we did not have sufficient data to demonstrate it. Such *Type II errors* [KZ89] can occur when data are insufficient and/or the data has too high a variance.

V.4.9.1 Key categories

First, we decide whether there are significant differences between key presses of different key categories within a single application and a single user. For each application and user, we will consider only the fifteen most frequent key categories, unless the top fifteen include some that occur less often than 30 times in the trace, in which case we use only those that occur at least 30 times. Table V.9 shows results for three typical applications and users. For results for other applications and users, see Section B.9.1.

We see that most event category pairs show significant difference from each other, even restricting considering only to the 99% level of confidence. There is no pair of event categories that consistently shows no significant difference from each other from application to application. There are applications, such as ssh, that treat many key categories similarly to each other; this is expected for ssh since it passes most key presses on uninterpreted via the network.

Next, we consider whether the situation for key release events. Table V.10 shows typical results from such tests; see Section B.9.1 for results for additional applications and users. We find that, in contrast to the results for key presses, many pairs of key categories are not significantly different from each other, even at only the 90% confidence level. In other words, many applications take quite similar amounts of time to process key release events corresponding to very different key categories. However, there is no pair of key categories that is never significantly different from each other across all applications. Nevertheless, this result is interesting as it suggests separating key categories into a large number of distinct groups for the purpose of estimating task work length is not as important for key releases as

User #1, netscape

	Modifiers	Number	Home/End	Punctuation	Letter	Shift-letter	Backspace	Arrow	Spacebar	Page up/down	Enter	Tab	Del	Ctrl-letter	Alt-letter
Modifiers		D	D	D	D	D	D	D	D	D	D	D	D	D	D
Number	D		D	D	D	D	D	D	D	D	D	D	D	D	D
Home/End	D	D			D	D	D	D	D	D	D	D	D	D	D
Punctuation	D	D			D	D	D	D	D	D	D	D	D	D	D
Letter	D	D	D	D		d	D	D	D	D	D	D	D	D	D
Shift-letter	D	D	D	D	d		D	D	D	D	D	D	D	D	D
Backspace	D	D	D	D	D	D		D	D	D	D	D	D	D	D
Arrow	D	D	D	D	D	D	D			D	D	D	D	D	D
Spacebar	D	D	D	D	D	D	D			D	D	D	D	D	D
Page up/down	D	D	D	D	D	D	D	D	D			D	D	D	D
Enter	D	D	D	D	D	D	D	D	D			D	D	D	D
Tab	D	D	D	D	D	D	D	D	D	D	D			D	D
Del	D	D	D	D	D	D	D	D	D	D	D				.
Ctrl-letter	D	D	D	D	D	D	D	D	D	D	D	D			
Alt-letter	D	D	D	D	D	D	D	D	D	D	D	D	.		

User #2, winword

	Modifiers	Arrow	Home/End	Spacebar	Letter	Number	Punctuation	Backspace	Shift-letter	Tab	Page up/down	Ctrl-F-key	Del	Enter	Ctrl-letter
Modifiers		D	D	D	D	D	D	D	D	D	D	D	D	D	D
Arrow	D			D	D		D	D	D	D	D	D	D	D	D
Home/End	D						D	D	D	D	D	D	D	D	D
Spacebar	D	D			d		D	D	D	D	D	D	D	D	D
Letter	D	D		d			d	D	D	D	D	D	D	D	D
Number	D								.	D	D	D	D	D	D
Punctuation	D	D	D	D	d				d	D	D	D	D	D	D
Backspace	D	D	D	D	D					d	D	D	D	D	D
Shift-letter	D	D	D	D	D	.	d			d	D	D	D	D	D
Tab	D	D	D	D	D	D	D	d	d		D	D	D	D	D
Page up/down	D	D	D	D	D	D	D	D	D	D			D	D	D
Ctrl-F-key	D	D	D	D	D	D	D	D	D	D			D	D	D
Del	D	D	D	D	D	D	D	D	D	D	D	D			D
Enter	D	D	D	D	D	D	D	D	D	D	D	D			d
Ctrl-letter	D	D	D	D	D	D	D	D	D	D	D	D	D	d	

User #4, ssh

	Modifiers	F-key	Tab	Shift-letter	Number	Punctuation	Letter	Spacebar	Backspace	Arrow	Enter	Escape	Ctrl-letter
Modifiers			d	d	D	D	D	D	D	D	D	D	
F-key			d	d	D	D	D	D	D	D	D	D	
Tab	d	d			D	D	D	D	D	D	D	D	
Shift-letter	d	d			d	d	D	D	D	D	D	D	
Number	D	D	D	d			D	d	D	D	d	d	
Punctuation	D	D	D	d			D	d	D	D	d	d	
Letter	D	D	D	D	D	D				d			
Spacebar	D	D	D	D	d	d							
Backspace	D	D	D	D	D	D							
Arrow	D	D	D	D	D	D	d						
Enter	D	D	D	D	d	d							
Escape	D	D	D	D	d	d							
Ctrl-letter													

Table V.9: This table shows the significance of differences of task length means for key **presses** for three applications and users. A 'D' indicates the row and column key categories are different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. The rows (and columns) are in increasing order of mean task length.

for key presses.

Given the different behavior of key presses and releases, we expect to see significantly different task work lengths for presses and releases of the same key category by the same application. Table V.11 shows the significance of differences between key presses and releases of the same key category for the top 25 applications of user #1; for the similar results for other users, see Section B.9.1. We find that by far the most common case is for key presses and releases to be significantly different from each other at the 99% level.

V.4.9.2 Mouse click categories

We next consider whether mouse click categories are different from each other. We will consider six main questions:

- Is a left mouse down significantly different from a left mouse up?
- Is a left mouse down significantly different from a left mouse double-click?
- Is a left mouse up significantly different from a left mouse double-click?
- Is a left mouse down significantly different from a right mouse down?
- Is a left mouse up significantly different from a right mouse up?
- Is a right mouse down significantly different from a right mouse up?

We answer these questions for each of the top applications for users #1–3 in Table V.12; for the remaining users, see Section B.9.2.

Some of the results are quite surprising in that it seems the answer should be "yes" but the answer turns out to be "no." For instance, iexplore for some users shows no significant difference between left mouse down and left mouse up. This is surprising, since the typical left mouse action in iexplore is to click on a web page, with the down action merely highlighting the link and the up action actually fetching the page. The main reason for these "no" answers is that, unlike key activity, mouse clicks can cause extremely different behavior depending on what is clicked, so mouse click event times tend to have high variance. Thus, the lack of an observed difference is likely due to a Type II error: without substantial data we may not be able to prove the presence of a significant difference despite the presence of

User #1, netscape

	Ctrl-letter	Escape	Page up/down	Punctuation	Number	Arrow	Backspace	Shift-letter	Letter	Home/End	Spacebar	Enter	Modifiers	Tab	Del
Ctrl-letter		D	D	D	D	D	D	D	D	D	D	D	D	D	D
Escape	D			D	D	D	D	D	D	D	D	D	D	D	D
Page up/down	D			D	D	D	D	D	D	D	D	D	D	D	D
Punctuation	D	D	D		D	D	D	D	D	D	D	D	D	D	D
Number	D	D	D	D		D	D	D	D	D	D	D	D	D	D
Arrow	D	D	D	D	D		D	D	D	D	D	D	D	D	D
Backspace	D	D	D	D	D	D				d	D	d	D	D	D
Shift-letter	D	D	D	D	D	D				d	D	d	D	D	D
Letter	D	D	D	D	D	D				d	D	d	D	D	D
Home/End	D	D	D	D	D	D	d	d	d		d	·	D	D	D
Spacebar	D	D	D	D	D	D	D	D	D	d			D	D	D
Enter	D	D	D	D	D	D	d	d	d	·			D	D	D
Modifiers	D	D	D	D	D	D	D	D	D	D	D	D		·	D
Tab	D	D	D	D	D	D	D	D	D	D	D	D	·		
Del	D	D	D	D	D	D	D	D	D	D	D	D	D		

User #2, winword

	Shift-letter	Letter	Arrow	Backspace	Spacebar	Punctuation	Home/End	Page up/down	Number	Modifiers	Del	Tab	Ctrl-F-key	Ctrl-letter	Enter
Shift-letter		·	d	D	D	D	D	D		D	D	D	D	D	D
Letter	·				·	d	D	D		D	D	D	D	D	D
Arrow	d					d	D	D		D	D	D	D	D	D
Backspace	D					d	d	d		D	D	D	D	D	D
Spacebar	D	·					·	·		D	D	D	D	D	D
Punctuation	D	d	d	d						D	D	D	D	D	D
Home/End	D	D	D	d	·					D	D	D	D	D	D
Page up/down	D	D	D	d	·					D	D	D	D	D	D
Number										D	D	D	d	D	D
Modifiers	D	D	D	D	D	D	D	D	D					d	D
Del	D	D	D	D	D	D	D	D	D					·	D
Tab	D	D	D	D	D	D	D	D	D					·	D
Ctrl-F-key	D	D	D	D	D	D	D	D	d						d
Ctrl-letter	D	D	D	D	D	D	D	D	D	d	·	·			
Enter	D	D	D	D	D	D	D	D	D	D	D	D	d		

User #5, iexplore

	Arrow	Number	Home/End	Spacebar	Letter	Del	Shift-letter	Punctuation	Modifiers	Backspace	Tab	Enter
Arrow		D		D	D	D	D	D	D	D	D	D
Number	D						d		d	d	D	D
Home/End											·	D
Spacebar	D						·		d	·	D	D
Letter	D								d	·	D	D
Del	D										d	D
Shift-letter	D	d		·							D	D
Punctuation	D										·	D
Modifiers	D	d		d	d						d	D
Backspace	D	d		·	·							D
Tab	D	D	·	D	D	d	D	·	d			D
Enter	D	D	D	D	D	D	D	D	D	D	D	

Table V.10: This table shows the significance of differences of task length means for key **releases** for various applications and users. A 'D' indicates the row and column key categories are different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. The rows (and columns) are in increasing order of mean task length.

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
netscape	D	D	D	D	D	D	D	D	D	D	D	D	D
acrord32	D	D	*	*	*	*	*	*	*	*	*	D	D
msdev	D	D	D	D	D	D	D		D		D	D	D
exceed	D	D	.	D	D	D	D	D		D	D	d	*
ssh	D		*	*	D	*	D	D	D	*	D	*	*
explorer	D	D	D	D	D	*	D	D	D	*	*	*	D
ntvdm	D	*	*	*	*	*	*	*	*	*	*	*	*
windbg	D	D	D	D	D	*	D	D	D	*	*	D	d
starcraft		*	*	*	*	*	*	*	*	*	*	*	*
emacs	D	D	D	D	D	D	D	D	D	*	D	D	*
iexplore	D	d	D	*	D	*	D	D	*	*	*	*	*
powerpnt	D	d	D	D	D	*	D	D	D	D	D	*	*
winword	D	D	D	D	D	*	D	D	D	D	*	D	*

Table V.11: This table shows the significance of differences between task length means for key presses and releases for **user #1**. A 'D' indicates key presses and releases are significantly different with 99% confidence, a 'd' indicates 95% significance, a '.' indicates 90% significance, and a '*' indicates insufficient data.

one. For example, for iexplore for user #2, the standard deviation of left mouse down event times is five times the mean, and for left mouse up events it is almost 40 times the mean!

Another reason we see no difference in the case of iexplore is that if the page fetch requires I/O activity from the network or disk, we ignore the task for the purpose of this experiment, since we are not considering tasks that consume I/O. Thus, page fetches not in the memory cache are not considered here, and the dominant mouse activity consists of things like clicking menus and scrollbars.

Despite the lack of power of our tests due to the high variance of mouse click event times, we find that each of the questions has a "yes" answer at a 99% confidence level for many applications. We conclude that it may in general be useful to consider different mouse click categories separately.

V.4.10 Significance of application differences

The next question we ask is whether different applications exhibit different task work lengths for the same event type or category. We would generally expect this, except that Windows provides default message handler procedures for applications to use for messages they don't need to handle, so it could occur that applications using the same default message handler procedure would exhibit similar behavior for certain user interface message types and

User #1

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
netscape	D	·	D	d	D	D
acrord32		D	D	*	*	*
msdev				D	D	D
exceed	D	D	D	D	D	D
ssh	·	*	*	*	*	*
explorer	D	D	D	D	D	D
windbg	D	D	D	*	*	*
starcraft	d	*	*	D		
emacs	D	*	*	*	*	*
iexplore	D		D	*	*	*
powerpnt		·		*	*	*
msiexec		*	*	*	*	*
winword	d	*	*	*	*	*

User #2

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
iexplore		D	D	D		D
starcraft			·		D	d
acrord32		D	D	*	*	*
winword	D		D	D	D	
psp	D	D	D	D	D	D
java	D	*	*	D	D	D
realplay	D	*	*	*	*	*
devenv	D	D	D	D	D	D
explorer	D	D	D	D	D	D
hotsync		*	*	*	*	*
msimn	D		D	D	D	D
powerpnt	D	D		D	D	D
txtpad32	D		D	D	*	*
mplayer2		*	*	*	*	*
ssh	D	D	d	*	*	*
appletviewer	D	*	*	D	D	
realjbox	·	*	*	*	*	*
textpad	D		d	*	*	*

User #3

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
aim	D	*	*	*	*	*
netscape		D	d	d	*	*
faxmain	D			D	D	D
psp	D	D	·	*	*	*
explorer	D	d	d	D	D	D
sitelink	D	*	*	*	*	*
sbtw	D	*	*	*	*	*
bartend	D		D	*	*	*
outlook		·		*	*	*
blackice	D	*	*	*	*	*
napster		*	*		D	D
iexplore		*	*	*	*	*
excel		d		*	*	*
winword				*	*	*
act	*	d	*	*	*	*

Table V.12: This table shows the significance of differences between task length means for different mouse click events for users #1–3. A 'D' indicates key presses and releases are significantly different with 99% confidence, a 'd' indicates 95% significance, a '.' indicates 90% significance, and a '*' indicates insufficient data.

categories.

We first consider whether different applications show significantly different task work lengths for mouse movement events. Table V.13 shows differences between applications' mouse movement task lengths for user #1, user #2, and user #6. For other, similar results for other users, see Section B.10.

We find that by far the most common case is significant difference at the 99% confidence level, so we conclude that applications tend to differ from each other significantly in the time they take to handle mouse movement events.

Next, we consider whether applications differ from each other in their handling of mouse click events. For simplicity, we only present results for left mouse clicks; these results, for each user, are in Table V.14 shows results for users #1–3; for the similar results for the other users, see Section B.10. We once again see significant differences between most pairs of applications, despite the high variance we discussed earlier in these mouse click times.

Finally, we consider whether different applications exhibit different task mean lengths for key presses. We choose to consider modifier press events, for two reasons. First, these events are irrelevant to most applications, so they present a likely opportunity to observe similarity between applications. Also, they are quite frequent, so if we find no similarity we can be relatively confident it is not due to insufficient data. Table V.15 gives these results for users #1–3; for similar results for other users, see Section B.10. We find, even for this event category, substantial differences between the task length means between different applications in most cases.

V.4.11 Distribution of task work requirements

As stated in the introduction, the distribution of task work requirements is of great importance in the evaluation of the effectiveness of dynamic voltage scaling algorithms. Therefore, in this section, we describe these distributions for various user interface tasks. Since the number of such distributions is large, we focus on showing those distributions that make an interesting point about the distributions.

In these analyses, we will restrict consideration to those events that trigger no I/O, unless stated otherwise. This is because dynamic voltage scaling algorithms cannot affect

User #1

	windbg	realplay	emacs	explorer	exceed	ssh	powerpnt	netscape	msdev	acrord32	msiexec	winword	iexplore	starcraft	ntvdm
windbg		D	D	D	D	D	D	D	D	D	D	D	D	D	d
realplay	D			D	D	D	D	D	D	D	D	D	D	D	d
emacs	D			D	D	d	D	D	D	D	D	D	D	D	d
explorer	D	D	D		D		D	D	D	D	D	D	D	D	d
exceed	D	D	D	D			D	D	D	D	D	D	D	D	d
ssh	D	D	d					D	D	D	D	D	D	D	d
powerpnt	D	D	D	D	D			D	D	D	D	D	D	D	d
netscape	D	D	D	D	D	D	D			D	·	D	D	D	d
msdev	D	D	D	D	D	D	D			D	·	D	D	D	d
acrord32	D	D	D	D	D	D	D	D	D			D	D	D	d
msiexec	D	D	D	D	D	D	D	·	·				d	D	d
winword	D	D	D	D	D	D	D	D	D	D			d	D	d
iexplore	D	D	D	D	D	D	D	D	D	D	d	d		D	d
starcraft	D	D	D	D	D	D	D	D	D	D	D	D	D		·
ntvdm	d	d	d	d	d	d	d	d	d	d	d	d	d	·	

User #2

	ss3dfo	explorer	ssh	devenv	powerpnt	iexplore	msimn	acrord32	winword	txtpad32	textpad	psp	java	appletviewer	starcraft
ss3dfo		D	D	D	D	D	D	D	D	D	D	D	D	D	D
explorer	D		D	D	D	D	D	D	D	D	D	D	D	D	D
ssh	D	D		D	D	D	D	D	D	D	D	D	D	D	D
devenv	D	D	D		D	D	D	D	D	D	D	D	D	D	D
powerpnt	D	D	D	D		D	D	D	D	D	D	D	D	D	D
iexplore	D	D	D	D	D		D	d	D	D	D	D	D	D	D
msimn	D	D	D	D	D	D		d	D	D	D	D	D	D	D
acrord32	D	D	D	D	D	d	d			·	d	D	D	D	D
winword	D	D	D	D	D	D	D			D	D	D	D	D	D
txtpad32	D	D	D	D	D	D	D	·	D		D	D	D	D	D
textpad	D	D	D	D	D	D	D	d	D	D		D	D	D	D
psp	D	D	D	D	D	D	D	D	D	D	D		D	D	D
java	D	D	D	D	D	D	D	D	D	D	D	D		D	D
appletviewer	D	D	D	D	D	D	D	D	D	D	D	D	D		D
starcraft	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

User #6

	aim	notify	explorer	netscape	grpwise	excel	powerpnt	acrord32	acrobat	winword	iexplore	realjbox	planner	eudora	ntvdm
aim		D	D	D	D	D	D	D	D	D	D	D	D	D	D
notify	D		D	D	D	D	D	D	D	D	D	D	D	D	D
explorer	D	D		D	D	D	D	D	D	D	D	D	D	D	D
netscape	D	D	D					D	D	D	D	D	D	D	D
grpwise	D	D	D					D	D	D	D	D	D	D	D
excel	D	D	D					D	D	D	D	D	D	D	D
powerpnt	D	D	D					D	D	D	D	D	D	D	D
acrord32	D	D	D	D	D	D	D		D	D	D	D	D	D	D
acrobat	D	D	D	D	D	D	D	D		D	D	D	D	D	D
winword	D	D	D	D	D	D	D	D	D		D	D	D	D	D
iexplore	D	D	D	D	D	D	D	D	D	D		D	D	D	D
realjbox	D	D	D	D	D	D	D	D	D	D	D		D	D	D
planner	D	D	D	D	D	D	D	D	D	D	D	D		D	D
eudora	D	D	D	D	D	D	D	D	D	D	D	D	D		D
ntvdm	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

Table V.13: This table shows the significance of differences between mouse movement task length means for different applications for users #1, 2, and 6. A ‘D’ indicates mouse movements of the two applications are significantly different with 99% confidence, a ‘d’ indicates 95% significance, and a ‘.’ indicates 90% significance. Results are reported only for applications with at least 30 mouse movement events. Applications are presented in increasing order of mean mouse movement task length.

User #1

	powerpnt	winword	explorer	msdev	ssh	emacs	netscape	exceed	iexplore	windbg	starcraft	acrord32	msiexec	ntvdm
powerpnt		·	D	D	D	D	D	D	D	D	D	D	D	·
winword	·			D	d	D	D	D	D	D	D	D	D	·
explorer	D			D	d	D	D	D	D	D	D	D	D	·
msdev	D	D	D			·	D	D	D	D	D	D	D	·
ssh	D	d	d				d	D	D	D	D	D	D	·
emacs	D	D	D	·					·	d	D	D	d	·
netscape	D	D	D	D	d				d	D	D	D	d	·
exceed	D	D	D	D	D				·	D	D	D	d	·
iexplore	D	D	D	D	D	·	d	·			·	D	d	·
windbg	D	D	D	D	D	d	D	D				D	d	·
starcraft	D	D	D	D	D	D	D	D	·			D	d	·
acrord32	D	D	D	D	D	D	D	D	D	D	D		·	
msiexec	D	D	D	D	D	d	d	d	d	d	d	·		
ntvdm	·	·	·	·	·	·	·	·	·	·	·			

User #2

	mplayer2	powerpnt	ssh	explorer	devenv	txtpad32	iexplore	msimn	psp	starcraft	winword	textpad	appletviewer	acrord32	java
mplayer2		D	D	D	D	D	D	D	D	D	D	D	D	D	D
powerpnt	D		d	D	D	D	D	D	D	D	D	D	D	D	D
ssh	D	d				d	D	D	D	D	D	D	D	D	D
explorer	D	D			d	D	D	D	D	D	D	D	D	D	D
devenv	D	D		d			D	D	D	D	D	D	D	D	D
txtpad32	D	D	d	D			d	D	D	D	D	D	D	D	D
iexplore	D	D	D	D	D	d		D	D	D	D	D	D	D	D
msimn	D	D	D	D	D	D	D		D	D	D	D	D	D	D
psp	D	D	D	D	D	D	D	D		D	D	D	D	D	D
starcraft	D	D	D	D	D	D	D	D	D				d	D	D
winword	D	D	D	D	D	D	D	D	D				D	D	D
textpad	D	D	D	D	D	D	D	D	D				d	D	D
appletviewer	D	D	D	D	D	D	D	D	D	d	D	d			
acrord32	D	D	D	D	D	D	D	D	D	D	D	D			
java	D	D	D	D	D	D	D	D	D	D	D	D			

User #3

	msoffice	excel	sbtw	iexplore	napster	blackice	outlook	explorer	winword	aim	netscape	faxmain	act	bartend	psp
msoffice		D	D	D	·	D	D	D	D	D	D	D	D	D	D
excel	D					D	D	D	D	D	D	D	d	D	D
sbtw	D					d	D	D	D	D	D	D	d	D	D
iexplore	D					·	d	D	D	D	D	D	d	D	D
napster	·								·	d	D	D	d	D	D
blackice	D	D	d	·					d	d	D	D	d	D	D
outlook	D	D	D	d					d	d	D	D	d	D	D
explorer	D	D	D	D					·	d	D	D	d	D	D
winword	D	D	D	D	·	d	d	·			d	D	·	D	D
aim	D	D	D	D	d	d	d	d				d		D	D
netscape	D	D	D	D	D	D	D	D	d			d		D	D
faxmain	D	D	D	D	D	D	D	D	D	d	d			D	D
act	D	d	d	d	d	d	d	d	·					·	D
bartend	D	D	D	D	D	D	D	D	D	D	D	D	·		D
psp	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

Table V.14: This table shows the significance of differences between left mouse down task length means for different applications for users #1–3. A ‘D’ indicates left mouse down events for the two applications are significantly different with 99% confidence, a ‘d’ indicates 95% significance, and a ‘.’ indicates 90% significance. Results are reported only for applications with at least 30 relevant events. Applications are presented in increasing order of task length mean.

User #1

	emacs	windbg	exceed	explorer	acrord32	iexplore	ssh	netscape	winword	powerpnt	msdev	starcraft
emacs		D	D	D	D	D	d	D	D	D	D	D
windbg	D		D	D	D	D	·	D	D	D	D	D
exceed	D	D		d	D	D	·	D	D	D	D	D
explorer	D	D	d		D	d		D	D	D	D	D
acrord32	D	D	D	D				D	D	D	D	D
iexplore	D	D	D	d					D	D	D	D
ssh	d	·	·						D	D	D	D
netscape	D	D	D	D	D				D	D	D	D
winword	D	D	D	D	D	D	D	D		d	D	D
powerpnt	D	D	D	D	D	D	D	D	d		D	D
msdev	D	D	D	D	D	D	D	D	D	D		D
starcraft	D	D	D	D	D	D	D	D	D	D	D	

User #2

	explorer	msimn	ssh	devenv	powerpnt	iexplore	winword	psp	appletviewer	textpad	txtpad32	java	acrord32	starcraft
explorer			D	D	D	D	D	D	D	D	D	D	d	D
msimn			D	D	D	D	D	D	D	D	D	D	d	D
ssh	D	D		D	D	D	D	D	D	D	D	D	d	D
devenv	D	D	D		D	D	D	D	D	D	D	D	d	D
powerpnt	D	D	D	D			D	D	D	D	D	D	d	D
iexplore	D	D	D	D			·	D	D	D	D	D	d	D
winword	D	D	D	D	D	·		D	D	D	D	D	d	D
psp	D	D	D	D	D	D	D			D	D	D	·	D
appletviewer	D	D	D	D	D	D	D				·	D	·	D
textpad	D	D	D	D	D	D	D	D			D	D	·	D
txtpad32	D	D	D	D	D	D	D	D	·	D		D	·	D
java	D	D	D	D	D	D	D	D	D	D	D			D
acrord32	d	d	d	d	d	d	d	·	·	·	·			
starcraft	D	D	D	D	D	D	D	D	D	D	D	D		

User #3

	napster	sbtw	explorer	aim	iexplore	excel	netscape	winword	sitelink	outlook	faxmain	bartend	psp
napster		D	D	·	D	D	D	D	D	D	D	D	D
sbtw	D		D	·	D	D	D	D	D	D	D	D	D
explorer	D	D			·	D	D	D	D	D	D	D	D
aim	·	·				d	d	D	D	D	D	D	D
iexplore	D	D	·			·	d	D	D	D	D	D	D
excel	D	D	D	d	·		·	D	D	D	D	D	D
netscape	D	D	D	d	d	·			D	D	D	D	D
winword	D	D	D	D	D	D			d	D	D	D	D
sitelink	D	D	D	D	D	D	D	d		d		D	D
outlook	D	D	D	D	D	D	D	D	d			D	D
faxmain	D	D	D	D	D	D	D	D				D	D
bartend	D	D	D	D	D	D	D	D	D	D	D		
psp	D	D	D	D	D	D	D	D	D	D	D		

Table V.15: This table shows the significance of differences between modifiers-only key press task length means for different applications for users #1–3. A ‘D’ indicates modifier key press events for the two applications are significantly different with 99% confidence, a ‘d’ indicates 95% significance, and a ‘.’ indicates 90% significance. Results are reported only for applications with at least 30 relevant events. Applications are presented in increasing order of task length mean.

how much time is spent on I/O, and should apply differently to tasks that involve I/O.

We will illustrate distributions by graphing their cumulative distribution function F. In other words, we will graph $F(w)$ where $F(w)$ is the percent of all corresponding tasks that completed using no more than w megacycles (Mc). We will use a logarithmic scale for the w axis, to illustrate the shape of the distribution at various time scales. Each amount of work corresponds to a certain amount of time, given the constant CPU speed on the traced machines. We will generally label the X axis on the bottom using megacycles and the top using milliseconds.

The top part of the cumulative distribution function, e.g., the part for which $F(w) \geq 0.90$, is especially interesting. This is because to achieve a certain fraction of deadlines, we must set the average pre-deadline speed for the task equal to that which will complete that percentile of the task's work distribution within the deadline. (This is equivalent to choosing a PDC, or number of pre-deadline cycles, as described in Chapter IV, since the average pre-deadline speed is the PDC divided by the deadline.) For this reason, when we graph a distribution in one figure we will often accompany it with another figure focusing on only the top part of the distribution. This will enable the reader to better see what value of w satisfies, for instance, $F(w) = 0.95$. If we want to tune a DVS algorithm to complete 95% of tasks within 50 ms, and the 95th percentile is 20 Mc, then we will require an average pre-deadline speed of 20 Mc/50 ms = 400 MHz. Note that with PACE we need not use a constant pre-deadline speed of 400 MHz. All that is necessary is that the average pre-deadline speed be 400 MHz; in general, we will begin running at a slower speed and increase the speed as the task progresses.

There are some useful rules of thumb for converting megacycles to required average speeds. If the deadline is 50 ms, then one can compute the required speed in MHz by multiplying the quantile in Mc by 20. So, for instance, 25 Mc requires a speed of $25 \cdot 20 = 500$ MHz. If the deadline is 100 ms, then the appropriate multiplying factor is 10 instead of 20.

The shape of the cumulative distribution function on the logarithmic scale provides interesting information about the trade-off between reducing CPU speed and reducing the percent of deadlines made. The slope at any point represents the increase in percent of deadlines made that is achieved by multiplying the clock speed by some constant amount.

The higher the slope at a point w, the more useful it is to raise the CPU speed beyond that which will complete only w units of work by the deadline. Thus, one approach for picking a good average pre-deadline speed is to look for a point on the curve above which the slope of the CDF on a logarithmic scale is low; often, this appears as a "knee" in the curve. We call such a point a *saturation point.* Dividing a saturation point by the deadline gives a *saturation speed*, a pre-deadline speed above which there is not much improvement in deadline performance for a given increase in speed. We can thus hypothesize that a user would find such a pre-deadline speed satisfactory for tasks of the given distribution, even in the presence of a higher available speed on his processor. Note that we have no technical definition of a saturation speed, as we do not have sufficient information about how users perceive and value varying deadline performance as a function of energy savings, and often we are only making a rough guess as to deadline as well. However, this notion allows us to make broad estimates of how DVS algorithms should choose pre-deadline speeds for tasks.

V.4.11.1 User interface event types

First, let us observe the distributions for the three different type of user interface event: key presses and releases, mouse movements, and mouse clicks. Figures V.1 and V.2 show these distributions.

We see that for all users, mouse movements take substantially less time than key presses/releases, and that they in turn take substantially less time than mouse clicks. Because mouse movements are more numerous than the other user interface event types, the overall distribution for all user interface events tends to be similar and close to that for just mouse movements.

It seems likely that we should use a different pre-deadline average speed for mouse movements, mouse clicks, and key press/releases due to their different work requirements. We now consider what average speed seems useful for each class of events.

For mouse movements, we will consider the soft deadline to be 50 ms, following Endo et al. [ES00]. We observe that mouse movements tend to take a small amount of time to process, so it is possible that always using the minimum speed before the deadline may be reasonable for mouse movements. Modern laptop computers with DVS all have a minimum speed of at least 200 MHz, and considering the trend toward faster processors,

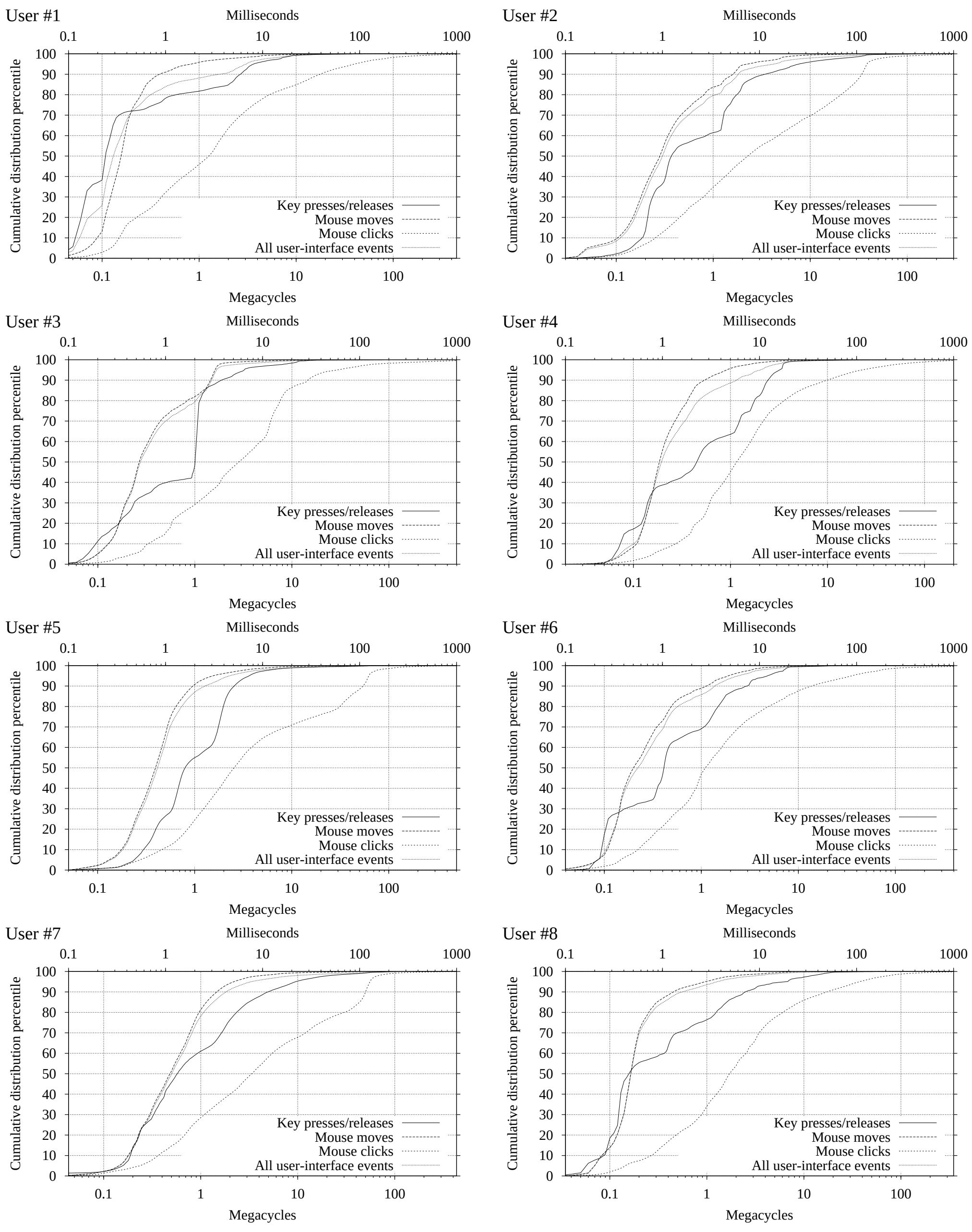

Figure V.1: The cumulative distribution function of CPU time required by various user interface event types for each user

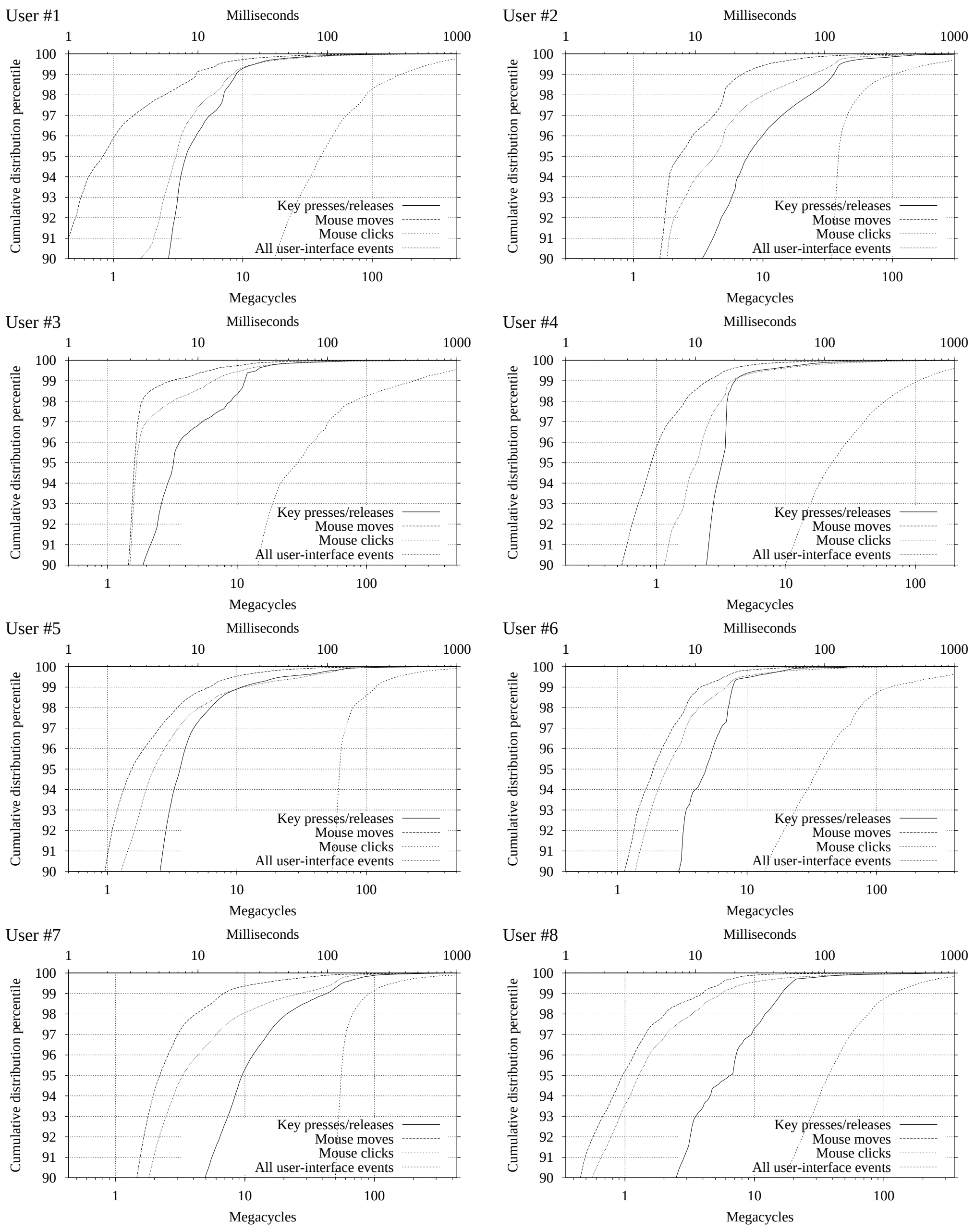

Figure V.2: The cumulative distribution function of CPU time required by various user interface event types for each user, but only above the 90th percentile

future ones will probably not have available speeds much lower than 200 MHz. In fact, this figure is extremely conservative, but is consistent with processor speeds for the machines traced. With such a speed, we will complete 10 Mc within 50 ms. For all users, this is above the 99th percentile, meaning this strategy would complete over 99% of tasks within the deadline. Note also that for all users the slope of the CDF is relatively low beyond 10 Mc of work, suggesting that the trade-off of speed for deadlines missed is worthwhile. In fact, the 99% quantiles for the various users range from 2.5–6.9 Mc, so we only need 138 MHz for this purpose. If we want to make 99.5% of all deadlines instead of 99%, we need to complete 3.5–13.3 Mc by the deadline, and we still can achieve this goal with the minimum speed as long as it is at least 266 MHz. It is not surprising that mouse movements are satisfied with low speeds, since people do not generally report poor interactive response time for mouse movement events on 133 MHz computers. We therefore recommend that mouse movements be scheduled using a simple strategy of always using the minimum available speed before the deadline.

Next, we consider key presses and releases. For this, we assume a soft deadline of 50 ms for each such task. Looking at the figures, we find an interesting dichotomy among our users. Users #1, 3, 4, 5, and 6 show relatively low CDF slopes beyond about 10.5 Mc. On the other hand, users #2, 7, and 8 show much steeper CDF slopes above 10.5 Mc. In other words, five of our users would probably not much mind if their processors ran at only 210 MHz before the deadline for all key press and release tasks, but three of our users would suffer significant changes in deadlines made and thus presumably notice the increased response time. Presumably, this signifies an important difference in the workload presented to computers by different users. Some users run applications that perform more substantial processing of keystrokes than others. This suggests that there is no single average pre-deadline speed that will work well for all users for key press and release tasks. It might be best for a DVS algorithm to monitor the percentage of deadlines made and adjust the speed if the frequency of deadlines missed gets too high. If it does not, the algorithm should at least allow the user to provide feedback about when the system appears too sluggish and to respond to such feedback by increasing the average pre-deadline speed. A DVS algorithm might do even better to consider different applications, and even different key categories within the same applications, differently; we will further examine this possibility later.

Finally, we look at the graphs for mouse click events. We will assume a 50 ms deadline for these events. With mouse clicks, we find general agreement among all users: the CDF has a relatively high slope above 10 Mc and continuing well beyond it. In other words, increasing the CPU speed pays off handsomely in reducing deadlines missed and thus improving user-perceived response time. In light of this, it seems that the average pre-deadline speed for processing mouse click events should be relatively high, although it probably does not have to be the maximum. With PACE, even a small reduction of the average pre-deadline speed can yield a large reduction in total energy consumption, as it allows us to begin the task at a low speed that may be sufficient to complete the entire task. For instance, suppose we have a CPU capable of varying between 200 and 500 MHz; if we use an average pre-deadline speed of 425 MHz, or 85% of the maximum, this theoretically allows us to start tasks running at 200 MHz for up to 12.5 ms, thereby completing 2.5 Mc of work using only the lowest speed. For user #1, this would complete 68.6% of all mouse click events at the lowest speed; for other users, it would complete between 43.8–73.9% of such events during this time. Of course, the optimal PACE schedule may not necessarily start with the lowest speed, but it will probably start with some speed significantly lower than the maximum.

In conclusion, the three types of user interface event (keystroke, mouse move, and mouse click) are different enough from each other in their task work requirements that they suggest significantly different handling by a DVS algorithm. In other words, it is quite useful for a DVS algorithm to be able to distinguish tasks at least by basic user interface type, since the best solution for each is different. Mouse movements should use the minimum available speed as its average pre-deadline speed; key presses and releases should use some intermediate speed that does not substantially increase user-perceived response time; and, mouse clicks should use a high average pre-deadline speed as they tend to require a large amount of processing time at least 10% of the time.

V.4.11.2 Differences between applications

Next, we consider how much applications vary from each other in terms of the distribution of their tasks' work requirements. Figures V.3 and V.4 show the task work distributions for key presses and releases for different applications. The same analysis is done for mouse click operations in figures V.5 and V.6. Finally, the graphs for mouse movement operations are in Figures V.7 and V.8. For ease of illustration, we only present a handful of applications, selected for having a large number of the given user interface task type.

For key press/release events, it is clear that different applications have quite different cumulative distribution functions, both in terms of their deadline performance and also their shapes. The former difference means that the effect of slowing down the processor will be felt differently by different applications, and the latter difference means that the optimal PACE schedule for one application will not work as well for another.

An interesting phenomenon with the key press/release graphs is that many CDF's have one or two "shelves," work ranges over which the CDF remains largely constant, indicating that few tasks have lengths within those ranges. The presence of one shelf suggests a vaguely bimodal shape, and the presence of two shelves suggests a vaguely trimodal shape. In other words, tasks often tend to be partitioned into a few different kinds of tasks with noticeably different lengths. We have already seen that key presses take different amounts of time from key releases, and that different key categories take different amounts of time from each other; these effects could account for these shelves. We will explore later the differences in task work distribution between key categories and between key presses and releases.

In the mouse click graphs, the we observe the following. First, different applications have quite different cumulative distribution functions, so as with key presses different speeds will have different effects on different applications, and PACE will find different schedules for different applications if it considers them separately. Second, far fewer "shelves" are present in these CDF's, suggesting less modal behavior.

If we examine where the "knees" in the CDF curves are, we can estimate at what average pre-deadline speed the user's experience with the application's mouse click response time will near saturation. Only vern, for user #4, shows saturation at 10 Mc, suggesting that this is the only application for which a user might be satisfied with a pre-deadline speed of 200 MHz when the CPU is capable of faster speeds. Other applications have higher apparent saturation points, but the same application does not necessarily have the same saturation point for different users. For instance, iexplore for user #4 seems to saturate at about 30 Mc, corresponding to a satisfactory speed of 600 MHz, while iexplore for user #6 saturates at about 50 Mc, corresponding to a satisfactory speed of 1 GHz. Note that our estimates of

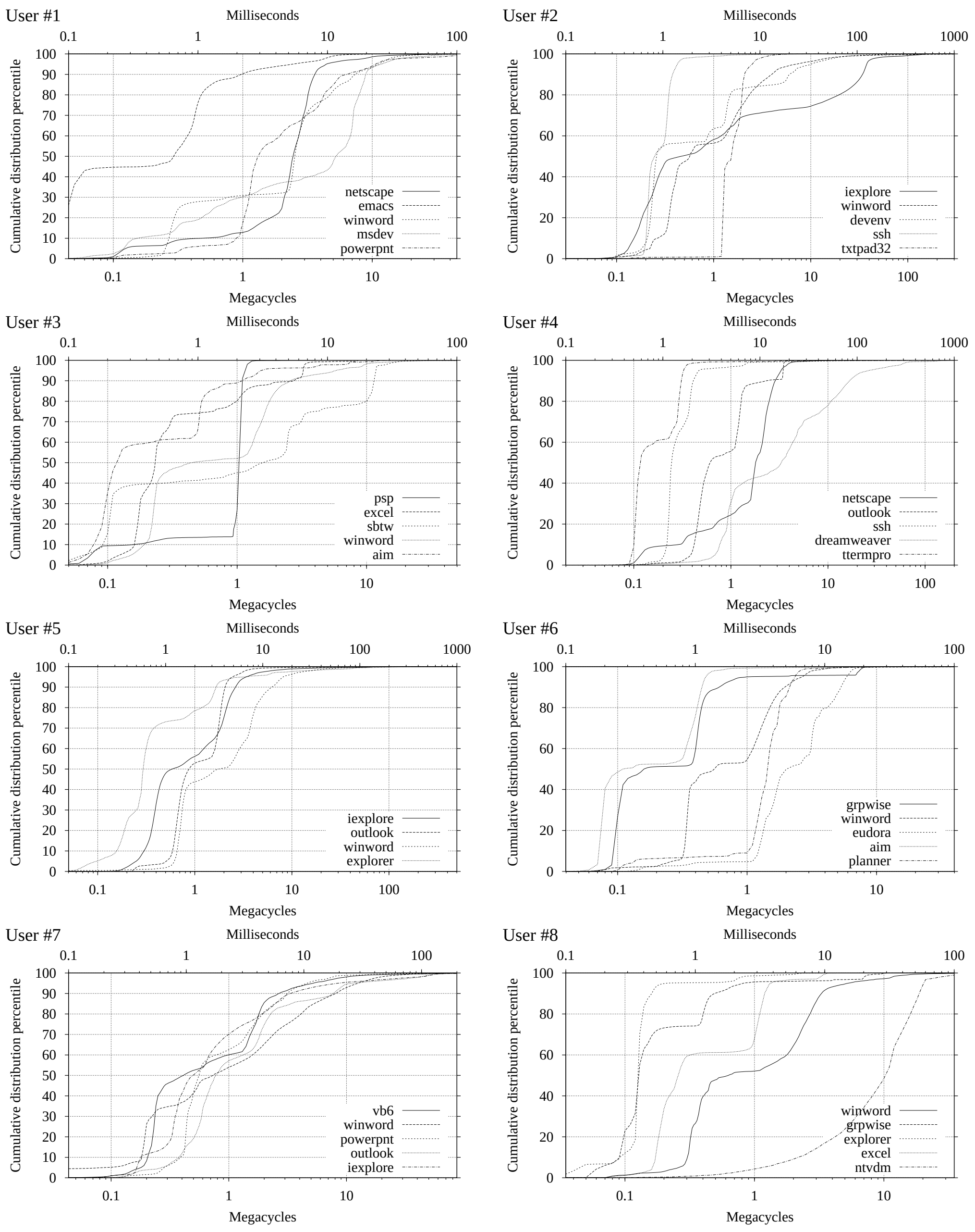

Figure V.3: The cumulative distribution function of CPU time required by key press and release events for different applications for each user

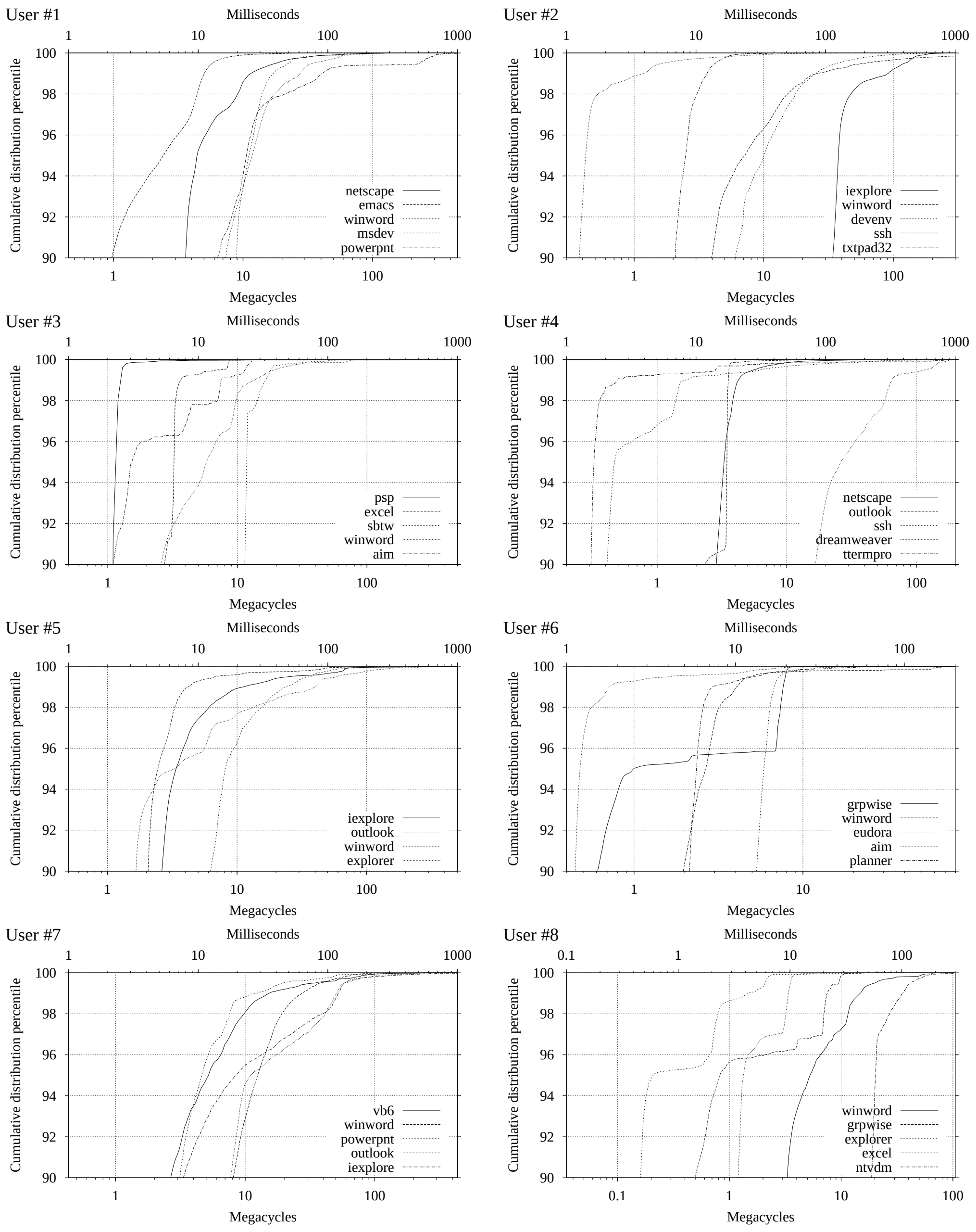

Figure V.4: The cumulative distribution function of CPU time required by key press and release events for different applications for each user, but only above the 90th percentile

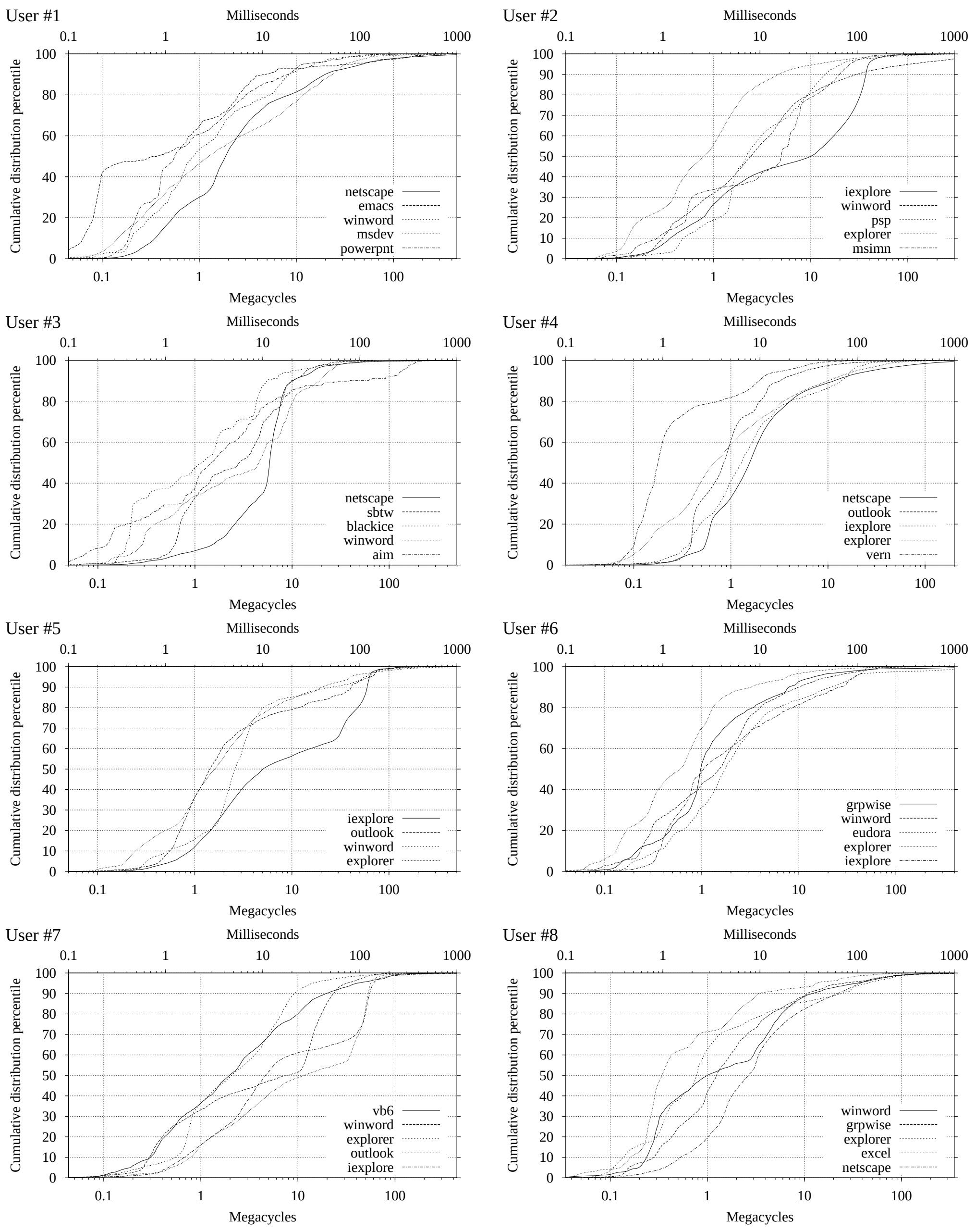

Figure V.5: The cumulative distribution function of CPU time required by mouse click events for different applications for each user

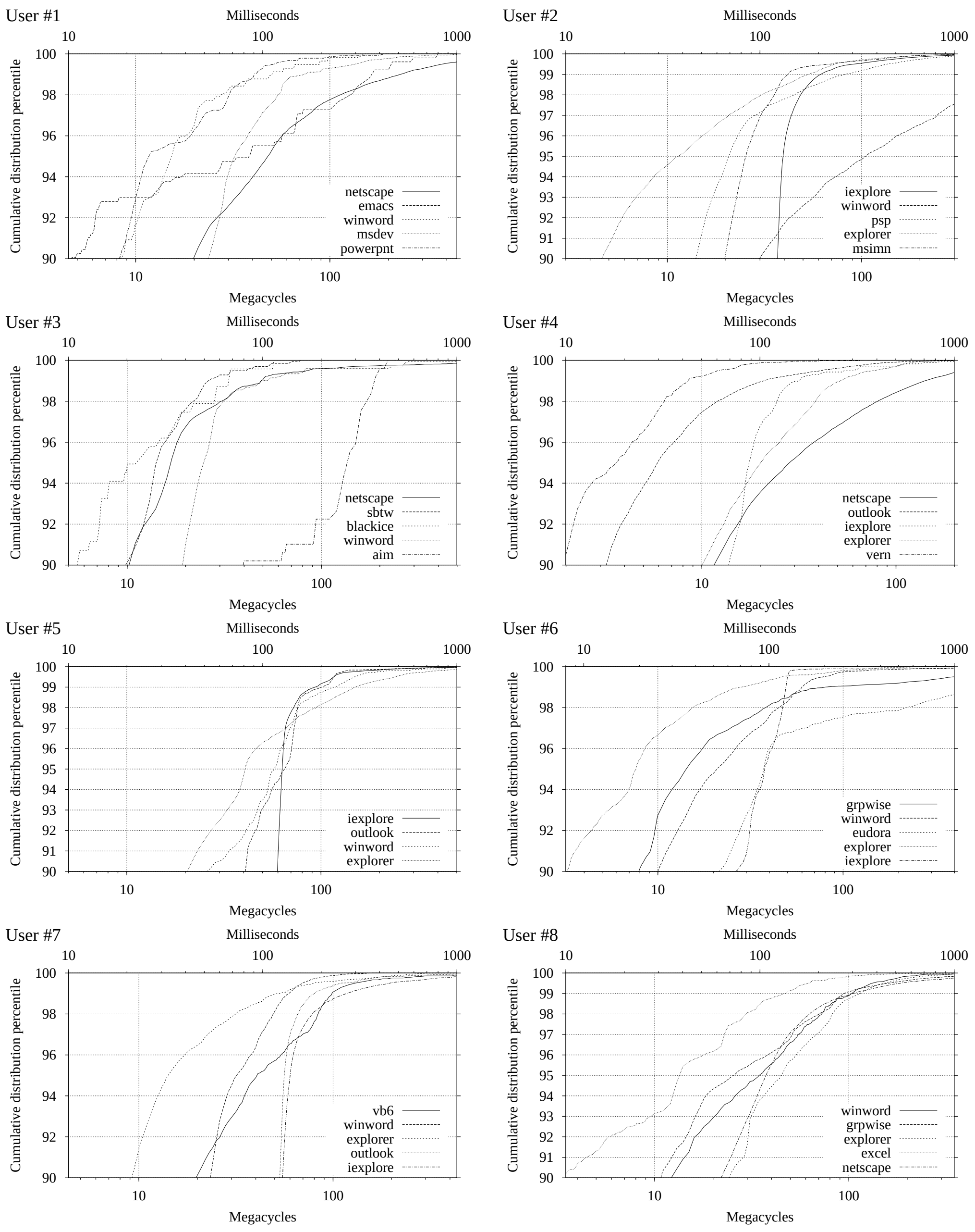

Figure V.6: The cumulative distribution function of CPU time required by mouse click events for different applications for each user, but only above the 90th percentile

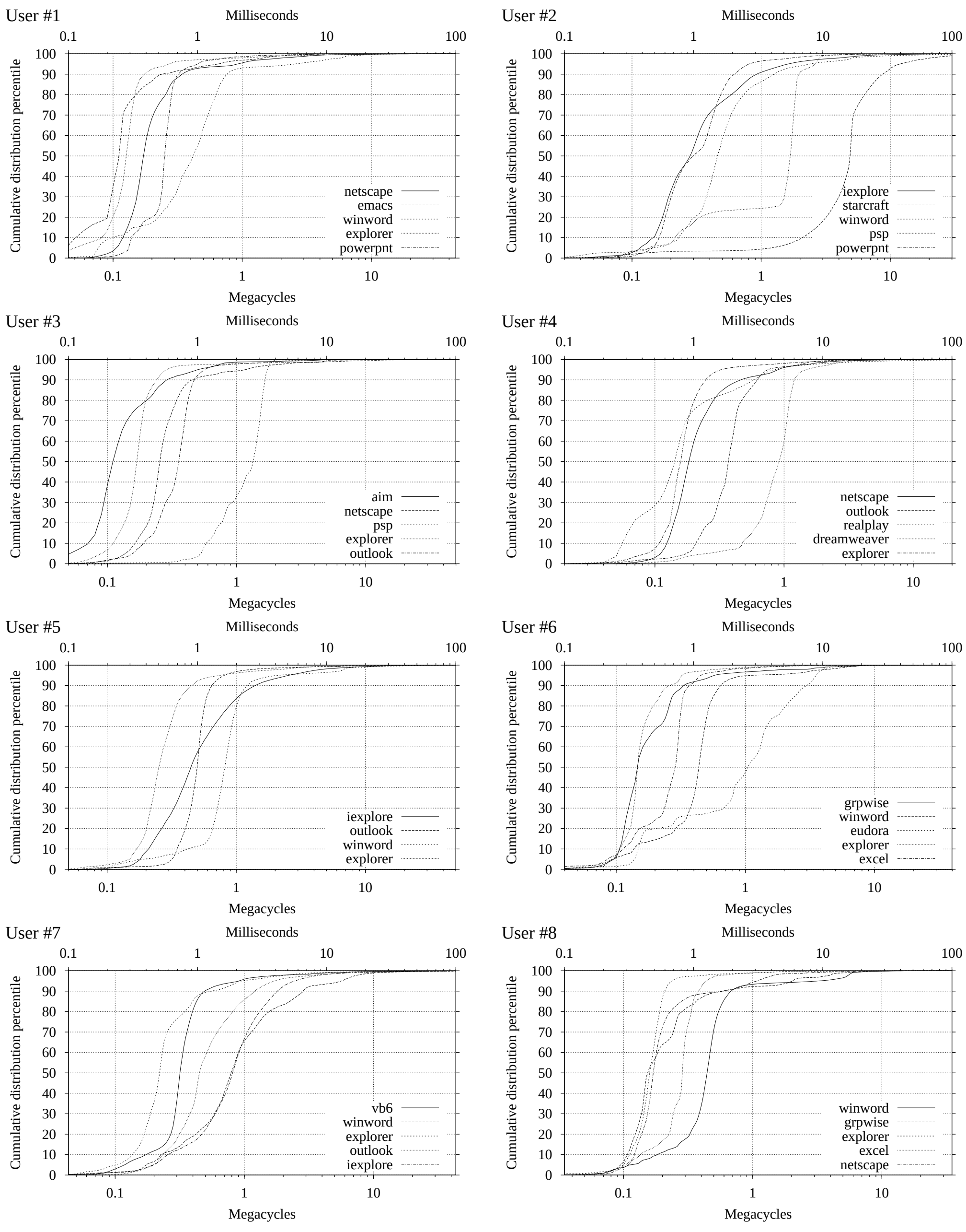

Figure V.7: The cumulative distribution function of CPU time required by mouse move events for different applications for each user

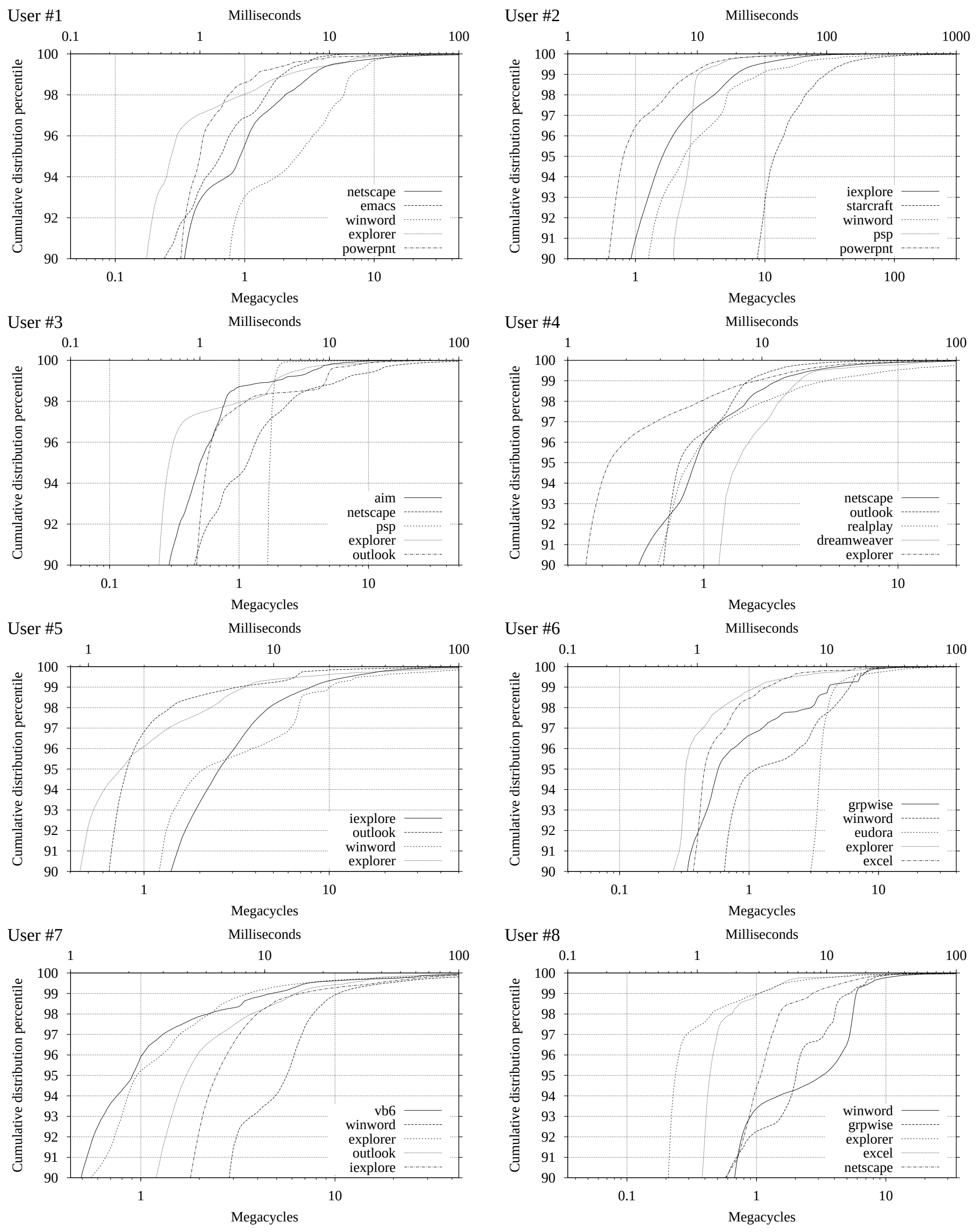

Figure V.8: The cumulative distribution function of CPU time required by mouse move events for different applications for each user, but only above the 90th percentile

saturation speeds are just estimates, and may be inaccurate due to misestimation of the true response time cutoff and/or the slope at which users become satisfied. However, it is clear that no matter what method is used for estimating a reasonable speed, different applications will give different results, as will the same application run by different users since different users make different demands on the same application.

Mouse movements show different distributions for different applications, but almost all of them show saturation before 10 Mc, suggesting that using the minimum speed would satisfy all these applications. The only exception is starcraft for user #2, which shows high sensitivity until about 35 Mc, suggesting that the response time of this computer game to mouse movements gains substantively up to a speed of about 700 MHz. A DVS algorithm could monitor mouse movement response time to watch for cases like these, or it could count on users playing computer games to turn off dynamic voltage scaling when it interferes with mouse movement response time.

V.4.11.3 Mouse click categories

Next, we investigate how the different mouse click categories compare to each other in terms of the task work distribution. For example, we consider how pushing the left mouse button down differs from releasing the left mouse button so that it goes up, and how pushing the left mouse button down differs from pushing the right mouse button down. To eliminate the confounding effect of differences between applications, we present task work distributions for different mouse click categories within the same application. Figure V.9 shows these distributions for each user running the same application, explorer. We chose explorer since it is the Windows file system desktop, so all users use it. Figure V.10 shows the distributions for various other applications for those users. To focus on the parts of these graphs above the 90th percentile, see Section B.11.1.

We find that for each application, different click categories tend to show quite different task work distributions, both in terms of their central tendencies and their shape. Thus, it appears useful for PACE to consider different click categories differently when estimating task work distributions.

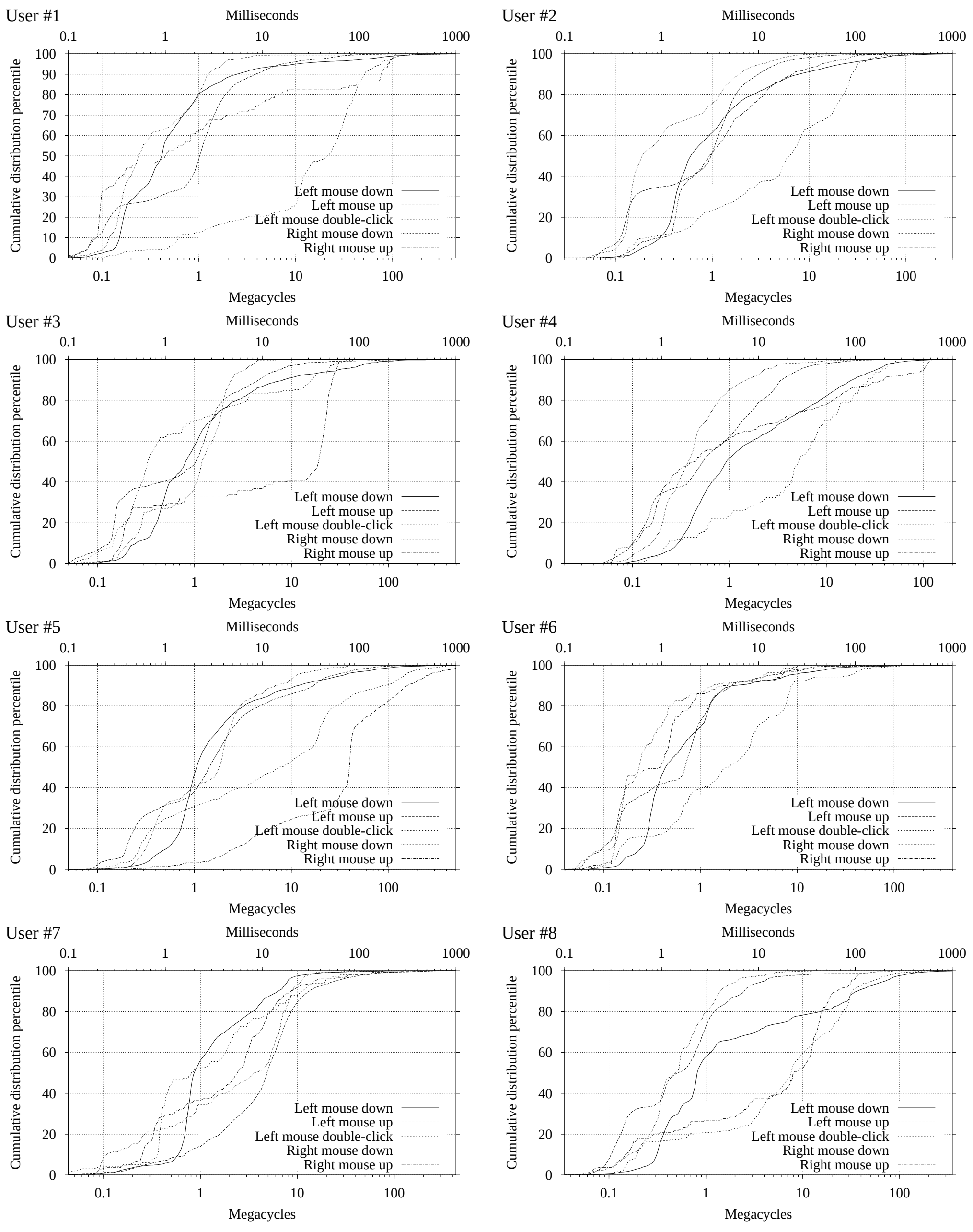

Figure V.9: For each user, the cumulative distribution function of CPU time required by explorer for mouse clicks of different categories

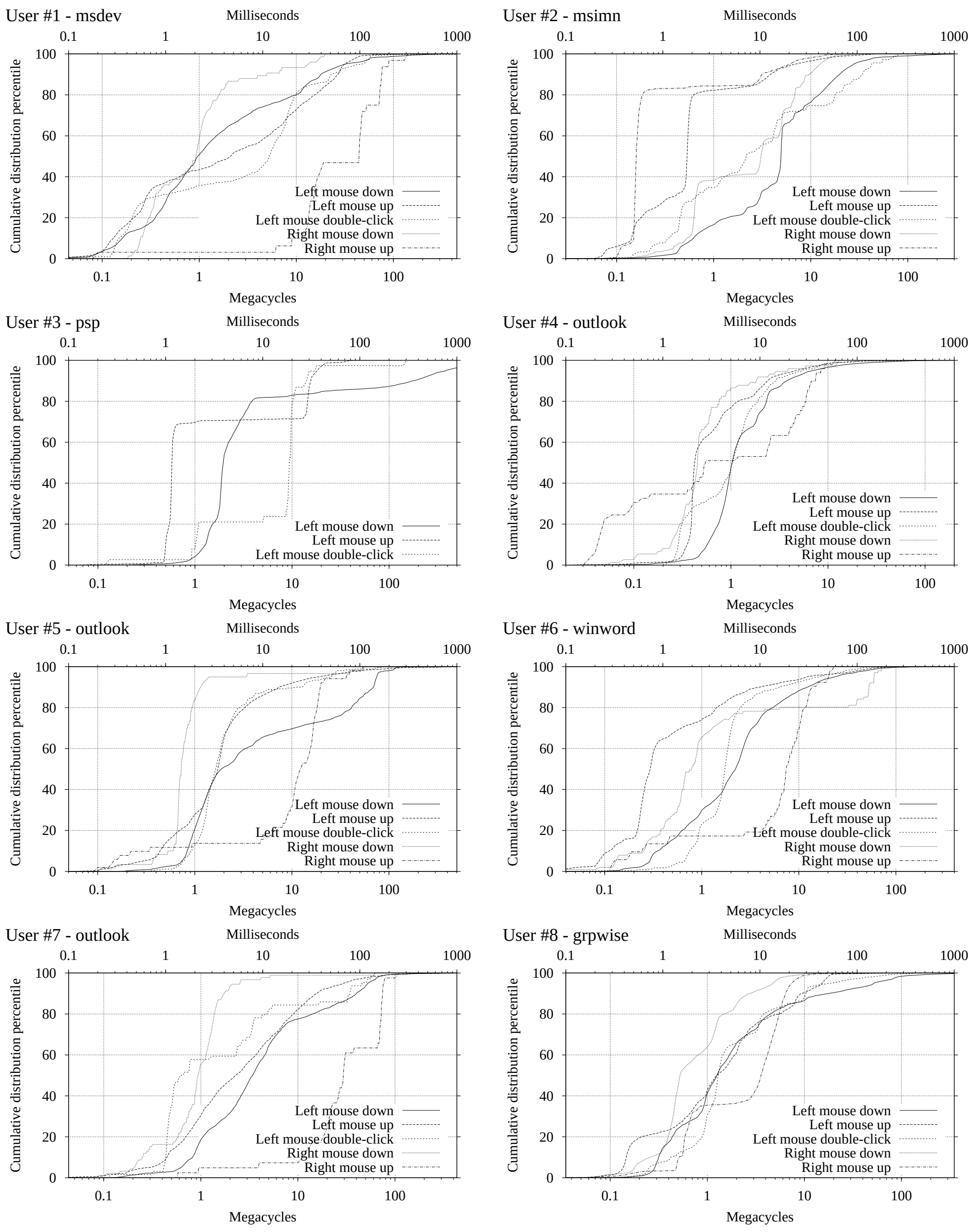

Figure V.10: For various applications and users, the cumulative distribution function of CPU time required by different categories of mouse click events

V.4.11.4 Key press vs. release

Next, we investigate how the different key press and release event categories compare to each other in terms of the task work distribution. First, we consider whether there is a substantial difference between key press and release events. Figure V.11 shows distributions for key up and key down events' task work.

We note that for all users, key presses take more time on average than key releases. Thus, it seems clear that key presses should be treated separately from key releases, at least by using separate PACE schedules and perhaps even by using different pre-deadline speed policies.

V.4.11.5 Key press categories

Now, we consider whether there is noticeable difference between different categories of keystrokes. First, we will focus on key presses rather than releases, since they take slightly more time. Figure V.12 shows the effect of differences in key category for various applications and users. In each graph, we restrict consideration to a single user and application, since we well know that there is significant variation from one application to another and one user to another. For graphs focusing on the parts of these cumulative distribution functions above the 90th percentile, see Section B.11.2; that section also also contains similar graphs for additional applications.

We observe that within an application, there is typically a noticeable difference from one key category to another. However, some applications show a great deal of similarity between certain sets of event categories. For instance, winword generally shows most key categories shown having similar distributions, except for enter. Grpwise shows similar behavior for the different key categories other than arrow and enter. This suggests that a DVS algorithm attempting to divide key presses into classes with similar characteristics to better predict their distributions may in some cases do well to combine different key categories in the same class. If an efficient clustering algorithm can be devised, it might help increase the sample size for each cluster, thus improving the estimation of task work distributions and thereby the efficacy of the computed PACE schedule.

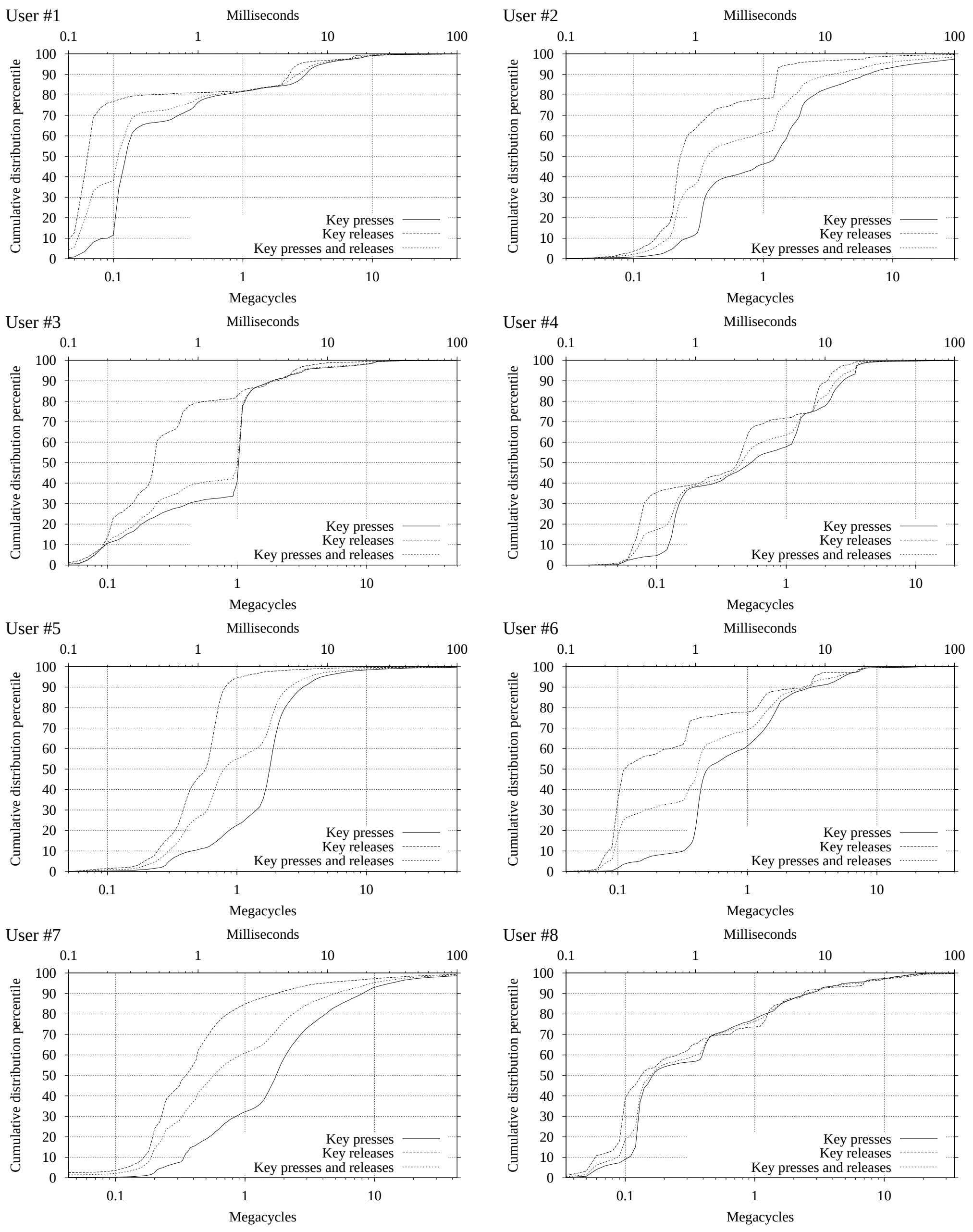

Figure V.11: For each user, the cumulative distribution function of CPU time required by key presses and releases

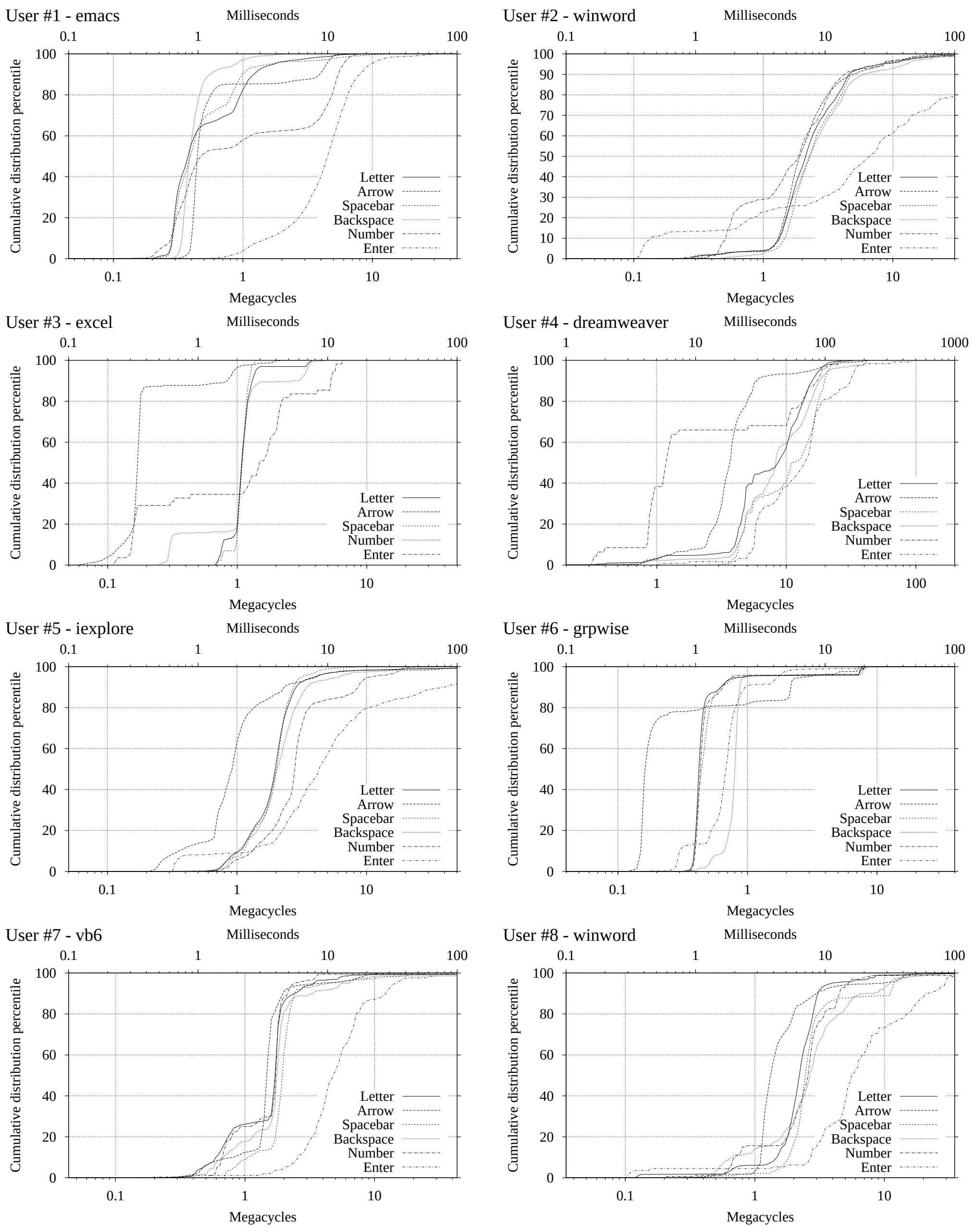

Figure V.12: The cumulative distribution function of CPU time required by different categories of keypress events for various applications and users

V.4.12 Application runs

When examining traces of user interface events, we observe that consecutive events tend to be destined for the same application. This is expected, since typically users use one application's user interface for a while before switching to another application's user interface. Now, recall that different applications have different task work distributions, so a DVS algorithm should, when estimating the task work distribution for a newly arrived task, consider only information about recent past tasks from that application. However, if users use a single application's user interface for a sufficiently long time, then most recent past tasks will tend to be from the same application as the current task. Thus, it may be reasonable for a DVS algorithm to effectively estimate task work distributions just by looking at all recent tasks, rather than making special effort to separate them by application. Thus, in this section, we will examine how long users use a user interface from the same application.

To do this, we divide a trace of user interface events into *runs*, each of which is a maximal-length consecutive sequence of events of some common type and/or category that belong to the same application. So, for instance, if a DVS algorithm makes a distinction between key events and mouse clicks into account, but not between tasks from different applications, it is relevant to consider the lengths of runs of consecutive key events and of consecutive mouse click events. If a DVS algorithm makes a distinction between keys of different categories, it is relevant to consider, for example, the lengths of runs of consecutive spacebar presses that belong to the same application.

Since we saw in Chapter IV that DVS algorithms do well at estimating task work distributions when looking at the last 28 or so events, we posit that it is reasonable to ignore application information when run lengths are frequently higher than 60 or so.

Figure V.13 shows the distributions of run lengths of different event classifications for various users. Table V.16 summarizes the average run lengths for these distributions.

We see that mouse click events do not have very high run lengths, with averages around 14. So, mouse click events are unlikely to be an event type for which application information can be ignored.

In contrast, key events have rather high average run lengths, with the average run length for most users over 100. However, when we look at the distributions we see that these

User #	Key	Letter press	Enter press	Mouse click	Left mouse down
1	210.3	180.5	21.4	8.2	4.9
2	110.7	100.4	8.3	14.6	8.1
3	133.4	31.6	4.7	10.5	6.1
4	301.5	160.1	17.6	13.2	8.3
5	236.7	125.9	6.9	12.0	6.3
6	409.2	210.7	7.1	13.5	7.8
7	54.3	50.1	11.5	13.4	7.7
8	269.0	59.3	8.9	21.8	14.3
Avg	215.6	114.8	10.8	13.4	8.0

Table V.16: Mean run length distributions for all users using various event classifications

high averages are only due to long tails in the run length distributions, and that high run lengths do not occur very frequently. We see that for all users, more than 46% of all run lengths are shorter than 60. So, key events are also unlikely to be an event type for which application information can be ignored.

Looking at the more specific events, such as letter press, enter press, and left mouse down, we largely see the same phenomenon of low run lengths. Interestingly, letter press shows the highest run lengths, presumably because when people type letters they tend to type several in a row to the same application. However, even for this key category, more than 46% of all run lengths are shorter than 60. So, it appears application run lengths for specific event categories are not long enough frequently enough for us to conclude that a DVS algorithm could ignore application information.

V.5 DVS algorithms

In this section, we will evaluate how our DVS techniques work on the given workloads. In the simulations here, we will assume a CPU capable of DVS with dynamic range between 200 MHz and 600 MHz. We assume that power consumption is proportional to the cube of the speed, with peak power consumption of 3 W at 600 MHz. This range is similar to that of the first AMD chip with DVS, and has a maximum CPU speed similar to that of the users traced in these workloads.

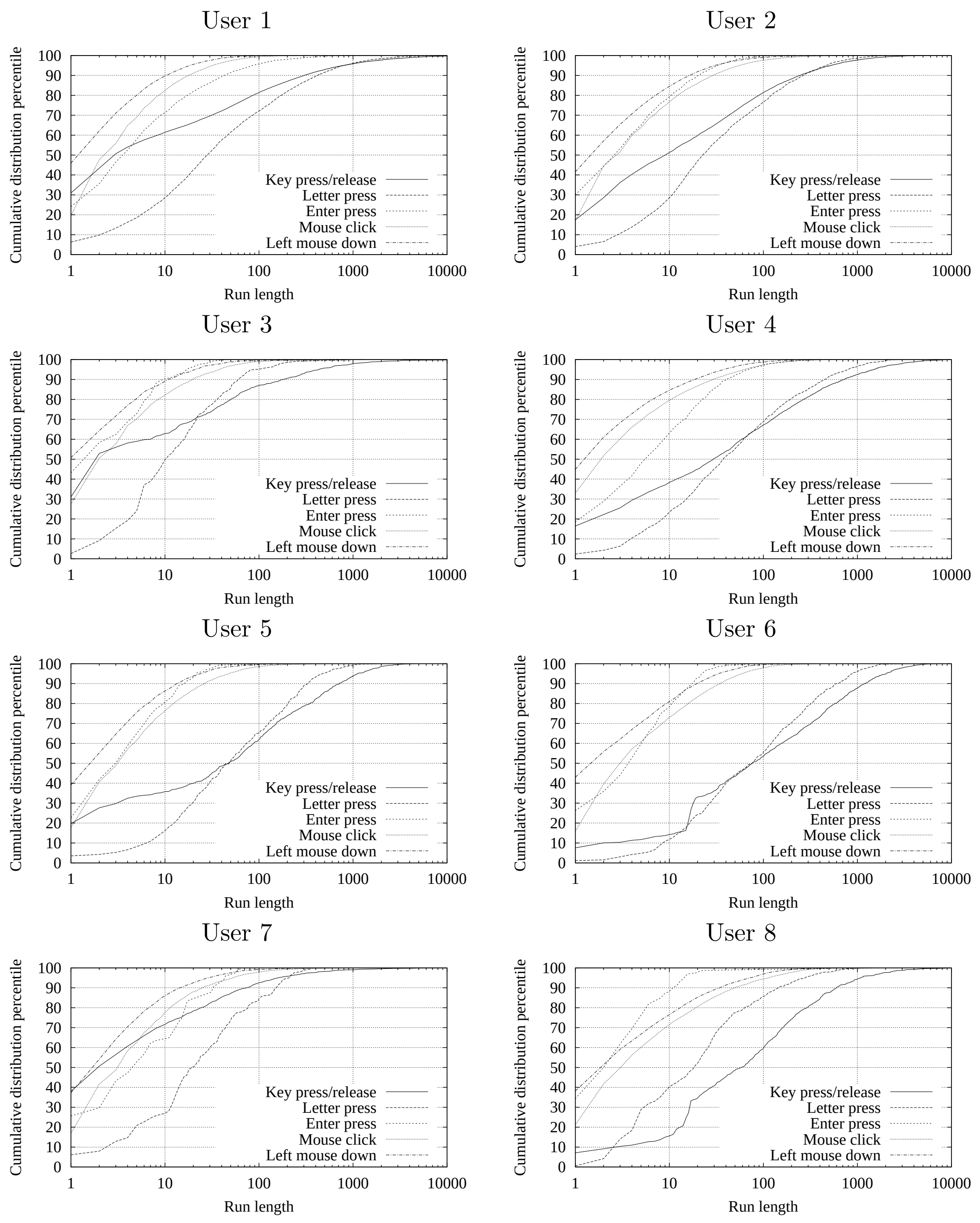

Figure V.13: Run length distributions for various users and various event classifications

User	Average energy without DVS	Average energy with DVS	% of possible deadlines made with DVS	Energy savings from DVS
1	2.054 mJ	0.651 mJ	99.8%	68.3%
2	3.955 mJ	0.989 mJ	99.6%	75.0%
3	3.101 mJ	0.646 mJ	99.8%	79.2%
4	1.690 mJ	0.272 mJ	99.9%	83.9%
5	3.480 mJ	0.776 mJ	99.7%	77.7%
6	2.568 mJ	0.410 mJ	99.9%	84.0%
7	4.548 mJ	1.096 mJ	99.6%	75.9%
8	1.763 mJ	0.378 mJ	99.9%	78.5%
Avg	2.895 mJ	0.653 mJ	99.8%	77.5%

Table V.17: Results from using suggested DVS algorithm on mouse movements

V.5.1 Mouse movements

For mouse movements, we observed that 200 MHz provides sufficient processing speed to meet a reasonable number of deadlines. We therefore recommended using the minimum available speed for the entire pre-deadline portion of the schedule, regardless of what application receives the mouse movement event. We use the maximum available speed after a deadline of 50 ms. Table V.17 gives results from this experiment.

We see that substantial energy, on average 77.5%, can be saved by observing that mouse movements only require the lowest speed, at least until the deadline, and setting the speed accordingly. It is thus useful to separate out mouse movements from other tasks to enable these large energy savings.

V.5.2 Key press/releases

We consider various possible DVS algorithms to evaluate our hypotheses about what task classifications DVS algorithms should consider as well as to evaluate the effectiveness of PACE on these workloads. For fair comparison among these algorithms, we will use the same PDC for all of them, namely one corresponding to a 400 MHz average pre-deadline speed. Although we recommend using a PDC tailored to each task for key press and release events, due to the different saturation points for different such tasks, we use a constant PDC in these experiments to provide a fair comparison between algorithms that take distinctions among task differences into account and those that do not. Algorithms that do not distinguish between tasks cannot provide a different pre-deadline speed to different ones. Also, for fairness, we use the same post-deadline algorithm for all algorithms: one that keeps the speed constant at its maximum, 600 MHz.

The six algorithms we consider are:

- **Flat.** The pre-deadline speed is constant at 400 MHz.
- **Stepped.** The pre-deadline speed begins at 200 MHz and is incremented by 100 MHz every 10 ms. This models algorithms such as that used by Transmeta's LongRun$^{\mathrm{TM}}$.
- **Past/Peg.** The pre-deadline speed is constant at 200 MHz for the first 25 ms, then is pegged to 600 MHz. This models the algorithm suggested by Grunwald et al. [GLF^{+}00].
- **PACE-NoClassify.** The pre-deadline speed schedule is computed by PACE using an estimate of task work distribution derived from the most recent key-triggered tasks, regardless of what category they are or what application they belong to.
- **PACE-NoApps.** The pre-deadline speed schedule for each task is computed by PACE using an estimate of task work distribution derived from the most recent key-triggered tasks of the same category (e.g., spacebar presses), regardless of what application they belong to.
- **PACE-Classify.** The pre-deadline speed schedule for each task is computed by PACE using an estimate of task work distribution derived from the most recent key-triggered tasks of the same category (e.g., spacebar presses) and same application.

Note that since all these algorithms have the same PDC and the same post-deadline part, all results will be the same except for pre-deadline energy. Thus, one can compare them by just looking at average per-task energy consumption. Figure V.14 gives average per-task energy consumption for each algorithm for each user, as well as the average for all users for each algorithm.

First of all, among the non-PACE algorithms we find there is a nearly universal pattern among all users. The Flat algorithm is always the worst, followed typically by the

Past/Peg, followed by Stepped. On average, using the Past/Peg algorithm reduces energy consumption by 22.8% relative to Flat, and using the Stepped algorithm reduces energy consumption by 10.5% relative to Past/Peg. Thus, without PACE, we suggest using the Stepped algorithm instead of these two others.

Unsurprisingly, all PACE algorithms reduce energy consumption relative to the best of the non-PACE algorithms. We find that PACE-NoClassify reduces energy consumption by 5.0% relative to Stepped on average, indicating PACE does reasonably well even without separating keystroke events into separate categories. PACE-Classify reduces energy consumption by 1.5% relative to PACE-NoClassify, indicating that it is (to a small extent) useful, and not detrimental, to separate keys by category and by application. On the other hand, PACE-NoApps, which separates keystroke events into categories but ignores application information always does worse than PACE-Classify and even does worse than PACE-NoClassify for three out of the eight users, supporting our conclusion in subsection V.4.12 that application run lengths are not long enough to reasonably ignore application information.

The fact that PACE improves when taking into account the application and category to which each key event belongs was not a foregone conclusion. If key events of different applications and categories were not sufficiently different from each other to have significantly different distributions, we would expect PACE to actually do worse by keeping tasks of different categories separate. This is because if PACE must infer a work distribution from a small set of nearly identical tasks rather than a larger set of merely similar tasks, it has fewer sample points to choose from, and thus must either use smaller sample sizes or rely on older information. The fact that PACE improves even using such older information indicates that the difference in distribution between different applications and task categories is significant.

Note that the specific percentage differences between energy consumption are likely to vary with processor characteristics and with choice of PDC, so they are less relevant than the trends we observed here.

In conclusion, we find that the Stepped algorithm gives a rather good general schedule for keystroke tasks, working best among the non-PACE algorithms we considered and only consuming 6.9% more energy than the best PACE-modified algorithm. PACE is better than all non-PACE algorithms, as expected, and this improvement is best when it separates keystroke events by category and application. We find that ignoring application information and using just category information is not useful for PACE.

V.5.3 Mouse clicks

We next consider the effectiveness of various possible DVS algorithms on mouse click events to evaluate our hypotheses about what task categories DVS algorithms should consider as well as to evaluate the effectiveness of PACE on these workloads. For fair comparison among these algorithms, we will use the same PDC for all of them, namely one corresponding to a 500 MHz average pre-deadline speed. Also, for fairness, we use the same post-deadline algorithm for all algorithms: one that keeps the speed constant at its maximum, 600 MHz.

The six algorithms we consider are:

- **Flat.** The pre-deadline speed is constant at 500 MHz.
- **Stepped.** The pre-deadline speed begins at 200 MHz and is incremented by 100 MHz every 5 ms. This models algorithms such as that used by Transmeta's LongRun$^{\text{TM}}$.
- **Past/Peg.** The pre-deadline speed is constant at 200 MHz for the first 12.5 ms, then is pegged to 600 MHz. This models the algorithm suggested by Grunwald et al. [GLF$^+$00].
- **PACE-NoClassify.** The pre-deadline speed schedule is computed by PACE using an estimate of task work distribution derived from the most recent mouse click-triggered tasks, regardless of what category they are or what application they belong to.
- **PACE-NoApps.** The pre-deadline speed schedule for each task is computed by PACE using an estimate of task work distribution derived from the most recent mouse click-triggered tasks of the same category (e.g., left mouse up), regardless of what application they belong to.
- **PACE-Classify.** The pre-deadline speed schedule for each task is computed by PACE using an estimate of task work distribution derived from the most recent key-triggered tasks of the same category (e.g., left mouse up) and same application.

Note that since all these algorithms have the same PDC and the same post-deadline part, all results will be the same except for pre-deadline energy. Thus, one can compare them

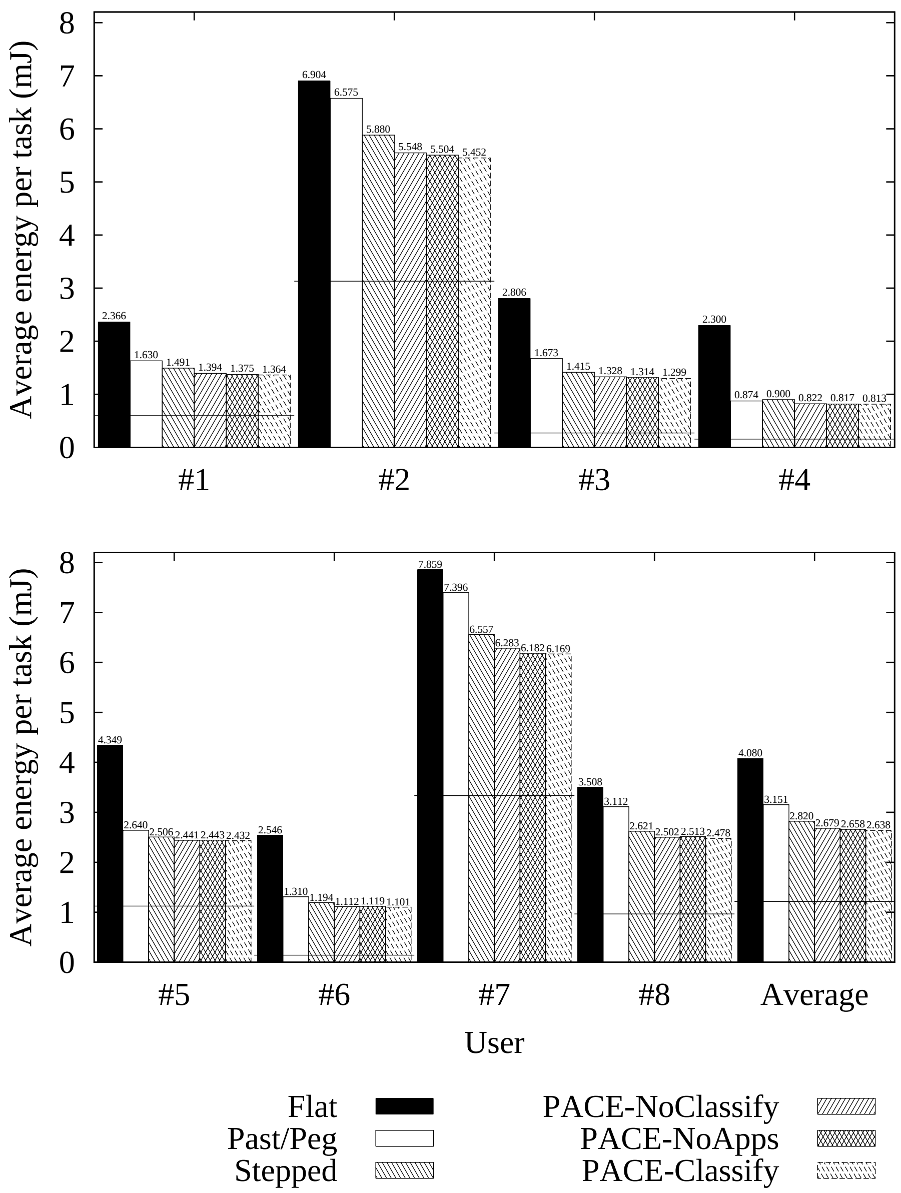

Figure V.14: These graphs show average per-task energy consumption for various DVS algorithms operating on various users' keystroke events. The horizontal lines show post-deadline energy consumption.

by just looking at average per-task energy consumption. Figure V.15 gives average per-task energy consumption for each algorithm for each user, as well as the average for all users for each algorithm.

We find another universal pattern among all users, but for mouse clicks it is somewhat different than for keystrokes. Here, the Past/Peg algorithm is always the worst, followed by Flat, followed by Stepped. On average, using the Flat algorithm reduces energy consumption by 0.7% relative to Past/Peg, and using the Stepped algorithm reduces energy consumption by 3.4% relative to Flat. Thus, without PACE, we suggest using the Stepped algorithm instead of these two others, just as we do for keystroke events.

Unsurprisingly, all PACE algorithms reduce energy consumption relative to the best of the non-PACE algorithms. We find that PACE-NoClassify reduces energy consumption by 1.5% relative to Stepped on average, indicating PACE does reasonably well even without separating mouse click events into separate categories. PACE-Classify reduces energy consumption by 0.5% relative to PACE-NoClassify, indicating that it is useful, albeit marginally, and not detrimental, to separate mouse clicks by category and by application. On the other hand, PACE-NoApps, which separates mouse click events into categories but ignores application information always does worse than PACE-Classify, and only improves over PACE-NoClassify by an average of 0.2%, meaning that we lose about 60% of the effectiveness of subtyping by ignoring application information. This supports our conclusion in subsection V.4.12 that application run lengths are not long enough to reasonably ignore application information.

In conclusion, we find that the Stepped algorithm gives a rather good general schedule for mouse click tasks, working best among the non-PACE algorithms we considered and only consuming 2.0% more energy than the best PACE-modified algorithm. PACE is better than all non-PACE algorithms, as expected, and this improvement is best when it separates mouse click events by category and application. We find that ignoring application information and using just category information is not useful for PACE for mouse click events. The improvement of PACE over Stepped is marginal enough that if PACE cannot be efficiently implemented it is probably not worthwhile for mouse clicks. However, this may or may not be the case for different processor characteristics.

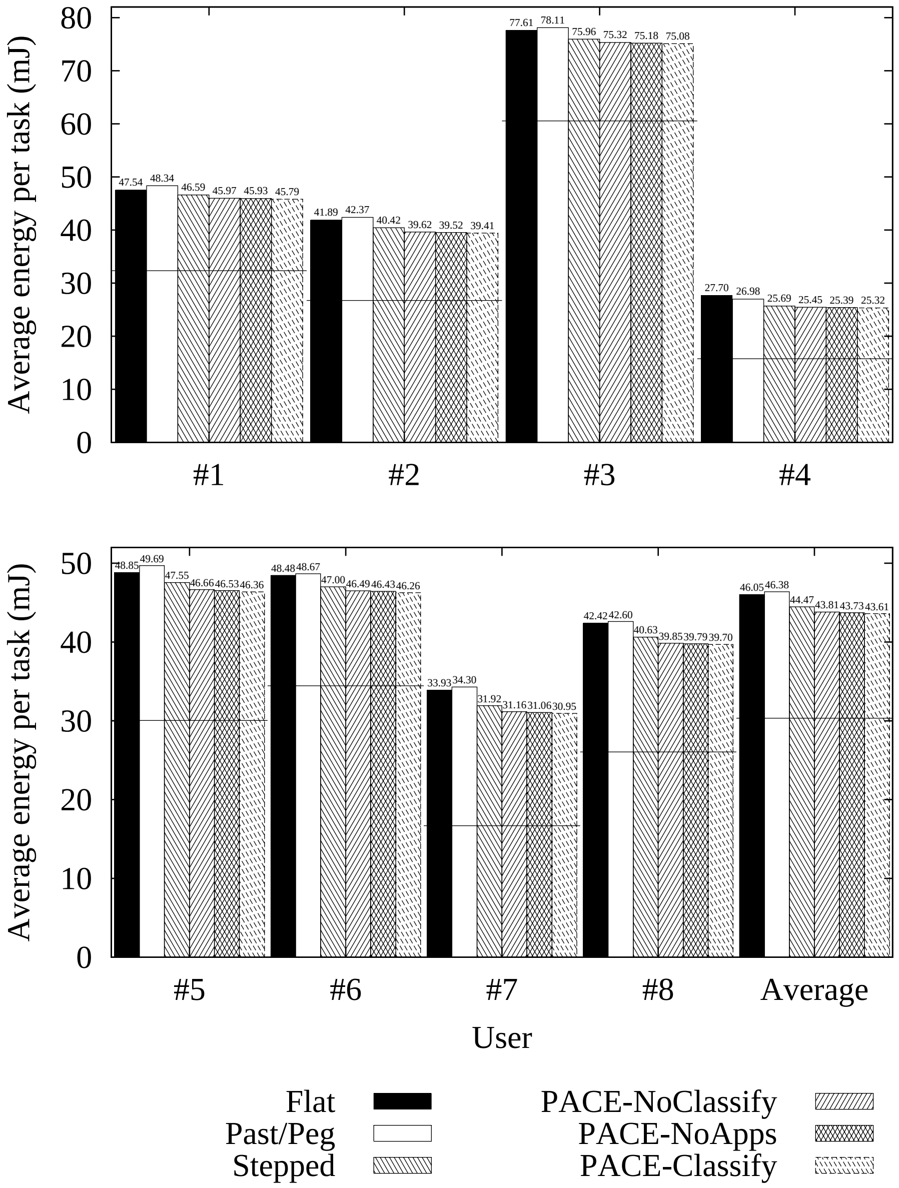

Figure V.15: These graphs show average per-task energy consumption for various DVS algorithms operating on various users' mouse click events. The horizontal lines show post-deadline energy consumption.

V.6 Conclusions

In this chapter, we analyzed months-long traces of user operations in order to draw conclusions about how to design DVS algorithms. Our main concerns are whether DVS algorithms can infer the occurrence of tasks triggered by user interface events, and how they should schedule processor speed to effectively trade off responsiveness and energy savings for those tasks that they can detect.

Observing overall characteristics of the traces, we learned the following. Overall, most of the time users' processors are idle, suggesting that DVS can save substantial energy by slowing down the processor and decreasing energy consumption. Most users run hundreds of applications on their machines over the course of months, but most of the CPU time is spent on only a few applications. The applications responsible for most CPU time usually have user interface components, such as web browser windows.

On average, we can attribute only 20.3% of CPU time to processing user interface events. This is largely because 35.1% of CPU time is spent responding to timer events instead of working directly on a user interface event. In addition, we were unable to accurately detect the cause of 21.9% of CPU time. For the remainder of our analyses, we focus on the characteristics of tasks, triggered by user interface events, that DVS algorithms can detect.

User interface events fall into three broad categories: keystroke, mouse click, and mouse movement. Mouse clicks account for the smallest percentage of such events, 1.3–4.0%, but account for the most CPU time spent processing such events, 25.4–54.6%. Keystrokes account for the next smallest percentage of events, 3.0–51.6%, and account for 4.4–46.3% of CPU time. Finally, mouse movements are quite frequent, accounting for 47.1–94.8% of all events, but they take little time, in aggregate 28.4–61.2% of all time. These wide ranges indicate the broad differences between different users' patterns of user interface usage; we also find that users differ in this way even when using the same application.

Since most techniques for inferring the ends of user interface tasks are complex and time-consuming to implement, we evaluated the possibility of considering a user interface task complete when the system is idle with no I/O or when the next user interface task arrives for the same application. We found that, excepting a couple of applications for which this technique may work badly, this technique is rather accurate, prematurely terminating fewer than 0.8–5.3% of all tasks. More research is needed to determine whether the misbehaving applications can be detected in some way, or whether their apparent misbehavior is only an artifact of our heuristics for identifying the ends of tasks.

When a task requires I/O, a DVS algorithm must treat it specially. We find that I/O occurred in only 0.3–3.5% of all tasks for the various users. If network I/O is ignored, as would be the case for a laptop running on batteries and disconnected from any wireless network, the percentage is much lower. One caveat is that mouse clicks require I/O far more often than the average user interface event, specifically 5.1–15.5% of the time, so DVS algorithms should probably leave spare time in their schedules for I/O when producing schedules for tasks triggered by mouse clicks.

Examining mouse clicks in more detail, we find that almost all such events, 87.9–96.0%, are due to presses or releases of the left mouse button. These account for 80.2–94.8% of the CPU time due to mouse clicks. Thus, DVS algorithms intending to treat mouse clicks differently based on the category of click can account for most such clicks by considering only these two events. If we also include left mouse double-clicks and right mouse presses and releases, we can account for even more, all but 0.9% of all mouse click events and all but 0.2% of all CPU time spent on mouse clicks.

We then considered 24 different categories of keys, including such categories as letter, punctuation, and arrows. We found that the ten most frequently used categories are letter, modifiers alone, arrow, spacebar, Eastern-language character, backspace, ctrl-letter, punctuation, number, and enter. However, though these account for 92.1% of all key events, they only account for 83.5% of CPU time spent on key events, as four other less frequent categories (del, tab, page up/down, and F-keys) account for a disproportionately large amount of CPU time. We suggest a DVS algorithm consider at least these fourteen categories when separating key tasks by category.

Next, we evaluated to what extent different event categories trigger tasks of significantly different average duration. We find, for key press categories, that pairs of categories tend to do so at a 99% confidence level, and found no pair of categories that never does so. For key release categories, pairs tend to show significant differences far less often, though no pair consistently shows no significant differences. For mouse click categories, we see surprisingly few pairs of categories showing significant differences at even the 90% level. However, this seems to be largely due to the high variance of these tasks' durations, greatly increasing the likelihood that if a difference exists we would not be able to detect it with high confidence.

We also evaluated to what extent different applications show significant differences in the average length of their tasks. We find that, for all three user interface event types, they frequently show differences significant enough to permit 99% confidence in their difference.

After this, we examined task work distributions to evaluate how DVS algorithms should choose pre-deadline speeds and to further evaluate how different task work distributions of different event types and categories are from each other. The first conclusions we drew were that different event types should use different approaches to choosing pre-deadline speeds. Mouse movements generally require just the minimum available speed. Keystrokes can require higher speeds though the specific speed is a function of the user, application, and key category. Finally, mouse clicks require a high pre-deadline speed for the user to be satisfied with the trade-off between response time and CPU speed and energy.

Observations of task work distributions for different applications and different categories confirmed what we found earlier, that differences in application and in category can have substantial effect on the distribution of task work. This means that DVS algorithms should do well to separate tasks by category and by application in both predicting task work distribution and in choosing PDC values.

We next considered the phenomenon of application runs, meaning the tendency of consecutive user interface events to belong to the same application. Sufficiently long application runs would render it less important for a DVS algorithm to keep separate track of different applications' tasks, since recent tasks would automatically tend to belong to the same application. However, we found that application runs are not long enough frequently enough for this to seem tenable.

Finally, we performed a limited set of experiments with DVS algorithms to evaluate in practice our suggestions for DVS algorithms. For mouse movements, we find that our suggestion to use the minimum speed as the pre-deadline speed works well, allowing 99.6–99.9% of all possible deadlines to be made while reducing energy consumption by 68.3–84.0% with an average of 77.5%. For keystroke operations, we find that the Stepped algorithm is the best among non-PACE algorithms, but even it is improved by PACE. PACE improves by considering tasks of different applications and different categories separately, validating our conclusions about the differences between applications and categories. For mouse click operations, we reach the same conclusions. However, we find that the effectiveness of PACE and of separating tasks by application and category is less for mouse clicks than for keystrokes. Given the processor characteristics considered in our simulation, the complexity of PACE is probably not worthwhile for mouse click operations.

In conclusion, the main fact we have discovered from our traces is that there are great differences between tasks arising from different circumstances. Differences in the user, the application, the user interface event type, and even the category of user interface event type, can have substantial effect on the CPU usage of a task. Therefore, a DVS algorithm can gain substantial information about the requirements a user places on his CPU, merely by monitoring details about the user interface tasks presented to applications.

Chapter VI

RightSpeed: A Task-Based Scheduler

VI.1 Introduction

Traditionally, systems capable of dynamic voltage scaling use *interval-based* strategies, which divide time into intervals of fixed length and set the speed at the beginning of each interval based on recent CPU utilization. However, CPU utilization is only a rough indicator of the speed the CPU requires. For instance, a process that polls continuously for input will keep the CPU utilization at 100% but does not actually require anything more than the lowest speed. Also, an interval-based strategy cannot distinguish an urgent task that must run at full speed to meet a tight deadline from a less important task that has several milliseconds to complete and little work to do in that time.

A better solution, as we have suggested in this dissertation, is to use *task-based scheduling.* Such scheduling considers the computer's work to consist of tasks with certain CPU requirements and deadlines. It then runs the CPU fast enough to meet those deadlines with reasonable probability. Recently, some researchers have even built such task-based schedulers [PS01, FRM01]. In this chapter, we describe how we built RightSpeed, a task-based scheduler with several improvements over these existing schedulers.

The key differentiating feature of RightSpeed is its *PACE calculator*, a component that, given performance requirements for a task, estimates the task's CPU requirements to determine the most energy efficient schedule that meets those performance requirements. In Chapter IV, we demonstrated that computing such a schedule requires estimating the probability distribution of the task's work requirements, and gave a method called PACE for using such a distribution to compute such a schedule. That method assumes a somewhat unrealistic model of real computers, so we had to extend it to deal with time spent waiting for I/O, the potential overlap of multiple simultaneous tasks, limited sets of available speed/voltage settings on real processors, and limited timer granularity available on the operating system.

Another new feature of RightSpeed is its *automatic task detector*. A task-based scheduler must have an interface letting applications specify information about their tasks: when those tasks begin, when they end, what their deadlines are, and what level of performance the application expects the scheduler to provide for those tasks. Note, however, that even if the system provides such a mechanism to applications, many application writers will not use it, either because they wrote the application before the mechanism was available or because they are unwilling or unable to provide task information. For this reason, a task-based scheduler should also have an automatic task detector to let it infer task information from such oblivious applications. Flautner et al.'s scheduler has such a detector, but it requires a great deal of complex, high-overhead, and Linux-specific system interposition [FRM01]. In Chapter V, we suggested a method for accomplishing automatic task detection with a more efficient heuristic, but did not demonstrate an implementation. RightSpeed demonstrates an implementation of this heuristic.

Another way our task-based scheduler is different from existing schedulers is that it runs on Windows 2000 rather than Linux. This is important because almost all portable computers sold today run Windows 2000 or its successor Windows XP. Implementing RightSpeed on Windows 2000 instead of Linux was challenging, since we lacked access to Windows 2000 source code, either to read or change it. Fortunately, Windows 2000 provides various documented mechanisms for extending the GUI and kernel, and we used these mechanisms to implement RightSpeed. By this we show that task-based voltage scaling can be accomplished even on a closed-source commodity operating system.

The goal of this chapter is to demonstrate that a task-based voltage scheduler with a PACE calculator and an automatic task detector can be implemented on a real machine running Windows 2000. This involves overcoming the challenges of real hardware and software issues, and demonstrating that the resulting scheduler places little overhead on the system.

The structure of this chapter is as follows. Section VI.2 gives related work on other task-based dynamic voltage scaling algorithms. Section VI.3 describes the characteristics of the processors to which we ported RightSpeed, and evaluates the potential effectiveness of DVS techniques on these processors. We find that these processors have a limited set of available settings, which means that PACE is not worthwhile on this generation of processors. Section VI.4 discusses the design of our task-based scheduler. We describe its task specification interface, its automatic task detector, and its PACE calculator. We implemented a PACE calculator in RightSpeed to demonstrate that it could be done; this will be important on future processors for which PACE is worthwhile. Section VI.5 discusses our implementation of the RightSpeed design. Section VI.6 gives results of benchmarks showing the impact of our modifications on performance and energy consumption. It shows that RightSpeed has modest overhead, about 1.2%, and its operations take only microseconds. It also shows that PACE becomes more effective at reducing energy consumption as more speeds are available on the CPU, and argues that processor manufacturers should provide such large speed ranges to help PACE reduce energy consumption. Section VI.7 discusses avenues for future work. Finally, Section VI.8 concludes.

VI.2 Related Work

Recently, researchers have begun building task-based voltage schedulers. These schedulers consider the work of the system to be divided into tasks with certain deadlines. The goal of a task-based voltage scheduler, then, is to use speeds just high enough to meet these deadlines with reasonable probability.

Yao et al. [YDS95] described how to compute an optimal schedule when task CPU requirements and deadlines are known. Hong et al. [HPS98] later showed how to compute such schedules more quickly using various heuristics. However, in general systems do not have certain knowledge of task CPU requirements, so these approaches are unrealistic.

Flautner et al. [FRM01] built a task-based voltage scheduler for Linux. One of the design goals was to require no modification of applications, so it infers all information about the system's tasks via heuristics. It infers that interactive tasks begin when a user interface event arrives, and uses a complex work-tracking heuristic to decide when such a task completes. It infers that periodic tasks begin when a periodic event occurs; they suggest considering an event periodic if the lengths of intervals between the last n events have a small variance. To determine the speed at which to run a task, it essentially computes the average of the speeds that would have completed past similar tasks on time.

Pillai et al. [PS01] built a task-based voltage scheduler designed for real-time embedded systems. This scheduler runs on Linux. This scheduler assumes complete knowledge of the deadlines and worst-case CPU requirements of all tasks in the system, and that these tasks are periodic. Their scheduler is notable in that it can use different algorithms, some of which make provisions for tasks completing before their deadlines, as follows. One such algorithm slows down the CPU when a task completes early and thereby creates slack in the schedule. The other algorithm anticipates the likelihood that tasks will complete early and therefore starts tasks as slowly as possible and only uses higher speeds when these become necessary to guarantee on-time completion.

VI.3 Platforms

In this section, we will examine the characteristics of the two processors to which we ported RightSpeed, one from Transmeta and one from AMD. As we do so, we will discuss how these characteristics influence how we should use PACE on these processors.

First, we introduce some definitions. A *setting* is a speed and voltage combination that a processor can stably obtain. The *efficacy* of a setting is the amount by which power consumption is reduced by using this setting instead of emulating its speed using other settings. For example, suppose there are three settings: 300 MHz consuming 2 W, 500 MHz consuming 3.6 W, and 700 MHz consuming 6 W. We can emulate 500 MHz by running half the time at 300 MHz and half the time at 700 MHz. This consumes 4 W, while the 500 MHz setting consumes only 3.6 W, so the 500 MHz setting has efficacy 10%. We can emulate 300 MHz by running 60% of the time at 500 MHz and turning the CPU off 40% of the time;

this emulation has average power consumption 2.16 W, so the 300 MHz setting has efficacy 7.4%. If a setting has efficacy of 0% or less, it is *unworthwhile*, i.e., there is no reason to ever use it since one can get lower power consumption at the same speed using other settings.

For PACE to be effective, a processor must have at least three worthwhile speed/voltage settings. Furthermore, the more settings, and the higher their efficacy, the more effective PACE will be. This is because PACE works by choosing among speed schedules with identical performance to find the one with least expected energy consumption. If there is little choice in such speed schedules, and/or if there is little difference between choosing one setting versus emulating that setting's speed with other settings, there will likely be little benefit to choosing among them.

PACE must also take into account transition overheads. The standard formula PACE uses assumes that a transition between settings is instantaneous. A more realistic model [BB00] is that a transition consumes nonzero time and energy, with each roughly proportional to the voltage differential between the two settings. Given this model, a better approach is to reduce the deadline by the transition time between the minimum and maximum speeds, as we showed in Section IV.4.5. If this model does not hold, and if transition time and/or energy is significant, one must take additional steps to incorporate the effect of transitions in the PACE formula.

To make judgments about what settings to use, PACE must have reliable information about the power consumption at each setting. Actually, it is sufficient for PACE to know the relative power consumption of the settings. We will examine whether the processors considered provide adequate information about relative power consumption for RightSpeed to apply PACE.

VI.3.1 Transmeta system

The Transmeta system contains a TM5400-633 Crusoe^{TM} processor and 128 MB of memory (64 MB of SDRAM and 64 MB of DDRAM). 16 MB of this memory is reserved for the Code-Morphing Software, whose primary function is to dynamically translate x86 code to the underlying machine language of the VLIW chip. This code also implements LongRun^{TM}, the DVS policy Transmeta chips use. Transmeta told us how to override LongRun^{TM} policies

Reported MHz	Measured MHz	Volts	CPU watts	Energy in nJ per cycle	Efficacy
300	297.3	1.2	1.349	4.537	0.5%
400	396.6	1.225	1.809	4.561	11.0%
500	497,8	1.35	2.714	5.461	11.8%
600	598.5	1.55	4.348	7.265	0.4%
633	631.1	1.6	4.915	7.787	N/A

Table VI.1: Characteristics of the Transmeta CPU at various settings

and change the speed ourselves.

VI.3.1.1 Settings

The processor can run at 300–633 MHz and 1.2–1.6 V. Table VI.1 gives the available speeds and voltages, as well as the power the CPU consumes at each level. We measured power consumption by running in a tight loop of additions while using hardware monitoring equipment Transmeta provided.

We see that the 300 MHz and 600 MHz settings have very low efficacies, and are therefore barely worthwhile. With only three reasonably worthwhile settings, we do not expect PACE to be very effective on this machine.

Incidentally, we note that the formula $1.179 \cdot 10^{-9} \cdot s^{3.41} + 3.681$, where s is speed, gives a very close approximation to the energy consumption in nJ/cycle for all but the 300 MHz setting. The power of 3.41 differs substantially from the power 2 predicted by simple scaling models, e.g., [WWDS94].

VI.3.1.2 Estimating power consumption

We find that power is very nearly a linear function of $V^2 f$, as predicted by CMOS models. An approximation of $0.00305V^2 f$ is always within 3.3% of the observed actual power consumption. We conclude that, in the absence of power measurement equipment, a DVS algorithm can reasonably estimate relative power consumption of different speed/voltage settings using $V^2 f$. This does not give absolute power figures, since the constant of proportionality is not known, but it suffices for comparing two different settings.

VI.3.1.3 Transitions

We also measured transition times on this CPU using the following experiment for each such transition. For a set of numbers ranging from 0 to 80,000, we determined how long it took to perform that number of simple loop iterations immediately after requesting the transition. Such a graph has an initial section with slope equal to the original speed, followed by a gap, followed by another section with slope equal to the new speed. We can interpret such a graph as follows. The duration of the first section indicates how long it takes to raise the voltage to the new level; the duration of the gap indicates how long the CPU is turned off while the frequency is raised to the new level. We observe the following results from these experiments:

- It takes about 60 μs to change between 1.2 V and 1.225 V (300 MHz and 400 MHz), or to change between 1.55 V and 1.6 V (600 MHz and 633 MHz).
- It takes about 254 μs to change between 1.225 V and 1.35 V (400 MHz and 500 MHz), or to change between 1.35 V and 1.55 V (500 MHz and 600 MHz).
- It takes about 16–17 μs to change speeds, slightly longer at slower speeds.
- The time to change speeds varies within a range about 10 μs wide, suggesting that the chip polls for change-speed requests every 10 μs.

Thus, we see that on this system, changing between two voltage levels takes time roughly proportional to the voltage differential, as predicted by the model Burd et al. use [BB00]. Incidentally, we learned later that the chip actually has 11 voltage levels, not just five: three of these are between 1.225 V and 1.35 V and three of these are between 1.55 V and 1.6 V. The voltage transition time is proportional to the number of transitions between internal voltage levels since the chip waits a constant amount of time for each such change [Ham01].

VI.3.1.4 Benchmarking

Accurately evaluating performance on the Transmeta system is difficult for two reasons: (a) the dynamic translation of code the chip performs can cause large differences from one run to another, and (b) confidentiality agreements preclude us from publishing certain

Speed	Voltage	CPU power	Energy per cycle	Efficacy
500 MHz	1.25 V	10.63 W	21.25 nJ	7.6%
600 MHz	1.3 V	13.79 W	22.99 nJ	1.4%
700 MHz	1.35 V	17.35 W	24.79 nJ	-0.9%
800 MHz	1.4 V	21.33 W	26.66 nJ	-3.6%
900 MHz	1.4 V	24.00 W	26.66 nJ	N/A

Table VI.2: This table shows characteristics of the AMD processor at various settings, with power and energy values approximated.

benchmarks of the prototype system. Therefore, we measure time to perform RightSpeed operations on the AMD systems only.

VI.3.2 AMD system

The AMD system contains a pre-production version of the 900 MHz Mobile Athlon 4 processor, based on the Palomino core, as well as 128 MB of memory. We were given documentation about PowerNow!$^{\text{TM}}$, the interface the chip uses for dynamically changing speed and voltage.

VI.3.2.1 Settings

The chip indicates it is capable of five settings, shown in Table VI.2. We were unable to determine the power consumption of each setting, so we estimate it using $P \propto V^2 f$; we saw this to be reasonably accurate for the Transmeta system. We assume a power consumption of 24 W at the maximum speed, as specified in the AMD data sheet [AMD01].

We see that the 700 MHz and 800 MHz settings have negative efficacy, so they are not worthwhile. (It is not surprising that the 800 MHz setting is not worthwhile, since it has the same voltage as the 900 MHz setting, and thus the same energy consumption, but it runs more slowly.) Furthermore, the 600 MHz setting has rather low efficacy. With only three worthwhile settings, one of which is only barely worthwhile, we expect PACE to be largely ineffective.

VI.3.2.2 Transitions

The AMD interface for changing CPU speed is to write a certain machine status register (MSR). With this write, one can specify not only the desired new speed and/or frequency, but also the time to force the CPU to halt while waiting for the new settings to stabilize. AMD documentation indicates one should force a wait of at least 10,000 bus clocks, or 100 μs for this [AMD01]. Furthermore, it suggests modifying speed and voltage in separate operations when raising the speed. However, in millions of attempts to crash the CPU by specifying waits of only 1 bus clock along with simultaneous modification of both speed and voltage, we never observed any problems. In these tests, we observed that the CPU simply stays halted longer than the 1 bus clock requested; presumably it does this to avoid glitches.

We attempted to measure transition times using the technique we used for the Transmeta system, but found that it did not work. The call to set the speed apparently takes effect immediately, and halts the CPU until the speed and voltage are changed, so there is no observation that will show the former speed in force. However, we can determine how long the CPU was halted by observing the gap in the CPU cycle count between one obtained immediately before the change request and one obtained as soon as possible afterward. We found that transition times were on the order of only 7–9 μs and, oddly, did not seem to depend on the amount by which the speed or voltage was changed. This suggests that the CPU has a hard-coded amount of time that it waits for speed and voltage to stabilize, and that it waits for this time without regard to how large the transition is. It is surprising that the transition times are so low, not only compared to the Transmeta chip but also compared to what AMD indicates is necessary.

VI.4 Design

VI.4.1 Overview

Figure VI.1 gives an overview of the design of RightSpeed. Applications convey information about their tasks to the operating system using system calls. This information includes when tasks begin and end and what performance targets the application expects

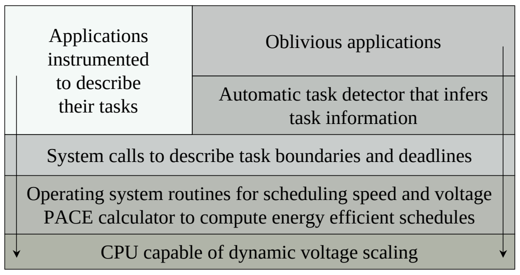
Applications instrumented to describe their tasks
Oblivious applications
Automatic task detector that infers task information
System calls to describe task boundaries and deadlines
Operating system routines for scheduling speed and voltage
PACE calculator to compute energy efficient schedules
CPU capable of dynamic voltage scaling

Figure VI.1: Overview of RightSpeed

those tasks to meet. Some applications are oblivious to the existence of these system calls, so an automatic task detector infers task information about them and generates task specification system calls on their behalf. The operating system uses information about the set of ongoing tasks to determine what speed to use at different times, and implements this schedule using timers and special processor instructions that change speed and voltage. The operating system uses a PACE calculator to compute the most energy efficient schedules that have the performance requested, and use those schedules.

In addition to the above functionality, we had three overall goals for RightSpeed. First, we wanted it to be efficient, creating low overhead on the system both when running in the background and when being actively invoked. Second, we wanted it to be stable, relying only on documented system interfaces so that it would continue to run even when the system was upgraded. Third, we wanted it to be easily portable to different processors despite such processors having different commands for querying and modifying speed and voltage settings.

VI.4.2 Task specification interface

A key piece of information an application must specify about a task is its *type*. An application may define types in any way it chooses; there are two reasons applications will want to classify different tasks into different types. First, it may want to specify different performance targets for different types of tasks. For example, an MPEG player may require a faster average speed for processing its I-frames than its P-frames, since I-frames are much larger. Second, tasks of different types may have different CPU requirements, and therefore it is helpful to direct PACE to only consider tasks of the same type when predicting the probability distribution of a task's CPU requirement.

RightSpeed keeps track of various persistent data about each task type. This data includes the performance targets associated with that task type, as well as a sample of the CPU usage of recent tasks of that type. Using this data, it can use the PACE calculator to compute an energy efficient schedule that meets performance targets for that task type. For efficiency, RightSpeed can pre-compute this schedule and include it in the task type data. Then, when the next task of that type begins, it can immediately begin the CPU speed schedule assigned to that task type.

RightSpeed uses this notion of task type to simplify its communication with applications. When an application begins a task, it need only tell RightSpeed the type of that task. RightSpeed can figure out all other information about the task from that type. RightSpeed can give the application a unique identifier to identify this task that just began, so the application can specify when the task completes by merely passing RightSpeed that identifier. RightSpeed can then determine how many cycles that task used and add this datum to the sample of CPU usage of recent tasks of that type.

An application specifies performance targets for task types via a separate part of the task specification interface. In theory, an application need only specify this data once, when the application is installed. Because task type data is persistent, i.e., it is retained even when the application terminates and even when the system shuts down, a logical abstraction to use for this data is a file. We should allow applications to create files containing data for their task types, and to describe tasks' types by referring to these files.

VI.4.3 Automatic task detector

Since RightSpeed has not been released, no application currently exists that communicates its task information to RightSpeed. Furthermore, even when it is released, we expect few application writers will be both willing and able to communicate such information. Therefore, for RightSpeed to be useful, we require an automatic task detector to infer task information from such applications and to call the task specification interface on their behalf.

Our approach is to focus on the tasks the user cares about most: those triggered by user interface events. User interface studies have shown that response times under 50–100 ms do not affect user think time [Shn98]; one can thus consider 50 ms the deadline for handling a user interface event. One exception is mouse movements, whose tracking may require lower response times of 25–50 ms [MW93]; one can thus consider 25 ms the deadline for handling mouse movement events.

Interposition of the GUI event delivery system will allow us to see when these events occur and what their characteristics are. By characteristics, we mean whether it is a keystroke, mouse movement, or mouse click; which key or mouse button was pressed or released; and what application the event was delivered to. We can assume a task begins whenever such an event occurs, and deduce the type of this task from the event characteristics. As we showed in Chapter V, separating tasks into types this way makes estimation of task work distribution more accurate, and also enables us to set different policies for, for instance, keystrokes and mouse clicks.

As we suggested in Chapter V, we use a pre-deadline speed for mouse movement events corresponding simply to the minimum available on the CPU. We use a default pre-deadline speed of $0.7M$ for keystroke events and $0.85M$ for mouse click events, where M is the maximum speed available on the machine. A better approach might be to compute a variable pre-deadline speed based on the distribution observed and the likelihood of missing deadlines at various pre-deadline speeds, as suggested in Chapter V. However, as we showed in IV.7.5.2, we do not yet know of a good way to do this, so we currently have not implemented this feature in RightSpeed.

We also need a heuristic to determine when such an inferred task is complete. One way, suggested by Flautner et al. [FRM01], is to keep track of the set of threads involved in the task. Whenever a thread involved in the task communicates with some other thread, that other thread becomes involved in the task. One considers the task to be complete when all threads involved in the task are blocked for reasons other than waiting for I/O, and when all data generated by those threads have been consumed. While this is a powerful method, it requires a great deal of modification of the operating system, which is largely infeasible for our purposes.

Thus, we instead use a heuristic that is far more efficiently implemented and that, according to workload analyses in Chapter V, gives quite similar results. We consider a task complete when either (a) all threads in the system above the idle priority level are blocked and no I/O is ongoing, or (b) another user interface event is delivered to the same application. A side effect of this is that time spent by unrelated threads is considered to be part of the task until all such threads block for reasons other than I/O. This might seem to be an unfortunate consequence, but it is actually an advantage. Recall that RightSpeed is not a real-time scheduler and thus applications must account for time spent by other unrelated processes when choosing a speed schedule for its tasks. By considering all time spent on other threads to be part of the task, we automatically take into account the work requirements of unrelated threads in the work requirement analysis for the task with a deadline. The speed schedule chosen will thus automatically account for the work that is performed by other threads while the task is running. However, this approach is not perfect: if other threads have lower priority than the task threads, it is not really necessary to complete their work before the deadline, and accounting for them will only make us unnecessarily increase the CPU speed to complete that work by the deadline as well.

We extend this mechanism for automatically detecting the ends of tasks to tasks explicitly generated by applications. This saves overhead, since applications that do not want to spend time communicating the ends of tasks to the scheduler need not do so. Furthermore, it aids in the automatic inclusion of other thread time in the estimate of task work requirements, as it causes the task end time to take into account work performed by other threads. Thus, whenever all threads in the system above the idle priority level are blocked and no I/O is ongoing, RightSpeed considers all ongoing tasks in the system to have ended.

VI.4.4 PACE calculator

We discussed earlier that two speed schedules can be performance equivalent yet have different energy consumption. Computing the optimal performance equivalent speed schedule requires knowledge of the distribution of task work requirements, which typically an application lacks. For this reason, RightSpeed keeps track of how long tasks of each type have taken, and uses this information to compute an optimal speed schedule conforming to the requirements of the task type. Incidentally, applications may request that PACE not be used for a task type, in which case the application must specify what algorithm to use for that task type and RightSpeed makes no effort to track how long tasks of that type take.

In Chapter IV, we described how to compute a PACE schedule. However, we made some assumptions that we showed in Section VI.3 do not hold in real systems. We assumed the CPU could attain any speed in a given range, while our real systems only have a finite set of speeds at which they can run. Also, we assumed the system could change the speed at arbitrary times, but most systems, including Windows 2000, only allow operations to be scheduled with a finite granularity, e.g., only once per millisecond.

Thus, for RightSpeed we need an algorithm that takes these realities into account yet still computes a near-optimal schedule. Our algorithm uses the following four steps.

1. Create an idealized schedule as described in Chapter IV. Such a schedule consists of consecutive *phases*, each having a constant speed.

2. For each phase, round its speed to the closest speed that available on the CPU.

3. Round the length of each phase to an integer multiple of the scheduling granularity.

4. Since the rounding may have altered the performance characteristics of the schedule, i.e., altered its pre-deadline speed, adjust the length of time spent at each speed by multiples of the scheduling granularity so that the performance is close to, and no less than, the requested performance.

Before we describe how we perform the last step, we make the following observation about speed schedules. Suppose some candidate schedule has n phases, and in phase i we run at constant speed s_i for time t_i. Suppose further that the performance constraint requires performing C cycles of work by deadline D. To satisfy this constraint, we require $\sum_{i=1}^{n} s_i t_i = C$ and $\sum_{i=1}^{n} t_i = D$, though one or both of these may not hold. If $\sum_{i=1}^{n-2} t_i \leq D$ and if

$$\frac{C - \sum_{i=1}^{n-2} s_i t_i}{D - \sum_{i=1}^{n-2} t_i}$$

is within the range of valid speeds, then there is a schedule that that both conforms to the performance constraint and has the same first $n-2$ phases as the candidate schedule;

furthermore, such a conforming schedule can be found algebraically. (This schedule may require an unavailable speed setting and/or a period that is not an integer multiple of the timer granularity, but rounding to the nearest setting and period should provide a sufficiently reasonable schedule.) Once a candidate schedule reaches a point that we can deduce a final schedule from it this way, we say the candidate schedule has become *feasible*. Once we find a feasible schedule, we are just a bit of algebra away from a final schedule. For example, suppose the CPU has range 200–600 MHz and our performance constraint is an average speed of 400 MHz before a 100 ms deadline. A candidate schedule of 250 MHz for 50 ms then 400 MHz for 20 ms then 600 MHz for 30 ms is feasible because there is a schedule that both satisfies the performance constraint and shares all but the last 2 phases with it. With algebra, one can find this schedule is 250 MHz for 50 ms then 500 MHz for 25 ms then 600 MHz for 25 ms.

We thus perform step 4 of our PACE algorithm in the following way. We begin with the first phase in the schedule and consider how much we would have to adjust its length for the schedule to be feasible. If we can adjust it in this way, we do so and are done; if it cannot, we adjust it as much as possible in the appropriate direction and move on to the next phase. If we reach the second-from-last phase and still have not created a feasible schedule, we give up and simply use a default schedule. Incidentally, the first approach we considered was to do a search through several possible schedules to find the one with the minimal energy consumption, instead of this greedy approach that seeks only the first one we can find. However, we found that the greedy approach yields practically identical results to the more involved, and far more time-consuming, search method.

The rounding that occurs in step 2 allows us to more efficiently perform the computations in step 1, as follows. Essentially, step 1 is a search for a positive real number K to plug into a general formula for a speed schedule. With knowledge of the available speeds, we can divide the real line into a small set of intervals such that any two values of K in the same interval produce the same speed schedule after rounding. Thus, when RightSpeed is installed, we can pick a single K from each such interval and pre-compute a speed schedule for each K using only knowledge of the CPU characteristics and using no knowledge of any task information. Thus, in step 1, the PACE calculator need not consider arbitrary values of K and plug them into a formula to compute speed schedules; instead, it merely determines which of the pre-computed schedules satisfies the performance constraint best. Since the schedules corresponding to increasing values of K correspond to higher speeds for each interval, we can save the pre-computed schedules in sorted order so that the PACE calculator needs only do a binary search through the schedules to find the lowest-energy one that nevertheless satisfies the constraint.

By default, the PACE calculator computes quantiles of the task work distribution using a gamma model. Although we have not yet implemented any other methods, such as kernel density estimation, we have structured RightSpeed so that other methods can be easily added. As an optimization, when the PACE calculator computes its estimate of α in the gamma model of the task work distribution, it rounds it to the nearest value among a limited set of values. We also hard-code quantiles of the gamma distribution corresponding to these values of α and assuming $\beta = 1$. Thus, we can quickly compute quantiles of any observed task work distribution by looking up the quantiles of $\mathsf{Gamma}[\hat{\alpha}, 1]$ and multiplying them by $\hat{\beta}$. Using a lookup table and a few multiplies saves considerable time, compared to other approaches for computing quantiles of the gamma distribution.

Note that there may be some settings that are available but not worthwhile, such as the 700 MHz and 800 MHz settings on the AMD system. A setting is not worthwhile if there is another pair of settings (possibly including the idle setting) that, on average, produce equal or lower power consumption than the setting in question. Thus, the RightSpeed installer, when it determines what settings are available on the system and what voltage they use, determines which of these settings are worthwhile and records those so that the PACE calculator considers only worthwhile settings.

The algorithm for picking worthwhile settings is as follows. Suppose there are n distinct settings available, and setting i has speed s_i and power consumption P_i. We seek a sequence $\{a_j\}$ consisting only of the indexes of worthwhile settings, by removing any non-worthwhile settings. First, define a_1 to be the index for the setting with the highest speed among those with the lowest energy consumption. In other words, define a_1 such that $P_{a_1}/s_{a_1} \leq P_i/s_i$ for all $i \leq n$ and such that if $P_{a_1}/s_{a_1} = P_i/s_i$ for any $i \leq n$ then $s_{a_1} > s_i$. We define the remaining worthwhile settings by induction. Given a_j, we assign to a_{j+1} the value i with the highest s_i among those with the lowest $(P_i - P_{a_j})/(s_i - s_{a_j})$ among those for which $s_i > s_{a_j}$. If no such i exists, then we terminate the sequence $\{a_j\}$ there.

VI.4.5 Dealing with I/O

I/O time, unlike CPU time, is unaffected by changes in CPU speed. An algorithm that schedules the CPU to achieve a certain slowdown of a task may not achieve the desired slowdown if I/O occurs. In particular, the model from which PACE arises accounts only for the CPU time of a task, so PACE does not give optimal results in the presence of I/O.

We deal with this in the following way. We consider time spent waiting for I/O to be separate from time the CPU spends running. Then, the problem is to complete the CPU work *and* the I/O by the deadline. Equivalently, this means completing the CPU work within a period equal to the deadline minus the I/O time. If we knew the I/O time in advance, PACE could compute the optimal schedule merely by substituting the deadline minus I/O time for the deadline. Since we do not know I/O time in advance, we initially assume it is 0; then, if I/O does occur, we determine how long the I/O wait was and accelerate the schedule to make up for the lost time.

Theoretically, accelerating the schedule properly requires performing a new complex calculation using the PACE formula. However, we can use a shortcut: we multiply all speeds in the schedule by a constant factor, where we choose that factor such that after rounding all resulting speeds to the nearest available speed we get a schedule that meets the new deadline constraint. The argument why this works is as follows. The distribution of task work remaining has by assumption not changed, but the deadline has effectively gotten shorter. Thus, all that has changed is the optimal value of K. This means the ratio of the new optimal speed to the old optimal speed is roughly the same for all points in the schedule, assuming that the function of energy versus speed has a reasonable shape. Even though this is inaccurate when our assumptions are inaccurate, the acceleration approach has the advantage of being fast. This is important considering that it is not invoked during idle time, but at a time when the system has completed an I/O and has more work to do to complete a task by its deadline.

VI.4.6 Scheduling multiple simultaneous tasks

When multiple tasks are ongoing, the ideal speed is not necessarily the sum of all the speeds requested for all those tasks. The reason for this is that power consumption is not a linear function of speed, so adding two speed schedules results in a total energy consumption that is not the same as running both one after the other. Unfortunately, computing a reasonable speed schedule that is the conjunction of two is extremely complex, so we avoid the issue by simply running at the maximum speed available when there are multiple ongoing tasks, and continue at that speed until no tasks remain. The complexity of handling multiple tasks any other way is probably unjustifiable, especially considering that in a mobile computer (and frequently in a desktop computer) there is only a single user and typically he will only notice the performance of the task with which he is currently actively involved. Therefore, there will typically be only one ongoing task at a time.

VI.4.7 Scheduling no ongoing tasks

When no tasks are ongoing, nothing of importance is occurring, so the best speed to use is generally the minimum available. However, since our inference of tasks is imperfect, there may be ongoing tasks even when RightSpeed is unaware of any ongoing tasks. The way we deal with this is by reverting to a traditional interval-based scheduler when we detect the absence of any tasks. Such a scheduler divides time into intervals of some fixed length and chooses a speed for each interval based on the CPU utilization of recent past intervals. This way, if the CPU becomes busy from working on a task we cannot detect, the interval-based scheduler will nevertheless increase the CPU speed to deal with this unknown work.

There is one caveat with this approach, however. When the number of tasks in the system becomes zero, recent past CPU utilization will likely be high because the system just finished working on a task. We know that this recent utilization is a poor predictor of future CPU utilization because it reflects a task that is no longer being worked on. However, an interval-based scheduler has no knowledge of tasks, so it will interpret the high recent utilization as a sign that the next intervals will have high utilization. Accordingly, it will use an unnecessarily high CPU speed. To prevent this problem, when the number of tasks becomes zero, RightSpeed waits for a short period of time at the minimum CPU speed before initiating the interval-based scheduler.

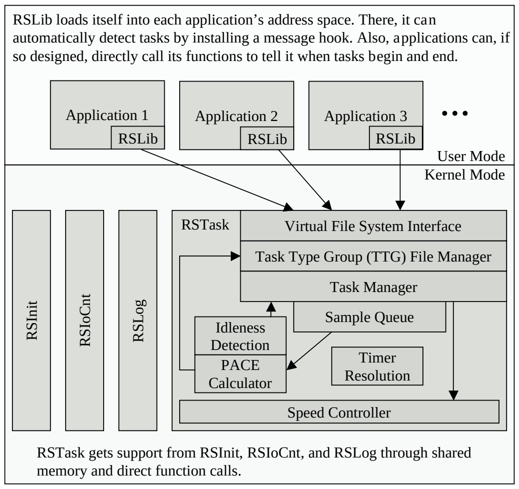
RSLib loads itself into each application's address space. There, it can automatically detect tasks by installing a message hook. Also, applications can, if so designed, directly call its functions to tell it when tasks begin and end.
Application 1
RSLib
Application 2
RSLib
Application 3
RSLib
...
User Mode
Kernel Mode
RSInit
RSIoCnt
RSLog
RSTask
Virtual File System Interface
Task Type Group (TTG) File Manager
Task Manager
Sample Queue
Idleness Detection
PACE Calculator
Timer Resolution
Speed Controller
RSTask gets support from RSInit, RSIoCnt, and RSLog through shared memory and direct function calls.

Figure VI.2: Architecture of RightSpeed

VI.5 Implementation

In this section, we discuss how we implemented our approach on Windows 2000.

VI.5.1 Architecture

Figure VI.2 shows the architecture of RightSpeed. The main component is RSTask, a kernel module that handles task scheduling. It handles requests to begin and end tasks, and schedules the CPU speed accordingly. Its main components are the speed controller, the task type group file manager, the automatic schedule computer, and the idleness detection thread, each of which we will discuss later. Alongside RSTask is RSIoCnt, another kernel module that interposes on all file system requests to monitor when any synchronous I/O's are ongoing. The next kernel component is RSLog, a low-overhead logger that we use for benchmarking and debugging; for a discussion on implementing a low-overhead logger in Windows 2000, see Appendix A. The last kernel mode component is RSInit, a driver that starts before all other drivers and facilitates communication between them. In user mode we have RSLib, a user-level library that the system loads into the address space of every application. It interacts with the GUI to interpose the user interface event delivery system and thereby implement the automatic task detector. This library also exports a set of functions that applications can use to communicate with RightSpeed if they choose to explicitly manage their tasks.

Incidentally, the reason RSTask is a kernel module instead of a user-level module is that on Transmeta's chips as well as AMD's, one changes CPU speed by executing privileged commands in kernel mode. Thus, requests to RSTask had to incur the cost of a transition to kernel mode anyway. So, we put all the associated related parts in kernel mode as well to gain additional advantages such as quick locking of globally shared data structures. Note, however, that we could have put much of the functionality of RSTask in a user-level module. (Indeed, we started out implementing it this way.)

VI.5.2 Speed controller

The lowest-level component of RSTask is the speed controller. This component accepts requests to start and stop speed schedules and to transition to idle and maximum speed states. A speed schedule consists of a sequence of phases, each with a speed to use and a duration in multiples of the scheduling granularity. The final phase of the schedule is always assumed to have indefinite duration. The speed controller internally handles any CPU-specific commands to change the speed. This modularity aided in porting RightSpeed to two different chips with different voltage scaling commands. The scheduler also keeps track of the total number of cycles that have passed since the current schedule began, so it can report this number to the automatic schedule computer.

The scheduler also exports routines to pause and resume the current schedule when the CPU starts and stops waiting for I/O. A pause records where in the schedule we start performing I/O and changes the speed to the minimum available. A resume determines how long the CPU spent waiting for I/O, accelerates the remaining part of the schedule accordingly, and resumes that schedule.

VI.5.3 Timer resolution controller

The default timer resolution on Windows 2000 machines is about 10 ms. If we left it this way, our speed schedules would not be able to change speed except on 10 ms boundaries, which we consider unacceptable. For this reason, we include a timer resolution controller, which reduces the timer resolution value as much as possible and restores the old resolution when RightSpeed is disabled. To do this, it uses well-documented system calls [Neb00]. On the systems we used, this allows us to attain a timer resolution, and thus a scheduling granularity, of 1 ms.

VI.5.4 Task type group file manager

Certain persistent information is associated with each task type: its performance target, a sample of recent task work requirements, and a schedule to use for the next task. Thus, it makes sense to consider task types to be part of a virtual file system. We could have used one file per task type, but instead we allow grouping information about multiple task types in a single file. Thus, a file in this virtual file system is a *task type group file*, containing information about multiple related task types. A task type is uniquely identified by its file and its index within that file.

RSTask thus exposes a virtual file system interface consisting of these files. In reality, the information in these virtual files are stored in real files in a reserved directory on the file system, but RSTask exposes them as existing in the special directory \\.\RSTask. (Unix users can think of this as /proc/rstask/.) Subdirectories of this directory are valid and supported; for instance, an application could choose to use a file called \\.\RSTask\AcmeCo\AcmeAppName\MyTasks.ttg. When an application opens a file in the virtual directory, RSTask opens the corresponding real file, checks it for validity, then locks it in nonpaged memory for quick access. RSTask can have multiple handles (file descriptors, in Unix terminology) open for the same file at the same time; it simply has them all refer to the same region of memory. Thus, multiple applications can easily share the same task type. One way we use this feature is by allowing all automatically detected mouse movement tasks for all applications to use the same task type, giving all of them the same policy. (A mouse movement task is triggered by a mouse-movement event, which indicates the mouse moved since its position was last sampled.) When all handles for the file are closed, RSTask writes back any changes it made to the virtual file to the real file. In case the system crashes before such writes occur, RSTask flushes changes to all open files every hour. (This period is a user-configurable parameter.)

Applications communicate with RSTask by performing I/O control requests on these virtual files. Supported control requests include beginning a task of a certain type and acquiring a task ID associated with that task, ending the task associated with a given task ID, changing the deadline for a task type, resetting the sample of recent work requirements for a task type, flushing a task type group file, and various other minor ones. RSTask supports *fast I/O* control requests [Nag97], a Windows 2000 optimization that allows very fast I/O requests to kernel mode. As a further optimization, RSTask also has an I/O control request that ends one task and begins another; the automatic task detector in RSLib uses this to quickly signal the end of the previous user interface task when an application receives a new one.

VI.5.5 Task manager and sample queue

RSTask keeps track of the set of ongoing tasks in a global shared data structure and makes appropriate calls to the scheduler when tasks begin and end. Also, when a task ends and PACE calculation is enabled for its task type, the task manager queues the information about how long this task took in the *sample queue*. The reason it does not immediately invoke the PACE calculator is that inserting a sample value into a task work distribution description and computing the resulting schedule can take time, and this is best saved for when the CPU is otherwise idle.

As stated in VI.4.7, when no tasks are ongoing, it is best to wait for a short period of time and then initiate an interval-based scheduler. We accomplish this on the Transmeta system in the following way. When the task manager detects the departure of the last ongoing task, it switches to the lowest available speed and sets a 50 ms timer. When this timer expires, it enters the LongRun$^{\mathrm{TM}}$ automatic speed scheduling mode, which uses an interval-based strategy. We chose 50 ms because this is further backward than LongRun$^{\mathrm{TM}}$'s scheduler ever looks. We have not yet implemented a scheme for using an interval-based scheduler on the AMD system; this is future work.

VI.5.6 Idleness detection thread and automatic schedule computer

The idleness detection thread is another major component of RSTask. Figure VI.3 gives a pseudocode description for this code. It runs at priority 5, just above the idle level, so that it can easily detect when no important threads remain unblocked. If it is scheduled when an I/O is ongoing, it tells RSTask to pause the current schedule; it will later tell RSTask to resume the schedule when no synchronous I/O's remain in the system. If the idleness detection thread runs when no I/O is ongoing, it notifies RSTask of this fact so that it can consider all ongoing tasks to have ended.

The other responsibility of the idleness detection thread is to invoke the PACE calculator on all unprocessed entries in the sample queue when the system is otherwise idle. Not only does this cause the overhead of PACE calculation to occur only when the system is idle, it also eliminates overhead due to saving and restoring floating-point registers. Since the idleness detection thread runs only in kernel mode, the PACE calculator does not have to save and restore floating-point registers before and after performing its floating-point calculations.

VI.5.7 I/O counter

The other kernel module we will discuss is RSIoCnt, whose job is to count the synchronous I/O's in the system and store this count in shared memory where RSTask can access it. This allows the idleness detection thread to determine whether any I/O's are ongoing. It is also RSIoCnt's responsibility to tell RSTask to resume any paused schedule whenever the last I/O completes.

We implemented RSIoCnt as a file system filter driver. A filter driver implements a filter device, a special kind of device extremely helpful in tracing system events in Windows NT/2000. A filter device can *attach* to an existing device, causing it to intercept any requests destined for that existing device. For more information about them, see [Nag97] and Appendix A. Our filter driver has low overhead because it merely counts the requests as they start and stop and pass them on.

Unfortunately, although Microsoft documents how to filter all file systems in this way, their approach limits one to filtering only non-network file systems. There are undocu-

```
while (1) {
  Wait for the generic wake-up signal indicating either the arrival
  of a new task or the departure of the last I/O in the system.

  // Since there's no way to stop Windows 2000 from boosting our
  // priority, and we never want to run while there are threads of
  // higher priority than 5 in the system, make sure before
  // continuing that our priority is still only 5:

  while (my priority is over 5)
    Lower my priority to 5.

  if (an I/O and a schedule are both ongoing)
    Pause the current schedule.

  if (no I/O's are ongoing)
    Indicate all tasks are now over.

  Call the PACE calculator to process all sample values in the queue.
}
```

Figure VI.3: Idleness detection thread

mented ways to filter network file systems as well, as we show in Appendix A, but we do not do this in our prototype, due to our stability goal. We also do not filter network activity, even though that is also I/O. Undocumented methods for this exist, as shown in Appendix A, but again, we have left them out of our prototype for stability.

VI.5.8 User-mode library

Microsoft documents a method for loading a user-mode library into the address space of every process that makes GUI calls. This involves placing the name of that library in the registry key HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs [Mic00]. (The registry is Windows's place for global configuration information, loosely similar to /etc in Unix.)

The main activity of this library is interposing on the delivery of user interface events to the application. This, too, can be done via an interface documented by Microsoft.

This interface allows one to create *message hooks* [Mic00]. With this mechanism, one can tell Windows to call a given function just before it successfully completes an application's request for the next message from the GUI. One can install such hooks globally or on a per-thread basis. The former is more expensive due to context switches so we use a separate hook for each thread. This also lets us implement an RSLib call that an application can make if it does not want its messages to be hooked, e.g., if it would prefer to make its own calls to indicate its tasks rather than have them automatically detected.

RSLib also exports a set of functions that applications can use to communicate with RSTask. Applications can always interact with RSTask without these calls, but they are helpful to application writers who prefer to deal with a function call interface rather than making I/O control calls to a virtual file system. RSLib also contains the code for the creation of new task type group files. RSLib creates these files without interacting with RSTask by placing them in the directory reserved for them. This functionality could have been placed in RSTask, but we preferred to do it in RSLib so that we could use C++.

RSLib exports miscellaneous other functions that applications may find helpful, such as one that converts a RightSpeed error message code into a text string explaining the error.

VI.6 Results

VI.6.1 General overhead

In this section, we evaluate the overhead on the system just from having RightSpeed run in the background, unused. There are two main sources of this overhead:

- Incrementing the timer frequency from every 10 ms to every 1 ms causes interrupt-processing time to increase.
- Filtering I/O requests to count them increases the time to perform each I/O.

To evaluate these effects, we ran the following benchmarks on the Pentium III system:

1. Read an uncached 32 KB file

2. Write a 100 KB file with write-through
3. Read 32 KB directly from the disk
4. Compile the RightSpeed logger device with the Windows DDK
5. Format this thesis (excluding this chapter) with LaTeX
6. Perform a CPU-intensive mathematical loop

We ran these benchmarks without any RightSpeed modules loaded, with only the RSTask module loaded, and with both the RSTask and the RSIoCnt modules loaded. In all cases, we disabled the network to avoid interference from network interrupts. None of these benchmarks *use* RightSpeed at all; indeed, we did not even install RSLib to perform these experiments. We ran each benchmark enough times that the 95% confidence interval about the sample mean included no values more than 0.01% away from the sample mean, or 10,000 runs occurred, or 2,000 seconds passed, whichever came first. Table VI.4 shows the results. Note that there are some anomalous results, such as all components of RightSpeed together producing less overhead than using either one separately; the reason for this is likely that the 95% confidence intervals do not take into account the error from running the benchmarks on different RightSpeed components at different times and thus with potentially different file system layouts for the files involved.

We see that RSIoCnt adds 0.3–1.5% overhead, with an average of 0.5%, due to filtering I/O operations. Microsoft could reduce this by incorporating this functionality into its operating system instead of requiring installing a filter just to count I/O's and notify drivers when all I/O's are complete. We also observe that RSTask, by virtue of it increasing timer resolution from 10 ms to 1 ms, increases operation times by 0.7–1.6% with an average of 1.1%, presumably due to the overhead of more frequent timer interrupts and the associated system response. Combined, the overhead amounts to 1.2% on average. So, we observe that just the need to filter I/O and to increase the timer granularity puts an overhead on the system of about 1.2%; if we could modify the operating system directly, these overheads would likely be lower, especially for the file system modifications.

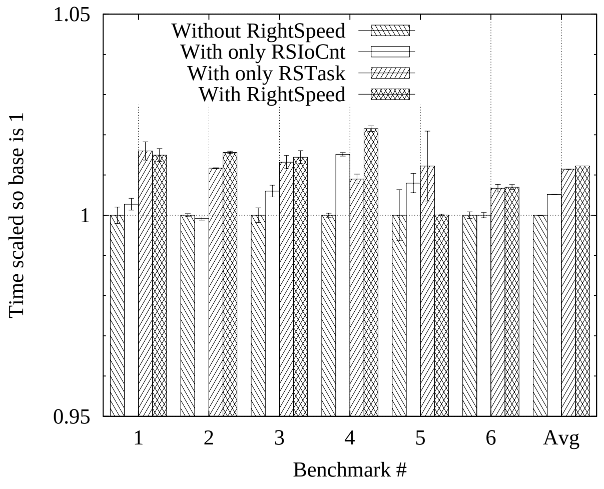

Figure VI.4: Time to perform various benchmarks without RightSpeed and with various components of RightSpeed enabled, shown with 95% confidence intervals. The six benchmarks are (1) read a 32 KB file, (2) write a 100 KB file, (3) read 32 KB directly from disk, (4), compile the logger, (5) format this thesis with LaTeX, and (6) a CPU-intensive mathematical loop.

VI.6.2 Time to perform RightSpeed operations

The next set of results evaluates the time to perform various RightSpeed operations. We performed these measurements on the AMD system, but in all cases speed changing was disabled to not confound the measurement of durations. So, all runs are at 900 MHz. Most of these results we measured directly by making an entry in the log each time an operation started or stopped. However, some of them, such as intercepting a message, involve hidden operating system overhead, so we measured these quantities by running with and without performing the operation and subtracting. We ran each operation 10,100 times, discarded the first 100, and reported the mean, standard deviation, and 95% confidence interval around the mean for the remaining 10,000. Table VI.3 shows these results.

We see that the overhead of linking RSLib into each application is about 1.4 ms; this occurs only once per application, when it starts. Some of this is RSLib's initialization, including installing the message hook and opening the automatic task type group files, but this accounts for very little of it. In these benchmarks, the application task type group file is

Operation	**Time**		**St. dev.**
Load and initialize RSLib for a process	1.401 ms	± 0.001 ms	0.055 ms
Install message hook	8.532 μs	± 0.040 μs	2.062 μs
Open systemwide auto task type group file	159.777 μs	± 0.168 μs	8.580 μs
Get application name	12.583 μs	± 0.033 μs	1.685 μs
Open per-app auto task type group file	121.229 μs	± 0.091 μs	4.619 μs
Intercept a non-user-interface message	2.265 μs	± 0.031 μs	1.556 μs
Intercept and handle a user interface message	7.605 μs	± 0.230 μs	11.722 μs
Evaluate message type	1.013 μs	± 0.029 μs	1.460 μs
End task and begin another	3.575 μs	± 0.032 μs	1.647 μs
Make a simple I/O control request of RSTask	1.162 μs	± 0.024 μs	1.201 μs
Begin a task	3.450 μs	± 0.019 μs	0.965 μs
Kernel-mode component	1.345 μs	± 0.008 μs	0.567 μs
End a task	2.530 μs	± 0.398 μs	0.008 μs
Kernel-mode component	1.134 μs	± 0.004 μs	0.305 μs
End one task and begin another	3.462 μs	± 0.023 μs	1.168 μs
Kernel-mode component	1.441 μs	± 0.010 μs	0.531 μs

Table VI.3: These benchmark results show how much time RightSpeed takes to perform common actions on the AMD machine.

in the file cache, but even if it were not the time to load would not be significantly more. In conclusion, an application will take under 2 ms extra to load due to the presence of RSLib.

The overhead of hooking all messages delivered to applications is also small. For non-user-interface messages, the overhead is only 2.3 μs per message. For user interface messages, the message hook must determine what kind of event it is and communicate information about the task's beginning and the previous task's ending to RightSpeed. The overhead for this is also small, approximately 7.6 μs per message. Considering that messages arrive on the order of every few milliseconds, and that user interface messages arrive even less often (at worst about every 14 ms in the case of rapid mouse movement, and more typically about once every 150 ms if the user is typing at 40 words per minute), the total overhead is quite low.

Looking at microbenchmarks of RightSpeed operations shows how this overhead arises. Each I/O control request takes about 1–2 μs due to the time to trap into kernel mode and to check and copy data from user buffers to kernel buffers. Inside RSTask, the time to begin a task is about 1.3 μs and the time to end a task is about 1.1 μs. The most common operation, processing a request to end one task and begin another, where the task to end has already been ended by the idle thread, takes about 1.4 μs of kernel time. Note that this is less than the sum of the time to begin a task and to end a task because of various optimizations for this case. For example, we do not have to look up the task type group file twice or acquire and release the spin lock controlling access to the ongoing tasks list twice.

VI.6.3 Effect on performance

RightSpeed can accept requests for any level of performance for any task type. However, since Windows 2000 is not a real-time operating system, scheduling decisions do not necessarily happen precisely when they should, so performance targets will not necessarily be met. In this subsection, we evaluate how well RightSpeed meets performance targets.

These evaluations require a workload consisting of a sequence of tasks. We derived the workloads from traces of users performing their normal business on desktop machines running Windows NT or Windows 2000. VTrace, a tracer described in Appendix A, generated these traces. The traces contain timestamped records describing events related to processes, threads, messages, disk operations, network operations, the keyboard, and the mouse. We deduce what work is done due to a user interface event as follows: we assume that a thread is working on such an event from the time it receives the message describing that event until the time it either performs a wait for a new event or requests and receives a message describing a different event. Furthermore, if the thread sends a message or signal to another thread while working on such an event, we assume that work done due to that message or signal is done due to the original event.

We use workloads derived from the same eight users described in Chapter V. The workloads consist of key and mouse click events generated by a particular application during the months that a given user traced his machine. We use two application workloads for each user; Table VI.4 describes these sixteen workloads and their characteristics. Note that for each user, we use two applications, one of which is explorer, the Windows GUI desktop. More information about these applications can be found in Section B.2.

For these experiments, we used a workload simulator that simulates the workloads derived from VTrace traces. The simulator reads a description of a task from a file describing

Workload	**User #**	**Application**	**Key events**	**Click events**
1	1	explorer	17,105	9,549
2	2	explorer	27,972	19,866
3	3	explorer	9,905	2,617
4	4	explorer	11,276	6,301
5	5	explorer	21,297	9,291
6	6	explorer	6,096	5,381
7	7	explorer	6,938	4,443
8	8	explorer	24,208	9,337
9	1	netscape	797,642	22,512
10	2	iexplore	193,823	59,667
11	3	psp	64,229	3,320
12	4	outlook	359,839	14,984
13	5	outlook	109,202	2,633
14	6	grpwise	275,972	13,576
15	7	winword	50,799	2,766
16	8	excel	13,891	2,016

Table VI.4: VTrace application workloads we use in certain simulations

its task type group and its duration in cycles. It simulates that task by performing additions repeatedly in a tight loop for the given number of cycles. It indicates the beginning of each task to RightSpeed with an explicit RSLib call, and sleeps for 2 ms at the end of each task to let RightSpeed automatically detect the end of the task. We run this workload simulator on the Transmeta machine to measure the performance obtained when RightSpeed schedules the speeds.

We evaluate RightSpeed's performance on these workloads as follows. For each such workload, we estimate how many deadlines it theoretically should miss, and how much total delay past deadlines it should achieve. We also report how many deadlines RightSpeed actually missed and the total delay it actually achieved. Table VI.5 gives results when RightSpeed uses the PACE calculator to compute optimal schedules to produce average pre-deadline speeds of 630 MHz for keystroke events and 765 MHz for mouse click events.

We find that RightSpeed misses 1.5% fewer to 0.3% more deadlines than the target, with an average absolute error of 0.4%. Also, it has delay from 0.5% less to 0.1% more than the target with an average absolute error of 0.2%. Since RightSpeed conservatively rounds

Workload	**Target number of deadlines to miss**	**Deadlines missed by RightSpeed**	**Error**	**Target total delay**	**Total delay using RightSpeed**	**Error**
1	560	554	-1.1%	33.9 sec	33.8 sec	-0.2%
2	682	681	-0.1%	33.0 sec	32.9 sec	-0.3%
3	104	104	0.0%	5.6 sec	5.6 sec	-0.3%
4	240	237	-1.3%	7.0 sec	7.0 sec	-0.5%
5	1,197	1,188	-0.8%	91.6 sec	91.5 sec	-0.1%
6	65	64	-1.5%	3.4 sec	3.4 sec	-0.2%
7	192	192	0.0%	13.8 sec	13.8 sec	-0.1%
8	1,085	1,081	-0.4%	57.7 sec	57.6 sec	-0.3%
9	4,822	4,809	-0.3%	481.4 sec	481.0 sec	-0.1%
10	19,058	19,067	+0.0%	451.0 sec	451.3 sec	+0.1%
11	575	575	0.0%	258.1 sec	258.1 sec	-0.0%
12	192	192	0.0%	9.2 sec	9.2 sec	-0.2%
13	1,035	1,031	-0.4%	35.5 sec	35.4 sec	-0.3%
14	583	580	-0.5%	178.0 sec	178.0 sec	-0.0%
15	675	677	+0.3%	13.2 sec	13.2 sec	-0.2%
16	58	58	0.0%	2.1 sec	2.1 sec	-0.4%
Average absolute error			0.4%			0.2%

Table VI.5: Deadlines missed and time taken past deadlines by RightSpeed on various workloads

speeds for intervals to maximize the probability of making deadlines, it is not surprising that it tends to miss fewer deadlines and have less delay than the target. Nevertheless, the absolute error is very low, showing that RightSpeed is effective at meeting performance targets even though it must use the millisecond-granularity timer of Windows 2000 and even though Windows 2000 makes no guarantees about when speed-changing routines will actually execute.

VI.6.4 Time to perform PACE calculations

We also measured the average time to perform PACE calculations for tasks. We performed this experiment for the user 1 workload running explorer. We found that adding the sample value to the task type group information and recomputing the schedule accordingly

Workload	**Energy without RightSpeed**	**Energy with RightSpeed mimicking LongRun$^{\text{TM}}$**	**Increase**
A	172.774 J	173.978 J	+0.7%
B	94.581 J	94.86 J	+0.2%
C	1007.228 J	1006.314 J	-0.1%
D	126.786 J	126.786 J	0.0%
E	379.056 J	376.047 J	-0.8%

Table VI.6: Comparison of using built-in LongRun$^{\text{TM}}$ scheduling versus doing this scheduling with RightSpeed

took an average of 4.447 μs $\pm$ 0.312 μs (the 95% confidence interval) with a standard deviation of 143.633 μs. Note that these were always performed at the slowest, 500 MHz setting. So, we see that PACE calculations can be made quite quickly given all our optimizations.

VI.6.5 Effect of RightSpeed overhead on energy consumption

To evaluate the effect of RightSpeed overhead on energy consumption, we ran some workloads on the Transmeta machine both with and without RightSpeed. To equalize performance, we instructed RightSpeed to not use the PACE calculator but instead use an algorithm identical to Transmeta's LongRun$^{\text{TM}}$ strategy. Table VI.6 shows the results for five short workloads derived from VTrace traces. We see that simulating LongRun$^{\text{TM}}$ with RightSpeed has little effect on the total energy consumption. In other words, the overhead of signaling the beginnings and ends of tasks, and of implementing the speed schedule in software instead of hardware is unnoticeable. Incidentally, although not shown here, the performance characteristics (deadlines missed and total delay) of RightSpeed mimicking LongRun$^{\text{TM}}$ are very close to that of LongRun$^{\text{TM}}$ by itself, so the direct comparison of energy consumption is valid.

VI.6.6 Effect of PACE on future processors

Because the real processors we implemented RightSpeed on do not have a large range of available and worthwhile speeds, PACE will not save sufficient energy on them to make its implementation worthwhile. To evaluate the effectiveness of our PACE calculator, in this section we conduct simulations assuming future processors with better DVS characteristics. Our simulations differ from those in Chapters IV and V in that we do not make the unrealistic assumptions about scheduling capabilities that those simulations do. In particular, we take into account a finite set of speeds and limited timer granularity.

For our simulations, we consider three processors, each with a minimum setting running at 200 MHz and consuming 1 W, and each with power consumption proportional to the cube of the speed. (The cubic relationship assumes either a very low threshold voltage or a threshold voltage that can be varied proportionally to supply voltage using technology such as that suggested in [KSM$^+$98].) The three processors differ only in their maximum speeds: 600 MHz, 800 MHz, and 1 GHz. We assume that the processors can only attain speeds that are multiples of 50 MHz. We also assume a timer capable of producing schedules with a granularity of 0.1 ms.

We will use eight workloads, each corresponding to a single traced user described in Chapter V. Since our simulations occur on virtual hardware, we can use longer workloads. Thus, for each workload we use the tasks triggered by keystroke events delivered to *all* applications.

All the algorithms we simulate, except for the no-DVS algorithm, will use the same performance target, so that we can compare them fairly using only energy consumption. The performance target is to have an average pre-deadline speed of 400 MHz and a post-deadline speed of 600 MHz. The four algorithms we consider are:

- **Flat.** The pre-deadline speed is constant.
- **Stepped.** The pre-deadline speed begins at 200 MHz and is incremented by 50 MHz after each interval. Interval length is chosen to achieve the desired average pre-deadline speed. This models algorithms such as that used by Transmeta's LongRun$^{\text{TM}}$ [Kla00].
- **Past/Peg.** The pre-deadline speed is constant at 200 MHz for the first interval, then is pegged to the maximum. Interval length is chosen to achieve the desired average

	Max speed 600 MHz					Max speed 800 MHz			Max speed 1 GHz		
User	No DVS	Flat	Past/Peg	Stepped	PACE	P/P	Step.	PACE	P/P	Step.	PACE
1	44.83	23.29	16.11	14.80	13.67	15.85	13.29	11.87	16.90	12.38	10.92
2	112.36	67.00	64.62	57.07	53.60	69.44	49.37	45.59	81.19	44.75	40.93
3	81.93	39.62	31.19	25.25	23.34	35.78	23.80	21.05	42.44	22.93	20.06
4	48.07	22.20	8.44	9.04	7.78	8.90	8.66	7.39	9.75	8.44	7.18
5	80.24	41.70	25.45	24.59	23.43	25.76	21.86	20.70	28.26	20.23	19.13
6	51.20	23.47	12.02	11.12	10.09	11.71	10.80	9.39	12.39	10.61	9.17
7	132.34	77.22	73.37	64.29	61.26	78.65	55.99	52.48	91.16	51.00	47.47
8	84.75	45.02	40.32	33.74	32.10	44.84	30.43	28.55	53.09	28.44	26.61
Avg	79.46	42.44	33.94	29.99	28.16	36.37	26.77	24.63	41.90	24.85	22.68

Table VI.7: Simulation results showing average per-task energy consumption, in mJ, for various algorithms, workloads, and maximum CPU speeds. All algorithms except "No DVS" achieve the same performance target by using a 400 MHz average pre-deadline speed and a 600 MHz constant post-deadline speed.

pre-deadline speed. This models the algorithm suggested in [GLF$^+$00].

- **PACE.** The pre-deadline speed schedule is computed by PACE using an estimate of task work distribution derived from the most recent tasks of the same type.

Results are in Table VI.7 and summarized in Figure VI.5. Note that Flat does not change its behavior based on the set of speeds available on the CPU, so we only present its results for a 600 MHz maximum speed.

One interesting observation is that the more speeds available on the CPU, the more energy efficient the Stepped and PACE algorithms become. For example, per-task average CPU energy consumption under PACE decreases 19.5% when switching from a CPU with maximum speed 600 MHz to one with maximum speed 1 GHz. It may seem surprising that a system with a faster CPU is more energy efficient, but there is a logical explanation. The availability of a higher speed on the CPU allows a schedule to begin a task running more slowly, since it can more easily make up for this slowness by running even faster later in the schedule. The ability to run slowly at the beginning saves energy in the common case where the task requires little work, since the schedule never proceeds past the low-energy beginning part. PACE takes advantage of the availability of a broader range of speeds to find a better schedule, while Stepped just happens to work better with a greater set of speeds. Past/Peg, on the other hand, does worse when a greater range of speeds becomes available. Essentially, Past/Peg ignores all but the two extreme settings of the CPU, and we see that this is costly in terms of energy consumption; we conclude that using intermediate speeds can save energy.

We also see from these results that PACE is always the best algorithm, followed by

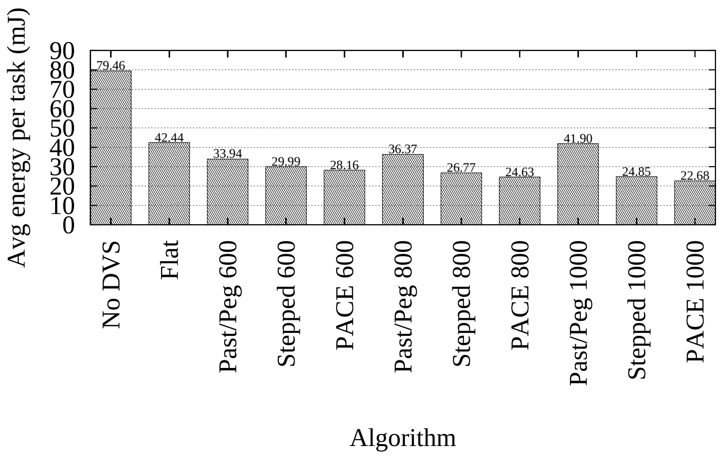

Figure VI.5: Simulation results showing average per-task energy consumption averaged over all workloads for various algorithms. Numbers after an algorithm identify the maximum CPU speed made available to that algorithm; for instance, "PACE 600" means PACE restricted to only use speeds no greater than 600 MHz. All algorithms except "No DVS" achieve the same performance target by using a 400 MHz average pre-deadline speed and a 600 MHz constant post-deadline speed.

Stepped, followed by Past/Peg, followed by Flat. This echoes the results from [LS02], and shows that even when we require PACE to deal with limited settings and timer granularity, it still gives an improvement over existing DVS algorithms.

Furthermore, we predicted in Section VI.3 that the more settings available, the better PACE would do in comparison to other algorithms, and we see this borne out in our simulation results. On the CPU with maximum speed 600 MHz, PACE reduces energy consumption by 6.1% compared to Stepped; with maximum speed 800 MHz, the reduction is 8.0%; with maximum speed 1 GHz, the reduction is 8.7%.

In conclusion, we find that even when there are a limited set of speeds available and limits to timer granularity, PACE is still an improvement over other algorithms. We find that having more, higher speeds available on the CPU helps PACE reduce energy consumption, and furthermore PACE does better the greater the range of speeds available on the CPU. This is an important lesson for chip designers, who may think that providing the capability of running at high voltages and therefore high speeds will increase energy consumption. We see here that with proper energy management using PACE, provision of higher speeds can actually *reduce* energy consumption.

VI.7 Future work

VI.7.1 Modifying applications

An important next step in this research is to insert calls to RightSpeed into various applications, such as movie players, to communicate task information to RightSpeed. We have shown that RightSpeed is good at meeting deadline targets, and this will pay off better once we modify applications in this manner.

VI.7.2 User testing

In this chapter, we have relied on user interface studies that suggest a connection between making deadlines and user-perceived response time instead of conducting user experiments ourselves. It will be important in future work to make sure that the performance targets RightSpeed assigns to automatically detected tasks ensure a satisfactory user experience.

VI.7.3 PACE calculator

We hope in future to test the PACE calculator on a real system with a large range of worthwhile settings to evaluate its actual effect on the energy consumption of such a system.

VI.7.4 Specification of performance targets

For some applications, the best way to specify performance targets may not be to describe the average pre-deadline speed or an equivalent DVS algorithm. It may be better to specify a target fraction of deadlines to make, for example. In Section IV.7.5.2, we discussed the difficulties in developing an algorithm that satisfies such a target, and more work is necessary to determine a solution to this problem.

VI.7.5 Predicting I/O

Our approach to dealing with I/O is somewhat unsatisfactory, as we do not consider the I/O time a task requires until after it actually occurs. A better approach would be to model the probability distribution of task I/O requirements and use this distribution to compute a more optimal schedule at the outset of the task. This requires a more complicated model of speed and voltage scheduling, and consequently a more complicated solution to computing an optimal schedule than PACE currently uses.

VI.8 Conclusions

We have implemented RightSpeed, a task-based speed and voltage scheduler for Windows 2000, to take advantage of dynamic voltage scaling capabilities on Transmeta and AMD chips (and potentially others). Unlike traditional dynamic voltage scaling systems, which use interval-based methods to change speed merely according to recent CPU usage, RightSpeed schedules CPU speed and voltage by taking tasks and their deadlines into account. RightSpeed accomplishes its goals using the interposition techniques of Appendix A, the PACE theory of Chapter IV, and what we discovered about Windows workloads in V.

RightSpeed obtains task information in two ways. First, applications can use its virtual file system interface to directly indicate when tasks begin and end, what task type they belong to, and what policy should be used for each task type. Second, RightSpeed uses an *automatic task detector* to infer task information for applications that do not use the RightSpeed task specification interface. To automatically detect tasks, RightSpeed interposes the user interface event delivery system; when a user interface event occurs, it assumes a task of a type corresponding to the type of the user interface event begins.

RightSpeed also features a *PACE calculator.* This allows RightSpeed to automatically monitor the work requirements of tasks as they complete, deduce a probability distribution of work requirements for each task type, and from those to compute optimal schedules for scheduling CPU speed when tasks of those type run. It then uses the theory of PACE as described in Chapter IV to compute these schedules.

The systems to which we have ported RightSpeed have DVS characteristics quite different from the idealized conditions given in Chapter IV. They have limited scheduling granularity, a limited supply of speeds, and an irregular relationship between speed and energy consumption. In addition, at least one system contains speeds that are not worthwhile for PACE schedules. We have therefore developed and described the techniques we used to apply PACE to such real systems.

We measured the overhead of RightSpeed to demonstrate the feasibility of using task-based deadline scheduling along with automatic task detection and PACE calculation. We found that the overhead due to low-level system modifications, including monitoring when I/O's occur and increasing the resolution of the timer interrupt, is small, on average only 1.2%. This would be lower if there were more operating system support for these modifications. We found that overhead due to other aspects of RightSpeed are quite modest, on the order of a few microseconds to perform most operations. Even PACE calculation, involving complex floating-point operations, takes on average about 4.4 μs per task, even running at the slowest available speed of 500 MHz. We were able to achieve this low computation time due to several optimizations we described.

We were also able to demonstrate that RightSpeed is effective at scheduling tasks in the manner requested, i.e., with the performance constraints specified by an application. It produces rates of deadline misses and quantities of excess time spent passed deadlines quite close to those expected from perfect scheduling. This is an important finding considering that Windows 2000 is not a real-time operating system and does not provide scheduling guarantees, even to the kernel-mode speed scheduler.

Unfortunately, the characteristics of the machines on which we implemented RightSpeed cannot demonstrate the usefulness of PACE in reducing energy consumption of tasks. Both processors have a limited set of available settings, and effectively have even fewer considering that several settings are of limited usefulness due to their relative energy efficiency. We believe that the next generation of processors will feature more worthwhile settings over a greater range of speeds, enabling greater energy savings from PACE.

We performed simulations of such processors, and found that our version of PACE, optimized for speed and modified to take into account limits of speed and time granularity on real systems, still saves energy compared to other algorithms. We find that as long as one uses the PACE algorithm, energy savings from DVS improve with larger ranges of available speeds. For example, on a CPU with a speed range of 200 MHz–1 GHz, we consume 19.5% less energy than on a CPU with a speed range of 200 MHz–600 MHz, even when power consumption is the same on both CPU's at identical speeds. Furthermore, PACE is more effective at improving algorithms when the CPU has a greater speed range. PACE reduces energy consumption compared to the Stepped algorithm by 6.1% when the speed range is 200 MHz–600 MHz; this improvement rises to 8.7% when the speed range expands to 200 MHz–1 GHz.

An important lesson from this is that the current trend for chip manufacturers to scale down the maximum speed of their processors for running in mobile environments may be misguided. Providing the ability to run at a high speed, even if it can only be for a limited amount of time due to thermal constraints, can not only make a processor more attractive to consumers evaluating them in terms of their maximum performance, but can also actually reduce energy consumption by providing DVS algorithms with more options. However, to take advantage of these options, the system needs to use an algorithm like PACE that only uses high speeds on those rare events when they are necessary to satisfy the performance constraints of particularly long tasks.

In conclusion, we have demonstrated that we can perform task-based speed and voltage scaling efficiently in Windows 2000 using documented operating system modification techniques, without modifying or even seeing proprietary Microsoft code. In addition, we have shown that even when applications are not written to communicate task and deadline information to the DVS system, we can infer when tasks begin and end using documented operating system interposition techniques. Finally, although our PACE calculator is not useful for modern processors due to limited ranges of useful settings, we expect future processors to have larger such ranges and for PACE to be more effective on them. We expect manufacturers will soon release such processors, capable of both very low and very high speeds, to satisfy consumers' demand for both energy efficiency and high performance.

Chapter VII

Conclusions and Future Work

VII.1 Conclusions

Limiting energy consumption of computers was once an issue only for supercomputers and small embedded systems. However, in the last decade, it has become a pervasive goal in computer design. The most important reason for this is the growing use of portable and embedded computers, which have limited battery capacities. Another reason is that high energy consumption by desktop computers translates into destructive heat, annoying fan noise, and increased expense. This latter problem has become increasingly important as energy prices have risen.

This thesis concerns techniques for reducing the energy consumption of computers by reducing the energy consumption of their processors. Modern processors have low-power states, such as sleep mode, that consume less power but also perform no work and cause some delay in order to resume full functionality. They also have the ability to dynamically alter their voltage in order to raise or lower energy consumption, with the caveat that the lower-energy voltage levels require slower operation. Thus, the difficulty in designing an energy reduction technique is ensuring that it does not overly impact performance.

The thesis of this work is that *operating systems should have a significant role in processor energy management.* Applications generally do not have enough knowledge of the totality of demands on the processor to make intelligent choices; furthermore, application developers typically do not welcome adding the complexity of hardware management to their role. In addition, while the BIOS and processor possess information about the current state of processing requirements, they lack contextual information that will aid them in evaluating future requirements, even a few cycles ahead. The operating system, on the other hand, knows what threads and applications are running, and can predict their future processor requirements based on past usage and on how they have been interacting with the user.

The contributions of this work were as follows. First, we motivated the use of software for controlling energy management decisions by describing how software has traditionally been successfully applied to this regime. The software approaches we surveyed fall into three categories: transition, load-change, and adaptation. Transition strategies choose when to switch a component from one state to another. Load-change strategies reduce the functionality needed by a component, so that it can operate in low-power modes more often. Finally, adaptation strategies modify system operation in order to accommodate substantially different hardware with lower energy consumption. We consider strategies that manage not only the processor but also the hard drive, the wireless interface, the display, and other components.

Next, we described operating system techniques we developed that improve the use of processor sleep states. We observed that considerable power can be saved by turning off the CPU when it is not doing useful work, but that in Apple's MacOS 7.5, idle time is often converted to busy waiting, and generally it is very hard to tell when no useful computation is occurring. We suggested several heuristic techniques for identifying this condition, and for temporarily putting the CPU in a low-power state. These techniques included turning off the processor when all processes are blocked, turning off the processor when processes appear to be busy waiting, and extending real time process sleep periods. We used trace-driven simulation, using processor run interval traces, to evaluate the potential energy savings and performance impact. We found that these techniques save considerable amounts of processor energy (as much as 66%), while having very little performance impact (less than 2% increase in run time). Implementing the proposed strategies should increase battery lifetime by approximately 20% relative to the CPU power management strategy of MacOS 7.5, since the CPU and associated logic are responsible for about 32% of power use; similar techniques should be applicable to operating systems with similar behavior.

Next, we turned our attention to techniques for choosing how to dynamically scale a processor's voltage and speed. Such scaling is useful and effective when it is immaterial when a task completes, as long as it meets some deadline. We showed how to modify any dynamic voltage scaling (DVS) algorithm to keep performance the same but minimize expected energy consumption. We refer to our approach as PACE (Processor Acceleration to Conserve Energy) since the resulting schedule increases speed as the task progresses. Since PACE depends on the probability distribution of the task's work requirement, we presented methods for estimating this distribution and evaluated these methods on a variety of real workloads. We also showed how to approximate the optimal schedule with one that changes speed a limited number of times. Using PACE causes very little additional overhead, and yields substantial reductions in CPU energy consumption. Simulations using real workloads showed that PACE reduces the CPU energy consumption of previously published algorithms by up to 49.5%, with an average of 20.6%, without any effect on performance.

Next, we analyzed extensive traces of system usage on real machines to determine what characteristics DVS algorithms should have, with an emphasis on how they should treat tasks triggered by user-interface events. We described a simple approach to estimate when such tasks are complete, and showed that this approach is far more efficient and nearly as effective as more complex approaches. Also, we found substantial differences between the amount of CPU time required to handle the three different types of user-interface event: mouse movements, mouse clicks, and keystrokes. This suggests that DVS algorithms should treat these user-interface events differently. In particular, mouse movements require so little CPU time that the best approach is likely to use the minimum speed available to process them. Furthermore, we found significant differences in CPU time requirements between different categories of the same event type, e.g., between pressing the spacebar and pressing the enter key. We also found significant differences between the requirements for events of the same type and category but delivered to different applications. This suggests that DVS algorithms should consider separately different categories of event and different applications. We support these findings with simulations assuming a processor with certain DVS characteristics. We found that using the minimum speed available for mouse movements reduces deadlines made by only 0.2% while reducing energy consumption by 77.5%. We found that for processing keystrokes, PACE reduces the energy consumption of the best non-PACE algorithm by 5.0%, but using data about the application and category of each user-interface event allows it to reduce energy consumption by an additional 1.5%. For mouse clicks, PACE reduces the energy consumption of the best non-PACE algorithm by 1.5%, but using data about the application and category of each user-interface event allows it to reduce energy consumption by an additional 0.5%.

Finally, we described RightSpeed, a task-based speed and voltage scheduler we designed for Windows 2000. This scheduler incorporates our system interposition techniques, our PACE theory, and our observations from workloads about desirable DVS algorithm properties. It runs on systems using Transmeta and AMD chips capable of dynamic voltage scaling. RightSpeed obtains task information either directly from applications using a customized interface or by automatic task detection, which it uses to infer the characteristics of tasks triggered by user-interface events. RightSpeed features automatic schedule computation, which uses the theory of PACE, suitably modified to operate under the constraints of a real system, to automatically optimize speed scheduling for each task type. We found that the low-level system modifications RightSpeed makes produce less than 2% time overhead on the system. Furthermore, we showed that individual operations RightSpeed performs consume very little overhead, typically on the order of microseconds. We also demonstrated that despite the lack of a real-time scheduler on Windows 2000, RightSpeed is able to use best-effort timers to produce schedules that have performance within 1.5% of target values. Unfortunately, modern processors do not have sufficient concavity in their power versus speed curves to demonstrate the energy that RightSpeed can potentially save using our methods. Hopefully, future generations of processors will have sufficient variation in power levels between speeds to allow RightSpeed to demonstrate its potential energy savings.

VII.2 Future Work

Much interesting and important work remains to be done in the area of processor energy management, much of it in the area of dynamic voltage scaling.

We have shown that PACE works best when many speed and voltage settings are available, when the power versus speed curve has a large second derivative, and when the time and energy to make a transition between settings is small. Processors of the current generation lack most of these features, but we hope that in a few years processors will have them. When such processors come out, it will be helpful to examine exactly how effective PACE is on those processors.

One failing of our algorithms is that they do not adequately deal with multiple overlapping tasks. Other researchers, such as Pillai et al. [PS01], have suggested methods for dealing with this issue, but they do not incorporate the lessons of PACE. A key area of future work is incorporating PACE into these methods.

Another problem with our algorithms is that they do not adequately deal with time tasks wait for I/O. We showed that certain types of task wait for I/O relatively often, and for such tasks it may be worthwhile to anticipate at the task's outset the probability distribution of the amount of I/O time the task will likely incur. Determining how to efficiently predict and use such information is useful future work.

We argued that it is important to infer task information from applications that do not provide such information themselves, but we only described a technique for inferring such information from user interface events. We showed that such events only trigger a fraction of the CPU usage of typical machines, so more techniques are necessary to infer more information about system tasks. One approach is to infer information about periodic tasks, as suggested by Flautner et al. [FRM01]. More work should be done to see how effective such techniques would be on real workloads.

We have concentrated our research on techniques applicable to laptops, but some of our results may be applicable to server environments. Some researchers have suggested that since idle servers consume a great deal of power even with the processor off, the key to energy management in server clusters is deciding when to turn off entire machines rather than choosing what processor speed to use [CAT$^+$01]. However, we feel that although this method deals well with workload variability with a granularity of minutes or hours, it does not adequately deal with short-term workload burstiness with a granularity of seconds. Future work is needed to decide whether dynamic voltage scaling can be beneficial in such circumstances.

Appendix A

Windows NT/2000 Operating System Interposition

A.1 Introduction

In this thesis, we frequently need to analyze energy usage patterns of modern portable computer operating systems. We expect our techniques to be applicable to laptops to appear in the next few years, so we use traces of an operating system representative of operating systems laptops will use in the future. Windows is the most popular operating system among such computers, so we decided to collect traces of Windows machines to characterize the usage of such computers. The upcoming release of Windows will be Windows XP, based on the Windows NT/2000 architecture, so we decided to trace machines running Windows NT and Windows 2000.

Writing a tracer for an operating system is difficult for many reasons. An operating system is a large, complex piece of software to analyze. Debugging code that runs before the system has fully started up is tricky. Many runs require a reboot of the computer, and failed runs can necessitate reinstalling the entire operating system or even reformatting the hard drive. However, writing a tracer for Windows NT/2000 is *especially* difficult, because source code is unavailable, descriptions of its internal operations are largely unavailable, and documentation of its interface is incomplete.

Our research required that we deal with all of these problems. Since we were studying the effect of varying the CPU voltage and clock speed, and powering down various system components, we needed to know when various power-consuming components (such as the CPU, the disk, and the network interface card) were active and what they were doing at each instant. Thus, we needed traces of several different types of system objects: processes, threads, messages, waitable objects, key presses, file systems, disks, and the network. Furthermore, we wanted the tracer to be non-intrusive and to respect the confidentiality of users' data so that users would agree to let us trace their systems. The tracer we developed, VTrace, contains over 30,000 lines of code in C, C++, and some assembler. In this appendix, we describe what we did to accomplish this so that the reader can repeat and extend these techniques.

The appendix is organized as follows. In Section A.2, we describe the information resources we found helpful. We report the difficulties we had in setting up a debugging environment and how we overcame them in Section A.3. Section A.4 describes how we adapted standard techniques to create drivers that perform logging and that filter accesses to the keyboard, file systems, network, and hard disks. In Section A.5, we describe our unique approach to logging context switches in Windows NT. Sections A.6 and A.7 describe how we logged Win32 system calls and NT kernel system calls. In Section A.8, we discuss how we parse the master file table of NTFS partitions to deduce file system metadata information. In Section A.9, we briefly consider some of the more interesting miscellaneous interesting features of the tracer. In Section A.10, we discuss how we modified the tracer to work on Windows 2000. We present results from benchmarks that show how much overhead VTrace places on the system in Section A.11. In Section A.12, we describe software that uses techniques similar to those used in VTrace. Finally, we summarize in Section A.13.

A.2 Information Resources

Writing software that traces system activity requires a great deal of information, both about how the operating system works and about how to write tracing tools. Microsoft provides a lot of this information through their developer tools, magazine, and web sites. Unfortunately, this information is not sufficient, since (1) Microsoft does not document many aspects of the internal operations and interfaces of Windows NT/2000, and (2) when Microsoft *does* describe something, this description may be difficult to understand or to generalize, e.g. when the documentation is simply one source code example. To be successful, therefore, we needed information from many sources: developer tools, USENET, magazines, books, and web sites.

Useful information came with a few of our development tools. Microsoft Developer Studio, naturally, has help that describes the interfaces to many well-documented Win32 system calls. It also has sufficient help on Microsoft Foundation Classes (MFC) that we were able to learn MFC programming without a book. The Windows NT and Windows 2000 Driver Development Kits (DDK's) have extensive help systems that describe basic and advanced driver development; they also include some useful sample source code. Finally, Microsoft's kernel-mode debugger, WinDbg, comes with some information about how to use it.

USENET was a good source for discussions of real-world problems and solutions in Windows NT/2000 programming. The useful articles are in the comp.os.ms-windows.programmer.* hierarchy. For our purposes, the most useful newsgroup was comp.os.ms-windows.programmer.nt.kernel-mode, which covers kernel-mode programming. To get the most out of USENET, we recommend using a site that can search USENET archives, such as Deja News (http://www.dejanews.com).

Magazines were another source of useful information. Open Systems Resources, Inc. (http://www.osr.com, ntinsider@osr.com) provides free subscriptions to *NT Insider*, which discusses many useful aspects of driver development. *MSDN Magazine* (http://msdn.microsoft.com) also contains many helpful articles, especially the Under the Hood columns by Matt Pietrek. Windows Developer's Journal is another good source. Finally, we recommend *Dr. Dobb's Journal* (http://www.ddj.com), especially for its articles by Mark Russinovich and Bryce Cogswell.

We also used books. "Inside Windows NT" and "Inside the Windows NT File System," both by Helen Custer, are useful for a general overview of the operating system and file system, although they do not offer aid in actual programming [Cus93, Cus94]. In contrast, "Windows NT File System Internals," by Rajeev Nagar, both describes how the file system works and provides practical advice for interfacing with it [Nag97]. For basic Windows NT programming strategies, we used "Windows NT 4 Programming from the Ground Up," by Herbert Schildt [Sch97]. Late in the process of VTrace's development, we discovered the book "Windows NT/2000 Native API Reference," by Gary Nebbett [Neb00]; this book provides documentation for a great deal of otherwise undocumented internal system calls and structures.

However, by far the most useful source of information, without which the tool might never have been developed, was the World Wide Web. Often, we found the solution to a problem simply by using a web search engine on key words or phrases. The web also has sites containing large, structured bodies of information on Windows NT/2000. For example, the Systems Internals web site (http://www.sysinternals.com) has a great deal of useful information, utilities, and even source code for Windows NT systems programming. Also, the Microsoft Developer Network Library (http://premium.microsoft.com/msdn/library) has helpful articles and documentation.

We thus observe that although Microsoft does not completely document Windows NT/2000, so many developers have used it and are willing to share information that it has become extensively, if unofficially, documented.

A.3 Creating a Debugging Environment

A tracer contains and interacts with a lot of code that runs in kernel mode, so we needed a kernel-mode debugger. Furthermore, since much of the code in a tracer gets run before the system has completely started up and thus before a debugger program can be launched, we did well to use a two-system debugging environment. In such a configuration, the debugger runs on the *host* machine (also the development machine), and the software under test runs on the *target* machine. The debugger monitors and controls the target machine via a serial cable connecting the two machines. Another advantage to this approach is that a reboot or reinstallation of the operating system on the test machine does not affect the development environment.

Unfortunately, setting up kernel debugging with our debugger, WinDbg, is notoriously difficult. Some of the hardest things are configuring the debugger program settings correctly and making the target machine communicate with a remote debugger. For this, documentation included with the debugger is helpful, as are stories on the web about user experiences. However, even with all this, we still had difficulty. We finally succeeded once we discovered we had an old, buggy version of WinDbg, and downloaded a fixed version from an obscure location at Microsoft described in an equally obscure USENET article posted by Paul Sanders (paulsan@microsoft.com) to the kernel-mode programming newsgroup on October 17, 1997. (The latest version of WinDbg is now easier to find and lacks these bugs.) We had other problems until we realized we needed to upgrade the debug symbol files to match the service pack installed on the target machine. We searched the web, and found we could download these files from Microsoft's FTP site.

Once we had the debugger set up, it worked very well, enabling us to easily set breakpoints in source code, step through source code, examine and change runtime variable values, and even view operating system code (albeit in uncommented assembler).

When the target machine being debugged is running Windows 2000, sometimes it will inexplicably hang while starting up. To get past this, use the Break command in the Debug menu of WinDbg.

A.4 Drivers

A.4.1 Background

Driver programming for Windows NT/2000 is a subject of tremendous breadth, so our treatment here will necessarily be incomplete. [Cus93], [Nag97], back issues of NT Insider, and the Windows NT and Windows 2000 DDK's discuss it more thoroughly.

In Windows NT/2000, a *device* is an object that can receive I/O requests, such as one that represents a disk drive, a file system, or a keyboard. Devices can be *layered* above each other, in that the top-level device processes its I/O requests by sending I/O requests to the lower-level device. For example, a device for a file system will typically be layered over a physical disk device so that file requests can be translated into disk requests. Some devices create *file objects*, which are pieces of state carried over between I/O requests. Despite the name, file objects can represent more than just open files; they can also represent things like network connection endpoints and open directories.

Device layering is accomplished naturally by the way I/O requests are handled. A device receives a structure called an *I/O request packet* (IRP), which represents an I/O request and contains a stack of *I/O stack locations*. This stack's top location contains a description of the request in a form the device understands. Before that device passes the I/O request to a lower-level device, it pushes onto the stack a new stack location describing the I/O request in a form the lower-level device understands. When that device finishes processing the IRP, the extra stack location is popped from the stack. Thus, when the top-level device gets the IRP back to complete its own processing, the top location of the stack is still the one relevant to that device. The most important fields of an I/O stack location are the major function code, describing the general request type; the minor function code, describing the request type more specifically; and the 16-byte parameters field, whose meaning depends on the function codes. See Figure A.1 for an illustration of this.

A *driver* is the code implementing a class of devices. For instance, all NTFS file system devices use the NTFS driver code for handling requests. This code includes a driver entry routine and several dispatch routines. The driver entry routine does per-driver initialization, including entering the dispatch routines into the dispatch routine table. The operating system uses the dispatch routine table, indexed by major function code, to determine which routine handles a given IRP.

A filter driver implements a filter device, a special kind of device extremely helpful in tracing system events in Windows NT/2000. A filter device can *attach* to an existing device, causing it to intercept any requests destined for that existing device. Typically, it modifies the request in some way, then passes it on to the existing device. This allows it to add functionality to that device, e.g., to turn a traditional file system into an encrypted file system. However, a filter can be used simply to record information about requests, pass those requests on unchanged to the device they were meant for, then record information about the results of those requests.

Figure A.4 gives an example of an initialization routine for a filter driver. This routine, DriverEntry, sets the MajorFunction entries in the driver object so the appropriate dispatch routine gets called for each request type. Those dispatch routines, such as the one shown in Figure A.2, log the request initiation, set a completion routine to be called when

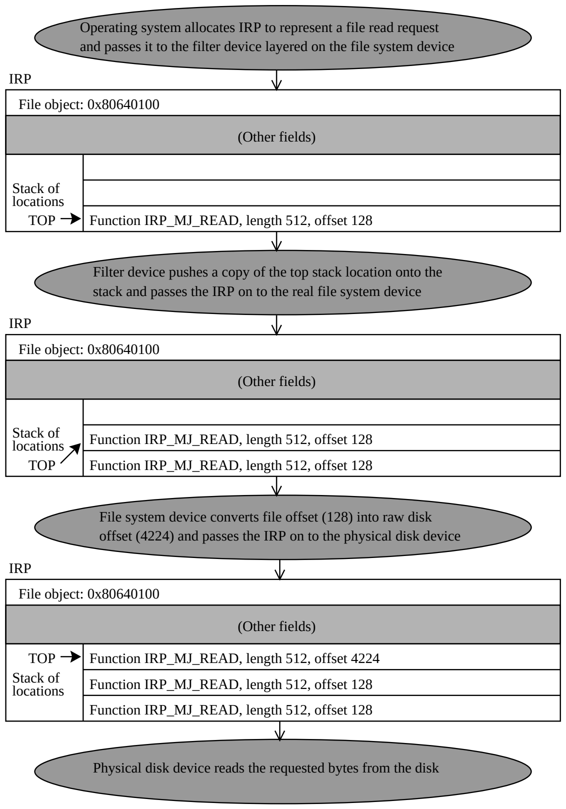
Operating system allocates IRP to represent a file read request and passes it to the filter device layered on the file system device
IRP
File object: 0x80640100
(Other fields)
Stack of locations
TOP
Function IRP_MJ_READ, length 512, offset 128
Filter device pushes a copy of the top stack location onto the stack and passes the IRP on to the real file system device
IRP
File object: 0x80640100
(Other fields)
Stack of locations
TOP
Function IRP_MJ_READ, length 512, offset 128
Function IRP_MJ_READ, length 512, offset 128
File system device converts file offset (128) into raw disk offset (4224) and passes the IRP on to the physical disk device
IRP
File object: 0x80640100
(Other fields)
TOP
Stack of locations
Function IRP_MJ_READ, length 512, offset 4224
Function IRP_MJ_READ, length 512, offset 128
Function IRP_MJ_READ, length 512, offset 128
Physical disk device reads the requested bytes from the disk

Figure A.1: Example of how layered devices use an IRP stack

the request completes, then call the lower-level driver to complete the request. Figure A.5 shows an example of a completion routine that logs the request completion.

A.4.2 Logger

Kernel-mode code can also use device control requests like this to communicate with the logger driver. But, it's more efficient for kernel-mode code to just call the logger driver's functions directly. To get pointers to these functions, a driver makes a single device control request to the logger driver requesting a structure containing all such function pointers.

A major component of VTrace is the *logger*, which accepts and serializes requests to add events to the in-memory log, and periodically writes the log to disk. We implemented

```
NTSTATUS VTrcFSDispatchReadWrite (PDEVICE_OBJECT FilterDevice, IN PIRP Irp)
{
  PIO_STACK_LOCATION  currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
  PIO_STACK_LOCATION  nextIrpStack    = IoGetNextIrpStackLocation(Irp);
  PFILE_OBJECT        fileObject      = currentIrpStack->FileObject;
  PFS_HOOK_EXTENSION  filterExtension = FilterDevice->DeviceExtension;
  PDEVICE_OBJECT      nextDevice      = filterExtension->attachedDevice;
  PCHAR               eventPosInLog   = NULL;
  ULONG               seq;
  KIRQL               oldirql;

  // If the file has a name, log the read or write request.

  if (fileObject->FileName.Buffer) {
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventPosInLog = (sharedState->logEventFunctionPointer)(
      (currentIrpStack->MajorFunction == IRP_MJ_READ ?
       ENTRY_TYPE_FILE_READ : ENTRY_TYPE_FILE_WRITE), 24);
    if (eventPosInLog) {
      seq = InterlockedIncrement(&globalSequenceNumber);
      RtlCopyMemory(&eventPosInLog[1], &seq, 4);
      RtlCopyMemory(&eventPosInLog[5], &fileObject, 4);
      RtlCopyMemory(&eventPosInLog[9],
                    &currentIrpStack->Parameters.Read.ByteOffset, 5);
      RtlCopyMemory(&eventPosInLog[14],
                    &currentIrpStack->Parameters.Read.Length, 4);
      RtlCopyMemory(&eventPosInLog[18], &Irp->Flags, 4);
      eventPosInLog[22] = currentIrpStack->MinorFunction;
      eventPosInLog[23] = currentIrpStack->Flags;
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }
```

Figure A.2: VTrace uses a dispatch routine like this one to handle read and write calls that its file system filter intercepts. This code is continued in Figure A.3.

```
  if (!eventPosInLog) {
    // If we didn't enter the request into the log, we just pass this
    // IRP on normally to the next lower device. We do this by backing
    // up the IRP stack location to reuse the current location for the
    // next lower device. In Windows 2000, use the
    // IoSkipCurrentIrpStackLocation macro for the following.

    Irp->CurrentLocation++;
    Irp->Tail.Overlay.CurrentStackLocation++;
  }
  else {
    // Copy parameters to the next position in the stack for the next
    // lower device. In Windows 2000, use the
    // IoCopyCurrentIrpStackLocationToNext macro for the following.

    *nextIrpStack = *currentIrpStack;

    // Set a completion routine, passing the sequence number as the
    // "context" parameter so that it can be used in the completion
    // log entry.

    IoSetCompletionRoutine(Irp, VTrcFSCompletionRoutine, (PVOID) seq,
                           TRUE, TRUE, TRUE);
  }

  // Pass the IRP on to the lower-level device.

  return IoCallDriver(nextDevice, Irp);
}
```

Figure A.3: This is a continuation of the function in Figure A.2.

```
NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject,
                      IN PUNICODE_STRING RegistryPath)
{
  NTSTATUS status;
  int i;

  // Read driver-shared state from initialization driver.

  status = GetSharedState(&sharedState);
  if (!NT_SUCCESS(status))
    return status;

  // Create dispatch points for all routines that must be handled.
  // All entry points are registered since we might filter a
  // file system that processes all of them.

  for (i = IRP_MJ_CREATE; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
    DriverObject->MajorFunction[i] = VTrcFSPassOnNormally;

  DriverObject->MajorFunction[IRP_MJ_CREATE] = VTrcFSDispatchCreate;
  DriverObject->MajorFunction[IRP_MJ_READ] = VTrcFSDispatchReadWrite;
  DriverObject->MajorFunction[IRP_MJ_WRITE] = VTrcFSDispatchReadWrite;

  // Set up the fast I/O dispatch table. (See the section on fast I/O
  // for details.)

  DriverObject->FastIoDispatch = &VTrcFSFastIoDispatchTable;

  // Note: It would be unwise to unload this driver, so we don't set an
  // unload routine. Otherwise, we would set DriverObject->DriverUnload.

  // Normally there would be code here to attach to some other device
  // or devices, but in VTrace we do this elsewhere.

  return STATUS_SUCCESS;
}
```

Figure A.4: VTrace uses a DriverEntry routine like this one to initialize the file system filter driver.

```
NTSTATUS VTrcFSCompletionRoutine (PDEVICE_OBJECT DeviceObject, PIRP Irp,
                                  PVOID Context)
{
  ULONG seq = (ULONG) Context;
  KIRQL oldirql;
  PCHAR eventPosInLog;

  // Log the return values.

  KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
  eventPosInLog = (sharedState->logEventFunctionPointer)(
    ENTRY_TYPE_FILE_COMPLETE_OPERATION, 13);
  if (eventPosInLog) {
    RtlCopyMemory(&eventPosInLog[1], &seq, 4);
    RtlCopyMemory(&eventPosInLog[5], &Irp->IoStatus.Status, 4);
    RtlCopyMemory(&eventPosInLog[9], &Irp->IoStatus.Information, 4);
  }
  KeReleaseSpinLock(&sharedState->mainMutex, oldirql);

  // Always do the following in a completion routine. (By the way, the
  // braces are necessary since IoMarkIrpPending is a macro that
  // expands to multiple statements.)

  if (Irp->PendingReturned) {
    IoMarkIrpPending(Irp);
  }

  return Irp->IoStatus.Status;
}
```

Figure A.5: VTrace uses a completion routine like this one to log the results of a file system request that just completed.

```
void LogEvent (char *eventDescription, DWORD descriptionLength)
{
  DWORD returnSize;

  HANDLE hDevice = CreateFile("\\\\.\\VTrcLog",
                              GENERIC_READ | GENERIC_WRITE, 0,
                              NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                              NULL);

  if (hDevice != INVALID_HANDLE_VALUE) {
    DeviceIoControl(hDevice,           // Handle to device
                    VTRACE_LOG_EVENT,  // Device I/O control code
                    eventDescription,  // Outbound communication buffer
                    descriptionLength, // Length of outbound buffer
                    NULL,              // Inbound communication buffer
                    0,                 // Length of inbound buffer
                    &returnSize,       // Bytes returned in inbound buffer
                    NULL);             // Unused (for async communication)
    CloseHandle(hDevice);
  }
}
```

Figure A.6: This function, which logs an event, illustrates how user-mode code communicates with a kernel-mode driver.

the logger as a device, so that its code could run in kernel mode. This enables other kernel-mode code, such as that comprising most of the tracer, to call it efficiently. It also allows user-mode code to call it without incurring a context switch, although with the overhead of a kernel trap. The logger driver implements several specialized device control I/O request types, including ones to start logging, stop logging, add an event to the log, flush the log to disk, and change the event mask (the set of event types the logger leaves out of the log). User-mode code makes such requests using code like that in Figure A.6.

The logger periodically changes the event mask (the set of event types the logger leaves out of the log) according to a fixed schedule. This lets us collect the full set of event types only part of the time, to reduce the space taken up by trace files on the user's disk. For example, we collect file-open events (which take up little space) all the time, but thread switches (which are very frequent) only part of the time.

A.4.3 Keyboard filter

To log key presses, we made straightforward modifications to the keyboard filter driver Ctrl2cap, whose code is freely available from the Systems Internals web site. Its original purpose was to exchange the functions of the control and caps lock keys. We simply made it encrypt and log the key presses instead of modifying them.

A.4.4 File system filter

To log file system activity, we made a few modifications to Filemon, another filter driver whose source code is available from the Systems Internals web site. This filter driver, described in [RC97a], logs and displays file system activity. The most important modification we made was in how to choose the file system devices to filter. Filemon requires the user to specify the file systems desired, but we wanted to filter all file systems. Furthermore, we could not simply find all the file systems at system start-up and filter those, since some file systems, such as floppy disks and CD's, may be added dynamically. So, instead, we arranged to hook the NT system call that opens files, check in that hook whether we have yet filtered the file system containing that file, and start filtering any unfiltered file system we find. Section A.7 will discuss how we hooked the file-open and other NT system calls.

One complication in filtering file system devices is an optimization called the *fast I/O path*. If a file system device can handle a request without involving a lower-level device, e.g., during a cache hit, the overhead of creating an IRP is unnecessary. A file system driver can specify a table of *fast* dispatch routines, one for each I/O request type, that can handle requests not packaged in IRP's. If the fast dispatch routine cannot handle the request, e.g. if it needs to use a lower-level device, it returns an error value, making the operating system send an IRP to the regular dispatch routine. To filter accesses that use fast I/O, we must specify a set of fast dispatch routines in our filter driver. These fast dispatch routines will log getting called and pass on calls to the fast dispatch routines of the lower-level driver. Figure A.7 shows a sample logging fast I/O routine. Also, the sample driver initialization code in Figure A.4 has a line to set up the fast I/O dispatch table.

```
BOOLEAN MyFastIoRead (PFILE_OBJECT FileObject, PLARGE_INTEGER FileOffset,
                      ULONG Length, BOOLEAN Wait, ULONG LockKey,
                      PVOID Buffer, PIO_STATUS_BLOCK IoStatus,
                      PDEVICE_OBJECT DeviceObject)
{
  PFS_HOOK_EXTENSION filterExtension = DeviceObject->DeviceExtension;
  PDEVICE_OBJECT nextDevice          = filterExtension->attachedDevice;
  PFAST_IO_DISPATCH nextFastIoTable  = nextDevice->DriverObject->
                                       FastIoDispatch;
  PCHAR eventPosInLog                = NULL;
  BOOLEAN retval;
  ULONG seq;
  KIRQL oldirql;

  // If the next lower driver has no fast I/O routine, return FALSE.

  if ((ULONG) &nextFastIoTable->FastIoRead - (ULONG) &nextFastIoTable >=
      nextFastIoTable->SizeOfFastIoDispatch ||
      nextFastIoTable->FastIoRead == NULL)
    return FALSE;

  // If there is a file name, record this call.

  if (FileObject->FileName.Buffer != NULL) {
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventPosInLog = (sharedState->logEventFunctionPointer)(
      ENTRY_TYPE_FILE_READ, 24);
    if (eventPosInLog) {
      seq = InterlockedIncrement(&globalSequenceNumber);
      RtlCopyMemory(&eventPosInLog[1], &seq, 4);
      RtlCopyMemory(&eventPosInLog[5], &FileObject, 4);
      RtlCopyMemory(&eventPosInLog[9], FileOffset, 5);
      RtlCopyMemory(&eventPosInLog[14], &Length, 4);
      RtlZeroMemory(&eventPosInLog[18], 4);      // no IRP flags
      eventPosInLog[22] = IRP_MN_NORMAL;
      eventPosInLog[23] = '\0';                  // no stack flags
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }
```

Figure A.7: VTrace uses a fast I/O routine like this one to handle fast-path read requests. This code is continued in Figure A.8.

```
  // Call the real fast I/O routine, recording the return value

  retval = nextFastIoTable->FastIoRead(FileObject, FileOffset, Length,
                                       Wait, LockKey, Buffer, IoStatus,
                                       nextDevice);

  // If the call completed successfully, and we logged the call, log
  // the return.

  if (retval && eventPosInLog) {
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventPosInLog = (sharedState->logEventFunctionPointer)(
      ENTRY_TYPE_FILE_COMPLETE_OPERATION, 13);
    if (eventPosInLog) {
      RtlCopyMemory(&eventPosInLog[1], &seq, 4);
      RtlCopyMemory(&eventPosInLog[5], &IoStatus->Status, 4);
      RtlCopyMemory(&eventPosInLog[9], &IoStatus->Information, 4);
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }

  // Return the real routine's return value.

  return retval;
}
```

Figure A.8: This is a continuation of the function in Figure A.7.

A.4.5 Raw disk partition filter

To log activity at the physical disk level, we modified a physical disk filter driver, DiskPerf, whose source code is included in the DDK. This driver collects and reports statistics about raw disk accesses, so it was straightforward to retool it so it instead logged information about each disk access.

A.4.6 Network filter

In contrast to the other filter drivers we needed, we found no source code for a network transport layer filter driver. We thus had to write one from scratch. More precisely,

Minor function code	Meaning
TDI_ASSOCIATE_ADDRESS	Associate a connection endpoint with a network address
TDI_DISASSOCIATE_ADDRESS	Disassociate a connection endpoint with the network address it was previously associated with
TDI_CONNECT	Establish a connection between a local connection endpoint and a specified remote address
TDI_LISTEN	Listen for requests from any of a set of remote addresses to a local connection endpoint
TDI_ACCEPT	Accept a connection request made by a remote address to a local connection endpoint
TDI_DISCONNECT	Terminate the connection in which a connection endpoint is participating
TDI_SEND	Send an ordered packet over a connection
TDI_RECEIVE	Receive an ordered packet over a connection
TDI_SEND_DATAGRAM	Send a datagram over a connection
TDI_RECEIVE_DATAGRAM	Receive a datagram over a connection
TDI_SET_EVENT_HANDLER	Establish a routine for handling a certain type of event, such as the arrival of a datagram
TDI_QUERY_INFORMATION	Get information about some network object, such as its network address
TDI_SET_INFORMATION	Set information about some network object
TDI_ACTION	Perform some transport-specific action

Table A.1: Minor function codes of some useful TDI internal device control requests, taken from the Windows NT DDK help

we had to write one mostly from scratch, since many things are the same from one filter driver to another, such as how to initialize the dispatch table, how to attach to lower-level drivers, etc.

Windows NT/2000 provides a single programming interface, called the transport driver interface (TDI), to the transport layers of all network protocols. (See Table A.1.) I/O requests passed to the transport layer all conform to the same format, described in the Windows NT DDK help and in the DDK files TDI.H and TDIKRNL.H. There were still challenges, however, in building a filter for these requests.

One problem we encountered is that some IRP's have the major function code "device control," and we found no description of the parameter format used by these IRP's. However, we learned from the DDK help that the first thing a device does upon receiving such a request is call the function TdiMapUserRequest to convert it to one with a major function code of “internal device control,” which we know how to interpret. In our filter driver dispatch routine for device control requests, we therefore first call TdiMapUserRequest.

Another problem is caused by an apparent bug in how Windows NT handles network filter devices. When it constructs an IRP, it must allocate enough stack space in it to account for the maximum depth of the device stack the IRP will pass through. To ensure this, each device object has a stack count field indicating how large the stack must be in IRP’s it receives. Unfortunately, Windows NT sometimes ignores the stack count field in our filter device objects and sends it an IRP with insufficient stack space. If we push a new location onto this stack and pass it on, eventually the stack overflows and the system crashes.

We solve the stack space problem in different ways, depending on whether we need to post-process the request. When we need to post-process, we create a new IRP with the appropriate stack space to pass on to the lower-level driver. When we do not need to post-process, we use a trick borrowed from Filemon: we do not push anything onto the stack, allowing the next device down to use the same stack location it used. We can only do this when we do not need to post-process, since if the request were passed back after doing this the stack would be empty.

Yet another difficulty stems from a unique aspect of network devices, namely that not all network I/O uses IRP’s. Specifically, I/O that happens in response to some event, such as a datagram arrival, is performed entirely by functions called *event handlers* and does not involve the dispatch routines. This is unfortunate, since while Windows NT provides filter devices as an elegant, well-supported approach to intercepting IRP’s sent to dispatch routines, it provides no special support for intercepting calls to event handlers.

We overcame this by developing our own technique for intercepting calls to event handlers. The key to this technique is our ability, thanks to filter devices, to intercept and change any request that specifies a new event handler for a file object. (These are the requests with minor function code TDI_SET_HANDLER.) Each of these requests contains the location of the event-handling function, the type of event it handles, and a four-byte context value to be passed to that function. All the driver must do, then, is allocate a structure to store this information, then modify the request so that instead of containing the location of the real event-handling function and the real four-byte context value, it contains the location of a special logging event-handling function and the four-byte address of the structure we allocated. In this way, whenever an event of the given type happens, our special logging event-handling function gets called and passed the address of the structure we allocated. This function logs the event, then inspects the structure so it can call the appropriate event-handling function with the appropriate context value. When that function returns, our special logging function can trace its return value. In actuality, we used a slightly different approach to memory allocation than described above: the driver allocates a single structure per file object, not per event handler; this permits it to quickly free all the memory allocated for a file object when it is closed.

A.5 Logging Context Switches

Logging context switches should be easy, since kernel-mode software can use the function KeSetSwapContextNotifyRoutine to set a function that gets called whenever context switches occur. Unfortunately, this call only works on the *checked build* of Windows NT/2000, a special version that contains extra hooks and symbols for use in driver development. Few people use this version, and we wanted our tracer to run on anyone's machine, so we designed a method that will work on the standard version.

We devised the method as follows. WinDbg comes with debug symbols for Windows NT files, so we used it to find the assembler code for the Windows NT function SwapContext. The function is more than five bytes long, and contains no jumps, branches, or calls in its first five bytes, enabling us to do the following. We overwrite, in memory, its first five bytes with a jump to our own function, NewSwapContext, as shown in Figures A.9 and A.10. NewSwapContext logs the context switch, including the thread being switched to; then executes the first five bytes of the original, pre-overwrite version of SwapContext; then jumps to the sixth byte of SwapContext. This method was inspired by an article in Microsoft Systems Journal unrelated to context switches[Pie94a].

Finding the location of SwapContext in memory is not straightforward, however. It is always in the in-memory image of the kernel executable, NTOSKRNL.EXE, which is loaded at address 0x80100000. However, its position within NTOSKRNL.EXE varies from version to version of Windows NT 4.0. For instance, in the original Windows NT 4.0 it's at 0x8013F4F0, but after applying Service Pack 3 it's at 0x80140CA0, and after applying

Before the context swap hook is in place, SwapContext looks like this:

```
SwapContext:
  mov byte ptr es:[esi+2Dh],2
  or cl,cl
  mov ecx,dword ptr [ebx]
  pushfd
  ... etc. ...
```

After the context swap hook is in place, SwapContext looks like this:

```
SwapContext:
  jmp <the address 6 bytes into NewSwapContext>
  or cl,cl
  mov ecx,dword ptr [ebx]
  pushfd
  ... etc. ...
```

(The 6-byte offset is to skip over the compiler-inserted stack set-up stuff at the beginning of NewSwapContext.)

Figure A.9: How we hook the context switch routine (NewSwapContext is shown in Figure A.10.)

Service Pack 5 it's at 0x80141E70. However, in all these versions, the instructions of the function are unchanged, and we expect this to remain the case in future versions of Windows NT 4.0. Thus, to find SwapContext, we simply search for the known first 28 bytes of the routine in the memory section where we expect it. (We check a few common locations first, since the routine is most likely to be in one of them.) Doing this check is dangerous, since an access to an invalid (e.g., paged out) memory location by kernel-mode software will crash the system. Thus, before checking any location, we first call MmIsAddressValid to make sure we can read it.

When Service Pack 6 became available, we tested this technique on it, and it worked perfectly.

One final complication in logging context switches involves locking. Normally, we require a thread to hold a spin lock when writing to the log. However, we found that the system sometimes crashed when the context-swap hook tried to acquire this spin lock. This may be because acquiring the spin lock can itself cause a context-swap, creating an infinite loop. Thus, we instead disable interrupts while the context-swap routine accesses the log.

```
// NewSwapContext logs the context switch, executes the overwritten
// instruction from the old context swap routine, then jumps to the point
// in the old swap routine past that instruction.

void FASTCALL NewSwapContext (void)
{
    // Save registers we may overwrite.
    push eax
    push ecx
    // Save interrupt mask and stop all interrupts. Since context-swaps
    // can't occur while a spinlock is held, we know no one else has the
    // spinlock.
    pushfd
    cli
    // eventPosInLog = LogEvent(ENTRY_TYPE_THREAD_SWITCH [= 0x0E], 5);
    push 5
    push 0Eh
    call LogEvent
    // if (eventPosInLog)
    cmp eax, 0
    je DoneLogging
    // Save eventPosInLog on stack for later use.
    push eax
    //   * (DWORD *) &eventPosInLog[1] = PsGetCurrentThreadId();
    call PsGetCurrentThreadId
    pop ecx    // Pop eventPosInLog from stack to use now.
    mov dword ptr [ecx+1], eax
  DoneLogging:
    // Restore the interrupt mask.
    popfd
    // Restore saved registers.
    pop ecx
    pop eax
    // Execute overwritten instruction from original swap routine.
    mov byte ptr es:[esi+2dh],2
    // Jump to point in original swap routine past overwritten part.
    jmp dword ptr globals.nonOverwrittenPartOfOrigSwapRoutine
  }
}
```

Figure A.10: VTrace uses this routine for logging context switches.

Code	Explanation
⋮ call _GetMessageA@16 ⋮	Application code. The application calls GetMessageA, which is compiled as an indirect call through __imp__GetMessageA@16.
__imp__PostThreadMessageA@16: 0x10001E50 __imp__GetMessageA@16: 0x10001410 __imp__PeekMessageA@20: 0x10001550 ⋮	Array of function pointers. The location __imp__GetMessageA@16 is part of an array of imported function pointers located in the import data section.
0x10001410: sub esp, 18h push ebx ⋮	Function body. The actual body of the function GetMessageA is at the specified memory location, 0x10001410. This location is part of the memory image of the USER32 DLL.

Figure A.11: An example of how Win32 system calls are performed

We can be sure that no other thread holds the spin lock, since context-swaps cannot occur on a uniprocessor while a thread holds a spin lock. This is the only aspect of VTrace that requires a uniprocessor; but for this operation, it would work as well on a multiprocessor.

A.6 Logging Win32 System Calls

The Windows NT/2000 kernel supports multiple user-level subsystems, such as Win32, POSIX, and OS/2. Thus, the term "system call" is vague; it could refer to a call to the Win32 subsystem, to some other subsystem such as OS/2, or even to the Windows NT/2000 kernel itself. In this section, we will discuss logging system calls to the Win32 subsystem. Section A.7 will discuss logging system calls to the kernel.

Our technique for logging Win32 system calls borrows heavily from the technique developed by Matt Pietrek for his APISPY32 program [Pie94a]. In this section, we describe Pietrek's technique only briefly; the reader is referred to [Pie94a] for a more complete description. We then describe the major ways in which our technique differs from it.

Pietrek's technique relies on a key observation about how applications make Win32 system calls. A Win32 system call is effected by calling a stub function which does an indirect jump to one of an array of function pointers. (See Figure A.11.) We must therefore merely find that array of function pointers (which is easy to do once the image and file format is understood [Kat93, Pie94b]), and replace the pointers to functions we want to log with pointers to our own logging functions. These logging functions, which reside in a special DLL that is part of the tracer software, will call the original functions and log those calls. The tracer must load this special DLL into every application's address space.

Pietrek describes several techniques for loading the logging DLL into every application's address space. We chose the simplest of those, putting the name of the DLL in the registry key HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs. This does not take effect until reboot, but our tracer requires a reboot anyway to install the raw disk filter driver.

APISPY32 was designed for Windows NT 3.5, and some USENET articles report it cannot be used with Windows NT 4.0. The reason is that virtual memory protections prevent the replacement of some of the function pointers. To fix this problem, we call VirtualProtect to temporarily change these protections.

Another problem with APISPY32 is that it only hooks system calls made directly by the application. If an application calls a DLL function, which in turn makes a system call, it does not notice that system call. This is because APISPY32 performs its function interception on the application executable image but not on the image of any loaded DLL. Our solution to this is twofold. First, when the logging DLL is loaded, it calls EnumerateLoadedModules to obtain the memory locations of the application and all its loaded DLL's. Then, it does the function interception in each of those modules. Second, we intercept the LoadLibrary... functions (even though we do not need to log them), so that when one completes we can call EnumerateLoadedModules to intercept functions in all the newly loaded library images. Note that it is not sufficient to consider only the library mentioned in the LoadLibrary... call, since that library may have implicitly or explicitly loaded other libraries. Also note that this implementation will miss logging some calls that libraries make when they initialize.

We found it useful for the logging DLL to be able to determine whether the current thread had performed any recent activity. In this way, we could perform online compression of the log entries logging calls to functions like PeekMessage and SendMessage when the thread was not doing anything else. However, since most thread activity is recorded at kernel level, this required either expensive, frequent communication between kernel and user level, or a region of memory that could be shared between these levels. We opted for the latter approach, to improve the performance of our tracer. Figure A.12 shows how a driver can map a region of non-paged pool to a user-level address. One important issue is that the driver must undo this mapping before the process exits, or the system will crash. Fortunately, to access the driver to request the mapping, a user process must create a "file" representing a link to the driver. When that process is about to terminate, it automatically closes this file. Therefore, VTrace stores the user-level address in the corresponding file object, and unmaps the address when it receives a close request for the file object.

Debugging a logging DLL can be difficult, since any bug in it can make the system unusable, e.g., by making a fundamental application like the login screen fail. If this happens, the only recourse may be to restore the registry to a previous state in which the DLL is not in the AppInit_DLLs list, or to delete the offending DLL file. Each of these approaches is annoying and time-consuming if the system cannot be run normally. One solution is to avoid putting the DLL into AppInit_DLLs and write test applications that explicitly load the DLL. However, this will not test how the DLL works with general applications. The best approach, suggested by a USENET post, is to put the DLL on a floppy disk and tell AppInit_DLLs to get it from there. In this way, if there is a bug, one can simply remove the floppy disk and the DLL will not get loaded into any application.

A.7 Logging NT System Calls

Our approach for logging system calls to the kernel is derived from the Regmon application, available from the Systems Internals web site and described in [RC97b]. The idea is to find the *service table list*, an in-memory array of system call function pointers indexed by system call number, and replace those function pointers with pointers to special logging functions. The trickiest part is figuring out what system call number corresponds to each system call.

Regmon accomplishes this by observing the following about how kernel-mode software makes these system calls. NTOSKRNL.EXE provides the interface to the system calls by exporting functions whose names have the prefix "Zw." Inspecting these functions, one

```
PVOID GetUserLevelAddress (PVOID kernelLevelAddress, ULONG length,
                           FILE_OBJECT *fileObject)
{
  PVOID address;
  PMDL mdl;

  mdl = IoAllocateMdl(kernelLevelAddress, length, FALSE, FALSE, NULL);
  if (mdl == NULL)
    return NULL;

  // Build the MDL for the kernel-level address, assumed to lie in
  // non-paged memory. Then, map it into a user-level address.

  MmBuildMdlForNonPagedPool(mdl);
  address = MmMapLockedPages(mdl, UserMode);
  if (address == NULL) {
    IoFreeMdl(mdl);
    return NULL;
  }

  // Save the address and MDL pointer so they can be unmapped and
  // freed, respectively, when this file object is closed.

  fileObject->FsContext = address;
  fileObject->FsContext2 = mdl;

  // In NT 4.0 SP3 and earlier, 'address' refers to the base virtual
  // address of the page instead of the actual virtual address of the
  // MDL. So, we use the following code, which will work whether it's
  // one of those versions or not.

  return (PVOID) ((ULONG)PAGE_ALIGN(address) + MmGetMdlByteOffset(mdl));
}
```

Figure A.12: VTrace uses a function like this one to map a region of kernel-level non-paged memory to user level.

can see that the first thing they do is load the system call number into register EAX. Thus, the system call number can be extracted from bytes 2–5 of the Zw function.

As mentioned in Subsection A.4.4, we hooked the system calls for opening files so we could make sure our file system filter driver attached a device to each active file system. Unfortunately, one of these system calls, ZwOpenFile, is undocumented in the DDK, so we did not know how to use its parameters to determine what file system to filter. Fortunately, we found this function documented in Nagar's book [Nag97].

Another problem we encountered is that NTOSKRNL.EXE does not export all the system calls we were interested in logging. Some, such as ZwSignalAndWaitForSingleObject, are only exported by NTDLL.DLL, with which we were unable to link our driver. (Regmon does not have this problem, since it only hooks system calls exported by NTOSKRNL.EXE.) So, our tracer reads and parses the file NTDLL.DLL to find the Zw function bodies. The file format it uses, called the portable executable (PE) file format, is well documented [Kat93, Pie94b], so parsing it is not difficult.

An important part of parsing a PE format file is translating virtual addresses into file positions. Many structures in the file refer to other structures in the file using the virtual addresses they will have when loaded into memory, but we need to know where in the *file* those structures are.

To translate from virtual addresses to file positions, we need the section header information. This is an array of IMAGE_SECTION_HEADER structures, each of which describes the absolute file position of a section, the length of that section, and the virtual address where that section will be loaded. Using this information, we can figure out which section contains a given virtual address, and from that the file position for that address. Figure A.13 shows where to find these section header structures in the file.

Once we can translate from virtual addresses to file positions, we can find the names and bodies of all the exported functions, using the outline shown in Figure A.13. This lets us find where the Zw function bodies are and what their first few bytes are.

As mentioned earlier, we hook the system calls for opening files so our file system filter driver can attach a device to each file system. Unfortunately, the DDK fails to document one of these system calls, ZwOpenFile, so we could not at first determine how to use its parameters to determine what file system to filter. Fortunately, Nagar's book documents

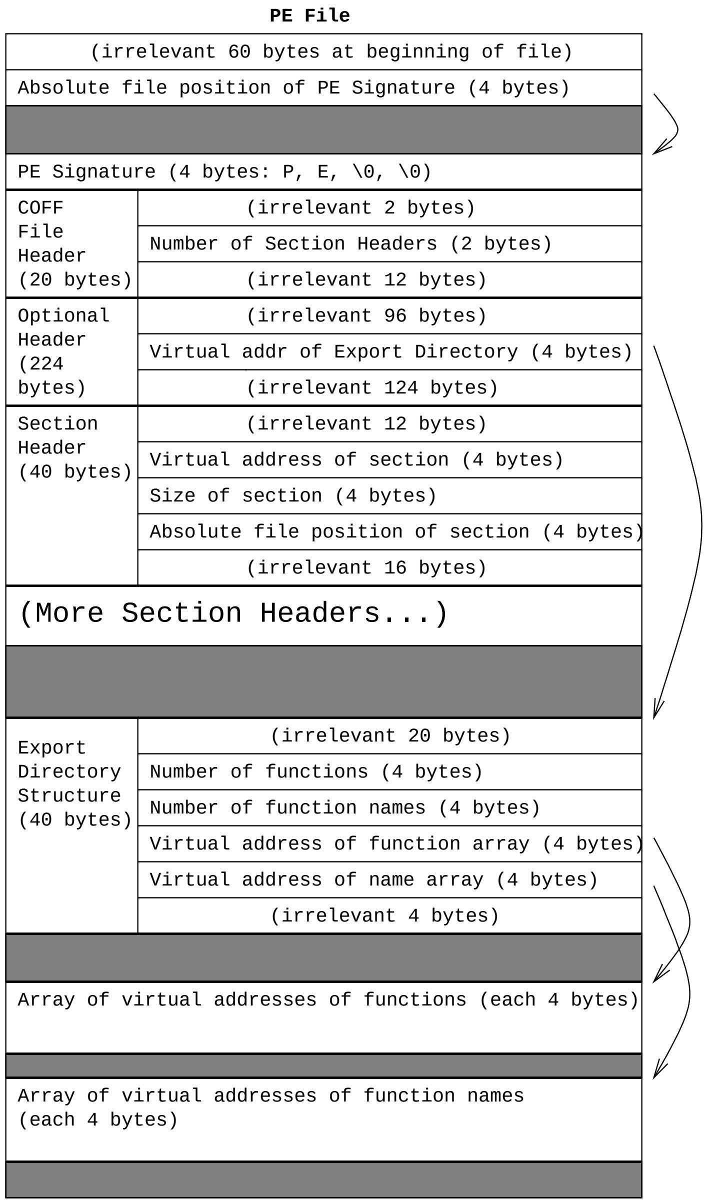
PE File
(irrelevant 60 bytes at beginning of file)
Absolute file position of PE Signature (4 bytes)
PE Signature (4 bytes: P, E, \0, \0)
COFF File Header (20 bytes)
(irrelevant 2 bytes)
Number of Section Headers (2 bytes)
(irrelevant 12 bytes)
Optional Header (224 bytes)
(irrelevant 96 bytes)
Virtual addr of Export Directory (4 bytes)
(irrelevant 124 bytes)
Section Header (40 bytes)
(irrelevant 12 bytes)
Virtual address of section (4 bytes)
Size of section (4 bytes)
Absolute file position of section (4 bytes)
(irrelevant 16 bytes)
(More Section Headers...)
Export Directory Structure (40 bytes)
(irrelevant 20 bytes)
Number of functions (4 bytes)
Number of function names (4 bytes)
Virtual address of function array (4 bytes)
Virtual address of name array (4 bytes)
(irrelevant 4 bytes)
Array of virtual addresses of functions (each 4 bytes)
Array of virtual addresses of function names (each 4 bytes)

Figure A.13: This outline of the structure of PE image files shows how to get what we need from NTDLL.DLL. It shows how to find the section headers you need to translate virtual addresses into absolute file positions. It also shows how to find the virtual addresses of the names and bodies of the exported functions.

this function [Nag97].

A.8 Parsing File System Metadata

Our design goals required us to take periodic snapshots of the file system metadata of each local NTFS partition. Helen Custer's book about NTFS [Cus94] discusses NTFS at a high level, but we needed detailed information about its layout. For this, we used the documentation and source code for the Linux NTFS driver.This documentation is at http://linux-ntfs.sourceforge.net/ntfs/, and the source code is at http://www.informatik.hu-berlin.de/∼loewis/ntfs/.

We learned that essentially all the data we needed is in a special file in each partition called the Master File Table (MFT). This file, named $MFT, contains fixed-length records describing the attributes of each file (and directory, since directories are basically just special files). However, there are at least three problems with recording the metadata by simply dumping this file. First, the file is sparse, because (1) many files' attributes do not use an entire record, and (2) many records are unused, having been allocated to files that have since been deleted. Second, an attribute can be *non-resident*, meaning that it is somewhere else on disk and only a pointer to it is in the MFT record. Third, the contents of a file are considered an attribute of the file, so recording the MFT might also record file contents; this would violate the confidentiality of our users' data.

This led us to the following approach: We do a depth-first search of the directory structure of each NTFS partition and, for each file found, we find and record certain non-data attributes of that file. Finding the metadata for a file requires knowing its *file number*, the number of the MFT record containing that file's attributes. The partition root always has file number 5.

We still have not explained how one reads directly from a disk, or how one finds specific MFT records. To read a raw disk on Windows NT, a user-mode program opens a file called "\\.\X:", replacing "X" with the appropriate drive letter. The first file block contains useful information: the size of a block (the 2-byte value at offset 0xB), the number of blocks in a cluster (the 1-byte value at offset 0xD), the number of clusters in an MFT record (the 1-byte value at offset 0x40), and the cluster number of the first MFT record (the 8-byte value at offset 0x30). The first MFT record is useful to find, since it contains the file attributes for $MFT itself. By parsing its data attribute information one can locate any MFT record. Parsing a file's MFT record reveals all the file's attributes. (This is not difficult, especially if one judiciously copies sections of the Linux NTFS driver code and reads the Linux NTFS documentation described above.) If the file is actually a directory, one can parse its index allocation attribute to find its subfiles' file numbers.

To determine what raw disk device a DOS disk name like "X:" corresponds to, we call QueryDosDevice. It takes a DOS disk name and returns the corresponding raw disk name, such as "\Device\Harddisk1\Partition3."

A.9 Miscellaneous Features of VTrace

A.9.1 Listing processes and threads

To log when processes and threads start and stop, we use the barely documented functions PsSetCreateProcessNotifyRoutine and PsSetCreateThreadNotifyRoutine. With them, we can set a logging function to be called when a process (or thread) is created or destroyed.

We also need to record a list of the existing processes and threads when the tracer starts logging. Unfortunately, there is no documented way to do this from kernel mode. Fortunately, there was a USENET article on comp.os.ms-windows.programmer.nt.kernel-mode by Fizal Khan describing how do this with the undocumented function ZwQuery-SystemInformation. (Later, a book documented this function [Neb00].) This function has the following prototype:

```
unsigned long ZwQuerySystemInformation
 (ULONG tag, VOID *buffer, ULONG bufSize, ULONG *returnedSize);
```

The tag parameter in this prototype indicates what kind of information is to be returned; the value 5, for instance, indicates process and thread information. Figure A.14 shows how we use this function to get a sequence of process information structures, one for each process. Figure A.15 illustrates the contents of each of these structures.

```
#define TAG_GET_PROC_THREAD_INFO                5
#define FIRST_GUESS_AT_PROC_THREAD_INFO_SIZE    8192
#define INCREMENT_FOR_PROC_THREAD_INFO_SIZE     1024

// GetProcessAndThreadInfo returns a pointer to an allocated buffer
// containing process and thread information. *bytesReturnedPtr will
// hold the useful length of this information. If an error occurs,
// this function returns NULL. Otherwise, the caller is expected to
// eventually call ExFreePool to deallocate the returned buffer.

char *GetProcessAndThreadInfo (ULONG *bytesReturnedPtr)
{
  char *buf;         // buffer to hold the process and thread information
  ULONG bufSize;     // size of the buffer
  NTSTATUS status;   // status code returned by ZwQuerySystemInformation

  bufSize = FIRST_GUESS_AT_PROC_THREAD_INFO_SIZE;
  while ((buf = ExAllocatePool(NonPagedPool, bufSize)) != NULL) {
    *bytesReturnedPtr = 0;
    status = ZwQuerySystemInformation(TAG_GET_PROC_THREAD_INFO,
                                      buf, bufSize, bytesReturnedPtr);
    if (status == STATUS_SUCCESS) return buf;

    // If the buffer was the wrong size, make the buffer bigger; use the
    // value returned in bytesReturnedPtr as a hint about the needed size.

    ExFreePool(buf);
    if (status == STATUS_BUFFER_OVERFLOW ||
        status == STATUS_INFO_LENGTH_MISMATCH)
      bufSize = MAX(*bytesReturnedPtr,
                    bufSize + INCREMENT_FOR_PROC_THREAD_INFO_SIZE);
    else
      return NULL;
  }
  return NULL;
}
```

Figure A.14: This function returns a buffer containing a sequence of process information structures, one for each process. It puts the length of the returned buffer in bytesReturnedPtr. The caller of this function is responsible for freeing the returned buffer if it is not NULL.

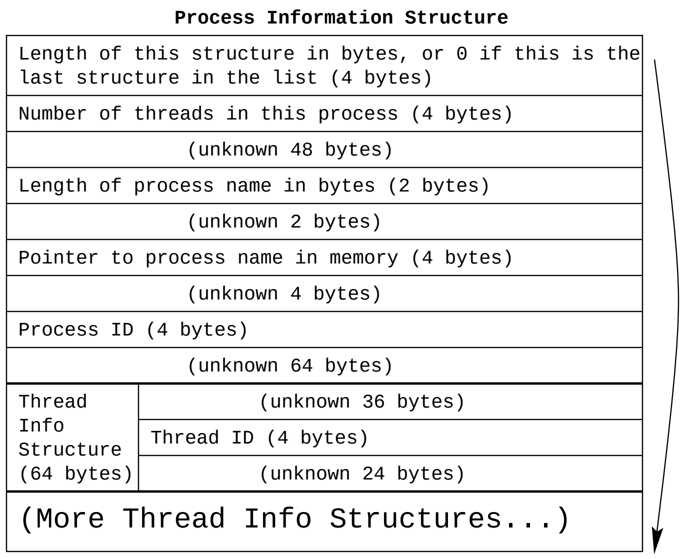
Process Information Structure
Length of this structure in bytes, or 0 if this is the last structure in the list (4 bytes)
Number of threads in this process (4 bytes)
(unknown 48 bytes)
Length of process name in bytes (2 bytes)
(unknown 2 bytes)
Pointer to process name in memory (4 bytes)
(unknown 4 bytes)
Process ID (4 bytes)
(unknown 64 bytes)
Thread Info Structure (64 bytes)
(unknown 36 bytes)
Thread ID (4 bytes)
(unknown 24 bytes)
(More Thread Info Structures...)

Figure A.15: NtQuerySystemInformation returns a list of process information structures, each of which looks like this.

We also use this undocumented feature to obtain and log the name of a process when we are notified of it starting, since the notification only tells us the process ID.

A.9.2 Idle timer

To keep trace file sizes down, the tracer should automatically stop logging when the user is idle for 10 minutes. Because we were logging keyboard and mouse messages, we could determine when the user was active; this permitted the following approach.

When the system starts up, we initialize a timer to go off in ten minutes, using code like that in Figure A.16. In this code, the constant is -6 billion, meaning 6 billion 100-ns units (10 minutes) from now (negativeness indicates relative time). The call to KeInitializeDpc initializes a deferred procedure call object by binding it to the function UserGoesIdleRoutine. KeInitializeTimer associates the timer with that deferred procedure call, so that its associated function is executed when the timer goes off. Finally, KeSetTimer initializes the timer to go off ten minutes later. When we detect user activity, we repeat the call to KeSetTimer, delaying when the timer will go off until ten minutes after *then*.

```
void InitializeUserInactivityWatch (void)
{
  LARGE_INTEGER tenMinsFromNow = { 0x9A5F4400, -2 };

  KeInitializeDpc(&globals.userGoesIdleDpc, &UserGoesIdleRoutine, NULL);
  KeInitializeTimer(&globals.userGoesIdleTimer);
  KeSetTimer(&globals.userGoesIdleTimer, tenMinsFromNow,
             &globals.userGoesIdleDpc);
}
```

Figure A.16: This code initializes a timer. If this timer is not canceled, it will call UserGoesIdleRoutine 10 minutes from now.

A.9.3 Disabling drivers at startup

Many of VTrace's drivers must be started automatically at startup time for them to work. This means that if they cause problems, it may be impossible to remove them by any means short of reinstalling the operating system. Thus, it is useful to be able to disable the drivers at startup.

To do this, we need access in the very early stages of startup to some state that the user can control. About the only thing the user can indicate to the system at this stage is what boot configuration to use. For instance, the user can choose to boot with the "last known good" configuration, meaning the most recent registry configuration that led to a successful boot. This is a natural signal we can use to decide to turn off VTrace.

To determine which configuration is in use, we open the registry key HKEY_LOCAL_MACHINE\System\Select and read the Current and LastKnownGood values. Each of these is an index into the list of registry configurations. If the current configuration in use is the same as the last known good configuration, we know that the user has chosen the last known good configuration and we turn off all components of VTrace.

A.9.4 User-level service

Some of VTrace's general operations are easier and safer to implement at user level than at kernel level. For this reason, VTrace includes a user-level service, VTrcSrvc, that is launched at startup and runs in the background to perform the following two operations:

```
void main (void)
{
  SERVICE_TABLE_ENTRY ServiceTable[] =
   { { "VTrcSrvc", (LPSERVICE_MAIN_FUNCTION) &ServiceMain },
     { NULL, NULL } };
  StartServiceCtrlDispatcher(ServiceTable);
}
```

Figure A.17: This is the main routine of VTrace's service, VTrcSrvc.

(1) After the user has been idle for 2 hours, it turns off tracing, takes a metadata snapshot, compresses all the trace and metadata files collected, uploads those files to our web site, deletes them from the local hard drive, and turns tracing on again. It waits 24 hours before doing any of this again. (2) Whenever a new user logs in or the logger signals that a new log file has started, it generates a log entry describing the current user's name.

Paula Tomlinson's article [Tom96] describes in detail how to write and install a user-level service, so we describe it only briefly here. The main routine initializes a table of service table entries and dispatches them. Ours looks like Figure A.17. The service main function, in our case called ServiceMain, registers a handler for service control messages (such as pause and stop), initializes other global state, launches threads to perform the service's tasks, then waits on an event set when a stop message is received. Throughout the initialization process, it calls SetServiceStatus to send messages to the service control manager indicating how far along it is.

One of the service control messages VTrcSrvc is programmed to respond to is one we made up called SERVICE_CONTROL_RELOAD_REGISTRY. When VTrcSrvc receives this message, it rechecks the VTrace parameters in the registry and starts using the new values if they have changed. This allows the utility that changes VTrace parameters to make those changes go into effect immediately without waiting for the next reboot.

A.9.5 Pentium cycle counter

To get accurate time stamps on each of our trace events, we use the Pentium cycle counter. This counter contains the number of cycles that have passed since the computer was

```
__asm {
  push eax         ; Save registers we will overwrite (eax, ebx, edx).
  push ebx
  push edx
  _emit 0x0F       ; The RDTSC instruction consists of these two bytes.
  _emit 0x31
  mov ebx, bufPtr  ; Put the address where the timestamp goes in ebx.
  mov [ebx], eax   ; Save low 4 bytes of timestamp there.
  mov [ebx+4], edx ; Save high 4 bytes of timestamp next.
  pop edx          ; Restore overwritten registers.
  pop ebx
  pop eax
}
```

Figure A.18: This code reads the Pentium cycle counter by invoking assembler in C.

started up. It is accessed via the RDTSC instruction, which can be coded in C by invoking assembler as in Figure A.18.

VTrace only needs the low four bytes of the counter for the following reason. Given the low four bytes of two consecutive event times, one can determine, modulo 2^{32}, how many cycles separated the events. The logger automatically places a null event in the log every $0.75 \cdot 2^{32}$ cycles, ensuring that no two consecutive events are separated by 2^{32} cycles or more. Thus, the time between any two consecutive events is unambiguous. (If the high four bytes were needed as well, they could be found in register EDX.)

To determine the time each event in the log occurred, the log reader divides the cycle count timestamp in the event by the CPU speed, then adds this number of seconds to the start time. However, we found that using this technique causes errors that compound as one goes further in the log. The reason for this is that the CPU speed reported by the processor, e.g., 450 MHz, is only an approximation of the number of cycles the processor performs each second. In reality, the speed is some less-round number, such as 451.022749 Hz.

To account for this, we have the installer perform an experiment to determine the actual speed, and record that instead of the processor-reported speed. To do this, the installer uses QueryPerformanceCounter to determine times and the RDTSC instruction to determine cycle counts. When it begins installing, it records the time and cycle count, and as it finishes installing it records the time and cycle count again. (If fewer than five seconds have passed since installation started it waits for five seconds to have passed before obtaining the second time and cycle count.) From the respective differences, it can determine an accurate estimate of the cycles per second on the machine.

A.9.6 File distribution

We wanted to use a self-extracting archive file to distribute our software, but one needs a distribution license to use the most common one, PKZIP's. In searching the web, we eventually found a free self-extracting archive creator, LHA. The file LHA255B.ZIP, containing this software, can be downloaded from various sites.

A.10 Windows 2000

Windows 2000 is substantially similar to Windows NT, but different enough that porting VTrace to it required some effort. In this section, we briefly describe some of the changes we made so VTrace would work on Windows 2000.

The biggest difference between Windows 2000 and Windows NT is that Windows 2000 has kernel-mode write protection. This means that any attempt to overwrite kernel code in memory causes a system crash. Since our method of hooking context switches requires that we overwrite the first instruction of the context-swap code in memory, this causes problems for VTrace. The solution is to map the memory to a writable address, as in Figure A.19.

The context swap routine is different in Windows 2000, so the tracer must look for this new routine in memory.

Windows 2000 expects drivers to provide two additional dispatch routines, to deal with power-management and plug-and-play requests. To pass on a power-management IRP, a dispatch routine must first call PoStartNextPowerIrp, and must use PoCallDriver instead of IoCallDriver. When a filter device receives a plug-and-play request, it must check whether the minor function number is IRP_MN_REMOVE_DEVICE. If this type of request completes successfully, the device to which the filter device is attached has removed itself, so the filter device should detach and delete itself.

```
// This function should only be called on Windows 2000. It's not
// necessary on Windows NT, and, besides, on NT 4.0 SP3 and earlier,
// MmMapLockedPages does not give the result we need.

NTSTATUS WriteReadOnlyMemory (char *dest, char *source, int length)
{
  KSPIN_LOCK tempSpinLock;
  KIRQL oldirql;
  PMDL mdl;
  PVOID writableAddress;

  mdl = IoAllocateMdl((PVOID) dest, length, FALSE, FALSE, NULL);
  if (mdl == NULL)
    return STATUS_UNSUCCESSFUL;
  MmBuildMdlForNonPagedPool(mdl);
  MmProbeAndLockPages(mdl, KernelMode, IoWriteAccess);
  writableAddress = MmMapLockedPages(mdl, KernelMode);
  if (writableAddress == NULL) {
    MmUnlockPages(mdl);
    IoFreeMdl(mdl);
    return STATUS_UNSUCCESSFUL;
  }

  // It is imperative that no context switch happens during the
  // copying, so we protect the write with a spin lock. (Context
  // switches are disabled while a spin lock is held.)

  KeInitializeSpinLock(&tempSpinLock);
  KeAcquireSpinLock(&tempSpinLock, &oldirql);
  RtlCopyMemory(writableAddress, source, length);
  KeReleaseSpinLock(&tempSpinLock, oldirql);

  MmUnmapLockedPages(writableAddress, mdl);
  MmUnlockPages(mdl);
  IoFreeMdl(mdl);
  return STATUS_SUCCESS;
}
```

Figure A.19: This function can overwrite part of the Windows 2000 kernel image in memory despite the Windows 2000 kernel memory write protection.

A particularly complicated plug-and-play request type to handle is IRP_MN_DEVICE_USAGE_NOTIFICATION. Such a request can indicate that a page file on the underlying device either started or stopped being used. A file system filter or disk filter must keep track of how many in-use page files the underlying device has, and update this count whenever it receives one of these notifications. Updating this count is complicated by the fact that the device must in some cases update the count when this request arrives, then undo it if the request fails. The DDK provides examples showing how to do this.

At the end of Section A.8, we discussed how to determine the disk and partition numbers of a given disk, such as “X:”. Unfortunately, this method does not work in Windows 2000, and the Windows 2000 method does not work in Windows NT. In Windows 2000, you must use a new device I/O control code called IOCTL_STORAGE_GET_DEVICE_NUMBER. Passing this code to an open file representing the raw disk yields a STORAGE_DEVICE_NUMBER structure containing the disk and partition numbers.

In Windows 2000, the process information structure is slightly changed from Figure A.15. The “unknown 64 bytes” in the header are actually 112 bytes long in Windows 2000.

The remaining the changes for Windows 2000 concern that operating system's filter plug-and-play feature, which makes it easier to attach a filter to every device of a certain type. To use this, our installer adds the name of the filter driver to the UpperFilters value of type REG_MULTI_SZ in the registry key corresponding to the class of devices it should filter (such as HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\{4D36E96B-E325-11CE-BFC1-08002BE10318} for keyboards). The filter driver's DriverEntry routine makes `DriverObject->DriverExtension->AddDevice` a pointer to a function that, like the example in Figure A.20, creates a filter device and has it filter a given device. Also, if a filter driver does not need to start at boot time, our installer gives it a Start parameter of SERVICE_DEMAND_START (instead of SERVICE_AUTO_START) when it sets this parameter with CreateService or ChangeServiceConfig.

When using a filter driver in this way, one must be extremely careful that the installer and uninstaller never leave the system in a state in which the driver does not exist but is listed as an upper filter. This is because the operating system will refuse to create any devices requiring a nonexistent filter. If the operating system fails to start the keyboard

```
NTSTATUS VTrcKbdAddDevice (IN PDRIVER_OBJECT driverObject,
                           IN PDEVICE_OBJECT physicalDeviceObject)
{
  PDEVICE_OBJECT       filterDevice;
  PKBD_HOOK_EXTENSION  filterExtension;
  NTSTATUS             status;

  // Create the filter device.

  status = IoCreateDevice(driverObject, sizeof(KBD_HOOK_EXTENSION), NULL,
                          FILE_DEVICE_KEYBOARD, 0, FALSE, &filterDevice);
  if (!NT_SUCCESS(status))
    return status;

  // Set flags for the filter device.

  filterDevice->Flags |= (DO_BUFFERED_IO | DO_POWER_PAGABLE);

  // Attach the filter device to the existing keyboard device.

  filterExtension = filterDevice->DeviceExtension;
  filterExtension->attachedDevice =
    IoAttachDeviceToDeviceStack(filterDevice, physicalDeviceObject);
  if (filterExtension->attachedDevice == NULL) {
    IoDeleteDevice(filterDevice);
    return STATUS_UNSUCCESSFUL;
  }

  // Indicate that the filter device is finished initializing.

  filterDevice->Flags &= ~DO_DEVICE_INITIALIZING;

  return STATUS_SUCCESS;
}
```

Figure A.20: VTrace uses a dynamic add-device function like this one under Windows 2000.

device, or, worse, the device corresponding to the disk containing the boot partition, the system can be unusable. For the same reason, one must ensure the DriverEntry routine never returns an error, because in this case as well, the operating system will refuse to create any devices that appear to require the failing filter. We think that these behaviors constitute a bug in the operating system, but Microsoft has not treated it as one.

As useful as the filter plug-and-play feature is, we only use it for our keyboard and raw disk filters. It appears that Windows 2000 will not support it on virtual device classes such as file systems and network protocols. We could filter these classes on a device-by-device basis using SetupDiSetDeviceRegistryProperty, but then if any additional devices were added to the system later on, they would not get filtered.

A.11 Benchmarks

Considering all the tracing that VTrace does, it is important to determine how much it slows down the system. We wanted to make the overhead unnoticeable, so users would let us install it on their systems. By this measure, we succeeded, since none of our users has ever complained about performance suffering.

That said, it can be hard for users to detect subtle differences, especially on today's fast machines. So we designed various benchmarks to show the effect of running VTrace. We ran each of these benchmarks on our PC, which has a 450 MHz Pentium III, is connected to a 100 Mbps switched Ethernet, has 128 MB of memory, has 10 GB divided among three SCSI disks, and is running Windows NT 4.0 with Service Pack 6a. We ran each benchmark (other than the compilation and document format benchmarks, which take too long) enough times that the 95% confidence interval about the sample mean included no values more than 0.1% away from the sample mean. We also instrumented VTrace to find out how much overhead there is just to write a single short log entry; on average, this takes 20.24 μs from user level but only 0.95 μs from kernel level.

Figure A.21 shows the results. We see that VTrace has almost no effect on simple reads and writes, since there is little to log and all the logging is at kernel level. Copying files incurs more tracing overhead, especially when VTrace is also tracing network operations. Calling various traced functions like ZwFlushInstructionCache, WaitMessage, and

Operation	**Time without VTrace**	**Time with VTrace**	**Slow-down**
Read an uncached 32KB file	9.16 ms	9.17 ms	0.1%
Write 1KB file (write-thru)	25.05 ms	25.05 ms	0%
Read 32 KB direct from disk	9.17 ms	9.17 ms	0%
Copy a 32 KB file locally	6.29 ms	6.57 ms	4.5%
Copy a 32 KB file remotely	27.73 ms	35.07 ms	26.4%
ZwFlushInstructionCache	2.78 μs	3.72μs	33.8%
WaitMessage	8.98 μs	64.84 μs	722%
TranslateMessage	0.11 μs	42.19 μs	40178%
Compile logger with DDK	10.23 s	11.60 s	13.4%
Format article with LaTeX	1.69 s	1.79 s	5.3%

Figure A.21: These benchmark result means show how much VTrace slows down various operations.

TranslateMessage incurs overhead essentially due to the overhead of writing log entries. As you can see, this is substantial for the latter two functions since they do little but are at user level, and furthermore because each call requires two log entries: one for the initiation of the function and one for its completion. Finally, one can see the “big picture” from the two application benchmarks, which show that VTrace makes a 10-second compilation take 13.4% longer and a 2-second document formatting take 5.3% longer.

These benchmarks suggest that the biggest area for improvement is the overhead of tracing user-level events. It would thus seem that we could substantially improve VTrace’s performance by having it trace user-level events entirely at user level. To test this, we wrote a version of VTrace that did separate kernel-level and user-level logging. This approach reduced the overhead for user-level logging tremendously, from about 20 μs to only about 0.25μs. However, the extra processing required to perform separate user-level and kernel-level tracing dominated these improvements, causing this separation approach to actually do slightly worse in the macrobenchmarks than our original approach. Thus, in the final version of VTrace, we perform all logging at kernel level.

A.12 Similar Software

VTrace is not the first piece of software to modify the operating system (without recompiling it) on machines used for normal operation. WMonitor [ZS00] uses the Windows message hook facility and installable file system support to trace application messages and file activity in Windows 95. SLIC [GPRA98] uses interposition to intercept system calls and signals in Solaris 2.5. KernInst [TM99] performs a structural analysis on the Solaris 2.5.1 kernel running on an UltraSPARC so that it can insert code at almost any point in the kernel without rebooting and without disturbing any live registers. COLA [KK92], like our Win32 system call hooking method, looks into each library loaded into a UNIX application to find all the points at which system calls are made, so it can intercept those system calls at those points. Instrumented Connectors [BG99] is a general system you can use to wrap your own function around any function exported by a Windows dynamically linked library. Michael Jones created an interposition agents toolkit [Jon93] that expresses the objects in the 4.3BSD operating system as C++ classes, so that you can extend their functionality by writing derived classes. And there are many other examples, including various commercial virus checkers and disk compression utilities.

A.13 Summary

Writing the tracer VTrace for Windows NT/2000 was a difficult task, because of both the inherent difficulty of system-level programming and the lack of official documentation. Nevertheless, with the help of many sources of information, including developer tools, magazines, books, web sites, and USENET, we achieved our goal. This appendix has described the major techniques we used in doing so.

Writing the tracer involved a lot of driver programming to implement special device types. The heart of our tracer is the logger device, which processes requests to add events to the log. The other devices we implemented were filter devices, which layer themselves above existing devices to intercept and log I/O requests destined for them. We filter file systems, the keyboard, disk partitions, and network transport layers.

We also did a substantial amount of hooking system functions. We developed a technique for intercepting calls to the context switch function. We also adapted to our purposes some published hacks for intercepting NT kernel and Win32 system calls.

We had to do many other things for our tracer, several of which we describe. We designed a file system metadata parser, used a technique to get a list of existing processes and threads, implemented an idle timer, and included a user-level service, among other things.

Although VTrace incurs overhead on the system, this overhead is relatively low considering how much it traces. No user has complained about the load VTrace places on the system. Logging common file operations incurs very little overhead. Benchmarks measuring the effect on realistic batch workloads show only a 5–13% increase in execution time.

It is our hope that the techniques we describe, as well as the references we give to more detailed descriptions of related techniques, will be helpful to future Windows NT/2000 system-level programmers. These techniques can be used for tracing many things VTrace does not trace, and for many things besides tracing.

Appendix B

Workload Characterization Details

B.1 Introduction

This appendix expands Chapter V by providing tables and figures showing greater detail about the characteristics of the workloads described in that chapter.

B.2 Application usage

In this section, we provide tables giving details of the top 25 applications used in the workloads. Tables B.1, B.2, B.3, B.4, B.5, B.6, B.7, and B.8 show, for each user, the 25 applications consuming the largest amount of total CPU time. They also show how much time the threads of these applications spent waiting for I/O of two types: disk and network. Note that the total time an application spends waiting for the disk or network can exceed the total tracing time, since these quantities are aggregated over all threads in all processes of that application name, and multiple threads can be waiting on the disk or network at the same time.

Executable	Description	CPU sec	Disk sec	Network sec
netscape	Netscape mail reader and web browser	31,686	4,950	1,590,673
acrord32	Acrobat document reader	12,887	174	9
msdev	Visual Studio software development	7,200	2,159	1,036
system	Operating system process	7,191	13,234	374,700
exceed	Exceed X server	6,695	1,187	1,844,201
ssh	Secure shell terminal	6,276	70	582,505
explorer	Windows file system desktop	5,841	5,994	3,591
symbolsx	Installation of Windows system symbols	4,360	135	0
ntvdm	Windows NT virtual DOS machine	3,877	133	8,185
hotsync	Palm Pilot synchronization	3,678	1,117	112
mcshield	McAfee virus scanner	2,932	4,437	0
cl	C++ Compiler	2,002	1,739	122
realplay	RealAudio player	1,519	155	43,402
windbg	Windows debugger	1,516	348	1,565
starcraft	StarCraft game	1,389	34	1,561
services	Miscellaneous services	1,193	261	12,485
emacs	Emacs text editor	1,188	472	116
benchmark	Custom benchmarking program	1,035	31	644
iexplore	Internet Explorer web browser	872	709	4,549
ksetup	Unknown	830	160	0
shstat	McAfee registry watcher	694	102	0
powerpnt	PowerPoint presentation design	682	111	162
msiexec	Windows installer	612	321	0
spoolss	Printer control	597	456	972
winword	Microsoft Word document editor	595	306	49
All	All applications (not just top 25)	115,455	48,665	100,821,979

Table B.1: Time taken by the 25 applications consuming the most CPU time for **user #1**.

Executable	Description	CPU sec	Disk sec	Network sec
iexplore	Internet Explorer web browser	62,197	9,213	12,496,809
ss3dfo	3D Flying Objects screen saver	16,067	48	7
starcraft	StarCraft game	14,669	47	34,598
acrord32	Acrobat document reader	14,372	211	10
winword	Microsoft Word document editor	12,627	1,162	725
psp	Paint Shop Pro graphics editor	9,822	289	163
java	Java virtual machine	8,669	1,694	1,931
system	Operating system process	8,222	19,327	1,962
realplay	RealAudio player	7,249	220	9,866,395
javac	Java compiler	5,853	427	2,582
devenv	Microsoft Visual Interdev	5,642	1,545	8,060
explorer	Windows file system desktop	5,243	5,431	7,352
hotsync	Palm Pilot synchronization	5,225	434	254
msimn	Outlook Express mail reader	5,009	740	43,709
powerpnt	PowerPoint presentation design	4,213	368	395
txtpad32	TextPad text editor	3,018	209	784
mplayer2	Microsoft media player	2,833	69	319,570
ssh	Secure shell terminal	2,632	394	2,080,066
mcshield	McAfee virus scanner	2,535	2,369	926
appletviewer	Java applet viewer	2,445	404	1,508
realjbox	RealAudio music jukebox	2,233	120	5,423
services	Miscellaneous services	2,021	515	1,818
textpad	TextPad text editor	1,331	178	88
shstat	McAfee registry watcher	1,294	195	0
ntguard	NetGuard firewall	1,229	701	139
All	All applications (not just top 25)	228,067	55,254	26,316,284

Table B.2: Time taken by the 25 applications consuming the most CPU time for **user #2**.

Executable	Description	CPU sec	Disk sec	Network sec
aim	AOL Instant Messenger chat	35,062	341	9,314
netscape	Netscape mail reader and web browser	16,386	1,208	70,998
realplay	RealAudio player	8,263	422	410
ntvdm	Windows NT virtual DOS machine	6,317	410	528
blackd	BlackICE Defender firewall daemon	2,722	89	0
faxmain	Fax Press network fax server	2,599	92	0
psp	Paint Sho Pro graphics editor	2,496	117	0
system	Operating system process	1,690	2,305	295
explorer	Windows file system desktop	1,664	2,762	804
acrord32	Acrobat document reader	1,474	72	1
sitelink	SiteLink web publisher	1,148	11	0
sbtw	ACCPAC Pro Series accounting software	749	136	590
bartend	BarTender label printing	564	57	0
outlook	Microsoft Outlook mail reader	485	1,028	230
blackice	BlackICE Defender firewall GUI	359	130	0
napster	Napster file sharing	352	136	12,039,172
iexplore	Internet Explorer web browser	351	386	12,203
excel	Microsoft Excel spreadsheet	291	130	5
winword	Microsoft Word document editor	238	211	0
emexec	Logitech enhanced mouse control	195	230	0
msoffice	Microsoft Office shortcut bar	183	383	6
winamp	Nullsoft Winamp media player	160	17	0
act	Symantec ACT! contact manager	150	87	82
ins5576	Installer for something?	136	68	115
winlogon	Windows logon prompt	121	992	3
All		85,336	15,249	12,149,110

Table B.3: Time taken by the 25 applications consuming the most CPU time for **user #3**.

Executable	Description	CPU sec	Disk sec	Network sec
netscape	Netscape mail reader and web browser	46,769	3,136	6,592,494
realplay	RealAudio player	36,322	1,259	22,612,927
msdtc	Microsoft transaction coordinator	10,986	37	0
outlook	Microsoft Outlook mail reader	8,686	2,167	19,363
ssh	Secure shell terminal	4,809	152	452,364
system	Operating system process	4,425	3,888	1,351
explorer	Windows file system desktop	3,627	3,794	4,233
exceed	Exceed X server	2,911	592	270,839
vern	Vern virtual desktop	2,365	66	0
services	Miscellaneous services	1,287	189	1,614
portmap	Legato Backup portmapper service	1,234	18	0
acrord32	Acrobat document reader	1,196	168	4
nsrexecd	Legato Backup remote exec service	1,168	31	2
dreamweaver	Macromedia Dreamweaver web design	1,147	74	46
nsrmmd	Legato NetWorker component	1,140	14	66
nsrd	Legato NetWorker component	922	45	7,142
ttermpro	Telnet	887	63	212,661
nsrindexd	Legator NetWorker component	876	34	106
shstat	McAfee registry watcher	874	140	0
nsrmmdbd	Legato NetWorker component	873	25	172
winamp	Nullsoft Winamp media player	794	30	7,416
iexplore	Internet Explorer web browser	553	130	721,970
winvnc	VNC remote display server	519	6	0
eudora	Eudora mail reader	506	35	71
icq	ICQ Internet chat	441	35	565
All		141,465	22,447	30,943,808

Table B.4: Time taken by the 25 applications consuming the most CPU time for **user #4**.

Executable	Description	CPU sec	Disk sec	Network sec
dwrcc	DameWare NT Utilities component	18,174	36	41,605,173
iexplore	Internet Explorer web browser	14,500	552	5,566,615
realplay	RealAudio player	12,898	778	303
explorer	Windows file system desktop	6,948	2,200	2,698
system	Operating system process	4,094	2,951	242
winmgmt	Windows Management Instrumentation	3,557	648	5,743
savenow	SaveNow web surfing “aid”	3,440	91	1,554
rtvscan	Norton Anti-Virus scanner	3,408	172	424,067
rapigator	Peer-to-peer file sharing	2,834	138	34,109,930
csrss	Windows client-server subsystem	2,683	28	285
outlook	Microsoft Outlook mail reader	2,174	284	203
winvnc	VNC remote display server	1,853	66	3
acrord32	Acrobat document reader	1,696	10	0
winamp	Nullsoft Winamp media player	1,580	28	553
creatr32	Adaptec Easy CD Creator	1,202	29	807
dntu	DameWare NT Utilities	1,062	70	4,272
services	Miscellaneous services	863	279	42,091
winword	Microsoft Word document editor	833	55	177
msiexec	Windows installer	630	343	22
fwenc	SecuRemote firewall	608	22	1
setup	Miscellaneous installer	403	6	404,883
photoshp	Photoshop graphics editor	342	55	1
fssrv	FileScreen file type blocker	340	35	376
clipbrd	Windows clipboard viewer	309	0	0
fwpolicy	CheckPoint security policy editor	303	0	2
All		93,364	14,078	89,969,037

Table B.5: Time taken by the 25 applications consuming the most CPU time for **user #5**.

Executable	Description	CPU sec	Disk sec	Network sec
grpwise	Novell Groupwise groupware	14,260	992	562,612
ntvdm	Windows NT virtual DOS machine	8,322	575	143
findfast	Windows file finder	5,013	2,057	5
eudora	Eudora mail reader	4,757	1,098	1,346,999
mcshield	McAfee virus scanner	4,405	8,831	0
netscape	Netscape mail reader and web browser	3,433	982	223,541
realplay	RealAudio player	2,709	172	1
system	Operating system process	2,692	6,003	52,043
hotsync	Palm Pilot synchronization	2,648	950	15
planner	Franklin Covey day planner	2,600	696	0
winword	Microsoft Word document editor	2,529	1,703	68
realjbox	RealAudio music jukebox	2,148	80	130
iexplore	Internet Explorer web browser	1,995	600	4,046,163
aim	AOL Instant Messenger chat	1,832	198	31,701
acrobat	Acrobat document reader	1,760	214	1
acrord32	Acrobat document reader	1,296	51	0
explorer	Windows file system desktop	1,080	2,625	100
services	Miscellaneous services	761	236	23,010,825
powerpnt	PowerPoint presentation design	468	850	1
notify	Groupwise event notification	467	175	99,352
shstat	McAfee registry watcher	421	208	0
tsystray	RealAudio system tray icon	393	41	1
excel	Microsoft Excel spreadsheet	346	277	17
ins0432	Installer for something?	269	23	0
photoprn	ArcSoft PhotoPrinter	249	0	0
All		68,670	35,330	30,166,702

Table B.6: Time taken by the 25 applications consuming the most CPU time for **user #6**.

Executable	Description	CPU sec	Disk sec	Network sec
v3webnt	V3.Web?	124,480	1,127	303,111
winmgmt	Windows Management Instrumentation	11,179	3,405	712
iexplore	Internet Explorer web browser	10,235	4,629	130,523,496
realplay	RealAudio player	7,098	2,819	811
explorer	Windows file system desktop	5,204	7,950	1,056
inetinfo	Microsoft Internet Information server	4,097	1,869	267,070
ndmonnt	Internet Neighborhood Pro	3,865	20	0
system	Operating system process	3,328	5,948	651
winamp	Nullsoft Winamp media player	3,295	672	12,606,819
outlook	Microsoft Outlook mail reader	3,060	6,370	657,870
vb6	Visual Basic compiler	2,844	3,934	7
services	Miscellaneous services	2,584	573	99,759
winword	Microsoft Word document editor	2,538	2,326	113
csrss	Windows client-server subsystem	2,462	99	188
netcaptor	NetCaptor web browser	2,054	750	82,633
warftpd	Jarle Aase's FTP daemon	1,927	108	4,787
wincmd32	Windows Commander file manager	1,919	2,273	6,768
qfmain	Homemade QuickFind?	1,847	2,113	245
acrord32	Acrobat document reader	1,709	437	0
getright	GetRight download manager	1,462	652	413,521
winmysql-admin	MySQL configuration	938	385	284
powerpnt	PowerPoint presentation design	861	969	2
sqlservr	SQL server	857	449	5,718
pro	Teleport Pro offline web browser	776	214	127,579
internat	Network address translation server?	673	264	0
All		212,731	72,354	145,701,953

Table B.7: Time taken by the 25 applications consuming the most CPU time for **user #7**.

Executable	Description	CPU sec	Disk sec	Network sec
netscape	Netscape mail reader and web browser	43,963	12,376	3,691,624
grpwise	Novell Groupwise groupware	11,305	6,557	3,819,838
realplay	RealAudio player	11,090	2,754	22,086,212
acrord32	Acrobat document reader	10,801	1,229	6
ntvdm	Windows NT virtual DOS machine	9,062	2,023	650
explorer	Windows file system desktop	5,571	10,882	175
system	Operating system process	5,182	8,981	21,739
mcshield	McAfee virus scanner	2,870	15,623	0
findfast	Windows file finder	1,914	3,768	53
remote	Macromedia Shockwave remote access	1,642	747	0
winword	Microsoft Word document editor	1,494	1,673	152
services	Miscellaneous services	868	1,590	10,736,944
delldmi	Dell remote client management	821	502	0
3dfot1	3-D photography?	727	255	0
acrobat	Acrobat document reader	618	480	4,211
shstat	McAfee registry watcher	564	768	0
powerpnt	PowerPoint presentation design	490	604	0
excel	Microsoft Excel spreadsheet	479	938	19
jerusalm	Virtual tour of Jerusalem	446	58	0
notify	Groupwise event notification	331	737	483,427
activeshare	Adobe ActiveShare photo organizer	315	148	3
vstskmgr	Network Associates virus scan manager	275	191	0
smartagt	drMON Edge Monitor network admin	261	110	0
iexplore	Internet Explorer web browser	246	155	162
winampa	NullSoft Winamp media player	241	353	0
All		113,947	100,724	41,508,637

Table B.8: Time taken by the 25 applications consuming the most CPU time for **user #8**.

B.3 Trigger events

Tables B.9, B.10, B.11, B.12, B.13, B.14, B.15, and B.16 show, for each user and each of the top 25 applications, how much CPU time is due to the various types of trigger event.

B.4 User interface event types

In this section, we show, for each user and each of the top 25 applications of that user, the breakdown of user interface messages among the different types of such messages: key press/release, mouse move, and mouse click. Tables B.17, B.18, B.19, B.20, B.21, B.22, B.23, and B.24 show these breakdowns.

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
netscape	42.6%	39.4%	3.6%	0.0%	9.4%	0.0%	0.0%	0.7%	0.9%	3.4%
acrord32	2.3%	97.5%	0.1%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%
msdev	62.4%	6.8%	7.0%	0.0%	22.1%	0.0%	0.1%	0.3%	1.3%	0.0%
system	0.0%	0.0%	0.0%	0.0%	43.2%	0.0%	56.7%	0.0%	0.0%	0.0%
exceed	42.4%	2.8%	0.7%	0.0%	53.6%	0.0%	0.4%	0.1%	0.1%	0.0%
ssh	1.6%	0.1%	0.3%	0.0%	0.5%	0.0%	0.0%	0.0%	0.0%	97.6%
explorer	33.1%	4.8%	18.4%	0.0%	36.9%	0.0%	2.0%	2.1%	2.7%	0.0%
symbolsx	0.0%	0.0%	0.0%	0.0%	98.7%	0.0%	1.0%	0.1%	0.2%	0.0%
ntvdm	87.7%	0.2%	9.6%	0.0%	1.5%	0.0%	0.0%	0.0%	1.0%	0.0%
hotsync	0.5%	92.4%	0.0%	0.0%	4.9%	0.0%	2.0%	0.1%	0.0%	0.0%
mcshield	0.0%	0.0%	0.0%	0.0%	12.2%	0.0%	0.0%	28.1%	59.7%	0.0%
cl	0.0%	0.0%	0.0%	0.0%	53.4%	0.0%	46.6%	0.0%	0.0%	0.0%
realplay	0.1%	96.6%	0.2%	0.0%	2.0%	0.0%	0.1%	0.0%	0.9%	0.0%
windbg	22.4%	0.1%	12.1%	0.0%	59.3%	0.0%	2.6%	2.4%	1.0%	0.0%
starcraft	78.4%	1.1%	0.3%	0.0%	20.1%	0.0%	0.0%	0.0%	0.1%	0.0%
services	0.0%	0.0%	0.0%	0.0%	97.9%	0.1%	0.0%	1.4%	0.6%	0.0%
emacs	86.8%	0.0%	1.2%	0.0%	4.0%	0.0%	2.2%	0.2%	5.6%	0.0%
benchmark	0.0%	0.0%	0.0%	0.0%	42.2%	0.0%	9.4%	48.3%	0.1%	0.0%
iexplore	34.1%	8.3%	12.2%	0.0%	38.0%	0.0%	1.5%	0.2%	5.7%	0.0%
ksetup	0.0%	0.0%	0.0%	0.0%	82.2%	0.1%	17.5%	0.2%	0.0%	0.0%
shstat	0.0%	21.3%	0.1%	0.0%	77.7%	0.1%	0.8%	0.0%	0.0%	0.0%
powerpnt	65.1%	2.4%	3.9%	0.0%	28.5%	0.0%	0.0%	0.0%	0.0%	0.0%
msiexec	51.4%	13.1%	10.0%	0.0%	19.1%	0.0%	6.1%	0.0%	0.2%	0.0%
spoolss	0.0%	0.0%	0.0%	0.0%	98.6%	0.0%	0.0%	1.3%	0.0%	0.0%
winword	81.2%	0.4%	0.6%	0.0%	3.2%	0.0%	0.2%	0.0%	14.4%	0.0%
All	29.5%	29.2%	3.6%	0.0%	23.4%	0.0%	2.9%	2.1%	2.7%	6.7%

Table B.9: CPU time triggered by each event type for **user #1**

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
iexplore	56.5%	13.2%	5.4%	0.0%	21.3%	0.0%	0.2%	0.9%	2.2%	0.2%
ss3dfo	0.7%	99.2%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%
starcraft	93.1%	0.4%	0.4%	0.0%	6.0%	0.0%	0.0%	0.0%	0.1%	0.0%
acrord32	2.3%	97.0%	0.4%	0.0%	0.2%	0.0%	0.1%	0.0%	0.0%	0.0%
winword	77.0%	1.5%	2.0%	0.0%	8.4%	0.0%	0.0%	0.1%	10.9%	0.0%
psp	53.9%	44.3%	1.3%	0.0%	0.2%	0.0%	0.3%	0.0%	0.0%	0.0%
java	65.9%	1.4%	0.4%	0.0%	13.1%	0.0%	9.7%	0.3%	9.1%	0.0%
system	0.0%	0.0%	0.0%	0.0%	61.6%	3.9%	34.5%	0.0%	0.0%	0.0%
realplay	1.3%	73.4%	4.0%	0.0%	2.8%	0.0%	0.1%	0.0%	9.0%	9.4%
javac	0.0%	0.0%	0.0%	0.0%	60.3%	0.0%	39.2%	0.5%	0.1%	0.0%
devenv	61.1%	2.3%	4.2%	0.0%	29.6%	0.0%	0.1%	0.9%	1.8%	0.0%
explorer	44.8%	6.6%	26.4%	0.0%	16.7%	0.0%	2.4%	0.9%	2.2%	0.0%
hotsync	0.1%	98.0%	0.1%	0.0%	0.1%	0.0%	1.7%	0.1%	0.0%	0.0%
msimn	69.2%	7.4%	4.4%	0.0%	18.4%	0.0%	0.0%	0.1%	0.5%	0.0%
powerpnt	84.2%	3.7%	1.9%	0.0%	9.6%	0.0%	0.1%	0.3%	0.2%	0.0%
txtpad32	92.9%	0.4%	5.0%	0.0%	1.3%	0.0%	0.4%	0.0%	0.0%	0.0%
mplayer2	1.7%	5.7%	0.3%	0.0%	14.0%	0.0%	0.2%	0.0%	23.0%	55.2%
ssh	40.8%	0.6%	3.0%	0.0%	55.2%	0.0%	0.3%	0.1%	0.0%	0.0%
mcshield	0.0%	0.0%	0.0%	0.0%	3.4%	0.0%	0.0%	20.6%	76.0%	0.0%
appletviewer	72.5%	3.3%	1.3%	0.0%	13.8%	0.0%	6.7%	0.2%	2.3%	0.0%
realjbox	10.0%	61.7%	3.3%	0.0%	19.4%	0.0%	0.1%	0.9%	0.0%	4.6%
services	0.0%	0.0%	0.0%	0.0%	99.3%	0.0%	0.0%	0.7%	0.0%	0.0%
textpad	66.8%	25.5%	4.8%	0.0%	2.5%	0.0%	0.2%	0.1%	0.0%	0.0%
shstat	0.1%	24.0%	0.0%	0.0%	75.0%	0.0%	0.9%	0.0%	0.0%	0.0%
ntguard	0.0%	0.0%	0.0%	0.0%	94.7%	0.0%	0.0%	5.1%	0.1%	0.0%
All	43.7%	27.2%	3.9%	0.0%	16.4%	0.1%	2.6%	0.9%	4.1%	1.2%

Table B.10: CPU time triggered by each event type for **user #2**

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
aim	0.1%	99.7%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
netscape	12.8%	18.7%	2.8%	0.0%	4.1%	0.0%	0.0%	0.4%	48.6%	12.7%
realplay	0.1%	98.4%	0.3%	0.0%	0.3%	0.0%	0.0%	0.1%	0.0%	0.8%
ntvdm	0.3%	91.0%	1.3%	0.0%	2.9%	0.0%	0.0%	1.0%	3.4%	0.0%
blackd	0.0%	0.0%	0.0%	0.0%	28.8%	0.0%	0.0%	19.2%	52.0%	0.0%
faxmain	5.2%	0.3%	0.4%	0.0%	90.7%	0.0%	0.0%	3.2%	0.3%	0.0%
psp	72.8%	26.5%	0.4%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%
system	0.0%	0.0%	0.0%	0.0%	99.1%	0.0%	0.9%	0.0%	0.0%	0.0%
explorer	25.6%	4.0%	55.9%	0.0%	11.4%	0.0%	1.6%	0.5%	0.9%	0.0%
acrord32	1.2%	97.3%	0.6%	0.0%	0.0%	0.0%	0.9%	0.0%	0.0%	0.0%
sitelink	0.6%	95.7%	0.1%	0.0%	3.5%	0.0%	0.0%	0.0%	0.1%	0.0%
sbtw	49.0%	46.5%	2.4%	0.0%	0.1%	0.0%	1.9%	0.1%	0.0%	0.0%
bartend	36.7%	55.6%	7.1%	0.0%	0.2%	0.0%	0.4%	0.0%	0.0%	0.0%
outlook	41.1%	19.3%	20.9%	0.0%	15.2%	0.0%	0.2%	0.5%	2.9%	0.0%
blackice	2.0%	97.1%	0.1%	0.0%	0.0%	0.0%	0.8%	0.0%	0.0%	0.0%
napster	15.7%	0.2%	8.7%	0.0%	52.3%	0.0%	0.3%	18.8%	0.4%	3.6%
iexplore	39.7%	14.7%	16.8%	0.0%	25.1%	0.0%	0.6%	0.2%	1.0%	1.8%
excel	33.6%	1.0%	1.1%	0.0%	64.1%	0.0%	0.2%	0.1%	0.0%	0.0%
winword	65.7%	1.0%	4.0%	0.0%	5.5%	0.0%	0.4%	0.1%	23.3%	0.0%
emexec	0.2%	74.7%	7.0%	0.0%	17.2%	0.0%	0.7%	0.2%	0.0%	0.0%
msoffice	24.8%	53.3%	16.6%	0.0%	1.4%	0.0%	2.6%	1.0%	0.3%	0.0%
winamp	8.2%	24.6%	19.2%	0.0%	25.5%	0.0%	0.4%	14.6%	0.0%	7.5%
act	28.1%	50.4%	3.7%	0.0%	16.1%	0.0%	1.5%	0.2%	0.0%	0.0%
ins5576	43.6%	53.0%	0.8%	0.0%	0.0%	0.0%	2.1%	0.1%	0.4%	0.0%
winlogon	0.0%	0.0%	0.0%	0.0%	4.8%	0.0%	0.0%	95.0%	0.0%	0.2%
All	7.3%	68.1%	2.5%	0.0%	6.4%	0.0%	0.2%	1.3%	11.6%	2.6%

Table B.11: CPU time triggered by each event type for **user #3**

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
netscape	39.9%	29.5%	2.7%	0.0%	18.4%	0.0%	0.1%	1.0%	8.1%	0.4%
realplay	2.6%	49.9%	2.8%	0.0%	4.0%	0.0%	0.1%	3.5%	35.1%	2.0%
msdtc	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.2%	97.8%	0.0%
outlook	54.0%	17.0%	4.4%	0.0%	22.4%	0.0%	0.0%	0.0%	2.2%	0.0%
ssh	3.2%	0.1%	0.8%	0.0%	95.8%	0.0%	0.0%	0.0%	0.0%	0.0%
system	0.0%	0.0%	0.0%	0.0%	8.0%	85.1%	6.9%	0.0%	0.0%	0.0%
explorer	49.2%	8.1%	24.3%	0.0%	14.9%	0.0%	0.3%	1.7%	1.5%	0.0%
exceed	37.1%	1.5%	1.3%	0.0%	58.4%	0.0%	0.4%	1.2%	0.0%	0.0%
vern	10.8%	85.0%	4.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%
services	0.0%	0.0%	0.0%	0.0%	98.4%	0.0%	0.0%	1.6%	0.0%	0.0%
portmap	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
acrord32	11.0%	86.7%	0.7%	0.0%	1.4%	0.0%	0.1%	0.0%	0.0%	0.0%
nsrexecd	0.0%	0.0%	0.0%	0.0%	99.2%	0.0%	0.0%	0.8%	0.0%	0.0%
dreamweaver	87.9%	7.1%	2.7%	0.0%	1.6%	0.0%	0.7%	0.0%	0.0%	0.0%
nsrmmd	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
nsrd	0.0%	0.0%	0.0%	0.0%	99.7%	0.0%	0.1%	0.0%	0.2%	0.0%
ttermpro	3.8%	0.0%	77.2%	0.0%	18.2%	0.0%	0.8%	0.0%	0.0%	0.0%
nsrindexd	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
shstat	0.0%	24.6%	0.0%	0.0%	74.7%	0.0%	0.6%	0.0%	0.0%	0.0%
nsrmmdbd	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
winamp	20.1%	5.1%	0.4%	0.0%	64.4%	0.0%	1.1%	4.8%	0.0%	4.2%
iexplore	50.7%	16.3%	12.5%	0.0%	19.9%	0.0%	0.3%	0.2%	0.1%	0.0%
winvnc	0.0%	8.2%	61.6%	0.0%	30.2%	0.0%	0.0%	0.0%	0.0%	0.0%
eudora	71.5%	7.4%	8.8%	0.0%	12.2%	0.0%	0.0%	0.0%	0.0%	0.0%
icq	9.5%	87.6%	1.9%	0.0%	0.9%	0.0%	0.1%	0.0%	0.0%	0.0%
All	22.9%	28.0%	3.9%	0.0%	21.6%	0.0%	0.4%	2.2%	20.4%	0.7%

Table B.12: CPU time triggered by each event type for **user #4**

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
dwrcc	0.0%	0.0%	0.0%	0.0%	22.0%	0.0%	26.2%	49.6%	2.3%	0.0%
iexplore	34.2%	22.4%	4.8%	0.1%	29.9%	0.0%	0.6%	0.5%	7.4%	0.0%
realplay	0.0%	98.8%	0.0%	0.0%	1.1%	0.0%	0.1%	0.0%	0.0%	0.0%
explorer	25.6%	11.2%	36.1%	0.0%	16.0%	0.0%	1.3%	2.4%	7.2%	0.0%
system	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%
winmgmt	0.0%	0.0%	0.0%	0.0%	95.5%	0.0%	0.0%	1.6%	2.9%	0.0%
savenow	0.0%	37.2%	0.0%	0.0%	43.5%	0.0%	2.0%	14.2%	3.1%	0.0%
rtvscan	0.0%	0.0%	0.0%	0.0%	9.3%	0.0%	0.0%	6.3%	84.5%	0.0%
rapigator	18.2%	0.6%	2.9%	0.0%	30.0%	0.0%	2.0%	32.1%	14.3%	0.0%
csrss	0.0%	0.0%	0.0%	0.0%	0.9%	0.0%	0.0%	99.1%	0.0%	0.0%
outlook	58.3%	22.2%	7.5%	0.0%	9.4%	0.0%	0.1%	0.4%	2.1%	0.0%
winvnc	0.0%	5.3%	40.4%	0.0%	53.9%	0.0%	0.2%	0.0%	0.2%	0.0%
acrord32	0.8%	98.4%	0.0%	0.0%	0.0%	0.0%	0.6%	0.0%	0.0%	0.0%
winamp	0.0%	0.0%	0.0%	0.0%	91.9%	0.0%	1.0%	7.0%	0.0%	0.1%
creatr32	0.0%	0.0%	0.0%	0.0%	96.8%	0.0%	0.3%	0.0%	2.8%	0.0%
dntu	0.0%	0.0%	0.0%	0.0%	51.1%	0.0%	6.9%	39.1%	2.9%	0.0%
services	0.0%	0.0%	0.0%	0.0%	65.6%	0.0%	0.0%	31.2%	2.1%	1.0%
winword	54.7%	0.5%	1.0%	0.0%	9.9%	0.0%	0.2%	0.6%	33.2%	0.0%
msiexec	2.8%	4.0%	0.6%	0.0%	53.1%	0.0%	0.7%	0.1%	38.8%	0.0%
fwenc	0.0%	0.0%	0.0%	0.0%	99.9%	0.0%	0.1%	0.0%	0.0%	0.0%
setup	2.2%	2.8%	0.8%	0.0%	90.4%	0.0%	3.5%	0.1%	0.2%	0.0%
photoshp	0.0%	0.0%	0.0%	0.0%	83.4%	0.0%	1.3%	14.7%	0.5%	0.0%
fssrv	0.0%	0.0%	0.0%	0.0%	78.7%	0.0%	0.0%	0.2%	21.1%	0.0%
clipbrd	0.0%	0.0%	0.0%	0.0%	99.9%	0.0%	0.0%	0.0%	0.0%	0.0%
fwpolicy	0.0%	0.0%	0.0%	0.0%	98.5%	0.0%	0.6%	0.0%	0.8%	0.0%
All	10.9%	23.4%	4.9%	0.1%	29.8%	0.0%	6.1%	16.8%	7.9%	0.0%

Table B.13: CPU time triggered by each event type for **user #5**

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
grpwise	16.1%	72.5%	1.5%	0.0%	9.0%	0.1%	0.2%	0.0%	0.7%	0.0%
ntvdm	2.6%	61.4%	0.9%	0.0%	16.2%	0.1%	0.2%	18.4%	0.2%	0.0%
findfast	0.0%	0.0%	0.0%	0.0%	98.6%	0.9%	0.1%	0.3%	0.0%	0.0%
eudora	52.6%	28.9%	6.2%	0.0%	8.0%	0.0%	1.4%	3.0%	0.0%	0.0%
mcshield	0.0%	0.0%	0.0%	0.0%	28.6%	0.0%	0.0%	34.0%	37.3%	0.0%
netscape	32.4%	46.8%	0.7%	0.0%	18.0%	0.0%	0.5%	0.1%	0.9%	0.6%
realplay	0.0%	98.6%	0.0%	0.0%	1.4%	0.0%	0.0%	0.0%	0.0%	0.0%
system	0.0%	0.0%	0.0%	0.0%	4.1%	0.3%	95.6%	0.0%	0.0%	0.0%
hotsync	0.8%	75.2%	0.1%	0.0%	19.8%	0.2%	3.8%	0.0%	0.1%	0.0%
planner	38.1%	51.6%	6.2%	0.0%	1.2%	0.0%	2.3%	0.0%	0.5%	0.0%
winword	71.7%	1.9%	0.7%	0.0%	6.0%	0.0%	0.2%	1.3%	18.2%	0.0%
realjbox	3.5%	36.9%	2.6%	0.0%	48.1%	0.0%	0.0%	0.6%	0.0%	8.2%
iexplore	46.3%	15.1%	3.4%	0.0%	34.6%	0.0%	0.4%	0.1%	0.1%	0.0%
aim	4.6%	87.1%	7.1%	0.0%	0.7%	0.0%	0.3%	0.0%	0.0%	0.0%
acrobat	11.8%	83.3%	1.8%	0.0%	1.3%	0.0%	1.7%	0.0%	0.0%	0.0%
acrord32	6.2%	92.0%	0.8%	0.0%	0.0%	0.0%	0.6%	0.4%	0.0%	0.0%
explorer	39.8%	7.0%	24.6%	0.0%	23.3%	0.1%	1.1%	0.8%	3.3%	0.0%
services	0.0%	0.0%	0.0%	0.0%	95.9%	0.1%	0.0%	2.3%	1.8%	0.0%
powerpnt	34.9%	0.3%	15.3%	0.0%	49.3%	0.0%	0.1%	0.0%	0.0%	0.0%
notify	4.8%	45.6%	0.2%	0.0%	44.7%	0.3%	3.6%	0.6%	0.1%	0.0%
shstat	0.1%	23.6%	0.1%	0.0%	74.6%	0.2%	1.5%	0.0%	0.0%	0.0%
tsystray	0.0%	73.1%	0.0%	0.0%	26.4%	0.0%	0.4%	0.0%	0.0%	0.0%
excel	87.0%	4.6%	1.2%	0.0%	5.7%	0.1%	0.3%	1.1%	0.0%	0.0%
ins0432	60.4%	30.9%	2.4%	0.0%	3.3%	0.0%	1.7%	0.0%	1.4%	0.0%
photoprn	0.0%	99.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%
All	17.7%	46.9%	2.3%	0.0%	22.7%	0.1%	1.0%	5.3%	3.8%	0.3%

Table B.14: CPU time triggered by each event type for **user #6**

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
v3webnt	0.0%	0.0%	0.0%	0.0%	37.0%	0.0%	0.0%	63.0%	0.0%	0.0%
winmgmt	0.0%	0.0%	0.0%	0.0%	85.7%	0.0%	0.0%	12.4%	2.0%	0.0%
iexplore	45.9%	14.2%	8.5%	0.1%	17.8%	0.1%	0.6%	0.3%	12.7%	0.0%
realplay	0.0%	97.9%	0.0%	0.0%	1.9%	0.0%	0.1%	0.0%	0.0%	0.0%
explorer	27.2%	20.9%	30.8%	0.0%	14.3%	0.0%	0.0%	1.6%	5.2%	0.0%
inetinfo	0.0%	0.0%	0.0%	0.0%	7.4%	0.0%	0.0%	0.1%	92.5%	0.0%
ndmonnt	0.0%	0.0%	0.0%	0.0%	1.1%	0.0%	0.0%	98.9%	0.0%	0.0%
system	0.0%	0.0%	0.0%	0.0%	3.2%	0.0%	96.8%	0.0%	0.0%	0.0%
winamp	0.0%	0.0%	0.0%	0.0%	89.2%	0.0%	0.5%	10.2%	0.0%	0.0%
outlook	51.0%	18.1%	8.0%	0.0%	13.5%	0.0%	0.1%	0.4%	9.0%	0.0%
vb6	25.5%	52.9%	2.5%	0.0%	16.2%	0.0%	0.2%	0.8%	1.9%	0.0%
services	0.0%	0.0%	0.0%	0.0%	74.0%	0.2%	0.0%	25.1%	0.1%	0.6%
winword	60.4%	4.8%	0.4%	0.0%	21.4%	0.0%	0.1%	0.6%	12.3%	0.0%
csrss	0.0%	0.0%	0.0%	0.0%	2.9%	0.0%	0.2%	96.9%	0.0%	0.0%
netcaptor	0.0%	0.0%	0.0%	0.0%	75.5%	0.0%	0.2%	1.8%	22.5%	0.0%
warftpd	0.0%	0.0%	0.0%	0.0%	99.1%	0.7%	0.1%	0.0%	0.0%	0.0%
wincmd32	0.0%	0.0%	0.0%	0.0%	81.2%	2.0%	1.2%	4.8%	10.7%	0.0%
qfmain	0.0%	0.0%	0.0%	0.0%	48.3%	0.0%	0.2%	49.8%	1.6%	0.0%
acrord32	13.5%	78.3%	0.4%	0.0%	3.7%	0.0%	0.1%	3.9%	0.0%	0.0%
getright	0.0%	0.0%	0.0%	0.0%	93.9%	0.4%	1.9%	2.4%	1.4%	0.0%
winmysqladmin	0.0%	0.0%	0.0%	0.0%	99.1%	0.5%	0.4%	0.0%	0.0%	0.0%
powerpnt	36.5%	10.2%	3.4%	0.0%	49.1%	0.0%	0.3%	0.5%	0.1%	0.0%
sqlservr	0.0%	0.0%	0.0%	0.0%	91.9%	0.0%	0.0%	7.0%	1.1%	0.0%
pro	0.0%	0.0%	0.0%	0.0%	81.4%	0.0%	0.0%	18.5%	0.1%	0.0%
internat	0.0%	0.0%	0.0%	0.0%	29.9%	0.0%	0.4%	69.8%	0.0%	0.0%
All	5.6%	6.8%	1.5%	0.0%	39.0%	0.0%	0.2%	43.0%	3.9%	0.0%

Table B.15: CPU time triggered by each event type for **user #7**

Executable	UI msg	Timer msg	Other msg	Timer object	Other object	Packet	Thread start	Session start	Time-out	APC
netscape	35.2%	39.8%	1.0%	0.0%	18.9%	0.0%	0.1%	0.1%	4.5%	0.4%
grpwise	33.4%	46.7%	3.1%	0.0%	15.0%	0.3%	0.3%	0.0%	1.2%	0.0%
realplay	0.5%	91.4%	0.8%	0.0%	2.4%	0.0%	0.1%	0.1%	0.1%	4.7%
acrord32	7.6%	91.1%	0.8%	0.0%	0.0%	0.0%	0.6%	0.0%	0.0%	0.0%
ntvdm	8.2%	71.0%	0.8%	0.0%	8.0%	0.1%	0.0%	11.9%	0.0%	0.0%
explorer	29.8%	55.2%	6.3%	0.0%	7.1%	0.2%	0.3%	0.3%	0.7%	0.0%
system	0.0%	0.0%	0.0%	0.0%	31.1%	30.5%	38.4%	0.0%	0.0%	0.0%
mcshield	0.0%	0.0%	0.0%	0.0%	22.0%	0.0%	0.0%	15.9%	62.1%	0.0%
findfast	0.0%	0.1%	0.0%	0.0%	98.0%	1.1%	0.4%	0.4%	0.0%	0.0%
remote	0.0%	60.6%	0.0%	0.0%	38.5%	0.1%	0.7%	0.0%	0.0%	0.0%
winword	80.8%	1.2%	0.7%	0.0%	6.0%	0.0%	0.5%	0.0%	10.7%	0.0%
services	0.0%	0.0%	0.0%	0.0%	97.7%	0.2%	0.0%	1.5%	0.6%	0.0%
delldmi	0.0%	99.3%	0.0%	0.0%	0.5%	0.0%	0.0%	0.2%	0.0%	0.0%
3dfot1	85.0%	0.0%	0.0%	0.0%	0.0%	0.0%	13.4%	1.6%	0.0%	0.0%
acrobat	46.3%	44.3%	3.3%	0.0%	4.2%	0.0%	1.9%	0.0%	0.0%	0.0%
shstat	0.0%	23.1%	0.0%	0.0%	74.7%	0.9%	1.3%	0.0%	0.0%	0.0%
powerpnt	70.1%	0.9%	0.0%	0.0%	28.8%	0.1%	0.1%	0.0%	0.0%	0.0%
excel	83.8%	1.2%	1.1%	0.0%	13.2%	0.0%	0.5%	0.2%	0.0%	0.0%
jerusalm	60.1%	35.0%	3.6%	0.0%	0.5%	0.0%	0.7%	0.0%	0.0%	0.1%
notify	2.9%	65.8%	0.2%	0.0%	25.5%	0.4%	4.3%	0.6%	0.4%	0.0%
activeshare	56.8%	18.1%	8.3%	0.0%	13.9%	0.7%	2.2%	0.0%	0.0%	0.0%
vstskmgr	0.0%	0.0%	0.0%	0.0%	38.2%	0.3%	0.1%	10.1%	51.4%	0.0%
smartagt	0.0%	0.0%	0.0%	0.0%	94.3%	0.1%	0.0%	3.6%	2.0%	0.0%
iexplore	6.3%	89.7%	0.4%	0.0%	3.0%	0.0%	0.5%	0.0%	0.0%	0.0%
winampa	0.0%	98.7%	0.3%	0.0%	0.0%	0.0%	0.9%	0.0%	0.0%	0.0%
All	24.5%	51.1%	1.4%	0.0%	16.0%	0.1%	0.6%	1.7%	4.0%	0.6%

Table B.16: CPU time triggered by each event type for **user #8**

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
netscape	3,378,305	22.9%	*44.5%*	75.4%	*21.3%*	1.8%	*34.2%*
acrord32	25,956	5.0%	*54.5%*	91.2%	*11.1%*	3.8%	*34.4%*
msdev	642,856	41.0%	*81.0%*	57.3%	*9.5%*	1.7%	*9.5%*
exceed	4,878,297	79.8%	*67.3%*	19.7%	*20.4%*	0.6%	*12.2%*
ssh	41,150	14.4%	*26.9%*	82.5%	*46.7%*	3.1%	*26.3%*
explorer	658,262	1.5%	*12.3%*	95.4%	*29.6%*	3.1%	*58.1%*
ntvdm	3,830	3.5%	*11.4%*	93.8%	*81.4%*	2.7%	*7.2%*
hotsync	1,218	2.6%	*0.1%*	93.5%	*93.4%*	3.9%	*6.5%*
realplay	2,005	0.7%	*6.3%*	97.4%	*53.2%*	1.9%	*40.5%*
windbg	207,498	12.6%	*48.6%*	85.4%	*14.3%*	1.9%	*37.1%*
starcraft	35,820	1.7%	*11.4%*	95.9%	*64.4%*	2.4%	*24.2%*
emacs	862,881	95.5%	*97.2%*	4.4%	*1.6%*	0.1%	*1.2%*
iexplore	85,243	3.3%	*10.6%*	94.4%	*64.2%*	2.4%	*25.2%*
shstat	373	0.5%	*9.8%*	98.4%	*77.5%*	1.1%	*12.7%*
powerpnt	154,068	12.9%	*67.5%*	85.1%	*24.7%*	2.0%	*7.8%*
msiexec	12,615	1.7%	*1.9%*	95.2%	*8.4%*	3.1%	*89.8%*
winword	74,838	66.0%	*85.6%*	33.0%	*10.7%*	1.0%	*3.6%*
All	11,456,238	51.6%	*46.3%*	47.1%	*28.4%*	1.3%	*25.4%*

Table B.17: This table shows, for **user #1**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
iexplore	5,015,889	2.6%	*15.7%*	92.2%	*35.2%*	5.1%	*49.2%*
ss3dfo	554,661	0.0%	*0.1%*	100.0%	*99.8%*	0.0%	*0.1%*
starcraft	342,054	2.7%	*14.5%*	94.2%	*58.6%*	3.0%	*26.9%*
acrord32	37,126	0.6%	*1.9%*	94.1%	*41.2%*	5.3%	*56.8%*
winword	1,040,335	15.1%	*20.1%*	81.3%	*36.3%*	3.6%	*43.6%*
psp	920,696	2.0%	*2.8%*	94.4%	*80.8%*	3.6%	*16.4%*
java	357,343	0.5%	*0.6%*	95.5%	*55.6%*	4.1%	*43.8%*
realplay	18,306	0.0%	*0.0%*	97.0%	*78.9%*	3.0%	*21.0%*
devenv	722,926	22.0%	*51.3%*	74.7%	*26.2%*	3.3%	*22.5%*
explorer	1,230,733	1.0%	*7.7%*	95.2%	*44.8%*	3.8%	*47.5%*
hotsync	7,509	0.0%	*0.2%*	94.0%	*54.6%*	5.9%	*45.2%*
msimn	860,859	11.4%	*8.4%*	83.4%	*48.6%*	5.2%	*43.1%*
powerpnt	858,505	8.7%	*26.1%*	87.7%	*34.9%*	3.6%	*38.9%*
txtpad32	563,769	40.7%	*46.1%*	56.9%	*47.9%*	2.4%	*6.0%*
mplayer2	16,544	0.1%	*0.2%*	96.8%	*83.5%*	3.1%	*16.4%*
ssh	737,046	69.0%	*67.3%*	29.8%	*24.0%*	1.1%	*8.7%*
appletviewer	133,734	0.2%	*0.1%*	95.9%	*74.6%*	3.9%	*25.3%*
realjbox	12,844	0.0%	*0.0%*	96.4%	*6.3%*	3.6%	*93.7%*
textpad	159,340	26.3%	*28.4%*	70.3%	*55.2%*	3.4%	*16.4%*
shstat	385	0.0%	*0.0%*	99.0%	*98.8%*	1.0%	*1.2%*
ntguard	232	0.0%	*0.0%*	95.3%	*26.9%*	4.7%	*73.1%*
All	15,075,773	10.6%	*17.1%*	85.4%	*44.5%*	4.0%	*38.3%*

Table B.18: This table shows, for **user #2**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
aim	24,614	6.1%	*8.0%*	92.6%	*32.8%*	1.3%	*59.2%*
netscape	775,214	1.2%	*2.1%*	96.4%	*67.0%*	2.4%	*30.9%*
realplay	3,790	0.0%	*0.0%*	98.9%	*39.6%*	1.1%	*60.4%*
ntvdm	11,850	1.8%	*5.5%*	96.9%	*57.8%*	1.3%	*36.8%*
faxmain	68,426	2.5%	*5.5%*	95.5%	*40.7%*	2.0%	*53.8%*
psp	521,664	11.6%	*6.2%*	86.9%	*60.5%*	1.5%	*33.3%*
explorer	392,801	2.0%	*4.5%*	96.3%	*56.7%*	1.7%	*38.8%*
acrord32	4,530	5.0%	*42.5%*	94.1%	*31.3%*	0.9%	*26.2%*
sitelink	5,835	1.2%	*1.7%*	96.9%	*70.1%*	1.9%	*28.2%*
sbtw	100,279	7.7%	*23.1%*	89.8%	*65.3%*	2.5%	*11.5%*
bartend	64,835	1.1%	*2.9%*	97.1%	*52.6%*	1.8%	*44.5%*
outlook	137,132	5.3%	*8.6%*	92.3%	*58.6%*	2.4%	*32.9%*
blackice	10,185	0.9%	*0.7%*	96.3%	*58.8%*	2.8%	*40.5%*
napster	60,800	9.8%	*2.2%*	89.0%	*50.7%*	1.3%	*47.1%*
iexplore	48,804	0.8%	*1.4%*	97.2%	*61.3%*	2.0%	*37.2%*
excel	72,430	15.3%	*18.3%*	82.8%	*64.5%*	1.9%	*17.2%*
winword	67,509	20.1%	*29.1%*	77.2%	*41.0%*	2.7%	*29.9%*
emexec	145	0.0%	*0.0%*	100.0%	*100.0%*	0.0%	*0.0%*
msoffice	45,678	0.7%	*2.8%*	98.2%	*52.1%*	1.1%	*45.1%*
winamp	3,994	0.1%	*0.0%*	98.3%	*46.2%*	1.5%	*53.8%*
act	3,936	8.6%	*5.6%*	85.1%	*48.2%*	6.3%	*46.2%*
ins5576	875	0.3%	*0.0%*	96.5%	*90.2%*	3.2%	*9.8%*
All	2,523,477	5.2%	*6.1%*	92.8%	*61.2%*	2.0%	*32.6%*

Table B.19: This table shows, for **user #3**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
netscape	3,395,471	11.4%	*20.2%*	85.5%	*36.1%*	3.0%	*43.7%*
realplay	148,275	27.7%	*7.5%*	70.1%	*39.3%*	2.2%	*53.3%*
outlook	1,245,865	27.6%	*39.7%*	69.7%	*45.4%*	2.7%	*14.9%*
ssh	53,004	59.7%	*67.8%*	39.3%	*21.4%*	1.0%	*10.8%*
explorer	631,672	1.0%	*4.3%*	96.6%	*50.6%*	2.4%	*45.1%*
exceed	813,999	58.6%	*35.3%*	40.2%	*39.7%*	1.1%	*25.0%*
vern	143,577	0.1%	*0.1%*	94.1%	*42.4%*	5.8%	*57.5%*
acrord32	16,943	0.1%	*0.0%*	96.3%	*35.8%*	3.6%	*64.1%*
dreamweaver	78,031	17.9%	*53.3%*	79.0%	*30.9%*	3.1%	*15.8%*
ttermpro	11,840	25.3%	*17.2%*	73.7%	*67.1%*	1.0%	*15.7%*
shstat	130	0.0%	*0.0%*	94.6%	*63.6%*	5.4%	*36.4%*
winamp	7,464	0.3%	*0.0%*	95.7%	*6.1%*	4.0%	*93.8%*
iexplore	50,826	2.9%	*5.8%*	93.4%	*67.8%*	3.7%	*26.4%*
eudora	44,312	3.9%	*4.2%*	91.2%	*72.3%*	4.9%	*23.6%*
icq	7,960	52.6%	*76.2%*	45.9%	*11.7%*	1.6%	*12.1%*
All	7,090,460	18.8%	*22.3%*	78.4%	*39.6%*	2.7%	*38.1%*

Table B.20: This table shows, for **user #4**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
iexplore	1,426,257	5.8%	*6.9%*	91.9%	*57.1%*	2.3%	*36.0%*
realplay	52	0.0%	*0.0%*	94.2%	*62.3%*	5.8%	*37.7%*
explorer	780,674	2.0%	*8.4%*	95.6%	*48.3%*	2.4%	*43.3%*
savenow	425	2.1%	*46.7%*	95.5%	*48.1%*	2.4%	*5.2%*
rapigator	2,776	1.5%	*0.0%*	71.4%	*11.2%*	27.1%	*88.7%*
outlook	724,781	14.6%	*23.8%*	84.1%	*59.6%*	1.2%	*16.6%*
acrord32	2,053	0.0%	*0.0%*	96.7%	*14.4%*	3.3%	*85.6%*
winword	128,027	11.4%	*20.4%*	85.8%	*62.5%*	2.8%	*17.1%*
msiexec	2,107	4.2%	*0.2%*	92.1%	*74.3%*	3.7%	*25.6%*
setup	1,260	0.0%	*0.0%*	96.7%	*68.7%*	3.3%	*31.3%*
All	3,527,579	6.7%	*9.8%*	91.1%	*53.6%*	2.1%	*36.6%*

Table B.21: This table shows, for **user #5**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
grpwise	1,463,341	17.8%	*16.6%*	80.0%	*42.4%*	2.2%	*40.9%*
ntvdm	19,684	28.3%	*64.2%*	70.5%	*29.0%*	1.2%	*6.7%*
findfast	15	0.0%	*0.0%*	93.3%	*63.2%*	6.7%	*36.8%*
eudora	482,345	17.3%	*24.5%*	80.9%	*53.5%*	1.8%	*22.1%*
netscape	264,339	2.3%	*2.8%*	95.9%	*31.6%*	1.8%	*65.7%*
realplay	803	0.0%	*0.0%*	97.5%	*75.1%*	2.5%	*24.9%*
hotsync	13,232	0.8%	*0.5%*	96.8%	*90.0%*	2.4%	*9.5%*
planner	249,709	10.7%	*10.2%*	86.8%	*44.2%*	2.5%	*45.6%*
winword	677,466	13.7%	*14.3%*	84.3%	*51.2%*	2.0%	*34.5%*
realjbox	21,400	2.4%	*3.6%*	94.3%	*49.5%*	3.4%	*47.0%*
iexplore	416,962	2.0%	*2.8%*	96.0%	*72.4%*	2.1%	*24.7%*
aim	126,682	35.2%	*40.6%*	63.7%	*31.4%*	1.1%	*28.0%*
acrobat	68,527	2.7%	*1.2%*	94.9%	*43.4%*	2.5%	*55.4%*
acrord32	20,286	0.0%	*0.2%*	97.3%	*28.8%*	2.7%	*71.1%*
explorer	427,230	0.6%	*0.6%*	96.9%	*66.2%*	2.5%	*33.2%*
powerpnt	16,111	2.5%	*2.0%*	94.0%	*25.7%*	3.5%	*72.3%*
notify	15,952	0.7%	*1.9%*	96.4%	*89.8%*	2.9%	*8.3%*
shstat	657	0.0%	*0.0%*	96.7%	*78.1%*	3.3%	*21.9%*
excel	194,422	7.0%	*7.0%*	90.8%	*73.7%*	2.2%	*19.3%*
ins0432	2,796	3.8%	*8.9%*	93.9%	*87.8%*	2.3%	*3.3%*
All	4,627,439	11.9%	*14.0%*	85.9%	*50.0%*	2.2%	*36.0%*

Table B.22: This table shows, for **user #6**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
iexplore	694,231	4.6%	*14.3%*	92.0%	*48.0%*	3.4%	*37.7%*
realplay	554	4.3%	*2.6%*	89.0%	*8.7%*	6.7%	*88.7%*
explorer	340,727	0.9%	*6.1%*	96.2%	*64.9%*	2.9%	*29.0%*
outlook	237,081	6.2%	*28.8%*	89.8%	*27.2%*	4.1%	*44.0%*
vb6	191,011	15.8%	*25.3%*	80.8%	*28.0%*	3.4%	*46.7%*
winword	214,154	22.8%	*34.4%*	72.7%	*44.7%*	4.5%	*20.9%*
acrord32	21,956	7.5%	*33.1%*	89.8%	*32.6%*	2.7%	*34.3%*
powerpnt	114,929	6.3%	*19.3%*	89.8%	*64.8%*	3.9%	*16.0%*
All	2,059,678	7.7%	*19.6%*	88.8%	*45.3%*	3.5%	*35.1%*

Table B.23: This table shows, for **user #7**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Application	UI events	Key press/release		Mouse move		Mouse click	
		events	*sec*	events	*sec*	events	*sec*
netscape	3,913,818	1.2%	*1.4%*	96.6%	*29.9%*	2.2%	*68.7%*
grpwise	1,733,163	8.0%	*5.0%*	89.7%	*60.6%*	2.3%	*34.4%*
realplay	35,931	0.0%	*0.1%*	97.1%	*26.9%*	2.9%	*73.1%*
acrord32	198,712	0.7%	*1.1%*	97.3%	*20.5%*	2.0%	*78.4%*
ntvdm	867,123	1.1%	*39.0%*	97.0%	*48.0%*	1.9%	*12.9%*
explorer	1,076,575	1.4%	*0.7%*	96.4%	*46.3%*	2.3%	*53.0%*
remote	47	0.0%	*0.0%*	95.7%	*85.7%*	4.3%	*14.3%*
winword	341,179	4.7%	*7.7%*	93.5%	*76.7%*	1.9%	*15.6%*
3dfot1	189	0.0%	*0.0%*	100.0%	*100.0%*	0.0%	*0.0%*
acrobat	55,136	0.0%	*0.1%*	97.8%	*28.9%*	2.2%	*71.0%*
shstat	274	0.0%	*0.0%*	94.5%	*67.0%*	5.5%	*33.0%*
powerpnt	18,341	0.7%	*11.2%*	94.8%	*39.2%*	4.5%	*49.6%*
excel	268,837	4.3%	*5.5%*	93.7%	*76.9%*	2.0%	*17.6%*
jerusalm	19,105	13.7%	*34.2%*	84.9%	*55.6%*	1.4%	*10.2%*
notify	8,717	2.1%	*5.9%*	95.0%	*78.2%*	2.9%	*15.9%*
activeshare	34,888	9.1%	*1.5%*	87.5%	*49.2%*	3.4%	*49.3%*
iexplore	6,788	0.9%	*0.4%*	97.3%	*44.4%*	1.9%	*55.2%*
All	8,976,067	3.0%	*4.4%*	94.8%	*41.0%*	2.2%	*54.6%*

Table B.24: This table shows, for **user #8**, how the user interface messages are divided among the major message types: key press/release, mouse move, and mouse click. It shows, for each application, what percentage of all such messages are of each type, and, in italics, what percentage of CPU time spent on user interface messages is spent on each type.

Appli-cation	# of requests	# of requests continuing past... system idle	next request	either boundary
netscape	3,378,305	20,982 (0.6%)	53,183 (1.6%)	55,528 (1.6%)
acrord32	25,956	1 (0.0%)	1 (0.0%)	1 (0.0%)
msdev	642,856	10,485 (1.6%)	11,937 (1.9%)	12,465 (1.9%)
exceed	4,878,297	1,211,517 (24.8%)	2,275,009 (46.6%)	2,292,936 (47.0%)
ssh	41,150	57 (0.1%)	709 (1.7%)	727 (1.8%)
explorer	658,262	1,575 (0.2%)	1,741 (0.3%)	2,546 (0.4%)
ntvdm	3,830	6 (0.2%)	22 (0.6%)	24 (0.6%)
hotsync	1,218	3 (0.2%)	6 (0.5%)	7 (0.6%)
realplay	2,005	3 (0.1%)	6 (0.3%)	8 (0.4%)
windbg	207,498	318 (0.2%)	120 (0.1%)	366 (0.2%)
starcraft	35,820	17 (0.0%)	1,853 (5.2%)	1,854 (5.2%)
emacs	862,881	2,284 (0.3%)	675 (0.1%)	2,772 (0.3%)
iexplore	85,243	3,166 (3.7%)	10,688 (12.5%)	10,832 (12.7%)
shstat	373	0 (0.0%)	0 (0.0%)	0 (0.0%)
powerpnt	154,068	328 (0.2%)	313 (0.2%)	357 (0.2%)
msiexec	12,615	0 (0.0%)	16 (0.1%)	16 (0.1%)
winword	74,838	139 (0.2%)	93 (0.1%)	176 (0.2%)
All	11,456,238	1,252,592 (10.9%)	2,357,960 (20.6%)	2,382,664 (20.8%)
(no exceed)	6,577,941	41,075 (0.6%)	82,951 (1.3%)	89,728 (1.4%)

Table B.25: For **user #1**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

B.5 Inferring task completion

In this section, we give tables showing how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application. Tables B.25, B.26, B.27, B.28, B.29, B.30, B.31, and B.32. break down this information by application for each user.

We repeated this experiment for users #1, 2, and 4 without considering object signaling to cause the receiver of the signal to be working on the same request. The results are in Tables B.33, B.34, and B.35.

Application	# of	# of requests continuing past...		
	requests	system idle	next request	either boundary
iexplore	5,178,978	96,341 (1.9%)	111,325 (2.1%)	114,051 (2.2%)
ss3dfo	554,661	1 (0.0%)	0 (0.0%)	1 (0.0%)
starcraft	342,054	0 (0.0%)	0 (0.0%)	0 (0.0%)
acrord32	37,126	2 (0.0%)	0 (0.0%)	2 (0.0%)
winword	1,040,335	220 (0.0%)	218 (0.0%)	296 (0.0%)
psp	920,696	77 (0.0%)	31 (0.0%)	82 (0.0%)
java	357,343	212,911 (59.6%)	236,223 (66.1%)	237,620 (66.5%)
realplay	18,306	82 (0.4%)	79 (0.4%)	89 (0.5%)
devenv	722,926	84,126 (11.6%)	87,522 (12.1%)	88,332 (12.2%)
explorer	1,230,733	613 (0.0%)	1,052 (0.1%)	1,414 (0.1%)
hotsync	7,509	2 (0.0%)	0 (0.0%)	2 (0.0%)
msimn	860,859	11,061 (1.3%)	11,827 (1.4%)	11,883 (1.4%)
powerpnt	858,505	116 (0.0%)	111 (0.0%)	154 (0.0%)
txtpad32	563,769	97 (0.0%)	68 (0.0%)	104 (0.0%)
mplayer2	16,544	137 (0.8%)	179 (1.1%)	186 (1.1%)
ssh	737,046	187 (0.0%)	435 (0.1%)	440 (0.1%)
appletviewer	133,734	8,048 (6.0%)	12,685 (9.5%)	12,941 (9.7%)
realjbox	12,844	15 (0.1%)	14 (0.1%)	15 (0.1%)
textpad	159,340	3,770 (2.4%)	3,536 (2.2%)	3,832 (2.4%)
shstat	385	0 (0.0%)	0 (0.0%)	0 (0.0%)
ntguard	232	0 (0.0%)	0 (0.0%)	0 (0.0%)
All	15,238,862	419,769 (2.8%)	467,337 (3.1%)	474,104 (3.1%)
(no java)	14,881,519	206,858 (1.4%)	231,114 (1.6%)	236,484 (1.6%)

Table B.26: For **user #2**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

B.6 I/O

In this section, we show what percentage of each application's user interface events perform I/O of two kinds: disk and network. Tables B.36, B.37, B.38, B.39, B.40, B.41, B.42, and B.43 show these results.

Application	**# of requests**	**# of requests continuing past...**		
		system idle	**next request**	**either boundary**
aim	24,614	119 (0.5%)	190 (0.8%)	256 (1.0%)
netscape	775,214	3,291 (0.4%)	11,331 (1.5%)	12,819 (1.7%)
realplay	3,790	21 (0.6%)	24 (0.6%)	27 (0.7%)
ntvdm	11,850	1 (0.0%)	8 (0.1%)	8 (0.1%)
faxmain	68,426	84 (0.1%)	10 (0.0%)	85 (0.1%)
psp	521,664	28 (0.0%)	10 (0.0%)	31 (0.0%)
explorer	392,801	572 (0.1%)	761 (0.2%)	1,117 (0.3%)
acrord32	4,530	8 (0.2%)	0 (0.0%)	8 (0.2%)
sitelink	5,835	10 (0.2%)	10 (0.2%)	11 (0.2%)
sbtw	100,279	49 (0.0%)	85 (0.1%)	107 (0.1%)
bartend	64,835	5 (0.0%)	4 (0.0%)	5 (0.0%)
outlook	137,132	1,431 (1.0%)	882 (0.6%)	1,805 (1.3%)
blackice	10,185	3 (0.0%)	4 (0.0%)	4 (0.0%)
napster	60,800	185 (0.3%)	227 (0.4%)	283 (0.5%)
iexplore	48,804	1,669 (3.4%)	2,268 (4.6%)	2,353 (4.8%)
excel	72,430	164 (0.2%)	1,069 (1.5%)	1,102 (1.5%)
winword	67,509	165 (0.2%)	177 (0.3%)	225 (0.3%)
emexec	145	0 (0.0%)	0 (0.0%)	0 (0.0%)
msoffice	45,678	4 (0.0%)	1 (0.0%)	5 (0.0%)
winamp	3,994	49 (1.2%)	69 (1.7%)	75 (1.9%)
act	3,936	33 (0.8%)	16 (0.4%)	38 (1.0%)
ins5576	875	12 (1.4%)	13 (1.5%)	22 (2.5%)
All	2,523,477	8,466 (0.3%)	17,382 (0.7%)	21,011 (0.8%)

Table B.27: For **user #3**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

Application	**# of requests**	**# of requests continuing past...**		
		system idle	**next request**	**either boundary**
netscape	3,395,471	29,516 (0.9%)	69,290 (2.0%)	74,650 (2.2%)
realplay	148,275	4,822 (3.3%)	6,683 (4.5%)	7,460 (5.0%)
outlook	1,245,865	4,031 (0.3%)	6,141 (0.5%)	6,820 (0.5%)
ssh	53,004	12 (0.0%)	101 (0.2%)	107 (0.2%)
explorer	631,672	1,394 (0.2%)	2,297 (0.4%)	2,763 (0.4%)
exceed	813,999	99,184 (12.2%)	311,122 (38.2%)	313,886 (38.6%)
vern	143,577	2,843 (2.0%)	3,312 (2.3%)	3,823 (2.7%)
acrord32	16,943	8 (0.0%)	0 (0.0%)	8 (0.0%)
dreamweaver	78,031	180 (0.2%)	264 (0.3%)	296 (0.4%)
ttermpro	11,840	2 (0.0%)	0 (0.0%)	2 (0.0%)
shstat	130	0 (0.0%)	0 (0.0%)	0 (0.0%)
winamp	7,464	15 (0.2%)	8 (0.1%)	15 (0.2%)
iexplore	50,826	3,354 (6.6%)	3,751 (7.4%)	3,883 (7.6%)
eudora	44,312	89 (0.2%)	106 (0.2%)	132 (0.3%)
icq	7,960	7 (0.1%)	9 (0.1%)	12 (0.2%)
All	7,090,460	146,716 (2.1%)	404,498 (5.7%)	415,569 (5.9%)
(no exceed)	6,276,461	47,532 (0.8%)	93,376 (1.5%)	101,683 (1.6%)

Table B.28: For **user #4**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

Application	**# of requests**	**# of requests continuing past...**		
		system idle	**next request**	**either boundary**
iexplore	1,426,257	68,881 (4.8%)	99,903 (7.0%)	102,044 (7.2%)
realplay	52	0 (0.0%)	10 (19.2%)	10 (19.2%)
explorer	780,674	4,105 (0.5%)	5,372 (0.7%)	6,775 (0.9%)
savenow	425	11 (2.6%)	4 (0.9%)	11 (2.6%)
rapigator	2,776	167 (6.0%)	110 (4.0%)	202 (7.3%)
outlook	724,781	3,090 (0.4%)	5,079 (0.7%)	5,550 (0.8%)
acrord32	2,053	1 (0.0%)	0 (0.0%)	1 (0.0%)
winword	128,027	71 (0.1%)	143 (0.1%)	177 (0.1%)
msiexec	2,107	1 (0.0%)	61 (2.9%)	61 (2.9%)
setup	1,260	1 (0.1%)	2 (0.2%)	3 (0.2%)
All	3,527,579	77,351 (2.2%)	112,057 (3.2%)	116,340 (3.3%)

Table B.29: For **user #5**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

Application	**# of requests**	**# of requests continuing past. . . system idle**	**next request**	**either boundary**
grpwise	1,461,773	4,928 (0.3%)	3,425 (0.2%)	5,423 (0.4%)
ntvdm	19,684	1 (0.0%)	1 (0.0%)	1 (0.0%)
findfast	15	0 (0.0%)	0 (0.0%)	0 (0.0%)
eudora	481,810	2,780 (0.6%)	3,844 (0.8%)	3,962 (0.8%)
netscape	264,339	1,241 (0.5%)	3,876 (1.5%)	4,397 (1.7%)
realplay	803	1 (0.1%)	19 (2.4%)	20 (2.5%)
hotsync	13,232	10 (0.1%)	6 (0.0%)	14 (0.1%)
planner	249,709	26,293 (10.5%)	29,883 (12.0%)	29,968 (12.0%)
winword	677,466	879 (0.1%)	2,160 (0.3%)	2,402 (0.4%)
realjbox	21,400	77 (0.4%)	90 (0.4%)	99 (0.5%)
iexplore	409,785	26,112 (6.4%)	41,342 (10.1%)	42,112 (10.3%)
aim	126,682	275 (0.2%)	275 (0.2%)	381 (0.3%)
acrobat	68,527	22 (0.0%)	0 (0.0%)	22 (0.0%)
acrord32	20,286	1 (0.0%)	0 (0.0%)	1 (0.0%)
explorer	426,113	957 (0.2%)	1,641 (0.4%)	2,019 (0.5%)
powerpnt	16,111	13 (0.1%)	70 (0.4%)	79 (0.5%)
notify	15,952	43 (0.3%)	47 (0.3%)	82 (0.5%)
shstat	657	0 (0.0%)	0 (0.0%)	0 (0.0%)
excel	194,422	361 (0.2%)	553 (0.3%)	700 (0.4%)
ins0432	2,796	6 (0.2%)	10 (0.4%)	15 (0.5%)
All	4,617,042	64,141 (1.4%)	87,421 (1.9%)	91,990 (2.0%)
(no planner)	4,367,333	37,848 (0.9%)	57,538 (1.3%)	62,022 (1.4%)

Table B.30: For **user #6**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

Application	**# of requests**	**# of requests continuing past...**		
		system idle	**next request**	**either boundary**
iexplore	681,442	43,564 (6.4%)	81,312 (11.9%)	82,351 (12.1%)
realplay	554	0 (0.0%)	0 (0.0%)	0 (0.0%)
explorer	339,371	274 (0.1%)	2,121 (0.6%)	2,203 (0.6%)
outlook	233,853	3,254 (1.4%)	9,560 (4.1%)	9,691 (4.1%)
vb6	191,011	539 (0.3%)	3,201 (1.7%)	3,437 (1.8%)
winword	208,400	84 (0.0%)	494 (0.2%)	525 (0.3%)
acrord32	21,956	11 (0.1%)	16 (0.1%)	20 (0.1%)
powerpnt	112,342	34 (0.0%)	73 (0.1%)	84 (0.1%)
All	2,033,964	53,628 (2.6%)	106,988 (5.3%)	108,703 (5.3%)

Table B.31: For **user #7**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

Application	**# of requests**	**# of requests continuing past...**		
		system idle	**next request**	**either boundary**
netscape	3,913,818	30,272 (0.8%)	54,801 (1.4%)	61,071 (1.6%)
grpwise	1,733,163	11,864 (0.7%)	12,362 (0.7%)	14,430 (0.8%)
realplay	35,931	129 (0.4%)	198 (0.6%)	235 (0.7%)
acrord32	198,712	4 (0.0%)	0 (0.0%)	4 (0.0%)
ntvdm	867,123	60 (0.0%)	61 (0.0%)	62 (0.0%)
explorer	1,076,575	881 (0.1%)	1,772 (0.2%)	2,173 (0.2%)
remote	47	0 (0.0%)	0 (0.0%)	0 (0.0%)
winword	341,179	492 (0.1%)	591 (0.2%)	673 (0.2%)
3dfot1	189	0 (0.0%)	0 (0.0%)	0 (0.0%)
acrobat	55,136	50 (0.1%)	30 (0.1%)	53 (0.1%)
shstat	274	0 (0.0%)	0 (0.0%)	0 (0.0%)
powerpnt	18,341	15 (0.1%)	58 (0.3%)	58 (0.3%)
excel	268,837	212 (0.1%)	489 (0.2%)	506 (0.2%)
jerusalm	19,105	19 (0.1%)	19 (0.1%)	21 (0.1%)
notify	8,717	43 (0.5%)	12 (0.1%)	48 (0.6%)
activeshare	34,888	34 (0.1%)	101 (0.3%)	134 (0.4%)
iexplore	6,788	14 (0.2%)	9 (0.1%)	15 (0.2%)
All	8,976,067	44,511 (0.5%)	71,472 (0.8%)	80,617 (0.9%)

Table B.32: For **user #8**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application.

Application	**# of requests**	**# of requests continuing past... system idle**	**next request**	**either boundary**
netscape	3,378,305	4,729 (0.1%)	25,331 (0.7%)	26,044 (0.8%)
acrord32	25,956	1 (0.0%)	1 (0.0%)	1 (0.0%)
msdev	642,856	8,803 (1.4%)	9,585 (1.5%)	9,856 (1.5%)
exceed	4,878,297	2,493 (0.1%)	19,223 (0.4%)	20,295 (0.4%)
ssh	41,150	13 (0.0%)	614 (1.5%)	620 (1.5%)
explorer	658,262	1,320 (0.2%)	1,463 (0.2%)	2,156 (0.3%)
ntvdm	3,830	0 (0.0%)	0 (0.0%)	0 (0.0%)
hotsync	1,218	0 (0.0%)	0 (0.0%)	0 (0.0%)
realplay	2,005	1 (0.0%)	2 (0.1%)	3 (0.1%)
windbg	207,498	297 (0.1%)	110 (0.1%)	355 (0.2%)
starcraft	35,820	0 (0.0%)	1 (0.0%)	1 (0.0%)
emacs	862,881	1 (0.0%)	1 (0.0%)	1 (0.0%)
iexplore	85,243	2,161 (2.5%)	9,288 (10.9%)	9,317 (10.9%)
shstat	373	0 (0.0%)	0 (0.0%)	0 (0.0%)
powerpnt	154,068	284 (0.2%)	265 (0.2%)	307 (0.2%)
msiexec	12,615	0 (0.0%)	0 (0.0%)	0 (0.0%)
winword	74,838	59 (0.1%)	39 (0.1%)	80 (0.1%)
All	11,456,238	21,644 (0.2%)	67,220 (0.6%)	70,755 (0.6%)

Table B.33: For **user #1**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application. Here we do not consider object signaling to cause the receiver of the signal to be working on the same request.

Application	**# of requests**	**# of requests continuing past...**		
		system idle	**next request**	**either boundary**
iexplore	5,178,978	88,120 (1.7%)	103,063 (2.0%)	103,739 (2.0%)
ss3dfo	554,661	1 (0.0%)	0 (0.0%)	1 (0.0%)
starcraft	342,054	0 (0.0%)	0 (0.0%)	0 (0.0%)
acrord32	37,126	2 (0.0%)	0 (0.0%)	2 (0.0%)
winword	1,040,335	113 (0.0%)	95 (0.0%)	142 (0.0%)
psp	920,696	77 (0.0%)	30 (0.0%)	81 (0.0%)
java	357,343	0 (0.0%)	16 (0.0%)	16 (0.0%)
realplay	18,306	37 (0.2%)	34 (0.2%)	38 (0.2%)
devenv	722,926	3,573 (0.5%)	3,477 (0.5%)	3,925 (0.5%)
explorer	1,230,733	532 (0.0%)	959 (0.1%)	1,319 (0.1%)
hotsync	7,509	1 (0.0%)	0 (0.0%)	1 (0.0%)
msimn	860,859	10,722 (1.2%)	11,400 (1.3%)	11,457 (1.3%)
powerpnt	858,505	80 (0.0%)	50 (0.0%)	93 (0.0%)
txtpad32	563,769	95 (0.0%)	68 (0.0%)	102 (0.0%)
mplayer2	16,544	2 (0.0%)	2 (0.0%)	3 (0.0%)
ssh	737,046	187 (0.0%)	432 (0.1%)	437 (0.1%)
appletviewer	133,734	0 (0.0%)	6 (0.0%)	6 (0.0%)
realjbox	12,844	1 (0.0%)	0 (0.0%)	1 (0.0%)
textpad	159,340	3,711 (2.3%)	3,504 (2.2%)	3,772 (2.4%)
shstat	385	0 (0.0%)	0 (0.0%)	0 (0.0%)
ntguard	232	0 (0.0%)	0 (0.0%)	0 (0.0%)
All	15,238,862	108,905 (0.7%)	124,764 (0.8%)	127,354 (0.8%)

Table B.34: For **user #2**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application. Here we do not consider object signaling to cause the receiver of the signal to be working on the same request.

Application	**# of requests**	**# of requests continuing past...**		
		system idle	**next request**	**either boundary**
netscape	3,395,471	11,962 (0.4%)	46,039 (1.4%)	48,295 (1.4%)
realplay	148,275	2,593 (1.7%)	4,196 (2.8%)	4,652 (3.1%)
outlook	1,245,865	2,175 (0.2%)	4,422 (0.4%)	4,761 (0.4%)
ssh	53,004	12 (0.0%)	101 (0.2%)	107 (0.2%)
explorer	631,672	1,211 (0.2%)	2,068 (0.3%)	2,523 (0.4%)
exceed	813,999	1,197 (0.1%)	5,911 (0.7%)	6,655 (0.8%)
vern	143,577	2,843 (2.0%)	3,312 (2.3%)	3,823 (2.7%)
acrord32	16,943	8 (0.0%)	0 (0.0%)	8 (0.0%)
dreamweaver	78,031	180 (0.2%)	264 (0.3%)	296 (0.4%)
ttermpro	11,840	2 (0.0%)	0 (0.0%)	2 (0.0%)
shstat	130	0 (0.0%)	0 (0.0%)	0 (0.0%)
winamp	7,464	0 (0.0%)	0 (0.0%)	0 (0.0%)
iexplore	50,826	3,151 (6.2%)	3,613 (7.1%)	3,649 (7.2%)
eudora	44,312	87 (0.2%)	80 (0.2%)	106 (0.2%)
icq	7,960	7 (0.1%)	9 (0.1%)	12 (0.2%)
All	7,090,460	26,512 (0.4%)	71,255 (1.0%)	76,364 (1.1%)

Table B.35: For **user #4**, how often the system remains working on a user interface request past the time the system goes idle with no I/O requests and/or past the time the next user interface event arrives for the same application. Here we do not consider object signaling to cause the receiver of the signal to be working on the same request.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
netscape	0.5%	*0.2%*	**0.6%**	0.2%	*0.1%*	**0.2%**	9.3%	*6.1%*	**12.3%**
acrord32	1.1%	*0.7%*	**1.8%**	0.2%	*0.0%*	**0.2%**	3.4%	*1.7%*	**4.0%**
msdev	1.3%	*8.8%*	**10.0%**	0.4%	*0.9%*	**1.3%**	13.1%	*8.1%*	**19.4%**
exceed	0.0%	*6.9%*	**6.9%**	0.0%	*5.0%*	**5.0%**	0.9%	*25.3%*	**26.1%**
ssh	0.6%	*22.8%*	**23.2%**	0.0%	*2.2%*	**2.2%**	2.8%	*3.5%*	**5.8%**
explorer	6.2%	*3.7%*	**8.1%**	0.2%	*0.1%*	**0.3%**	12.3%	*3.9%*	**13.9%**
ntvdm	3.0%	*0.7%*	**3.7%**	0.6%	*0.5%*	**1.1%**	21.2%	*0.0%*	**21.2%**
hotsync	0.0%	*0.0%*	**0.0%**	0.3%	*0.4%*	**0.4%**	8.5%	*0.0%*	**8.5%**
realplay	0.0%	*0.0%*	**0.0%**	0.4%	*0.2%*	**0.4%**	5.3%	*0.0%*	**5.3%**
windbg	0.9%	*0.3%*	**1.2%**	0.0%	*0.0%*	**0.1%**	2.3%	*2.1%*	**3.9%**
starcraft	1.6%	*20.8%*	**21.6%**	0.2%	*5.6%*	**5.7%**	5.0%	*17.6%*	**20.1%**
emacs	0.7%	*0.2%*	**0.8%**	0.2%	*0.0%*	**0.2%**	8.8%	*4.7%*	**11.6%**
iexplore	3.9%	*1.0%*	**4.1%**	0.5%	*0.1%*	**0.5%**	25.7%	*9.3%*	**26.1%**
shstat	0.0%	*0.0%*	**0.0%**	0.5%	*0.0%*	**0.5%**	0.0%	*0.0%*	**0.0%**
powerpnt	0.5%	*0.9%*	**1.3%**	0.1%	*0.1%*	**0.2%**	5.4%	*3.9%*	**7.8%**
msiexec	3.7%	*0.0%*	**3.7%**	0.2%	*0.0%*	**0.2%**	6.2%	*1.0%*	**6.2%**
winword	1.1%	*0.4%*	**1.3%**	2.5%	*0.2%*	**2.6%**	21.2%	*5.0%*	**23.3%**
All	0.3%	*5.0%*	**5.2%**	0.2%	*1.1%*	**1.2%**	8.3%	*9.3%*	**15.5%**

Application	All user interface events		
	disk	*network*	**either**
netscape	0.4%	*0.2%*	**0.5%**
acrord32	0.3%	*0.1%*	**0.4%**
msdev	1.0%	*4.2%*	**5.2%**
exceed	0.0%	*6.6%*	**6.6%**
ssh	0.2%	*5.2%*	**5.4%**
explorer	0.6%	*0.3%*	**0.8%**
ntvdm	1.3%	*0.5%*	**1.7%**
hotsync	0.6%	*0.3%*	**0.7%**
realplay	0.4%	*0.1%*	**0.5%**
windbg	0.2%	*0.1%*	**0.3%**
starcraft	0.3%	*6.2%*	**6.4%**
emacs	0.7%	*0.2%*	**0.8%**
iexplore	1.2%	*0.4%*	**1.3%**
shstat	0.5%	*0.0%*	**0.5%**
powerpnt	0.3%	*0.3%*	**0.5%**
msiexec	0.4%	*0.0%*	**0.4%**
winword	1.8%	*0.4%*	**2.0%**
All	0.3%	*3.2%*	**3.5%**

Table B.36: These tables show, for **user #1**, what percentage of various user interface tasks require waiting for disk I/O and/or the network.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
iexplore	2.1%	*0.8%*	**2.5%**	0.3%	*0.3%*	**0.5%**	4.2%	*2.3%*	**5.5%**
ss3dfo	0.0%	*0.0%*	**0.0%**	0.0%	*0.0%*	**0.0%**	0.0%	*22.2%*	**22.2%**
starcraft	0.7%	*20.3%*	**20.5%**	0.1%	*8.4%*	**8.5%**	3.3%	*16.2%*	**17.5%**
acrord32	0.9%	*0.0%*	**0.9%**	0.5%	*0.0%*	**0.5%**	8.7%	*1.3%*	**9.8%**
winword	1.2%	*0.2%*	**1.3%**	0.3%	*0.2%*	**0.5%**	4.8%	*1.4%*	**5.7%**
psp	1.0%	*1.0%*	**1.9%**	0.0%	*0.0%*	**0.1%**	3.1%	*0.6%*	**3.5%**
java	2.8%	*0.0%*	**2.8%**	0.3%	*0.1%*	**0.4%**	3.5%	*1.3%*	**4.5%**
realplay	0.0%	*0.0%*	**0.0%**	0.2%	*0.1%*	**0.2%**	9.1%	*11.8%*	**19.2%**
devenv	1.4%	*1.9%*	**2.9%**	0.3%	*0.1%*	**0.4%**	7.5%	*4.0%*	**10.1%**
explorer	5.6%	*3.4%*	**7.5%**	0.2%	*0.1%*	**0.3%**	13.8%	*4.4%*	**16.1%**
hotsync	0.0%	*33.3%*	**33.3%**	0.0%	*0.0%*	**0.1%**	9.4%	*0.9%*	**9.9%**
msimn	0.2%	*0.0%*	**0.3%**	0.1%	*0.1%*	**0.1%**	1.5%	*0.8%*	**1.9%**
powerpnt	0.7%	*0.1%*	**0.8%**	0.1%	*0.0%*	**0.1%**	3.3%	*1.1%*	**4.0%**
txtpad32	0.2%	*0.7%*	**0.8%**	0.1%	*0.3%*	**0.4%**	2.7%	*2.9%*	**5.2%**
mplayer2	0.0%	*0.0%*	**0.0%**	0.4%	*0.7%*	**0.9%**	3.5%	*2.7%*	**6.0%**
ssh	0.3%	*13.8%*	**14.0%**	0.2%	*0.0%*	**0.2%**	5.5%	*0.7%*	**5.9%**
appletviewer	5.1%	*0.0%*	**5.1%**	0.3%	*0.1%*	**0.4%**	3.0%	*3.2%*	**5.3%**
realjbox	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	7.4%	*0.2%*	**7.4%**
textpad	0.5%	*0.3%*	**0.8%**	0.1%	*0.2%*	**0.3%**	3.4%	*2.0%*	**5.0%**
shstat	N/A	*N/A*	**N/A**	1.0%	*0.0%*	**1.0%**	0.0%	*0.0%*	**0.0%**
ntguard	N/A	*N/A*	**N/A**	0.0%	*0.0%*	**0.0%**	9.1%	*0.0%*	**9.1%**
All	0.7%	*5.4%*	**5.9%**	0.2%	*0.4%*	**0.6%**	5.0%	*2.7%*	**6.8%**

Application	All user interface events		
	disk	*network*	**either**
iexplore	0.6%	*0.4%*	**0.8%**
ss3dfo	0.0%	*0.0%*	**0.0%**
starcraft	0.2%	*9.0%*	**9.1%**
acrord32	1.0%	*0.1%*	**1.0%**
winword	0.6%	*0.2%*	**0.8%**
psp	0.2%	*0.1%*	**0.2%**
java	0.5%	*0.1%*	**0.6%**
realplay	0.4%	*0.4%*	**0.8%**
devenv	0.8%	*0.6%*	**1.3%**
explorer	0.8%	*0.3%*	**1.0%**
hotsync	0.6%	*0.1%*	**0.7%**
msimn	0.2%	*0.1%*	**0.2%**
powerpnt	0.3%	*0.1%*	**0.3%**
txtpad32	0.2%	*0.6%*	**0.7%**
mplayer2	0.5%	*0.8%*	**1.1%**
ssh	0.3%	*9.5%*	**9.8%**
appletviewer	0.5%	*0.2%*	**0.6%**
realjbox	0.4%	*0.0%*	**0.4%**
textpad	0.3%	*0.3%*	**0.6%**
shstat	1.0%	*0.0%*	**1.0%**
ntguard	0.4%	*0.0%*	**0.4%**
All	0.5%	*1.1%*	**1.4%**

Table B.37: These tables show, for **user #2**, what percentage of various user interface events require waiting for disk I/O and/or the network.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
aim	3.5%	*0.3%*	**3.5%**	0.3%	*0.0%*	**0.3%**	23.6%	*0.6%*	**23.9%**
netscape	1.0%	*0.1%*	**1.0%**	0.3%	*0.2%*	**0.4%**	9.0%	*2.8%*	**10.9%**
realplay	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	17.1%	*0.0%*	**17.1%**
ntvdm	5.6%	*0.0%*	**5.6%**	1.1%	*0.0%*	**1.1%**	36.4%	*0.0%*	**36.4%**
faxmain	0.2%	*0.0%*	**0.2%**	0.1%	*0.0%*	**0.1%**	10.5%	*0.0%*	**10.5%**
psp	0.0%	*0.0%*	**0.0%**	0.0%	*0.0%*	**0.0%**	6.8%	*0.0%*	**6.8%**
explorer	1.7%	*0.2%*	**1.9%**	0.3%	*0.0%*	**0.3%**	25.5%	*3.2%*	**27.1%**
acrord32	1.8%	*0.4%*	**1.8%**	0.5%	*0.3%*	**0.8%**	41.5%	*19.5%*	**48.8%**
sitelink	0.0%	*0.0%*	**0.0%**	0.0%	*0.0%*	**0.0%**	7.1%	*0.0%*	**7.1%**
sbtw	1.9%	*11.1%*	**11.6%**	0.6%	*2.4%*	**2.6%**	1.9%	*20.5%*	**20.9%**
bartend	0.4%	*0.0%*	**0.4%**	0.0%	*0.0%*	**0.0%**	4.8%	*0.1%*	**4.8%**
outlook	0.4%	*0.0%*	**0.4%**	0.6%	*0.0%*	**0.6%**	19.6%	*0.6%*	**19.6%**
blackice	0.0%	*0.0%*	**0.0%**	0.0%	*0.0%*	**0.0%**	16.5%	*0.0%*	**16.5%**
napster	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	6.5%	*0.5%*	**6.8%**
iexplore	4.3%	*0.3%*	**4.3%**	0.8%	*0.2%*	**0.9%**	12.2%	*2.0%*	**12.6%**
excel	1.4%	*0.0%*	**1.4%**	0.7%	*0.0%*	**0.7%**	18.4%	*0.8%*	**18.7%**
winword	2.1%	*0.0%*	**2.1%**	1.1%	*0.0%*	**1.1%**	17.1%	*0.1%*	**17.1%**
emexec	N/A	*N/A*	**N/A**	2.8%	*0.0%*	**2.8%**	N/A	*N/A*	**N/A**
msoffice	2.3%	*1.0%*	**3.3%**	0.0%	*0.0%*	**0.0%**	46.9%	*10.2%*	**48.0%**
winamp	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	8.2%	*0.0%*	**8.2%**
act	0.0%	*9.7%*	**9.7%**	0.2%	*1.7%*	**1.9%**	15.8%	*17.0%*	**26.7%**
ins5576	0.0%	*0.0%*	**0.0%**	12.9%	*4.3%*	**16.8%**	25.0%	*3.6%*	**25.0%**
All	0.7%	*0.7%*	**1.4%**	0.3%	*0.2%*	**0.4%**	12.5%	*2.9%*	**14.5%**

Application	All user interface events		
	disk	*network*	**either**
aim	0.8%	*0.0%*	**0.8%**
netscape	0.5%	*0.3%*	**0.7%**
realplay	0.3%	*0.0%*	**0.3%**
ntvdm	1.7%	*0.0%*	**1.7%**
faxmain	0.3%	*0.0%*	**0.3%**
psp	0.1%	*0.0%*	**0.1%**
explorer	0.8%	*0.1%*	**0.8%**
acrord32	0.9%	*0.5%*	**1.2%**
sitelink	0.2%	*0.0%*	**0.2%**
sbtw	0.7%	*3.5%*	**3.8%**
bartend	0.1%	*0.0%*	**0.1%**
outlook	1.0%	*0.0%*	**1.0%**
blackice	0.5%	*0.0%*	**0.5%**
napster	0.1%	*0.0%*	**0.1%**
iexplore	1.1%	*0.3%*	**1.1%**
excel	1.1%	*0.0%*	**1.2%**
winword	1.7%	*0.0%*	**1.7%**
emexec	2.8%	*0.0%*	**2.8%**
msoffice	0.6%	*0.1%*	**0.6%**
winamp	0.3%	*0.0%*	**0.3%**
act	1.2%	*3.4%*	**4.2%**
ins5576	13.3%	*4.2%*	**17.0%**
All	0.5%	*0.3%*	**0.7%**

Table B.38: These tables show, for **user #3**, what percentage of various user interface events require waiting for disk I/O and/or the network.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
netscape	0.1%	*0.8%*	**0.8%**	0.1%	*0.7%*	**0.8%**	2.8%	*11.6%*	**12.9%**
realplay	0.0%	*0.1%*	**0.1%**	0.1%	*0.2%*	**0.4%**	6.6%	*15.4%*	**20.3%**
outlook	0.1%	*0.1%*	**0.1%**	0.1%	*0.1%*	**0.2%**	4.3%	*1.8%*	**5.5%**
ssh	0.1%	*29.1%*	**29.2%**	0.1%	*0.6%*	**0.7%**	3.5%	*4.3%*	**7.3%**
explorer	3.4%	*3.9%*	**5.8%**	0.2%	*0.1%*	**0.3%**	15.2%	*12.0%*	**22.4%**
exceed	0.0%	*12.4%*	**12.4%**	0.1%	*12.6%*	**12.6%**	1.1%	*39.7%*	**40.6%**
vern	3.7%	*0.0%*	**3.7%**	0.0%	*0.0%*	**0.0%**	42.1%	*0.0%*	**42.1%**
acrord32	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	6.2%	*1.6%*	**7.9%**
dreamweaver	0.2%	*0.2%*	**0.4%**	0.1%	*0.1%*	**0.2%**	2.0%	*3.1%*	**4.2%**
ttermpro	0.5%	*13.7%*	**14.2%**	0.3%	*0.9%*	**1.1%**	6.1%	*4.4%*	**10.5%**
shstat	N/A	*N/A*	**N/A**	0.0%	*0.0%*	**0.0%**	28.6%	*0.0%*	**28.6%**
winamp	0.0%	*0.0%*	**0.0%**	0.7%	*0.0%*	**0.7%**	5.0%	*2.7%*	**6.3%**
iexplore	5.0%	*0.7%*	**5.4%**	0.4%	*0.3%*	**0.5%**	3.6%	*3.7%*	**6.4%**
eudora	0.8%	*0.9%*	**1.6%**	0.1%	*0.0%*	**0.1%**	4.8%	*1.8%*	**5.4%**
icq	0.0%	*0.3%*	**0.3%**	0.1%	*0.0%*	**0.2%**	11.3%	*1.6%*	**12.9%**
All	0.1%	*5.4%*	**5.5%**	0.1%	*1.1%*	**1.2%**	6.1%	*10.0%*	**14.6%**

Application	All user interface events		
	disk	*network*	**either**
netscape	0.2%	*1.0%*	**1.1%**
realplay	0.3%	*0.5%*	**0.7%**
outlook	0.2%	*0.1%*	**0.3%**
ssh	0.2%	*17.6%*	**17.8%**
explorer	0.6%	*0.4%*	**0.9%**
exceed	0.1%	*12.8%*	**12.8%**
vern	2.5%	*0.0%*	**2.5%**
acrord32	0.3%	*0.1%*	**0.4%**
dreamweaver	0.2%	*0.2%*	**0.4%**
ttermpro	0.4%	*4.2%*	**4.5%**
shstat	1.5%	*0.0%*	**1.5%**
winamp	0.9%	*0.1%*	**1.0%**
iexplore	0.6%	*0.4%*	**0.9%**
eudora	0.4%	*0.1%*	**0.4%**
icq	0.2%	*0.2%*	**0.4%**
All	0.3%	*2.2%*	**2.4%**

Table B.39: These tables show, for **user #4**, what percentage of various user interface events require waiting for disk I/O and/or the network.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
iexplore	0.2%	*0.2%*	**0.4%**	0.1%	*0.2%*	**0.2%**	1.3%	*1.5%*	**2.6%**
realplay	N/A	*N/A*	**N/A**	0.0%	*0.0%*	**0.0%**	66.7%	*33.3%*	**66.7%**
explorer	0.6%	*0.5%*	**1.0%**	0.1%	*0.1%*	**0.2%**	5.6%	*5.5%*	**9.4%**
savenow	0.0%	*0.0%*	**0.0%**	0.0%	*0.0%*	**0.0%**	0.0%	*0.0%*	**0.0%**
rapigator	0.0%	*0.0%*	**0.0%**	0.1%	*1.4%*	**1.5%**	13.3%	*16.1%*	**22.5%**
outlook	0.1%	*0.0%*	**0.2%**	0.1%	*0.0%*	**0.1%**	4.8%	*2.6%*	**6.9%**
acrord32	N/A	*N/A*	**N/A**	0.3%	*0.0%*	**0.3%**	1.5%	*0.0%*	**1.5%**
winword	0.2%	*0.0%*	**0.3%**	0.1%	*0.0%*	**0.1%**	0.5%	*0.4%*	**0.9%**
msiexec	0.0%	*0.0%*	**0.0%**	2.6%	*0.0%*	**2.6%**	12.8%	*0.0%*	**12.8%**
setup	N/A	*N/A*	**N/A**	0.2%	*1.3%*	**1.5%**	0.0%	*0.0%*	**0.0%**
All	0.2%	*0.1%*	**0.3%**	0.1%	*0.1%*	**0.2%**	3.1%	*2.7%*	**5.1%**

Application	All user interface events		
	disk	*network*	**either**
iexplore	0.1%	*0.2%*	**0.3%**
realplay	3.8%	*1.9%*	**3.8%**
explorer	0.2%	*0.2%*	**0.4%**
savenow	0.0%	*0.0%*	**0.0%**
rapigator	3.7%	*5.4%*	**7.1%**
outlook	0.1%	*0.0%*	**0.2%**
acrord32	0.3%	*0.0%*	**0.3%**
winword	0.1%	*0.0%*	**0.1%**
msiexec	2.9%	*0.0%*	**2.9%**
setup	0.2%	*1.3%*	**1.4%**
All	0.2%	*0.2%*	**0.3%**

Table B.40: These tables show, for **user #5**, what percentage of various user interface events require waiting for disk I/O and/or the network.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
grpwise	0.1%	*0.0%*	**0.1%**	0.1%	*0.1%*	**0.1%**	4.8%	*8.6%*	**10.7%**
ntvdm	0.1%	*5.8%*	**5.9%**	0.6%	*0.3%*	**0.8%**	6.6%	*5.3%*	**11.1%**
findfast	N/A	*N/A*	**N/A**	0.0%	*0.0%*	**0.0%**	100.0%	*0.0%*	**100.0%**
eudora	0.2%	*0.1%*	**0.3%**	0.3%	*0.1%*	**0.3%**	18.4%	*2.7%*	**19.4%**
netscape	1.1%	*1.2%*	**1.6%**	0.3%	*0.5%*	**0.7%**	12.2%	*13.1%*	**19.2%**
realplay	N/A	*N/A*	**N/A**	0.1%	*0.0%*	**0.1%**	10.0%	*0.0%*	**10.0%**
hotsync	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	14.0%	*1.9%*	**14.0%**
planner	0.1%	*0.0%*	**0.1%**	0.1%	*0.0%*	**0.1%**	8.9%	*0.0%*	**8.9%**
winword	1.1%	*0.2%*	**1.3%**	0.5%	*0.2%*	**0.7%**	15.1%	*2.3%*	**16.5%**
realjbox	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	4.1%	*0.0%*	**4.1%**
iexplore	2.0%	*0.2%*	**2.1%**	0.2%	*0.1%*	**0.3%**	6.8%	*6.1%*	**11.0%**
aim	0.2%	*0.6%*	**0.8%**	0.1%	*0.0%*	**0.1%**	8.7%	*4.0%*	**11.0%**
acrobat	0.0%	*0.0%*	**0.0%**	0.1%	*0.0%*	**0.1%**	2.7%	*0.1%*	**2.7%**
acrord32	20.0%	*0.0%*	**20.0%**	0.1%	*0.0%*	**0.1%**	2.4%	*0.0%*	**2.4%**
explorer	1.1%	*0.2%*	**1.1%**	0.3%	*0.1%*	**0.4%**	16.0%	*3.0%*	**17.5%**
powerpnt	20.0%	*0.0%*	**20.0%**	1.0%	*0.0%*	**1.0%**	28.4%	*1.1%*	**28.4%**
notify	15.9%	*17.8%*	**17.8%**	0.9%	*1.6%*	**1.6%**	4.7%	*6.9%*	**9.0%**
shstat	N/A	*N/A*	**N/A**	0.3%	*0.0%*	**0.3%**	18.2%	*0.0%*	**18.2%**
excel	0.9%	*0.0%*	**0.9%**	0.4%	*0.0%*	**0.4%**	11.7%	*1.9%*	**12.8%**
ins0432	0.0%	*0.0%*	**0.0%**	4.0%	*0.0%*	**4.0%**	7.8%	*0.0%*	**7.8%**
All	0.4%	*0.2%*	**0.6%**	0.2%	*0.1%*	**0.3%**	10.0%	*5.1%*	**13.3%**

Application	All user interface events		
	disk	*network*	**either**
grpwise	0.2%	*0.3%*	**0.4%**
ntvdm	0.6%	*1.9%*	**2.4%**
findfast	6.7%	*0.0%*	**6.7%**
eudora	0.6%	*0.1%*	**0.7%**
netscape	0.5%	*0.8%*	**1.0%**
realplay	0.4%	*0.0%*	**0.4%**
hotsync	0.5%	*0.1%*	**0.5%**
planner	0.4%	*0.0%*	**0.4%**
winword	0.9%	*0.3%*	**1.1%**
realjbox	0.2%	*0.0%*	**0.2%**
iexplore	0.4%	*0.2%*	**0.6%**
aim	0.2%	*0.3%*	**0.4%**
acrobat	0.1%	*0.0%*	**0.1%**
acrord32	0.2%	*0.0%*	**0.2%**
explorer	0.7%	*0.1%*	**0.8%**
powerpnt	2.5%	*0.1%*	**2.5%**
notify	1.1%	*1.9%*	**2.0%**
shstat	0.9%	*0.0%*	**0.9%**
excel	0.7%	*0.1%*	**0.7%**
ins0432	3.9%	*0.0%*	**3.9%**
All	0.5%	*0.2%*	**0.6%**

Table B.41: These tables show, for **user #6**, what percentage of various user interface events require waiting for disk I/O and/or the network.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
iexplore	4.2%	*1.2%*	**4.6%**	0.5%	*0.2%*	**0.5%**	8.2%	*2.8%*	**9.0%**
realplay	0.0%	*0.0%*	**0.0%**	0.0%	*0.0%*	**0.0%**	29.7%	*0.0%*	**29.7%**
explorer	8.5%	*0.0%*	**8.5%**	0.6%	*0.0%*	**0.6%**	11.9%	*0.1%*	**12.0%**
outlook	4.5%	*1.7%*	**4.8%**	0.6%	*0.1%*	**0.6%**	16.9%	*2.3%*	**17.4%**
vb6	1.7%	*0.0%*	**1.7%**	0.2%	*0.0%*	**0.2%**	12.0%	*0.3%*	**12.0%**
winword	1.5%	*0.0%*	**1.5%**	0.5%	*0.0%*	**0.5%**	7.0%	*0.1%*	**7.0%**
acrord32	8.1%	*0.0%*	**8.1%**	0.2%	*0.0%*	**0.2%**	5.6%	*0.0%*	**5.6%**
powerpnt	2.6%	*0.0%*	**2.6%**	0.3%	*0.0%*	**0.3%**	5.8%	*0.0%*	**5.8%**
All	2.6%	*0.4%*	**2.7%**	0.5%	*0.1%*	**0.5%**	10.0%	*1.5%*	**10.5%**

Application	All user interface events		
	disk	*network*	**either**
iexplore	0.9%	*0.3%*	**1.0%**
realplay	2.0%	*0.0%*	**2.0%**
explorer	1.0%	*0.0%*	**1.0%**
outlook	1.5%	*0.3%*	**1.6%**
vb6	0.9%	*0.0%*	**0.9%**
winword	1.0%	*0.0%*	**1.0%**
acrord32	0.9%	*0.0%*	**0.9%**
powerpnt	0.7%	*0.0%*	**0.7%**
All	1.0%	*0.2%*	**1.0%**

Table B.42: These tables show, for **user #7**, what percentage of various user interface events require waiting for disk I/O and/or the network.

Application	Key press/release			Mouse move			Mouse click		
	disk	*network*	**either**	disk	*network*	**either**	disk	*network*	**either**
netscape	1.4%	*0.8%*	**1.5%**	0.3%	*0.2%*	**0.4%**	23.1%	*14.8%*	**23.8%**
grpwise	0.2%	*0.0%*	**0.3%**	0.2%	*0.1%*	**0.3%**	9.7%	*3.9%*	**11.7%**
realplay	0.0%	*0.0%*	**0.0%**	0.3%	*0.0%*	**0.3%**	23.0%	*1.1%*	**23.2%**
acrord32	1.4%	*0.0%*	**1.4%**	0.2%	*0.0%*	**0.2%**	9.7%	*3.0%*	**9.7%**
ntvdm	0.1%	*4.8%*	**4.9%**	0.0%	*0.0%*	**0.0%**	0.8%	*0.2%*	**0.9%**
explorer	1.4%	*0.0%*	**1.4%**	0.3%	*0.0%*	**0.4%**	21.6%	*1.7%*	**22.2%**
remote	N/A	*N/A*	**N/A**	2.2%	*0.0%*	**2.2%**	0.0%	*0.0%*	**0.0%**
winword	2.8%	*0.0%*	**2.8%**	1.4%	*0.0%*	**1.4%**	21.3%	*1.1%*	**21.3%**
3dfot1	N/A	*N/A*	**N/A**	24.9%	*0.0%*	**24.9%**	N/A	*N/A*	**N/A**
acrobat	4.2%	*0.0%*	**4.2%**	0.2%	*0.0%*	**0.2%**	14.9%	*0.7%*	**15.4%**
shstat	N/A	*N/A*	**N/A**	0.0%	*0.0%*	**0.0%**	13.3%	*0.0%*	**13.3%**
powerpnt	32.8%	*0.0%*	**32.8%**	1.8%	*0.0%*	**1.8%**	21.0%	*0.1%*	**21.0%**
excel	1.5%	*0.0%*	**1.5%**	0.6%	*0.0%*	**0.6%**	14.6%	*0.2%*	**14.6%**
jerusalm	0.1%	*0.0%*	**0.1%**	0.3%	*0.0%*	**0.3%**	11.7%	*0.0%*	**11.7%**
notify	7.7%	*1.6%*	**8.2%**	0.5%	*0.4%*	**0.6%**	18.6%	*11.9%*	**20.6%**
activeshare	0.2%	*0.0%*	**0.2%**	0.7%	*0.0%*	**0.7%**	19.1%	*0.9%*	**19.3%**
iexplore	0.0%	*0.0%*	**0.0%**	1.0%	*0.1%*	**1.0%**	19.0%	*3.2%*	**19.8%**
All	0.8%	*2.5%*	**3.1%**	0.3%	*0.1%*	**0.4%**	17.2%	*7.9%*	**18.1%**

Application	All user interface events		
	disk	*network*	**either**
netscape	0.8%	*0.5%*	**0.9%**
grpwise	0.5%	*0.2%*	**0.5%**
realplay	0.9%	*0.0%*	**1.0%**
acrord32	0.4%	*0.1%*	**0.4%**
ntvdm	0.0%	*0.1%*	**0.1%**
explorer	0.8%	*0.1%*	**0.9%**
remote	2.1%	*0.0%*	**2.1%**
winword	1.8%	*0.0%*	**1.8%**
3dfot1	24.9%	*0.0%*	**24.9%**
acrobat	0.5%	*0.0%*	**0.5%**
shstat	0.7%	*0.0%*	**0.7%**
powerpnt	2.8%	*0.0%*	**2.8%**
excel	0.9%	*0.0%*	**0.9%**
jerusalm	0.4%	*0.0%*	**0.4%**
notify	1.2%	*0.8%*	**1.3%**
activeshare	1.3%	*0.0%*	**1.3%**
iexplore	1.3%	*0.1%*	**1.4%**
All	0.7%	*0.3%*	**0.8%**

Table B.43: These tables show, for **user #8**, what percentage of various user interface events require waiting for disk I/O and/or the network.

B.7 Mouse clicks

In this section we show, for each user and each of the top 25 applications of that user, how many tasks and how much work each category of mouse click event triggers. Tables B.44, B.45, B.46, B.47, B.48, B.49, B.50, and B.51 show this information.

B.8 Key presses and releases

In this section we show, for each user and each of the top 25 applications of that user, how many tasks and how much work each category of keystroke event triggers. Tables B.52, B.53, B.54, B.55, B.56, B.57, B.58, and B.59 show this information.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category. . .						
netscape	52.3%	41.9%	1.8%	3.7%	0.3%	0.0%
acrord32	72.4%	11.3%	15.7%	0.6%	0.0%	0.0%
msdev	45.0%	45.6%	8.0%	0.7%	0.7%	0.0%
exceed	44.6%	38.6%	1.6%	4.6%	6.2%	4.3%
ssh	55.9%	42.3%	1.8%	0.0%	0.0%	0.0%
explorer	42.7%	45.2%	8.5%	1.7%	1.7%	0.0%
ntvdm	69.2%	26.9%	3.8%	0.0%	0.0%	0.0%
hotsync	57.4%	40.4%	2.1%	0.0%	0.0%	0.0%
realplay	65.8%	34.2%	0.0%	0.0%	0.0%	0.0%
windbg	68.1%	27.4%	4.4%	0.0%	0.0%	0.0%
starcraft	30.9%	31.5%	0.6%	15.2%	18.5%	3.4%
emacs	56.4%	43.3%	0.3%	0.0%	0.0%	0.0%
iexplore	49.7%	47.3%	2.2%	0.3%	0.3%	0.0%
shstat	50.0%	50.0%	0.0%	0.0%	0.0%	0.0%
powerpnt	48.7%	46.1%	4.9%	0.2%	0.2%	0.0%
msiexec	55.3%	44.2%	0.5%	0.0%	0.0%	0.0%
winword	50.1%	45.6%	3.7%	0.3%	0.3%	0.0%
All	49.1%	41.6%	3.6%	2.9%	1.8%	0.9%
Percent of CPU time due to each category. . .						
netscape	48.0%	43.3%	1.2%	7.3%	0.1%	0.0%
acrord32	68.5%	9.7%	21.6%	0.2%	0.0%	0.0%
msdev	23.7%	60.5%	13.3%	0.1%	2.4%	0.0%
exceed	87.1%	10.9%	0.2%	0.3%	0.9%	0.6%
ssh	66.8%	33.1%	0.1%	0.0%	0.0%	0.0%
explorer	34.2%	44.3%	14.3%	0.2%	6.9%	0.0%
ntvdm	66.0%	33.5%	0.5%	0.0%	0.0%	0.0%
hotsync	26.6%	73.1%	0.2%	0.0%	0.0%	0.0%
realplay	66.0%	34.0%	0.0%	0.0%	0.0%	0.0%
windbg	88.9%	7.5%	3.6%	0.0%	0.0%	0.0%
starcraft	19.7%	26.8%	6.1%	1.4%	41.0%	5.0%
emacs	94.2%	5.3%	0.5%	0.0%	0.0%	0.0%
iexplore	57.5%	34.8%	7.2%	0.0%	0.5%	0.0%
shstat	12.2%	87.8%	0.0%	0.0%	0.0%	0.0%
powerpnt	46.8%	46.5%	6.6%	0.0%	0.1%	0.0%
msiexec	16.5%	83.5%	0.1%	0.0%	0.0%	0.0%
winword	35.8%	55.8%	7.1%	0.0%	1.4%	0.0%
All	45.1%	43.5%	4.5%	4.3%	2.5%	0.2%

Table B.44: For **user #1**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category...						
iexplore	49.1%	45.9%	2.3%	1.0%	1.1%	0.6%
ss3dfo	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
starcraft	35.1%	37.6%	2.6%	11.1%	12.3%	1.3%
acrord32	65.7%	14.5%	18.6%	0.7%	0.4%	0.2%
winword	52.0%	42.0%	3.7%	1.0%	1.0%	0.3%
psp	43.6%	36.1%	0.9%	8.4%	9.7%	1.3%
java	49.4%	42.9%	0.2%	3.8%	3.8%	0.0%
realplay	64.1%	33.2%	2.5%	0.2%	0.0%	0.0%
devenv	46.5%	42.2%	8.5%	1.2%	1.3%	0.3%
explorer	45.7%	41.1%	5.1%	4.0%	4.0%	0.0%
hotsync	49.9%	49.0%	1.1%	0.0%	0.0%	0.0%
msimn	48.4%	45.2%	0.8%	2.7%	2.7%	0.2%
powerpnt	51.7%	43.1%	4.0%	0.6%	0.6%	0.0%
txtpad32	52.2%	45.7%	1.1%	0.4%	0.4%	0.2%
mplayer2	60.1%	38.8%	0.4%	0.4%	0.4%	0.0%
ssh	56.3%	42.3%	1.4%	0.0%	0.0%	0.0%
appletviewer	51.7%	44.2%	0.2%	1.8%	1.8%	0.2%
realjbox	52.5%	43.3%	4.2%	0.0%	0.0%	0.0%
textpad	51.9%	44.9%	1.2%	0.4%	0.5%	1.2%
shstat	50.0%	50.0%	0.0%	0.0%	0.0%	0.0%
ntguard	63.6%	36.4%	0.0%	0.0%	0.0%	0.0%
All	48.7%	43.0%	3.1%	2.4%	2.5%	0.4%
Percent of CPU time due to each category...						
iexplore	29.2%	64.8%	2.6%	0.2%	3.1%	0.1%
ss3dfo	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%
starcraft	10.8%	69.1%	0.5%	11.5%	7.8%	0.3%
acrord32	61.8%	17.1%	21.0%	0.1%	0.0%	0.0%
winword	28.8%	68.4%	2.5%	0.1%	0.1%	0.0%
psp	61.0%	23.5%	1.6%	9.8%	1.8%	2.1%
java	33.5%	56.5%	0.3%	1.1%	8.6%	0.0%
realplay	82.5%	15.7%	1.8%	0.0%	0.0%	0.0%
devenv	31.3%	54.6%	12.1%	1.7%	0.2%	0.1%
explorer	52.9%	21.8%	12.4%	0.6%	12.4%	0.0%
hotsync	44.8%	54.5%	0.7%	0.0%	0.0%	0.0%
msimn	49.4%	47.6%	1.2%	1.2%	0.6%	0.0%
powerpnt	53.8%	41.8%	4.1%	0.1%	0.2%	0.0%
txtpad32	76.0%	21.3%	1.9%	0.2%	0.5%	0.1%
mplayer2	56.5%	38.9%	3.6%	0.0%	1.0%	0.0%
ssh	91.1%	8.3%	0.6%	0.0%	0.0%	0.0%
appletviewer	78.2%	20.1%	0.6%	1.0%	0.2%	0.0%
realjbox	2.1%	97.9%	0.0%	0.0%	0.0%	0.0%
textpad	83.1%	14.4%	1.9%	0.1%	0.4%	0.1%
shstat	18.9%	81.1%	0.0%	0.0%	0.0%	0.0%
ntguard	92.7%	7.3%	0.0%	0.0%	0.0%	0.0%
All	33.7%	57.2%	2.9%	2.4%	3.6%	0.1%

Table B.45: For **user #2**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category...						
aim	60.4%	25.9%	4.4%	4.7%	4.7%	0.0%
netscape	48.6%	38.7%	1.4%	11.1%	0.3%	0.0%
realplay	68.3%	31.7%	0.0%	0.0%	0.0%	0.0%
ntvdm	59.7%	39.6%	0.0%	0.6%	0.0%	0.0%
faxmain	50.2%	39.2%	4.0%	3.3%	3.3%	0.0%
psp	54.0%	44.9%	0.5%	0.2%	0.3%	0.1%
explorer	43.5%	39.6%	6.0%	5.5%	5.4%	0.0%
acrord32	65.9%	29.3%	0.0%	4.9%	0.0%	0.0%
sitelink	55.8%	38.9%	0.0%	2.7%	2.7%	0.0%
sbtw	53.4%	45.4%	0.0%	0.6%	0.6%	0.0%
bartend	49.1%	44.4%	6.1%	0.3%	0.3%	0.0%
outlook	55.5%	35.4%	9.0%	0.1%	0.1%	0.0%
blackice	52.0%	42.2%	5.4%	0.4%	0.0%	0.0%
napster	41.3%	34.7%	0.5%	11.9%	11.6%	0.0%
iexplore	53.8%	42.1%	0.0%	2.1%	2.1%	0.0%
excel	51.1%	39.2%	9.4%	0.1%	0.1%	0.0%
winword	51.2%	39.9%	5.4%	1.7%	1.7%	0.0%
msoffice	48.4%	50.2%	0.4%	1.0%	0.0%	0.0%
winamp	52.5%	47.5%	0.0%	0.0%	0.0%	0.0%
act	53.4%	30.8%	15.8%	0.0%	0.0%	0.0%
ins5576	50.0%	50.0%	0.0%	0.0%	0.0%	0.0%
All	50.3%	40.7%	3.2%	4.2%	1.6%	0.0%
Percent of CPU time due to each category...						
aim	57.8%	7.5%	5.9%	0.2%	28.6%	0.0%
netscape	39.6%	53.0%	0.4%	6.8%	0.1%	0.0%
realplay	81.3%	18.7%	0.0%	0.0%	0.0%	0.0%
ntvdm	30.1%	69.3%	0.0%	0.7%	0.0%	0.0%
faxmain	79.8%	7.7%	10.8%	0.2%	1.5%	0.0%
psp	92.5%	7.2%	0.1%	0.1%	0.0%	0.0%
explorer	58.9%	21.6%	6.2%	0.8%	12.5%	0.0%
acrord32	60.3%	37.8%	0.0%	1.9%	0.0%	0.0%
sitelink	83.4%	12.1%	0.0%	0.1%	4.4%	0.0%
sbtw	22.3%	77.3%	0.0%	0.1%	0.3%	0.0%
bartend	58.3%	33.1%	8.1%	0.3%	0.2%	0.0%
outlook	19.5%	75.2%	5.3%	0.0%	0.0%	0.0%
blackice	82.0%	11.7%	4.8%	1.5%	0.0%	0.0%
napster	11.6%	84.7%	0.1%	0.9%	2.7%	0.0%
iexplore	18.2%	76.6%	0.0%	0.1%	5.1%	0.0%
excel	38.1%	51.4%	10.2%	0.0%	0.3%	0.0%
winword	34.8%	51.8%	12.2%	0.2%	1.0%	0.0%
msoffice	2.4%	96.7%	0.0%	0.9%	0.0%	0.0%
winamp	3.6%	96.4%	0.0%	0.0%	0.0%	0.0%
act	38.6%	60.3%	1.2%	0.0%	0.0%	0.0%
ins5576	0.3%	99.7%	0.0%	0.0%	0.0%	0.0%
All	59.2%	34.7%	2.3%	2.2%	1.6%	0.0%

Table B.46: For **user #3**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category…						
netscape	55.2%	42.6%	1.2%	1.0%	0.0%	0.0%
realplay	62.7%	37.0%	0.2%	0.0%	0.0%	0.0%
outlook	51.6%	46.5%	1.3%	0.3%	0.3%	0.1%
ssh	58.0%	36.9%	0.7%	0.6%	0.0%	3.7%
explorer	45.4%	37.9%	3.2%	6.8%	6.7%	0.1%
exceed	50.1%	37.9%	0.1%	0.2%	0.2%	11.5%
vern	49.6%	50.2%	0.0%	0.1%	0.1%	0.0%
acrord32	74.4%	20.7%	4.4%	0.3%	0.2%	0.0%
dreamweaver	50.2%	43.1%	0.7%	2.9%	3.0%	0.0%
ttermpro	78.8%	21.2%	0.0%	0.0%	0.0%	0.0%
shstat	57.1%	42.9%	0.0%	0.0%	0.0%	0.0%
winamp	46.3%	50.7%	3.0%	0.0%	0.0%	0.0%
iexplore	47.9%	48.2%	2.5%	0.7%	0.7%	0.0%
eudora	49.1%	42.1%	5.8%	1.2%	1.3%	0.3%
icq	52.4%	37.9%	3.2%	3.2%	3.2%	0.0%
All	53.3%	42.7%	1.5%	1.2%	0.7%	0.6%
Percent of CPU time due to each category…						
netscape	55.3%	41.8%	1.3%	1.5%	0.0%	0.0%
realplay	53.9%	46.1%	0.0%	0.0%	0.0%	0.0%
outlook	43.2%	54.5%	1.3%	0.2%	0.7%	0.0%
ssh	89.7%	8.9%	0.4%	0.3%	0.0%	0.8%
explorer	50.0%	14.9%	5.7%	0.5%	28.8%	0.0%
exceed	84.4%	14.5%	0.3%	0.0%	0.0%	0.7%
vern	4.1%	95.7%	0.0%	0.0%	0.2%	0.0%
acrord32	83.9%	10.9%	5.1%	0.1%	0.0%	0.0%
dreamweaver	61.7%	31.8%	0.4%	0.3%	5.8%	0.0%
ttermpro	99.6%	0.4%	0.0%	0.0%	0.0%	0.0%
shstat	24.8%	75.2%	0.0%	0.0%	0.0%	0.0%
winamp	0.6%	81.7%	17.7%	0.0%	0.0%	0.0%
iexplore	34.3%	55.4%	2.1%	0.1%	8.1%	0.0%
eudora	46.9%	39.7%	11.6%	0.2%	1.6%	0.0%
icq	51.8%	35.1%	5.2%	0.3%	7.6%	0.0%
All	54.1%	40.7%	1.9%	1.1%	2.2%	0.0%

Table B.47: For **user #4**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category...						
iexplore	49.7%	44.3%	1.1%	1.9%	1.9%	1.1%
realplay	66.7%	33.3%	0.0%	0.0%	0.0%	0.0%
explorer	38.1%	43.8%	7.7%	5.1%	5.1%	0.3%
savenow	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rapigator	1.9%	94.8%	0.0%	0.0%	3.3%	0.0%
outlook	54.0%	38.5%	4.2%	0.9%	0.9%	1.4%
acrord32	82.1%	10.4%	7.5%	0.0%	0.0%	0.0%
winword	49.9%	37.1%	4.1%	1.6%	1.5%	5.9%
msiexec	48.7%	47.4%	1.3%	1.3%	1.3%	0.0%
setup	52.4%	47.6%	0.0%	0.0%	0.0%	0.0%
All	46.1%	43.0%	4.4%	2.8%	2.8%	0.9%
Percent of CPU time due to each category...						
iexplore	23.4%	65.4%	2.7%	0.5%	7.9%	0.1%
realplay	14.3%	85.7%	0.0%	0.0%	0.0%	0.0%
explorer	27.1%	26.0%	24.3%	0.9%	21.8%	0.0%
savenow	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rapigator	0.7%	95.4%	0.0%	0.0%	3.9%	0.0%
outlook	73.7%	22.7%	2.3%	0.2%	1.0%	0.1%
acrord32	97.3%	0.0%	2.7%	0.0%	0.0%	0.0%
winword	69.4%	23.9%	1.6%	2.0%	1.7%	1.5%
msiexec	11.9%	87.5%	0.0%	0.0%	0.6%	0.0%
setup	2.2%	97.8%	0.0%	0.0%	0.0%	0.0%
All	27.4%	52.8%	9.0%	0.6%	10.0%	0.1%

Table B.48: For **user #5**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category…						
grpwise	49.9%	35.0%	12.8%	1.2%	1.2%	0.0%
ntvdm	56.0%	42.8%	1.2%	0.0%	0.0%	0.0%
findfast	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
eudora	49.1%	39.1%	7.7%	2.1%	2.0%	0.0%
netscape	64.2%	28.9%	6.5%	0.2%	0.2%	0.0%
realplay	55.0%	45.0%	0.0%	0.0%	0.0%	0.0%
hotsync	51.3%	48.7%	0.0%	0.0%	0.0%	0.0%
planner	47.0%	43.4%	4.3%	2.8%	2.5%	0.0%
winword	57.1%	32.6%	8.5%	0.9%	0.9%	0.0%
realjbox	43.6%	37.3%	3.6%	7.7%	7.7%	0.0%
iexplore	46.5%	48.1%	4.7%	0.3%	0.3%	0.0%
aim	66.4%	23.9%	8.9%	0.5%	0.2%	0.0%
acrobat	66.4%	14.1%	18.9%	0.4%	0.2%	0.0%
acrord32	76.8%	5.5%	17.0%	0.7%	0.0%	0.0%
explorer	36.5%	47.0%	12.5%	1.9%	2.0%	0.1%
powerpnt	48.4%	38.9%	11.9%	0.4%	0.4%	0.0%
notify	51.7%	48.3%	0.0%	0.0%	0.0%	0.0%
shstat	50.0%	45.5%	0.0%	4.5%	0.0%	0.0%
excel	52.7%	36.2%	7.9%	1.6%	1.6%	0.0%
ins0432	50.0%	50.0%	0.0%	0.0%	0.0%	0.0%
All	50.4%	37.5%	9.6%	1.3%	1.3%	0.0%
Percent of CPU time due to each category…						
grpwise	67.7%	18.6%	13.1%	0.1%	0.6%	0.0%
ntvdm	58.4%	41.5%	0.0%	0.0%	0.0%	0.0%
findfast	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
eudora	68.9%	23.1%	7.0%	0.2%	0.9%	0.0%
netscape	61.7%	37.2%	0.5%	0.6%	0.1%	0.0%
realplay	34.4%	65.6%	0.0%	0.0%	0.0%	0.0%
hotsync	20.4%	79.6%	0.0%	0.0%	0.0%	0.0%
planner	12.9%	78.5%	6.2%	0.5%	1.9%	0.0%
winword	21.5%	72.2%	5.3%	0.6%	0.5%	0.0%
realjbox	48.5%	41.3%	1.2%	1.6%	7.4%	0.0%
iexplore	41.1%	46.4%	11.9%	0.0%	0.6%	0.0%
aim	83.9%	6.0%	7.1%	0.5%	2.5%	0.0%
acrobat	73.7%	18.0%	8.1%	0.1%	0.1%	0.0%
acrord32	93.1%	2.0%	4.7%	0.2%	0.0%	0.0%
explorer	31.8%	33.6%	29.7%	0.3%	4.6%	0.0%
powerpnt	3.0%	8.9%	87.9%	0.0%	0.1%	0.0%
notify	18.6%	81.4%	0.0%	0.0%	0.0%	0.0%
shstat	29.1%	53.2%	0.0%	17.7%	0.0%	0.0%
excel	39.9%	51.6%	8.1%	0.2%	0.2%	0.0%
ins0432	24.3%	75.7%	0.0%	0.0%	0.0%	0.0%
All	48.6%	40.2%	10.0%	0.3%	0.8%	0.0%

Table B.49: For **user #6**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category...						
iexplore	50.4%	47.3%	0.7%	0.8%	0.8%	0.0%
realplay	56.8%	43.2%	0.0%	0.0%	0.0%	0.0%
explorer	45.4%	47.4%	1.1%	3.0%	3.1%	0.0%
outlook	47.4%	46.9%	2.5%	1.6%	1.6%	0.0%
vb6	46.3%	47.9%	4.4%	0.7%	0.7%	0.0%
winword	47.8%	48.1%	1.8%	1.1%	1.1%	0.0%
acrord32	55.9%	42.4%	0.5%	1.0%	0.2%	0.0%
powerpnt	48.7%	48.9%	0.6%	0.9%	0.9%	0.0%
All	48.1%	47.2%	1.8%	1.5%	1.4%	0.0%
Percent of CPU time due to each category...						
iexplore	20.7%	74.9%	0.1%	0.1%	4.2%	0.0%
realplay	72.6%	27.4%	0.0%	0.0%	0.0%	0.0%
explorer	16.0%	74.9%	0.4%	0.7%	8.0%	0.0%
outlook	31.6%	61.0%	2.9%	0.1%	4.4%	0.0%
vb6	31.0%	54.9%	12.3%	0.8%	1.0%	0.0%
winword	41.0%	48.4%	8.1%	1.1%	1.4%	0.0%
acrord32	31.6%	67.8%	0.1%	0.5%	0.0%	0.0%
powerpnt	26.8%	46.2%	13.3%	1.1%	12.4%	0.0%
All	28.9%	63.5%	3.0%	0.5%	4.0%	0.0%

Table B.50: For **user #7**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Application	Left down	Left up	Left double	Right down	Right up	Other
Percent of tasks of each category…						
netscape	61.0%	29.6%	7.4%	1.4%	0.6%	0.0%
grpwise	51.0%	33.5%	7.4%	4.1%	4.0%	0.0%
realplay	59.8%	38.3%	1.1%	0.8%	0.1%	0.0%
acrord32	74.0%	10.9%	14.1%	0.7%	0.3%	0.0%
ntvdm	40.4%	43.1%	4.8%	5.7%	5.9%	0.2%
explorer	47.3%	37.4%	10.0%	2.6%	2.6%	0.0%
remote	50.0%	50.0%	0.0%	0.0%	0.0%	0.0%
winword	53.5%	37.4%	4.2%	2.5%	2.5%	0.0%
acrobat	60.3%	30.5%	6.4%	2.4%	0.4%	0.0%
shstat	53.3%	40.0%	0.0%	6.7%	0.0%	0.0%
powerpnt	49.9%	44.4%	3.3%	1.2%	1.2%	0.0%
excel	56.3%	34.6%	4.8%	2.0%	2.1%	0.1%
jerusalm	44.4%	45.9%	1.5%	4.1%	4.1%	0.0%
notify	49.8%	45.1%	4.3%	0.4%	0.4%	0.0%
activeshare	33.3%	51.7%	6.8%	3.8%	4.3%	0.1%
iexplore	72.2%	21.4%	3.2%	1.6%	1.6%	0.0%
All	55.1%	33.0%	7.3%	2.5%	2.1%	0.0%
Percent of CPU time due to each category…						
netscape	38.9%	55.6%	2.3%	3.0%	0.2%	0.0%
grpwise	73.8%	17.9%	6.1%	0.5%	1.6%	0.0%
realplay	81.0%	18.9%	0.0%	0.0%	0.0%	0.0%
acrord32	79.8%	14.0%	6.1%	0.1%	0.0%	0.0%
ntvdm	53.6%	35.4%	5.8%	4.8%	0.3%	0.0%
explorer	67.5%	10.3%	13.7%	0.2%	8.4%	0.0%
remote	27.0%	73.0%	0.0%	0.0%	0.0%	0.0%
winword	67.6%	22.5%	5.0%	2.7%	2.2%	0.0%
acrobat	56.8%	37.6%	4.9%	0.7%	0.0%	0.0%
shstat	51.1%	39.7%	0.0%	9.2%	0.0%	0.0%
powerpnt	1.6%	98.1%	0.0%	0.3%	0.0%	0.0%
excel	67.7%	24.5%	5.7%	0.6%	1.6%	0.0%
jerusalm	27.5%	71.0%	1.4%	0.1%	0.0%	0.0%
notify	36.8%	57.3%	5.8%	0.0%	0.0%	0.0%
activeshare	4.2%	67.0%	27.5%	0.1%	1.3%	0.0%
iexplore	44.4%	49.2%	6.2%	0.0%	0.2%	0.0%
All	46.1%	47.0%	3.7%	2.3%	0.8%	0.0%

Table B.51: For **user #8**, what percentage of events and CPU time from mouse click events is due to the various categories of mouse click events. Event percentages are shown in normal type, while CPU time percentages are shown in parenthesized italics.

Key description	Key press		Key release		Key press or release	
Letter	33.39%	*27.14%*	15.86%	*15.92%*	17.53%	*11.21%*
Modifier(s) alone	14.72%	*8.39%*	10.00%	*2.94%*	4.72%	*5.45%*
Arrows	5.37%	*9.93%*	4.61%	*8.90%*	0.76%	*1.03%*
Spacebar	6.30%	*8.59%*	3.35%	*4.76%*	2.95%	*3.83%*
Backspace	3.53%	*3.25%*	2.27%	*2.48%*	1.26%	*0.77%*
Ctrl-letter	23.18%	*9.64%*	17.76%	*8.04%*	5.43%	*1.60%*
Punctuation	3.74%	*3.25%*	1.96%	*2.43%*	1.78%	*0.82%*
Number	1.64%	*1.12%*	0.74%	*0.77%*	0.89%	*0.35%*
Enter	2.46%	*10.30%*	1.23%	*5.30%*	1.23%	*5.01%*
Shift-letter	2.09%	*2.17%*	1.35%	*1.57%*	0.74%	*0.60%*
Del	0.23%	*2.68%*	0.15%	*2.08%*	0.08%	*0.59%*
Modified-arrow	0.23%	*0.56%*	0.19%	*0.46%*	0.04%	*0.10%*
Tab	0.96%	*4.68%*	0.20%	*0.60%*	0.76%	*4.08%*
Page up/down	0.80%	*2.27%*	0.56%	*1.99%*	0.24%	*0.27%*
Home/End	0.04%	*0.08%*	0.02%	*0.05%*	0.02%	*0.03%*
F-keys	0.13%	*3.12%*	0.07%	*1.81%*	0.06%	*1.30%*
Escape	0.21%	*0.14%*	0.10%	*0.09%*	0.10%	*0.05%*
Alt-letter	0.05%	*0.76%*	0.04%	*0.75%*	0.01%	*0.00%*
Ctrl-spacebar	0.23%	*0.04%*	0.11%	*0.03%*	0.11%	*0.01%*
Ctrl-F-keys	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Others	0.72%	*1.91%*	0.24%	*1.17%*	0.48%	*0.74%*

Table B.52: For **user #1**, what percentage of events and CPU time from key events are due to the various categories of key events.

Key description	Key press		Key release		Key press or release	
Letter	36.19%	*23.73%*	17.88%	*16.61%*	18.31%	*7.12%*
Modifier(s) alone	12.11%	*6.21%*	7.75%	*2.03%*	4.36%	*4.17%*
Arrows	18.11%	*20.09%*	11.74%	*18.36%*	6.37%	*1.73%*
Spacebar	6.56%	*4.29%*	3.36%	*3.03%*	3.20%	*1.26%*
Backspace	5.77%	*3.77%*	3.18%	*3.03%*	2.59%	*0.73%*
Ctrl-letter	2.59%	*4.89%*	1.56%	*4.53%*	1.04%	*0.36%*
Punctuation	3.51%	*4.02%*	1.81%	*2.97%*	1.70%	*1.05%*
Number	1.40%	*2.24%*	0.66%	*0.99%*	0.74%	*1.24%*
Enter	2.22%	*10.60%*	1.12%	*3.41%*	1.09%	*7.18%*
Shift-letter	2.12%	*1.20%*	1.22%	*1.01%*	0.91%	*0.19%*
Del	0.82%	*2.25%*	0.43%	*2.00%*	0.39%	*0.24%*
Modified-arrow	3.51%	*3.24%*	2.00%	*2.63%*	1.52%	*0.60%*
Tab	0.60%	*1.49%*	0.14%	*0.21%*	0.45%	*1.28%*
Page up/down	1.57%	*6.74%*	0.81%	*6.17%*	0.76%	*0.56%*
Home/End	1.51%	*1.25%*	0.75%	*1.00%*	0.76%	*0.25%*
F-keys	0.23%	*1.62%*	0.12%	*0.57%*	0.12%	*1.05%*
Escape	0.15%	*0.50%*	0.08%	*0.38%*	0.08%	*0.12%*
Alt-letter	0.04%	*0.07%*	0.03%	*0.06%*	0.01%	*0.01%*
Ctrl-spacebar	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Ctrl-F-keys	0.02%	*0.07%*	0.01%	*0.06%*	0.01%	*0.01%*
Others	0.96%	*1.74%*	0.45%	*1.50%*	0.51%	*0.24%*

Table B.53: For **user #2**, what percentage of events and CPU time from key events are due to the various categories of key events.

Key description	Key press		Key release		Key press or release	
Letter	11.52%	*12.13%*	5.75%	*7.56%*	5.77%	*4.57%*
Modifier(s) alone	59.09%	*32.91%*	57.20%	*31.30%*	1.89%	*1.61%*
Arrows	13.55%	*22.65%*	9.75%	*21.36%*	3.79%	*1.29%*
Spacebar	2.86%	*2.84%*	1.75%	*2.56%*	1.11%	*0.28%*
Backspace	0.83%	*0.76%*	0.45%	*0.48%*	0.38%	*0.28%*
Ctrl-letter	1.14%	*3.60%*	0.89%	*3.36%*	0.25%	*0.24%*
Punctuation	0.82%	*1.02%*	0.41%	*0.45%*	0.40%	*0.57%*
Number	2.48%	*2.75%*	1.24%	*1.87%*	1.24%	*0.88%*
Enter	1.43%	*10.75%*	0.65%	*6.50%*	0.78%	*4.24%*
Shift-letter	1.29%	*1.21%*	0.66%	*0.92%*	0.63%	*0.30%*
Del	0.52%	*1.74%*	0.27%	*1.56%*	0.25%	*0.18%*
Modified-arrow	2.17%	*1.27%*	1.48%	*1.15%*	0.68%	*0.11%*
Tab	0.31%	*1.00%*	0.15%	*0.76%*	0.15%	*0.24%*
Page up/down	1.06%	*3.46%*	0.83%	*3.24%*	0.24%	*0.22%*
Home/End	0.22%	*0.52%*	0.11%	*0.45%*	0.11%	*0.07%*
F-keys	0.08%	*0.34%*	0.04%	*0.25%*	0.04%	*0.08%*
Escape	0.10%	*0.61%*	0.04%	*0.45%*	0.05%	*0.16%*
Alt-letter	0.05%	*0.05%*	0.03%	*0.04%*	0.03%	*0.01%*
Others	0.50%	*0.41%*	0.29%	*0.35%*	0.21%	*0.05%*

Table B.54: For **user #3**, what percentage of events and CPU time from key events are due to the various categories of key events.

Key description	Key press		Key release		Key press or release	
Letter	44.71%	*41.39%*	23.46%	*24.77%*	21.26%	*16.63%*
Modifier(s) alone	10.61%	*4.22%*	8.51%	*2.31%*	2.10%	*1.91%*
Arrows	9.59%	*8.95%*	7.64%	*6.83%*	1.95%	*2.13%*
Spacebar	9.72%	*16.53%*	6.10%	*12.90%*	3.62%	*3.63%*
Backspace	7.40%	*9.02%*	4.77%	*6.26%*	2.63%	*2.76%*
Ctrl-letter	0.59%	*1.63%*	0.32%	*1.46%*	0.27%	*0.17%*
Punctuation	3.40%	*2.39%*	1.70%	*1.40%*	1.70%	*1.00%*
Number	4.21%	*1.18%*	3.62%	*0.88%*	0.59%	*0.30%*
Enter	4.14%	*6.39%*	2.20%	*4.20%*	1.94%	*2.19%*
Shift-letter	1.94%	*2.20%*	0.99%	*1.41%*	0.95%	*0.80%*
Del	0.13%	*0.24%*	0.08%	*0.23%*	0.05%	*0.01%*
Modified-arrow	0.86%	*1.02%*	0.66%	*0.73%*	0.20%	*0.29%*
Tab	1.25%	*0.88%*	0.57%	*0.52%*	0.68%	*0.36%*
Page up/down	0.14%	*0.35%*	0.10%	*0.28%*	0.04%	*0.08%*
Home/End	0.32%	*0.47%*	0.16%	*0.23%*	0.16%	*0.25%*
F-keys	0.06%	*0.99%*	0.03%	*0.51%*	0.03%	*0.48%*
Escape	0.24%	*0.19%*	0.12%	*0.14%*	0.12%	*0.04%*
Alt-letter	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Ctrl-spacebar	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Ctrl-F-keys	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Others	0.68%	*1.94%*	0.34%	*1.49%*	0.34%	*0.45%*

Table B.55: For **user #4**, what percentage of events and CPU time from key events are due to the various categories of key events.

Key description	Key press		Key release		Key press or release	
Letter	55.00%	*42.46%*	27.37%	*33.24%*	27.62%	*9.22%*
Modifier(s) alone	10.84%	*4.75%*	8.85%	*3.82%*	1.99%	*0.93%*
Arrows	4.74%	*5.78%*	3.32%	*4.43%*	1.43%	*1.34%*
Spacebar	10.84%	*7.49%*	5.47%	*5.85%*	5.37%	*1.64%*
Backspace	2.81%	*2.91%*	1.56%	*2.48%*	1.25%	*0.43%*
Ctrl-letter	0.01%	*0.02%*	0.00%	*0.02%*	0.00%	*0.00%*
Punctuation	4.08%	*4.34%*	2.30%	*3.71%*	1.78%	*0.64%*
Number	2.09%	*2.01%*	1.04%	*1.71%*	1.05%	*0.30%*
Enter	1.75%	*13.05%*	0.83%	*6.91%*	0.93%	*6.14%*
Shift-letter	2.81%	*2.37%*	1.53%	*1.97%*	1.28%	*0.40%*
Del	2.33%	*10.58%*	1.34%	*8.77%*	1.00%	*1.81%*
Modified-arrow	0.37%	*0.72%*	0.27%	*0.48%*	0.10%	*0.24%*
Tab	1.12%	*1.38%*	0.56%	*1.18%*	0.56%	*0.20%*
Page up/down	0.02%	*0.04%*	0.01%	*0.03%*	0.01%	*0.00%*
Home/End	0.31%	*0.22%*	0.16%	*0.16%*	0.16%	*0.05%*
F-keys	0.05%	*1.09%*	0.02%	*0.98%*	0.02%	*0.11%*
Escape	0.02%	*0.02%*	0.01%	*0.01%*	0.01%	*0.00%*
Alt-letter	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Ctrl-spacebar	0.01%	*0.05%*	0.00%	*0.05%*	0.00%	*0.00%*
Ctrl-F-keys	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Others	0.80%	*0.74%*	0.35%	*0.48%*	0.45%	*0.26%*

Table B.56: For **user #5**, what percentage of events and CPU time from key events are due to the various categories of key events.

Key description	Key press		Key release		Key press or release	
Letter	60.56%	*54.87%*	30.12%	*35.19%*	30.44%	*19.69%*
Modifier(s) alone	6.57%	*4.02%*	4.55%	*2.78%*	2.02%	*1.23%*
Arrows	4.49%	*4.08%*	3.18%	*3.33%*	1.31%	*0.75%*
Spacebar	13.06%	*11.82%*	6.63%	*7.86%*	6.43%	*3.95%*
Backspace	2.95%	*3.32%*	1.59%	*2.42%*	1.36%	*0.89%*
Ctrl-letter	0.00%	*0.02%*	0.00%	*0.02%*	0.00%	*0.00%*
Punctuation	2.68%	*2.74%*	1.43%	*1.86%*	1.25%	*0.88%*
Number	2.54%	*3.03%*	1.29%	*1.92%*	1.25%	*1.11%*
Enter	1.34%	*5.93%*	0.66%	*3.04%*	0.68%	*2.90%*
Shift-letter	3.42%	*3.27%*	1.95%	*2.44%*	1.47%	*0.83%*
Del	1.64%	*2.89%*	1.09%	*2.36%*	0.55%	*0.53%*
Tab	0.49%	*2.32%*	0.24%	*1.25%*	0.24%	*1.07%*
Page up/down	0.08%	*0.22%*	0.05%	*0.19%*	0.04%	*0.03%*
Home/End	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
F-keys	0.03%	*0.42%*	0.01%	*0.15%*	0.01%	*0.27%*
Escape	0.01%	*0.01%*	0.00%	*0.01%*	0.00%	*0.01%*
Alt-letter	0.00%	*0.01%*	0.00%	*0.01%*	0.00%	*0.00%*
Others	0.14%	*1.04%*	0.07%	*0.41%*	0.07%	*0.63%*

Table B.57: For **user #6**, what percentage of events and CPU time from key events are due to the various categories of key events.

Key description	Key press		Key release		Key press or release	
Letter	22.09%	*5.60%*	5.33%	*4.02%*	16.77%	*1.58%*
Modifier(s) alone	12.35%	*7.91%*	9.19%	*1.13%*	3.16%	*6.78%*
Arrows	10.43%	*7.12%*	5.97%	*6.27%*	4.46%	*0.85%*
Spacebar	1.96%	*0.50%*	0.58%	*0.32%*	1.39%	*0.19%*
Eastern-language-char	29.30%	*40.71%*	12.97%	*6.99%*	16.33%	*33.72%*
Backspace	3.44%	*2.07%*	1.45%	*0.98%*	1.99%	*1.09%*
Ctrl-letter	0.92%	*1.63%*	0.66%	*1.58%*	0.26%	*0.04%*
Punctuation	2.03%	*0.53%*	0.95%	*0.42%*	1.08%	*0.11%*
Number	1.96%	*0.43%*	0.96%	*0.34%*	1.01%	*0.09%*
Enter	1.53%	*8.86%*	0.75%	*4.09%*	0.78%	*4.77%*
Shift-letter	1.15%	*0.22%*	0.57%	*0.17%*	0.58%	*0.04%*
Del	2.04%	*7.81%*	1.02%	*3.64%*	1.02%	*4.17%*
Modified-arrow	2.89%	*0.74%*	1.56%	*0.64%*	1.33%	*0.10%*
Tab	0.91%	*1.86%*	0.24%	*0.14%*	0.67%	*1.72%*
Page up/down	3.34%	*8.36%*	1.75%	*7.20%*	1.59%	*1.16%*
Home/End	1.65%	*0.82%*	0.81%	*0.62%*	0.84%	*0.20%*
F-keys	0.29%	*3.35%*	0.15%	*2.85%*	0.14%	*0.50%*
Escape	0.11%	*0.12%*	0.05%	*0.09%*	0.06%	*0.03%*
Modified Eastern-char	0.40%	*0.40%*	0.18%	*0.10%*	0.22%	*0.29%*
Alt-letter	0.05%	*0.14%*	0.05%	*0.14%*	0.01%	*0.00%*
Others	1.13%	*0.81%*	0.55%	*0.69%*	0.59%	*0.12%*

Table B.58: For **user #7**, what percentage of events and CPU time from key events are due to the various categories of key events.

Key description	Key press		Key release		Key press or release	
Letter	33.33%	*27.83%*	16.53%	*15.88%*	16.79%	*11.95%*
Modifier(s) alone	37.06%	*11.04%*	35.53%	*9.63%*	1.53%	*1.41%*
Arrows	3.48%	*10.11%*	2.54%	*8.27%*	0.93%	*1.85%*
Spacebar	6.31%	*6.90%*	3.46%	*4.64%*	2.85%	*2.26%*
Backspace	2.45%	*2.05%*	1.30%	*1.43%*	1.15%	*0.63%*
Ctrl-letter	0.08%	*0.06%*	0.04%	*0.06%*	0.04%	*0.00%*
Punctuation	2.11%	*4.40%*	1.10%	*1.82%*	1.01%	*2.58%*
Number	5.03%	*5.20%*	2.51%	*3.01%*	2.52%	*2.19%*
Enter	3.32%	*10.07%*	1.86%	*8.22%*	1.46%	*1.85%*
Shift-letter	2.27%	*2.29%*	1.31%	*1.54%*	0.96%	*0.75%*
Del	2.37%	*2.63%*	1.66%	*2.35%*	0.71%	*0.28%*
Modified-arrow	0.01%	*0.00%*	0.01%	*0.00%*	0.00%	*0.00%*
Tab	1.70%	*8.86%*	0.98%	*5.23%*	0.72%	*3.63%*
Home/End	0.01%	*0.01%*	0.00%	*0.01%*	0.00%	*0.00%*
F-keys	0.19%	*7.01%*	0.10%	*1.17%*	0.09%	*5.84%*
Escape	0.02%	*0.02%*	0.01%	*0.01%*	0.01%	*0.00%*
Alt-letter	0.00%	*0.00%*	0.00%	*0.00%*	0.00%	*0.00%*
Others	0.25%	*1.52%*	0.13%	*0.65%*	0.12%	*0.88%*

Table B.59: For **user #8**, what percentage of events and CPU time from key events are due to the various categories of key events.

B.9 Significance of user interface event category differences

In this section, we show tables indicating to what extent different user interface event categories have significantly different processing requirements.

B.9.1 Key categories

In this subsection, we show to what extent different keystroke categories exhibit significantly different CPU requirements. For each application and user, we show only the fifteen most frequent key categories, unless the top fifteen include some that occur less often than 30 times in the trace, in which case we show only those that occur at least 30 times. Tables B.60 and B.61, show results for key presses. Tables B.62 and B.63 show results for key releases.

In addition, we show to what extent key presses and releases are significantly different for the same application and user. Tables B.64 and B.65 show the significance of differences between key presses and releases of the same key category for the top 25 applications of each user except user #1. Results for user #1 were included in Section V.4.9.1.

B.9.2 Mouse click categories

In this subsection, we show to what extent different mouse click categories exhibit significantly different CPU requirements. Tables B.66 and B.67 show results for users #4–9. Results for users #1–3 were included in Section V.4.9.2.

B.10 Significance of application differences

In this section, we show tables indicating the extent to which different applications have significantly different task length means for the same event type or category. Tables B.68 and B.69 show differences between applications' mouse movement task lengths for various users for whom this data was not shown in Section V.4.10. Tables B.70 and B.71 show differences between applications' left mouse down task lengths for various users for whom

User #2, powerpoint

	Modifiers	Home/End	Arrow	Escape	Number	Del	Shift-letter	Letter	Backspace	Ctrl-letter	Punctuation	Tab	Spacebar	Enter	Page up/down
Modifiers		D	D	D	D	D	D	D	D	D	D	D	D	D	D
Home/End	D		D	D	D	D	D	D	D	D	D	D	D	D	D
Arrow	D	D		d	D	D	D	D	D	D	D	D	D	D	D
Escape	D	D	d		D	D	D	D	D	D	D	D	D	D	D
Number	D	D	D	D			D	D	D	D	D	D	D	D	D
Del	D	D	D	D			·	D	D	d	D	D	D	D	D
Shift-letter	D	D	D	D	D	·		d	D	·	D	D	D	D	D
Letter	D	D	D	D	D	D	d				D	D	D	D	D
Backspace	D	D	D	D	D	D	D				D	D	D	D	D
Ctrl-letter	D	D	D	D	D	d	·					D	D	D	D
Punctuation	D	D	D	D	D	D	D	D	D			D	D	D	D
Tab	D	D	D	D	D	D	D	D	D	D	D			D	D
Spacebar	D	D	D	D	D	D	D	D	D	D	D			D	D
Enter	D	D	D	D	D	D	D	D	D	D	D	D	D		D
Page up/down	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

User #3, outlook

	Modifiers	Arrow	Number	Shift-letter	Letter	Punctuation	Backspace	Enter	Spacebar
Modifiers		D	D	D	D	D	D	D	D
Arrow	D		D	D	D	D	D	D	D
Number	D	D		·	d	D	D	D	D
Shift-letter	D	D	·				D	D	D
Letter	D	D	d				D	D	D
Punctuation	D	D	D				d	D	D
Backspace	D	D	D	D	D	d		D	D
Enter	D	D	D	D	D	D	D		D
Spacebar	D	D	D	D	D	D	D	D	

User #5, iexplore

	Modifiers	Home/End	Arrow	Spacebar	Del	Shift-letter	Punctuation	Letter	Backspace	Number	Tab	Enter
Modifiers		D	D	D	D	D	D	D	D	D	D	D
Home/End	D			d	D	D	D	D	D	D	D	D
Arrow	D				d	D	D	D	D	D	D	D
Spacebar	D	d			D	D	D	D	D	D	D	D
Del	D	D	d	D				D	D	D	D	D
Shift-letter	D	D	D	D				D	D	D	D	D
Punctuation	D	D	D	D					d	D	D	D
Letter	D	D	D	D	D	D			·	D	D	D
Backspace	D	D	D	D	D	D	d	·		D	D	D
Number	D	D	D	D	D	D	D	D	D		·	D
Tab	D	D	D	D	D	D	D	D	D	·		D
Enter	D	D	D	D	D	D	D	D	D	D	D	

Table B.60: This table shows the significance of differences of task length means for key **presses** for various applications and users. A 'D' indicates the row and column key categories are different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. The rows (and columns) are in increasing order of mean task length.

User #6, eudora

	Number	Modifiers	Arrow	Punctuation	Shift-letter	Letter	Spacebar	Backspace	Del	Tab	Enter
Number		D	D	D	D	D	D	D	D	D	D
Modifiers	D		D	D	D	D	D	D	D	D	D
Arrow	D	D		·	D	D	D	D	D	D	D
Punctuation	D	D	·		D	D	D	D	D	D	D
Shift-letter	D	D	D	D		D	D	D	D	D	D
Letter	D	D	D	D	D		D	D	D	D	D
Spacebar	D	D	D	D	D	D			·	d	D
Backspace	D	D	D	D	D	D					D
Del	D	D	D	D	D	D	·				D
Tab	D	D	D	D	D	D	d				D
Enter	D	D	D	D	D	D	D	D	D	D	

User #7, vb6

	Modifiers	Shift-letter	Home/End	Arrow	Number	Tab	Letter	Spacebar	Eastern char	Del	Backspace	Punctuation	Enter	Ctrl-letter	F-key
Modifiers		D	D	D	D	D	D	D	D	D	D	D	D	D	D
Shift-letter	D			D		D	D	D	D	D	D	D	D	D	D
Home/End	D					·	d	D	D	D	d	D	D	D	D
Arrow	D	D				D	D	D	D	D	d	D	D	D	D
Number	D								d	D	d	D	D	D	D
Tab	D	D	·	D				·	D	D	d	D	D	D	D
Letter	D	D	d	D				·	D	D	d	D	D	D	D
Spacebar	D	D	D	D		·	·		·	D		D	D	D	D
Eastern char	D	D	D	D	d	D	D	·		·		D	D	D	D
Del	D	D	D	D	D	D	D	D	·				D	D	D
Backspace	D	D	d	d	d	d	d						D	D	D
Punctuation	D	D	D	D	D	D	D	D	D				D	D	D
Enter	D	D	D	D	D	D	D	D	D	D	D	D		·	D
Ctrl-letter	D	D	D	D	D	D	D	D	D	D	D	D	·		D
F-key	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

User #8, excel

	Modifiers	Arrow	Backspace	Letter	Del	Punctuation	Spacebar	Tab	Shift-letter	Number
Modifiers		D	D	D	D	D	D	D	D	D
Arrow	D		D	D	D	D	D	D	D	D
Backspace	D	D		D	D	D	D	d	D	D
Letter	D	D	D			D	D	·	D	D
Del	D	D	D			D	D	·	D	D
Punctuation	D	D	D	D	D				D	D
Spacebar	D	D	D	D	D				D	D
Tab	D	D	d	·	·					D
Shift-letter	D	D	D	D	D	D	D			D
Number	D	D	D	D	D	D	D	D	D	

Table B.61: This table shows the significance of differences of task length means for key **presses** for various applications and users. A ‘D’ indicates the row and column key categories are different with 99% confidence, a ‘d’ indicates 95% significance, and a ‘.’ indicates 90% significance. The rows (and columns) are in increasing order of mean task length.

User #2, powerpnt

	Home/End	Number	Shift-letter	Backspace	Ctrl-letter	Arrow	Letter	Punctuation	Del	Spacebar	Modifiers	Enter	Escape	Tab	Page up/down
Home/End		D	D	D	D	D	D	D	D	D	D	D	D	D	D
Number	D			·			D	d	D	D	D	D	D	D	D
Shift-letter	D			·			D	d	D	D	D	D	D	D	D
Backspace	D	·	·				d		D	d	D	D	D	D	D
Ctrl-letter	D								D	d	D	D	D	D	D
Arrow	D								d	d	D	d	D	D	D
Letter	D	D	D	d					D	d	D	D	D	D	D
Punctuation	D	d	d						D	d	D	D	D	D	D
Del	D	D	D	D	D	d	D	D			·		D	D	D
Spacebar	D	D	D	d	d	d	d	d					D	D	D
Modifiers	D	D	D	D	D	D	D	D	·				D	D	D
Enter	D	D	D	D	D	d	D	D					D	D	D
Escape	D	D	D	D	D	D	D	D	D	D	D	D		D	D
Tab	D	D	D	D	D	D	D	D	D	D	D	D	D		D
Page up/down	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

User #3, outlook

	Arrow	Spacebar	Letter	Number	Punctuation	Shift-letter	Modifiers	Enter
Arrow			·		D	D	·	d
Spacebar					d	D	·	·
Letter	·				d	D	·	·
Number								·
Punctuation	D	d	d					·
Shift-letter	D	D	D					·
Modifiers	·	·	·					
Enter	d	·	·	·	·	·		

User #4, ssh

	F-key	Escape	Number	Backspace	Punctuation	Arrow	Shift-letter	Ctrl-letter	Letter	Spacebar	Tab	Modifiers	Enter
F-key						·			D	d	·	d	D
Escape									D	d	·	d	D
Number									D	d	·	d	D
Backspace									D	d	·	d	D
Punctuation									D	d	·	d	D
Arrow	·								D	d	·	d	D
Shift-letter										·		·	D
Ctrl-letter										·		·	D
Letter	D	D	D	D	D	D				·		·	D
Spacebar	d	d	d	d	d	d	·	·	·				·
Tab	·	·	·	·	·	·							
Modifiers	d	d	d	d	d	d	·	·	·				
Enter	D	D	D	D	D	D	D	D	D	·			

Table B.62: This table shows the significance of differences of task length means for key **releases** for various applications and users. A 'D' indicates the row and column key categories are different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. The rows (and columns) are in increasing order of mean task length.

User #6, eudora

	Number	Punctuation	Enter	Tab	Modifiers	Shift-letter	Arrow	Letter	Spacebar	Backspace	Del
Number		D	D	D	D	D	D	D	D	D	D
Punctuation	D		D	D	D	D	D	D	D	D	D
Enter	D	D		·	D	D	D	D	D	D	D
Tab	D	D	·					·	d	D	D
Modifiers	D	D	D					D	D	D	D
Shift-letter	D	D	D					d	D	D	D
Arrow	D	D	D					d	D	D	D
Letter	D	D	D	·	D	d	d		d	D	D
Spacebar	D	D	D	d	D	D	D	d		D	D
Backspace	D	D	D	D	D	D	D	D	D		
Del	D	D	D	D	D	D	D	D	D		

User #7, vb6

	Shift-letter	Backspace	Punctuation	Letter	Number	Ctrl-letter	Del	Home/End	Spacebar	Enter	Arrow	Modifiers	Eastern char	F-key	Tab
Shift-letter		d	d	D	·		D	d	d	D	d	D	D	D	D
Backspace	d			·			d	·	d	D	d	D	D	D	D
Punctuation	d									D	d	D	D	D	D
Letter	D	·								D	d	D	D	D	D
Number	·									D	·	D	D	D	D
Ctrl-letter										·		D	D	D	D
Del	D	d								d	·	D	D	D	D
Home/End	d	·								d		D	D	D	D
Spacebar	d	d								d		D	D	D	D
Enter	D	D	D	D	D	·	d	d	d			D	D	D	D
Arrow	d	d	d	d	·		·					·	D	D	D
Modifiers	D	D	D	D	D	D	D	D	D	D	·		D	D	D
Eastern char	D	D	D	D	D	D	D	D	D	D	D	D		d	D
F-key	D	D	D	D	D	D	D	D	D	D	D	D	d		d
Tab	D	D	D	D	D	D	D	D	D	D	D	D	D	d	

User #8, excel

	Backspace	Modifiers	Del	Spacebar	Letter	Tab	Punctuation	Shift-letter	Number	Arrow
Backspace				·	d		d	D	D	d
Modifiers								·	D	
Del										
Spacebar	·								d	
Letter	d								d	
Tab										
Punctuation	d									
Shift-letter	D	·								
Number	D	D		d	d					
Arrow	d									

Table B.63: This table shows the significance of differences of task length means for key **releases** for various applications and users. A 'D' indicates the row and column key categories are different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. The rows (and columns) are in increasing order of mean task length.

User #2

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
iexplore	D	D	D	D	D	D	D	D	D	D	D	D	D
starcraft	D	.	*	D	D	*	D	d		*	*	*	*
winword	D	D	D	D	D	D	D	d	D	D	d	D	*
psp	*	D	*	*	*	D	*	D	*	D	D	*	*
java	D	.	*	*	D	*	*	*	*	*	*	*	*
devenv	D	D	D	D	D	D	D	D	D	D		D	D
explorer	D	D	D	D	D	.	D	.	d		*	D	D
msimn	D	D	D	D	D	D	D	D	D	D	*	*	*
powerpnt	D	D	D	D	D	D	D	D	D	D		D	*
txtpad32	D	D	D	D	D	D	D	D	D	D	D	D	D
ssh	D	D	D	D	D	D	D	D	D	D	D	*	*
appletviewer	*	d	*	*	*	*	*	*	*	*	*	*	*
textpad	D	D	D	D	D	D	D	D	D	D	D	D	*

User #3

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
aim	D	d	*	D	D	*	*	*	*	*	*	*	*
netscape	D	D	D	d	D	D		D		*	d	D	*
ntvdm	D	*	*	*	*	*	*	*	*	*	*	*	*
faxmain	D	D		*	*	*	D	D	*	*	*	D	*
psp	*	D	*	*	*	*	*	*	*	*	*	*	*
explorer	D		D	d	*	*	D	D	*	D	*	*	*
sbtw		*	D	D	*	*	*	D	D	*	D	D	*
bartend	*	*	d	*	D	*	*	D	*	*	*	*	*
outlook	D		D	D	*	*	D	D	D	*	*	*	*
blackice	*	*	*	*	*	*	*	D	*	*	*	*	*
napster	D		*	*	*	*	*	*	*	*	*	*	*
iexplore	D	*	*	*	*	*	*	*	*	*	*	*	*
excel	D		D	D	*	D	D	D		*	*	D	*
winword	D		D	D	D	*	D	D	D	D	*	D	*
msoffice		*	*	*	*	*	*	*	*	*	*	*	*
act		*	*	*	*	*	*	*	*	*	*	*	*

User #4

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
netscape	D	D	D	D	D	D	D	D	D	*	D	D	*
realplay		*		*	*	*	*	*	*	*	*	*	*
outlook	D	D	D	D	D	D	D	D	D	*		D	*
ssh	D	.	D		D		D	D	d	*		*	D
explorer	D	D	D	*	D	*	D	D	.	*	*	*	D
exceed	D	D	D	D	D	D	D	D	D	D	D		D
dreamweaver	D	d	D	D	D	D			D	*	*	*	*
ttermpro	D		D	D	D	*	D	D		*	*	*	*
iexplore	D		D	*	D	*	D	*	D	*	*	*	*
eudora	d	D		*	.	*		*	*	*	*	*	*
icq	D	D	*	D	D	*	D	*	*	*	*	*	*

Table B.64: This table shows the significance of differences between task length means for key presses and releases for users #2–4. A ‘D’ indicates key presses and releases are significantly different with 99% confidence, a ‘d’ indicates 95% significance, a ‘.’ indicates 90% significance, and a ‘*’ indicates insufficient data.

User #5

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
iexplore	D	d	D	D	D	*	D	D	D	D	D	*	*
explorer	D	.	D	D	D	*	d		d	D	*	*	*
outlook	D		D	D	D	*	D	D	D	D	D	*	*
winword	D	D	D	D	D	*	D	D	D	D	D	*	*

User #6

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
grpwise	D	d	D	D	D	*	D	D	D	D	d	*	*
ntvdm			*	D		*			*	D		D	*
eudora	D	D	D	D	D	*	D	D	D	D	D	*	*
netscape	D	D		.		*	D	D	*	*	D	*	*
planner	D			D	D	*	D		.	*	D	*	*
winword	D		D	D	D	*	D	D	D	D	D	*	*
realjbox	d	*	*	*	*	*	*	*	*	*	*	*	*
iexplore	D	D	D	D	D	*	D	D	*	*	*	*	*
aim	D	D	D	D	D	*	D	D	D	*	*	*	*
acrobat	D	D	*	D	D	*	*	D	*	D	*	*	*
explorer	D	D	*	*	*	*	D	D	*	*	*	*	*
powerpnt	*	*	*	*	*	*	*	*	*	*	*		*
excel	D	D	D	D	D	*	D	D	D	*	*	*	*
ins0432	.	*	*	*	*	*	*	*	*	*	*	*	*

User #7

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
iexplore	D	D	D	D	D	*	D	D	D	D	D	D	d
explorer	D	.		*	*	*	D	D	*	*	*	*	*
outlook	D	D	D	D	D	*	D	D	D	D	D	*	*
vb6	D	d	D	D	D	D	D	D	D	D	D	*	D
winword	D	D	D	D	D	.	D	D	d	D	*	D	*
acrord32	*	D	D	*	*	*	*	*	*	*	*	D	*
powerpnt	D	d		D	D	d	*	D	*	D	*	D	*

User #8

	Letter	Modifiers	Arrow	Spacebar	Backspace	Ctrl-letter	Punctuation	Number	Enter	Del	Tab	Page up/down	F-key
netscape	D	D	D	D	D	*	D	D	D		D	*	*
grpwise	D	D		D	D	*	d	.	D	.	*	*	*
acrord32	d		*	D	*	*	*	*	*	*	*	*	*
ntvdm		D	*	*		*	D		*	*	D	*	*
explorer	D		*	*	D	*	*	D	*	D	*	*	*
winword	D		D	D	D	*	D	D	D	D	*	*	*
powerpnt	*	*	D	*	*	*	*	*	*	*	*	*	*
excel	D	D	D	D	D	*	D	D	*	D	D	*	*
jerusalm	*	D	D	*	*	*	*	*	*	*	*	*	*
activeshare	D		*	D	*	*	*	*	*	*	*	*	*

Table B.65: This table shows the significance of differences between task length means for key presses and releases for users #5–8. A 'D' indicates key presses and releases are significantly different with 99% confidence, a 'd' indicates 95% significance, a '.' indicates 90% significance, and a '*' indicates insufficient data.

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
netscape	D	D	D	D	D	d
realplay	D	*	*	*	*	*
outlook	D	D		D	D	D
ssh	D	*	*	*	*	*
explorer	D	D	D	D	D	D
exceed	D	*	*	*	*	*
vern	D	*	*	*	*	*
acrord32	D	*	*	*	*	*
dreamweaver	D	*	*	D	D	D
winamp		*	*	*	*	*
iexplore			d	*	*	*
eudora	.		d	*	*	*
icq		*	*	*	*	*

Table B.66: This table shows the significance of differences between task length means for different mouse click events for **user #4**. A 'D' indicates key presses and releases are significantly different with 99% confidence, a 'd' indicates 95% significance, a '.' indicates 90% significance, and a '*' indicates insufficient data.

this data was not shown in Section V.4.10. Finally, Table B.72 shows these results for modifier key presses instead of left mouse clicks.

B.11 Distribution of task work requirements

In this section, we show graphs of task work requirements for various sets of tasks triggered by user interface events in the workloads.

B.11.1 Mouse click categories

Here, we show graphs illustrating the differences between different categories of mouse clicks, focusing on the parts of these graphs above the 90th percentile. Figure B.1 shows these graphs for all users running explorer. Figure B.2 shows these graphs for various other applications and users.

B.11.2 Key press categories

Here, we show graphs illustrating the differences between different categories of keystrokes. Figures B.3 and B.4 show the effect of differences in key category for various applications and users. Figure B.5 shows this effect for the same applications as in Section V.4.11.5, focusing on the parts above the 90th percentile.

User #5

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
iexplore		·		D	d	d
explorer	D	D	D	D	D	D
outlook	D	D		D	D	D
winword	D	D		*	*	*

User #6

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
grpwise	D	D	D	D	D	D
ntvdm		*	*	*	*	*
eudora	D	D	D	D	D	D
netscape	D	D	D	*	*	*
hotsync	D	*	*	*	*	*
planner	D	D		d	d	D
winword	D	D		d	D	
realjbox		*	*	D		D
iexplore	D	D	D	*	*	*
aim	D	D	D	*	*	*
acrobat	D	D		*	*	*
acrord32	D	D		*	*	*
explorer	D	D	D	D		
powerpnt	D		d	*	*	*
notify	D	*	*	*	*	*
excel	D	·	D	D	D	d

User #7

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
iexplore	D	D	D	D	D	D
explorer	D		D	D	d	
outlook	D	d		D	D	D
vb6		D	D		*	*
winword	D	d	D		D	
acrord32	D	*	*	*	*	*
powerpnt	D	*	*	D	D	D

User #8

	Left down vs. left up	Left down vs. left double	Left up vs. left double	Left down vs. right down	Left up vs. right up	Right down vs. right up
netscape		d		D	D	D
grpwise	D	D	D	D		D
realplay	D	*	*	*	*	*
acrord32	D	D	D	*	*	*
ntvdm	D	D	D	D	D	D
explorer	D	D	D	D	D	D
winword	D	·	D		D	
acrobat	D		.	*	*	*
powerpnt	D	*	*	*	*	*
excel	D	d	D	D		
jerusalm		*	*	*	*	*
notify		*	*	*	*	*
activeshare	D	D	D	D		D

Table B.67: This table shows the significance of differences between task length means for different mouse click events for users #5–8. A 'D' indicates key presses and releases are significantly different with 99% confidence, a 'd' indicates 95% significance, a '.' indicates 90% significance, and a '*' indicates insufficient data.

User #3

	napster	aim	blackice	explorer	msoffice	excel	ntvdm	faxmain	winword	outlook	netscape	iexplore	sbtw	bartend	psp
napster			D	D	D	D	D	D	D	D	D	D	D	D	D
aim			D	D	D	D	D	D	D	D	D	D	D	D	D
blackice	D	D		D	D	D	D	D	D	D	D	D	D	D	D
explorer	D	D	D		D	D	D	D	D	D	D	D	D	D	D
msoffice	D	D	D	D		D	d	D	D	D	D	D	D	D	D
excel	D	D	D	D	D			D	D	D	D	D	D	D	D
ntvdm	D	D	D	D	d			D	D	D	D	D	D	D	D
faxmain	D	D	D	D	D	D	D		D	D	D	D	D	D	D
winword	D	D	D	D	D	D	D	D		D	D	D	D	D	D
outlook	D	D	D	D	D	D	D	D	D		D	D	D	D	D
netscape	D	D	D	D	D	D	D	D	D	D		·	D	D	D
iexplore	D	D	D	D	D	D	D	D	D	D	·		D	D	D
sbtw	D	D	D	D	D	D	D	D	D	D	D	D		D	D
bartend	D	D	D	D	D	D	D	D	D	D	D	D	D		D
psp	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

User #4

	shstat	vern	ttermpro	exceed	explorer	icq	winamp	ssh	netscape	outlook	realplay	acrord32	iexplore	dreamweaver	eudora
shstat					D	D	D	D	D	D	D	D	D	D	D
vern				D	D	D	D	D	D	D	D	D	D	D	D
ttermpro				·	D	D	D	D	D	D	D	D	D	D	D
exceed		D	·		D	D	D	D	D	D	D	D	D	D	D
explorer	D	D	D	D		D	D	D	D	D	D	D	D	D	D
icq	D	D	D	D	D			d	D	D	D	D	D	D	D
winamp	D	D	D	D	D			D	D	D	D	D	D	D	D
ssh	D	D	D	D	D	d	D		D	D	D	D	D	D	D
netscape	D	D	D	D	D	D	D	D		D	D	D	D	D	D
outlook	D	D	D	D	D	D	D	D	D		·	D	D	D	D
realplay	D	D	D	D	D	D	D	D	D	·		D	D	D	D
acrord32	D	D	D	D	D	D	D	D	D	D	D		D	D	D
iexplore	D	D	D	D	D	D	D	D	D	D	D	D		D	D
dreamweaver	D	D	D	D	D	D	D	D	D	D	D	D	D		D
eudora	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

Table B.68: This table shows the significance of differences between mouse movement task length means for different applications for users #3–4. A 'D' indicates mouse movements of the two applications are significantly different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. Results are reported only for applications with at least 30 mouse movement events. Applications are presented in increasing order of mean mouse movement task length.

User #5

	savenow	acrord32	explorer	msiexec	outlook	iexplore	winword	setup	realplay	rapigator
savenow					D	D	D	D	d	D
acrord32					D	D	D	D	·	D
explorer					D	D	D	D	·	D
msiexec						d	D	D	·	D
outlook	D	D	D			D	D	D	·	D
iexplore	D	D	D	d	D		D	d		D
winword	D	D	D	D	D	D				D
setup	D	D	D	D	D	d				d
realplay	d	·	·	·	·					
rapigator	D	D	D	D	D	D	D	d		

User #7

	realplay	vb6	explorer	powerpnt	outlook	iexplore	winword	acrord32
realplay		D	D	D	D	D	D	D
vb6	D			D	D	D	D	D
explorer	D			D	D	D	D	D
powerpnt	D	D	D		D	D	D	D
outlook	D	D	D	D		D	D	D
iexplore	D	D	D	D	D		D	D
winword	D	D	D	D	D	D		
acrord32	D	D	D	D	D	D		

User #8

	shstat	realplay	ntvdm	notify	explorer	acrord32	iexplore	powerpnt	excel	netscape	grpwise	acrobat	winword	activeshare	jerusalm
shstat			·	D	D	D	D	D	D	D	D	D	D	D	D
realplay			D	D	D	D	D	D	D	D	D	D	D	D	D
ntvdm	·	D		D	D	D	D	D	D	D	D	D	D	D	D
notify	D	D	D		D	D	D	D	D	D	D	D	D	D	D
explorer	D	D	D	D		D	D	D	D	D	D	D	D	D	D
acrord32	D	D	D	D	D		d	D	D	D	D	D	D	D	D
iexplore	D	D	D	D	D	d				d	D	D	D	D	D
powerpnt	D	D	D	D	D	D			·	D	D	D	D	D	D
excel	D	D	D	D	D	D		·		d	D	D	D	D	D
netscape	D	D	D	D	D	D	d	D	d		D	D	D	D	D
grpwise	D	D	D	D	D	D	D	D	D	D		D	D	D	D
acrobat	D	D	D	D	D	D	D	D	D	D	D		D	D	D
winword	D	D	D	D	D	D	D	D	D	D	D	D		D	D
activeshare	D	D	D	D	D	D	D	D	D	D	D	D	D		D
jerusalm	D	D	D	D	D	D	D	D	D	D	D	D	D	D	

Table B.69: This table shows the significance of differences between mouse movement task length means for different applications for users #5, 7, and 8. A 'D' indicates mouse movements of the two applications are significantly different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. Results are reported only for applications with at least 30 mouse movement events. Applications are presented in increasing order of mean mouse movement task length.

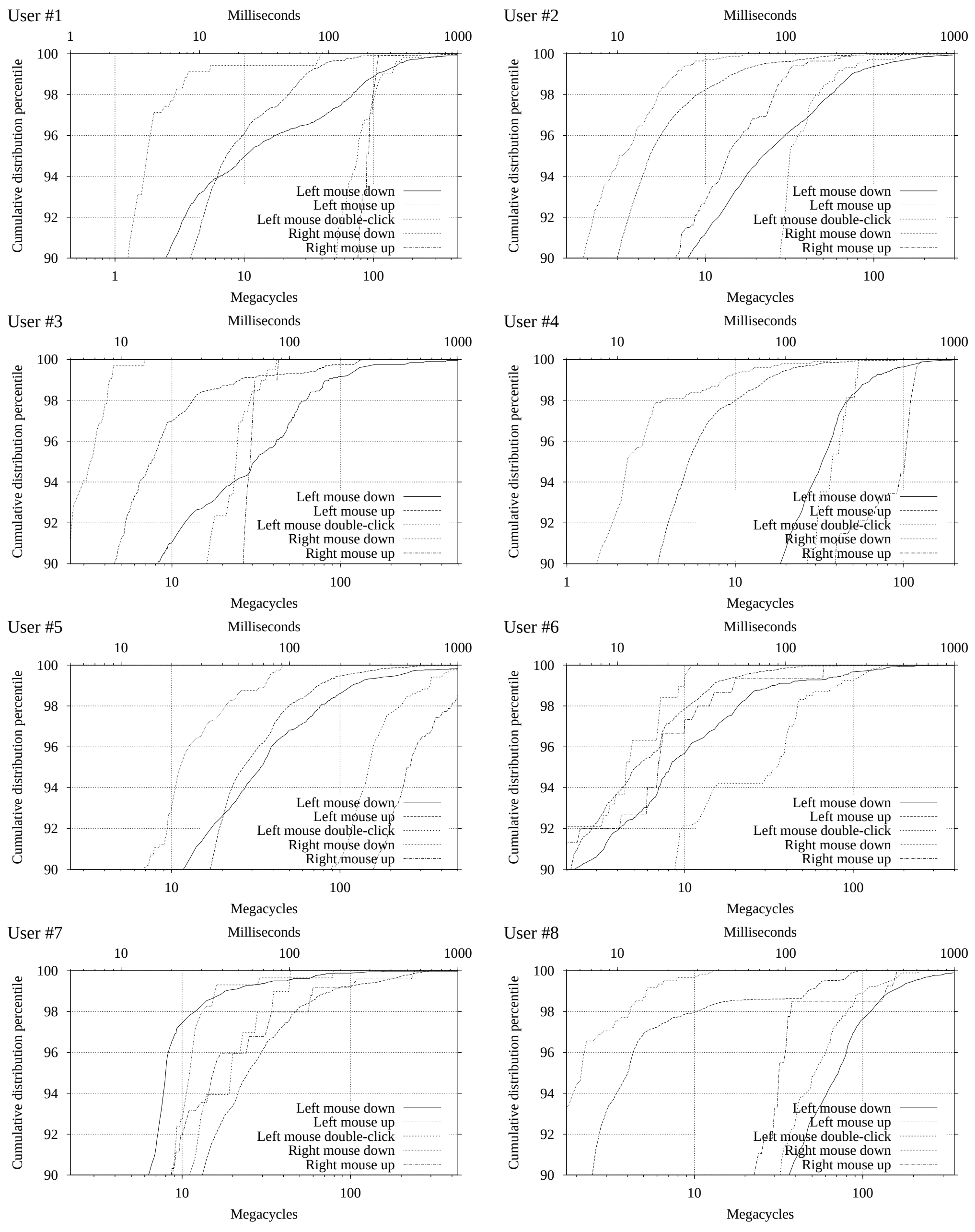

Figure B.1: For each user, the cumulative distribution function of CPU time required by explorer for mouse clicks of different categories, but only above the 90th percentile

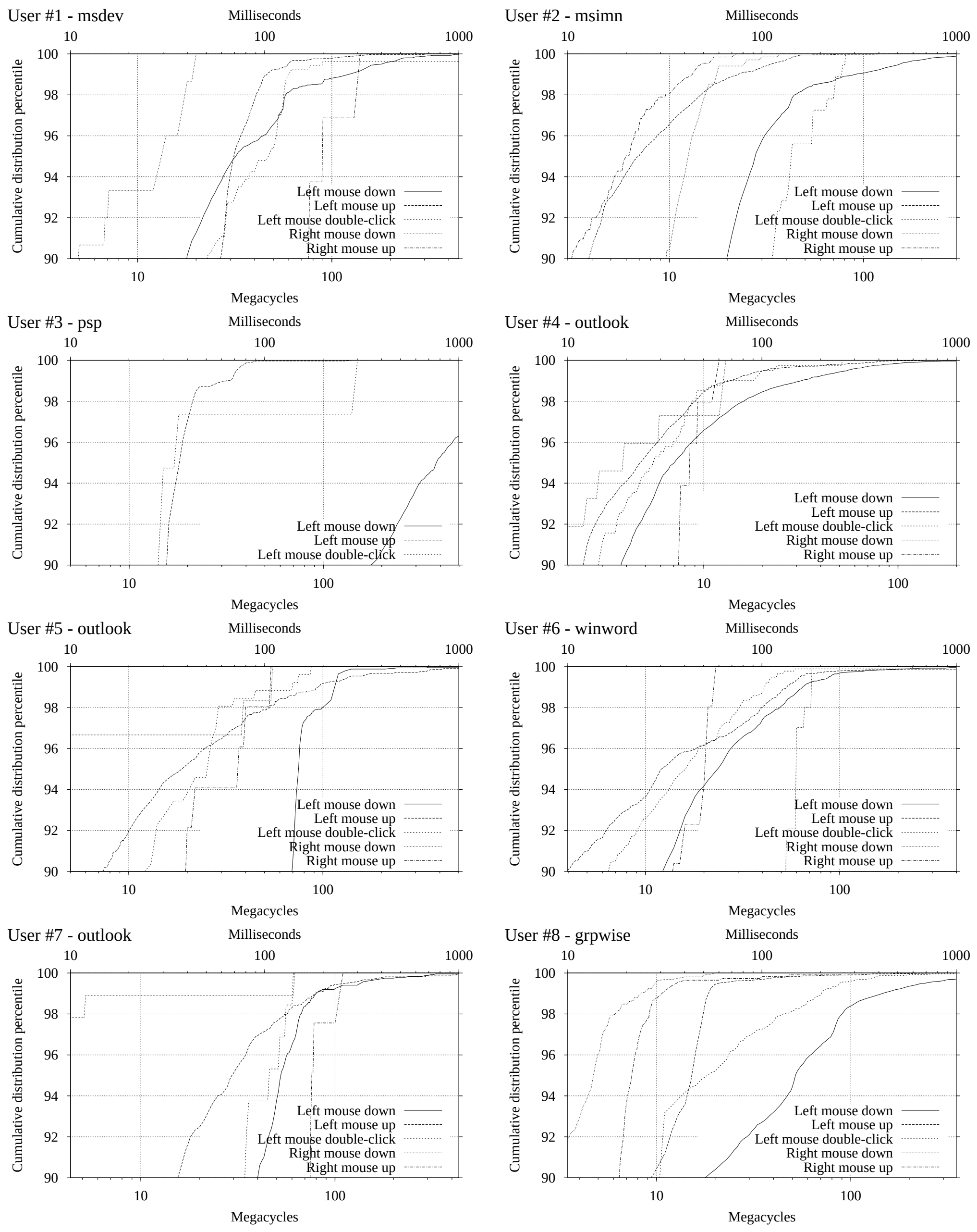

Figure B.2: For various apps and users, the cumulative distribution function of CPU time required by different categories of mouse click events, but only above the 90th percentile

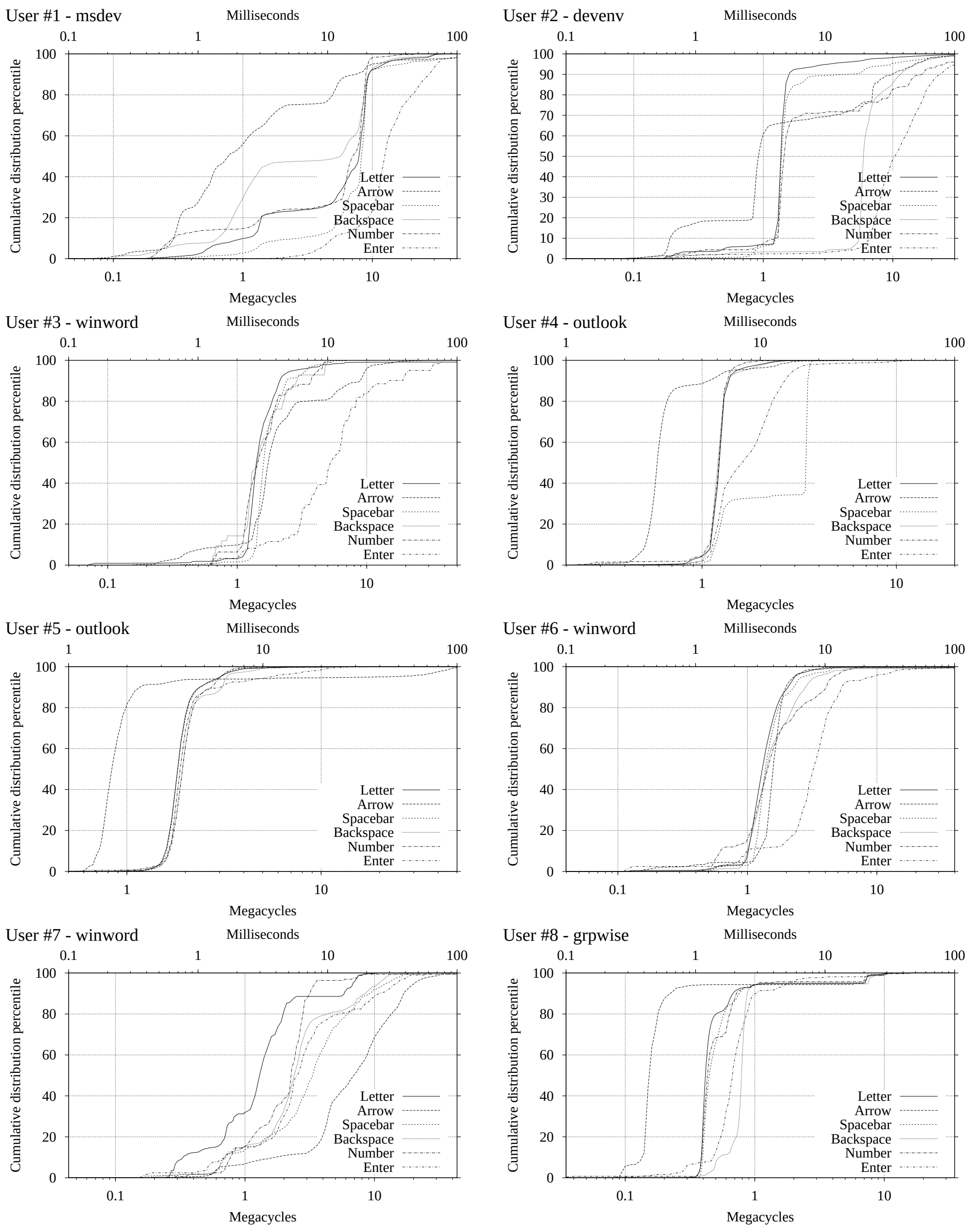

Figure B.3: The cumulative distribution function of CPU time required by different categories of keypress events for various applications and users

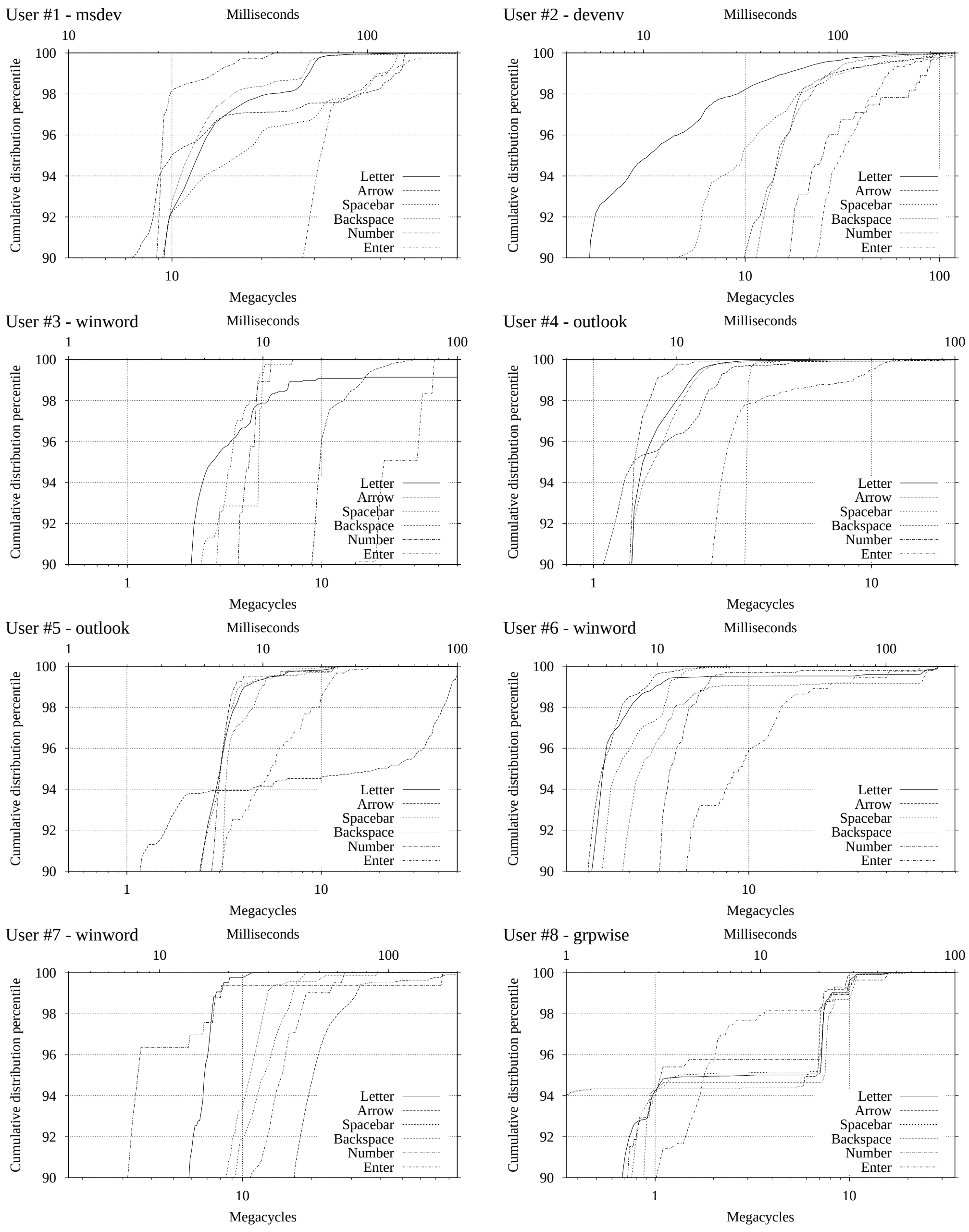

Figure B.4: The cumulative distribution function of CPU time required by different categories of keypress events for various applications and users, but only above the 90th percentile

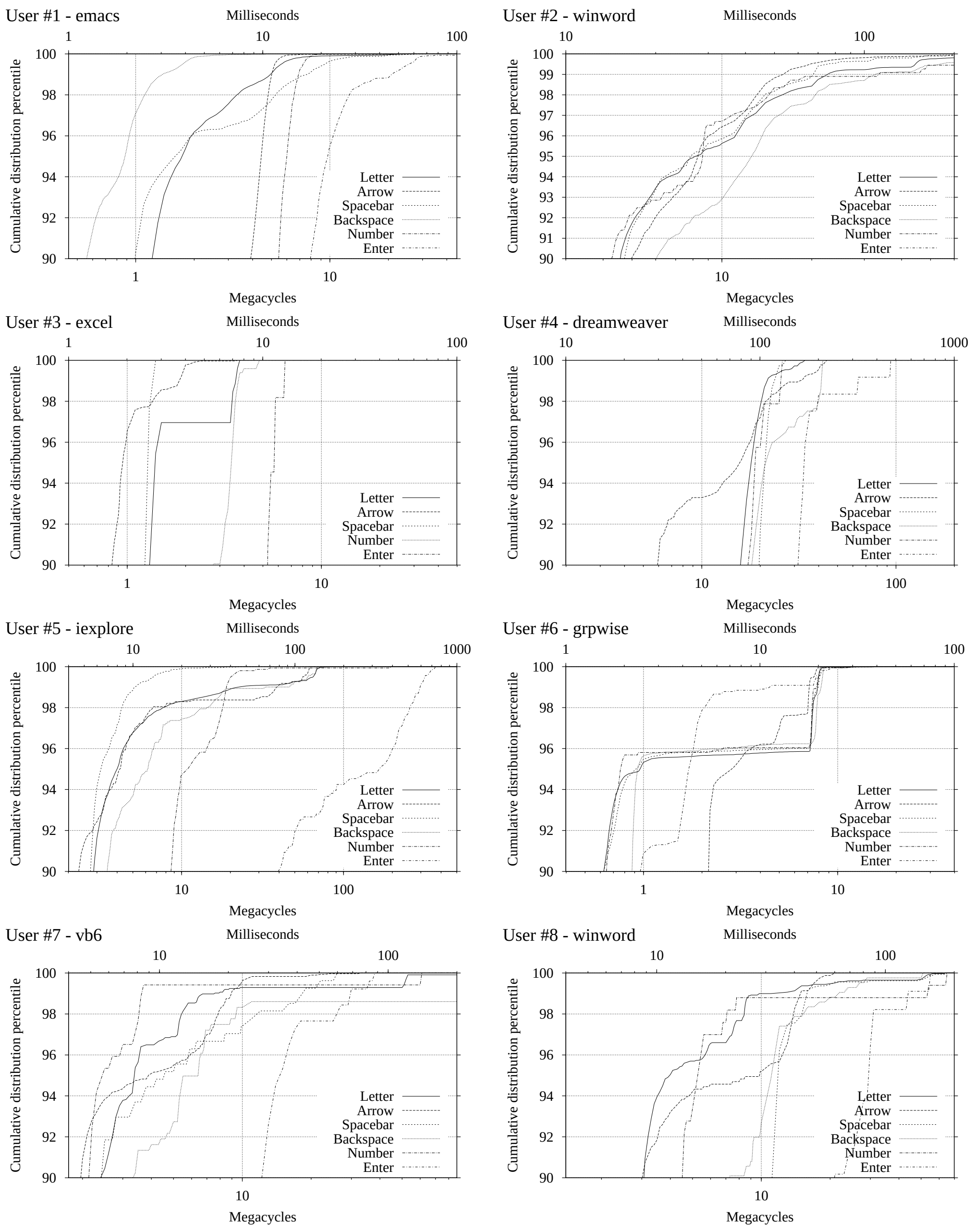

Figure B.5: The cumulative distribution function of CPU time required by different categories of keypress events for various applications and users, but only above the 90th percentile

User #4

	vern	winamp	outlook	iexplore	explorer	eudora	icq	ssh	realplay	netscape	exceed	ttermpro	dreamweaver	acrord32
vern		d	D	D	D	D	D	D	D	D	D	D	D	D
winamp	d		D	D	D	D	D	D	D	D	D	D	D	D
outlook	D	D		D	D	D	d	D	D	D	D	D	D	D
iexplore	D	D	D		D	d		d	D	D	D	d	D	D
explorer	D	D	D	D					D	D	D	·	D	D
eudora	D	D	D	d					D	D	D		D	D
icq	D	D	d						·	d	D		D	D
ssh	D	D	D	d					·	D	D		D	D
realplay	D	D	D	D	D	D	·	·						D
netscape	D	D	D	D	D	D	d	D						D
exceed	D	D	D	D	D	D	D	D						D
ttermpro	D	D	D	d	·									D
dreamweaver	D	D	D	D	D	D	D	D						D
acrord32	D	D	D	D	D	D	D	D	D	D	D	D	D	

User #5

	msiexec	iexplore	explorer	outlook	winword	acrord32
msiexec		D	D	D	D	D
iexplore	D			D	D	D
explorer	D			D	D	D
outlook	D	D	D		D	d
winword	D	D	D	D		d
acrord32	D	D	D	d	d	

Table B.70: This table shows the significance of differences between left mouse down task length means for different applications for users #4–5. A 'D' indicates left mouse down events for the two applications are significantly different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. Results are reported only for applications with at least 30 relevant events. Applications are presented in increasing order of task length mean.

User #6

	notify	hotsync	powerpnt	explorer	excel	winword	aim	planner	iexplore	grpwise	realjbox	acrobat	eudora	acrord32	netscape
notify		D	D	D	D	D	D	D	D	D	D	D	D	D	D
hotsync	D		D	D	D	D	D	D	D	D	D	D	D	D	D
powerpnt	D	D			·	D	D	D	D	D	D	D	D	D	D
explorer	D	D				D	D	D	D	D	D	D	D	D	D
excel	D	D	·			D	D	D	D	D	D	D	D	D	D
winword	D	D	D	D	D			·	D	D	D	D	D	D	D
aim	D	D	D	D	D				d	D	D	D	D	D	D
planner	D	D	D	D	D	·			d	D	D	D	D	D	D
iexplore	D	D	D	D	D	D	d	d		D	D	D	D	D	D
grpwise	D	D	D	D	D	D	D	D	D			D	D	D	D
realjbox	D	D	D	D	D	D	D	D	D			D	D	D	D
acrobat	D	D	D	D	D	D	D	D	D	D	D		·	D	D
eudora	D	D	D	D	D	D	D	D	D	D	D	·			
acrord32	D	D	D	D	D	D	D	D	D	D	D	D			
netscape	D	D	D	D	D	D	D	D	D	D	D	D			

User #7

	powerpnt	explorer	iexplore	vb6	winword	outlook	acrord32
powerpnt		D	D	D	D	D	D
explorer	D		D	D	D	D	D
iexplore	D	D			D	D	D
vb6	D	D			D	D	D
winword	D	D	D	D		D	D
outlook	D	D	D	D	D		D
acrord32	D	D	D	D	D	D	

User #8

	powerpnt	notify	activeshare	ntvdm	excel	realplay	winword	netscape	explorer	iexplore	grpwise	jerusalm	acrobat	acrord32
powerpnt			·	D	D	D	D	D	D	D	D	D	D	D
notify				D	D	D	D	D	D	D	D	D	D	D
activeshare	·			d	D	D	D	D	D	D	D	D	D	D
ntvdm	D	D	d		D	D	D	D	D	D	D	D	D	D
excel	D	D	D	D		D	D	D	D	D	D	D	D	D
realplay	D	D	D	D	D		D	D	D	D	D	d	D	D
winword	D	D	D	D	D	D			D	·	D	·	D	D
netscape	D	D	D	D	D	D			d		d	·	D	D
explorer	D	D	D	D	D	D	D	d					D	D
iexplore	D	D	D	D	D	D	·						D	D
grpwise	D	D	D	D	D	D	D	d					D	D
jerusalm	D	D	D	D	D	d	·	·					D	D
acrobat	D	D	D	D	D	D	D	D	D	D	D	D		
acrord32	D	D	D	D	D	D	D	D	D	D	D	D		

Table B.71: This table shows the significance of differences between left mouse down task length means for different applications for users #6–8. A ‘D’ indicates left mouse down events for the two applications are significantly different with 99% confidence, a ‘d’ indicates 95% significance, and a ‘.’ indicates 90% significance. Results are reported only for applications with at least 30 relevant events. Applications are presented in increasing order of task length mean.

User #4

	realplay	exceed	explorer	netscape	ttermpro	ssh	iexplore	icq	outlook	eudora	dreamweaver
realplay			D	D		D	D	D	D	D	D
exceed			D	D		D	D	D	D	D	D
explorer	D	D		D		D	D	D	D	D	D
netscape	D	D	D			D	D	D	D	D	D
ttermpro								·	·	d	D
ssh	D	D	D	D				D	D	D	D
iexplore	D	D	D	D						·	D
icq	D	D	D	D	·	D			D		D
outlook	D	D	D	D	·	D		D			D
eudora	D	D	D	D	d	D	·				D
dreamweaver	D	D	D	D	D	D	D	D	D	D	

User #5

	rapigator	explorer	iexplore	outlook	winword
rapigator		D	D	D	D
explorer	D			·	D
iexplore	D				D
outlook	D	·			D
winword	D	D	D	D	

User #6

	aim	explorer	excel	acrobat	grpwise	netscape	winword	realjbox	iexplore	planner	eudora	ntvdm
aim		D	D	D	D	d	D	D	D	D	D	D
explorer	D			D	D		D	d	D	D	D	D
excel	D			D	D		D	d	D	D	D	D
acrobat	D	D	D				D	·	D	D	D	D
grpwise	D	D	D				D	·	D	D	D	D
netscape	d								D	D	D	D
winword	D	D	D	D	D				D	D	D	D
realjbox	D	d	d	·	·					D	D	D
iexplore	D	D	D	D	D	D	D			D	D	D
planner	D	D	D	D	D	D	D	D	D		D	D
eudora	D	D	D	D	D	D	D	D	D	D		D
ntvdm	D	D	D	D	D	D	D	D	D	D	D	

User #7

	iexplore	acrord32	vb6	explorer	outlook	powerpnt	winword
iexplore					D	D	D
acrord32						D	D
vb6						D	D
explorer							
outlook	D					d	D
powerpnt	D	D	D		d		
winword	D	D	D		D		

User #8

	explorer	netscape	acrord32	activeshare	excel	grpwise	winword	ntvdm	jerusalm
explorer		D	D	D	D	D	D	D	D
netscape	D		D	D	D	D	D	D	D
acrord32	D	D			D	D	D	D	D
activeshare	D	D			D	D	D	D	D
excel	D	D	D	D			D	D	D
grpwise	D	D	D	D			D	D	D
winword	D	D	D	D	D	D		D	D
ntvdm	D	D	D	D	D	D	D		D
jerusalm	D	D	D	D	D	D	D	D	

Table B.72: This table shows the significance of differences between modifiers-only key press task length means for different applications for users #4–8. A 'D' indicates modifier key press events for the two applications are significantly different with 99% confidence, a 'd' indicates 95% significance, and a '.' indicates 90% significance. Results are reported only for applications with at least 30 relevant events. Applications are presented in increasing order of task length mean.

Bibliography

[AD99] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 232–246, December 1999.

[AMD01] AMD. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. On the World Wide Web at http://www.amd.com/products/cpg/athlon/techdocs/pdf/24319.pdf, August 2001.

[BB00] Tom Burd and Robert W. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 9–14, July 2000.

[BBB$^+$94] B. Barringer, Tom Burd, Frederick L. Burghardt, Andrew J. Burstein, Anantha Chandrakasan, Roger Doering, Shankar Narayanaswamy, Trevor Pering, Brian C. Richards, Tom Truman, Jan M. Rabaey, and Robert W. Brodersen. Infopad: a system design for portable multimedia access. In *Proceedings of the Calgary Wireless 1994 Conference*, July 1994.

[BBB$^+$95] Eric Brewer, Tom Burd, Frederick L. Burghardt, Andrew J. Burstein, Roger Doering, Ken Lutz, Shankar Narayansaramy, Trevor Pering, Brian C. Richards, Tom Truman, Randy H. Katz, Jan M. Rabaey, and Robert W. Brodersen. Design of wireless portable systems. In *Proceedings of the IEEE International Computer Society Conference (COMPCON 95*, pages 169–176, March 1995.

[BD96] P. Bellanger and W. Diepstraten. 802.11 MAC entity: MAC basic access mechanism / privacy and access control, March 1996.

[BEP$^+$96] Jacek Błażewicz, Klaus H. Ecker, Erwin Pesch, Günter Schmidt, and Jan Węglarz. *Scheduling Computer and Manufacturing Processes.* Springer-Verlag, Berlin, Germany, 1996.

[BG99] Robert M. Balzer and Neil M. Goldman. Mediating connectors. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, pages 73–77, May/June 1999.

[CAT$^+$01] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2001.

[CDLM93] Ramón Cáceres, Fred Douglis, Kai Li, and Brian Marsh. Operating system implications of solid-state mobile computers. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, pages 21–27, October 1993.

[CGW95] Edwin Chan, Kinshuk Govil, and Hal Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM 95)*, pages 13–25, November 1995.

[Con92] James L. Conger. *Windows API Bible*. Waite Group Press, Corte Madera, CA, 1992.

[Cox95] Donald C. Cox. Wireless personal communications: what is it? *IEEE Personal Communications*, 2(2):20–35, April 1995.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[Cus94] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, Redmond, WA, 1994.

[Del01] Dell Computer Corporation. Dell Computer Corporation. On the World Wide Web at http://www.dell.com, June 2001.

[DENP96] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MOBICOM 96)*, pages 1–14, November 1996.

[DKB95] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*, pages 121–137, April 1995.

[DKM$^+$94a] Fred Douglis, Frans Kaashoek, Brian Marsh, Ramón Cáceres, Kai Li, and Joshua Tauber. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 25–37, November 1994.

[DKM94b] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *Proceedings of the 1994 Winter USENIX Conference*, pages 293–306, 1994.

[ES00] Yasuhiro Endo and Margo Seltzer. Improving interactive performance using TIPME. In *Proceedings of the 2000 ACM SIGMETRICS Conference*, pages 240–251, June 2000.

[FGBA96] Armando Fox, Steve D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 160–170, October 1996.

[FRM01] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the Seventh ACM International Conference on Mobile Computing and Networking (MOBICOM 2001)*, July 2001.

[FS99] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.

[Fuj97] Fuji Photo Film Co., Ltd. Fujifilm develops new generation lithium ion secondary battery. On the World Wide Web at http://www.fujifilm.co.jp/eng/news_e/nr079.html, January 1997.

[Fuj01] Fujitsu Limited. 2.5-inch magnetic disk drives: Mhl2300at, mhm2200at/mhm2150at/mhm2100at. On the World Wide Web at http://hdd.fujitsu.com/global/drive/mhl2xxx/mhl2xxx.html, June 2001.

[GDE+94] Sonya Gary, Carl Dietz, Jim Eno, Gianfranco Gerosa, Sung Park, and Hector Sanchez. The PowerPC^{TM} 603 microprocessor: a low-power design for portable applications. In *Proceedings of the IEEE International Computer Society Conference*, pages 307–315, 1994.

[GLF+00] Dirk Grunwald, Philip Levis, Keith I. Farkas, Charles B. Morrey III, and Michael Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[GPRA98] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: an extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 39–52, June 1998.

[Gre94] Paul M. Greenawalt. Modeling power management for hard disks. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 62–66, February 1994.

[Ham01] Rowan Hamilton. Personal communication, April 2001.

[HDP$^+$95] Erik P. Harris, Steven W. Depp, William E. Pence, Scott Kirkpatrick, M. Sri-Jayantha, and Ronald R. Troutman. Technology directions for portable computers. *Proceedings of the IEEE*, 83(4):636–657, April 1995.

[Hew96] Hewlett Packard. Infrared IRDA compliant transceiver HSDL-1001 technical data. On the World Wide Web at ftp://ftp.hp.com/pub/accesshp/HP-COMP/ir/hsdl1001.pdf, November 1996.

[Hit01] Hitachi Limited. Dk22aa disk drive specifications. On the World Wide Web at http://www.hitachi.com/storage/products/hdd/2.5%20inch%20disk%20drive%20manuals/DK22AA%7E2.PDF, June 2001.

[HLS96] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. Dynamic disk spin-down technique for mobile computing. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MOBICOM 96)*, pages 130–142, November 1996.

[HPS98] Inki Hong, Miodrag Potkonjak, and Mani B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of the International Conference on Computer Aided Design*, pages 653–656, November 1998.

[IBM01] IBM Corporation. Hard disk drive specifications: Travelstar 48gh, 30gn, & 15 gn. On the World Wide Web at http://www.storage.ibm.com/techsup/hddtech/prodspec/t48gh30gn15gn_sp.pdf, May 2001.

[IM93] Intel Corporation and Microsoft Corporation. *Advanced Power Management (APM) BIOS interface specification revision 1.1*, September 1993.

[Int01] Intel Corporation. *Mobile Intel Pentium III processor in BGA2 and micro-PGA2 packages, revision 7.0*, May 2001.

[IVB94] Tomasz Imielinski, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 25–36, May 1994.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, Inc., New York, NY, 1991.

[JK70] Norman L. Johnson and Samuel Kotz. *Continuous Univariate Distributions - I: Distributions in Statistics.* John Wiley & Sons, Inc., New York, NY, 1970.

[Jon93] Michael B. Jones. Interposition agents: transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, December 1993.

[Kat93] Randy Kath. The portable executable file format, from top to bottom. on the World Wide Web at http://premium.microsoft.com/msdn/library/techart msdn_pefile.htm, June 1993.

[KK92] Eduardo Krell and Balachander Krishnamurthy. COLA: customized overlaying. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 3–7, January 1992.

[Kla00] Alexander Klaiber. The technology behind Crusoe™ processors. White paper, Transmeta Corporation, January 2000.

[KLV95] P. Krishnan, Philip M. Long, and Jeffrey Scott Vitter. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. Technical Report CS-1995-08, Duke University, 1995.

[KMMO94] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6):542–571, June 1994.

[KNM95] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the 1995 USENIX Technical Conference*, pages 155–164, January 1995.

[KSM$^+$98] Tadahiro Kuroda, Kojiro Suzuki, Shinji Mita, Tetsuya Fujita, Fumiyuki Yamane, Fumihiko Sano, Akihiko Chiba, Yoshinori Watanabe, Koji Matsuda, Takeo Maeda, Takayasu Sakurai, and Tohru Furuyama. Variable supply-voltage scheme for low-power high-speed CMOS digital design. *IEEE Journal of Solid-State Circuits*, 33(3):454–462, March 1998.

[Kut95] David Kuty. Personal communication, 1995.

[KZ89] Geoffrey Keppel and Sheldon Zedeck. *Data Analysis for Research Designs: Analysis of Variance and Multiple Regression/Correlation Approaches.* W. H. Freeman and Company, New York, NY, 1989.

[LB96] Yuming Lu and Robert W. Brodersen. Unified power control, error correction coding and scheduling for a cdma downlink system. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM 96)*, pages 1125–1132, March 1996.

[LKHA94] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the 1994 Winter USENIX Conference*, pages 279–291, 1994.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[Lor95a] Jacob R. Lorch. A complete picture of the energy consumption of a portable computer. Master's thesis, Computer Science Division, EECS Department, University of California at Berkeley, 1995.

[Lor95b] Jacob R. Lorch. Modeling the effect of different processor cycling techniques on power consumption. Technical Report 179, Performance Evaluation Group, ATG Integrated Systems, Apple Computer, Inc., November 1995.

[LS96] Jacob R. Lorch and Alan Jay Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MOBICOM 96)*, pages 143–154, November 1996.

[LS98] Jacob R. Lorch and Alan Jay Smith. Energy consumption of Apple Macintosh computers. *IEEE Micro*, 18(6):54–63, November/December 1998.

[LS02] Jacob R. Lorch and Alan Jay Smith. Dynamic voltage scaling implications from analyses of personal computer workloads. Submitted to somewhere for publication, January 2002.

[Luc96a] Lucent Technologies. WaveLAN/PCMCIA Card. On the World Wide Web at ftp://ftp.wavelan.com/pub/pdf-file/pcmcia/fs-pcm.pdf, November 1996.

[Luc96b] Lucent Technologies. WaveLAN/PCMCIA card user's guide. On the World Wide Web at ftp://ftp.wavelan.com/pub/pdf-file/pcmcia/pcman3x.pdf, October 1996.

[Mac91] Jim MacDonald. Power management for 386DXL-based notebook computers. In *Proceedings of the Silicon Valley Personal Computing Conference*, pages 735–748, 1991.

[MDK93] Brian Marsh, Fred Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Sciences*, pages 451–460, January 1993.

[Mic00] Microsoft Corporation. *Platform SDK Documentation*, 2000.

[Mic01] Micron Technology, Inc. Synchronous DRAM. On the World Wide Web at http://images.micron.com/pdf/datasheet/dram/256MSDRAM_C.pdf, June 2001.

[MS95] William Mangione-Smith. Low power communications protocols: paging and beyond. In *1995 IEEE Symposium on Low Power Electronics Digest of Technical Papers*, pages 8–11, October 1995.

[MSGN+96] William Mangione-Smith, Phil Seong Ghang, Sean Nazareth, Paul Lettieri, Walt Boring, and Rajeev Jain. A low power architecture for wireless multimedia systems: lessons learned from building a power hog. In *1996 International Symposium on Low Power Electronics and Design Digest of Technical Papers*, pages 23–28, August 1996.

[MW93] I. Scott MacKenzie and Colin Ware. Lag as a determinant of human performance in interactive systems. In *Proceedings of INTERCHI '93*, pages 24–29, April 1993.

[Nag97] Rajeev Nagar. *Windows NT File System Internals.* O'Reilly and Associates, Inc., Sebastopol, CA, 1997.

[Neb00] Gary Nebbett. *Windows NT/2000 Native API Reference.* Macmillan Technical Publishing, Indianapolis, IN, 2000.

[NPS95] Brian D. Noble, Morgan Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. *Computing Systems*, 8(4):345–363, 1995.

[PBB98] Trevor Pering, Tom Burd, and Robert W. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[Pie93] Matt Pietrek. *Windows Internals.* Addison-Wesley, Reading, MA, 1993.

[Pie94a] Matt Pietrek. Learn system-level Win32 coding techniques by writing an API spy program. *Microsoft Systems Journal*, 9(12):17–38, December 1994.

[Pie94b] Matt Pietrek. Peering inside the PE: a tour of the Win32 portable executable file format. *Microsoft Systems Journal*, 9(3):15–32, March 1994.

[Pow95] Robert A. Powers. Batteries for low power electronics. *Proceedings of the IEEE*, 83(4):687–693, April 1995.

[Pre92] William H. Press. *Numerical Recipes in C: the Art of Scientific Computing.* Cambridge University Press, Cambridge, MA, 1992.

[PS01] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102, October 2001.

[PSR93] Ketan Patel, Brian Smith, and Lawrence Rowe. Performance of a software MPEG video decoder. In *Proceedings of the First ACM International Conference on Multimedia*, pages 75–82, August 1993.

[RB89] J. C. W. Rayner and D. J. Best. *Smooth Tests of Goodness of Fit.* Oxford University Press, New York, NY, 1989.

[RB96] John M. Rulnick and Nicholas Bambos. Mobile power management for maximum battery life in wireless communication networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM 96)*, pages 443–450, March 1996.

[RC97a] Mark Russinovich and Bryce Cogswell. Examining the Windows NT filesystem. *Dr. Dobb's Journal*, 22(2):42–50, February 1997.

[RC97b] Mark Russinovich and Bryce Cogswell. Windows NT system-call hooking. *Dr. Dobb's Journal*, 22(1):42–46, January 1997.

[RDH+80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: an operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, 1980.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[SBB72] Stephen Sherman, Forest Baskett III, and J. C. Browne. Trace-driven modeling and analysis of CPU scheduling in a multiprogramming system. *Communications of the ACM*, 15(12):1063–1069, December 1972.

[SCB96] Mani B. Srivastava, Anantha P. Chandrakasan, and Robert W. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):42–55, 1996.

[Sch97] Herbert Schildt. *Windows NT Programming from the Ground Up.* Osborne/McGraw-Hill, Berkeley, CA, 1997.

[She01] Richard Sherwin. Memory on the move. *Spectrum*, 38(5):55–59, May 2001.

[Shn98] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, Reading, MA, 1998.

[SI94] Brad W. Suessmith and George Paap III. PowerPC 603 microprocessor power management. *Communications of the ACM*, 37(6):43–46, 1994.

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis.* Chapman and Hall, London, England, 1986.

[SK97] Mark Stemm and Randy Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *Institute of Electronics, Information, and Communication Engineers Transactions on Communications*, E80-B(8):1125–1131, August 1997.

[SKM+93] M. Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. Experience with disconnected operation in a mobile computing environment. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pages 11–28, August 1993.

[SML94] J. H. Snyder, J. B. McKie, and B. N. Locanthi. Low-power software for low-power people. In *1994 IEEE Symposium on Low Power Electronics Digest of Technical Papers*, pages 32–35, October 1994.

[Soh94] Phil Sohn. SETC: a system event tracer and counter. Technical report, Performance Evaluation Group, ATG Integrated Systems, Apple Computer, Inc., 1994.

[Soh95] Phil Sohn. Personal communication, 1995.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SU93] Naoshi Suzuki and Shunya Uno. Information processing system having power saving control of the processor clock. United States Patent #5,189,647, 1993.

[TM99] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 117–130, February 1999.

[TMWL96] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(3):223–238, September 1996.

[Tom96] Paula Tomlinson. How to write an NT service. *Windows Developer's Journal*, 7(2):6–18, February 1996.

[Tos01] Toshiba America Information Systems. Toshiba MK3017GAP (HDD2159) functional specifications. On the World Wide Web at http://www.toshiba.com/taecdpd/techdocs/MK3017/3017spec.shtml, June 2001.

[Wer94] Kenneth I. Werner. Flat panels fill the color bill for laptops. *Circuits and Devices*, 10(4):21–29, July 1994.

[WWDS94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.

[WZ94] Michael Wu and Willy Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 86–97, October 1994.

[YDS95] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE 36th Annual Symposium on Foundations of Computer Science*, pages 374–382, October 1995.

[ZR97] Michele Zorzi and Ramesh R. Rao. Error control and energy consumption in communications for nomadic computing. *IEEE Transactions on Computers*, 46(3):279–289, March 1997.

[ZS97] Barbara T. Zivkov and Alan Jay Smith. Disk caching in large database and timeshared systems. In *Proceedings of the Fifth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 97)*, pages 184–195, January 1997.

[ZS00] Min Zhou and Alan Jay Smith. Tracing Windows 95. *Journal of Microprocessors and Microsystems*, 24(7):333–347, November 2000.