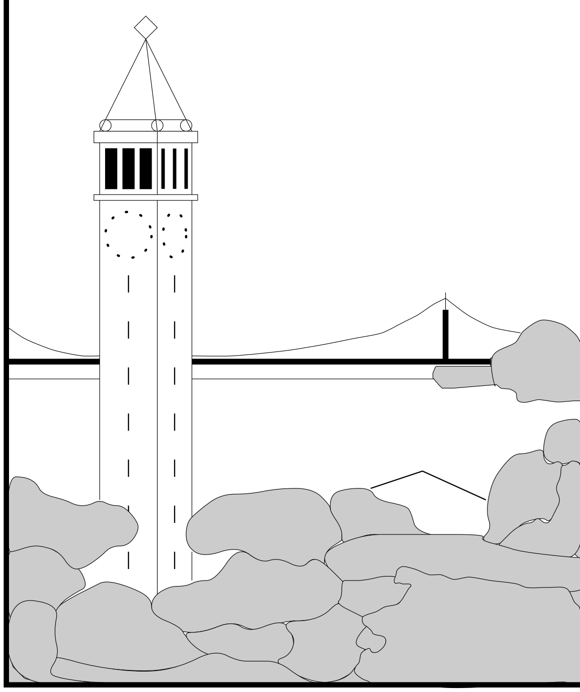
Using User Interface Event Information in Dynamic Voltage Scaling Algorithms

Jacob R. Lorch *Alan Jay Smith*



Report No. UCB/CSD-02-1190

August 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Using User Interface Event Information in Dynamic Voltage Scaling Algorithms*

Jacob R. Lorch† Alan Jay Smith‡

August 2002

Abstract

Increasingly, mobile computers are using dynamic voltage scaling (DVS) to increase their battery life. We analyze traces of real user workloads to determine how DVS algorithms should treat tasks triggered by user interface events. We compare different DVS algorithms and find that for a given level of performance, Lorch et al.'s PACE (Processor Acceleration to Conserve Energy) algorithm always uses the least energy and the Stepped algorithm always uses the second least. The Stepped algorithm increases speed at a constant rate, while PACE increases speed according to an energy-minimizing schedule derived from the probability distribution of a task's CPU requirement. We find that different types of user interface event (mouse movements, mouse clicks, and keystrokes) trigger tasks with significantly different CPU requirements, suggesting that DVS algorithms should use different speeds for these different types of event. For example, mouse movements require little enough CPU time that using the minimum speed available for them reduces deadlines made by only 0.2% while reducing CPU energy consumption for those events by 77.5%. We also find significant differences in CPU requirements between different categories of the same event type, e.g., between pressing the spacebar and pressing the enter key, and between events delivered to different applications. Conditioning the speed schedule on the event category, however, yields only small improvements in energy consumption: in our experiments, by an additional 1.5% for keystrokes and 0.5% for mouse clicks. Finally, we evaluate a novel heuristic for inferring when user interface tasks complete and find it is more efficient and nearly as effective as more complex approaches.

1 Introduction

Reducing energy consumption of portable computers and other mobile devices is important, mainly because this increases their battery lifetime. Transmeta, AMD, and Intel now sell processors with *dynamic voltage scaling* (DVS), a feature that lets the system dynamically alter CPU speed and save energy while running at low speeds. However, to take advantage of this feature, the system must determine when it is worthwhile to trade off speed for energy savings.

Research suggests that to accomplish this, DVS algorithms should consider what tasks the system is working on, how much CPU time those tasks use, and when those tasks should complete [FRM01, LS01, PS01, PLS01]. The time a task should complete is called its *deadline*; this may be a hard deadline, meaning it must be made, or it may be a soft deadline, meaning the task may miss the deadline but doing so impacts perceived performance negatively. Unfortunately, most modern systems lack an interface allowing applications to specify such task information. Even if such an interface were added to a system, applications already released could not use it to provide task information, and writers of new applications might be unwilling or unable to provide accurate information via this interface. For these reasons, authors such as Flautner et al. [FRM01] and Lorch et al. [LS01] suggest estimating task information for applications oblivious to task-specification interfaces.

One way to infer task information is from user interface events. User interface studies have shown that users are satisfied with response times to user interface events as long as they do not exceed 50–100 ms [Shn98]. A DVS algorithm can thus infer that a task begins when a user interface event arrives, and assume that its deadline is 50–100 ms. The algorithm must also guess when a task completes, and Flautner et al. [FRM01] suggest a heuristic for determining this. Using such methods, a DVS algorithm can, most of the time, set the CPU speed high enough to adequately respond to pending user interface events, yet not so high that it wastes energy.

Researchers have used short benchmarks to study how to estimate task information from user interface events. In such a benchmark, a user uses an application in some specified manner. However, we feel that to design DVS algorithms generally applicable to any user running any application, it is beneficial to study how users act "in the wild" and to thereby answer several questions not easily answered with simple benchmarks. Our questions include:

- What fraction of all tasks result from user interface events?
- Is there an efficient heuristic for deducing when user interface tasks end?
- In real scenarios, how often does handling user interface events require waiting for I/O? This is relevant for DVS algorithms because I/O time does not scale with CPU frequency.
- How quickly should we run the CPU to achieve good response time goals for user interface tasks?
- Which DVS algorithms work best for user interface tasks?
- Is there significant enough difference between different categories of user interface event (e.g., between pressing the spacebar and the enter key) that a DVS algorithm should handle different categories differently?

*This material is based upon work supported by the State of California MICRO program, AT&T Research, Cisco Systems, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation, and Veritas Software.

†Microsoft Research, lorch@microsoft.com

‡Computer Science Division, EECS Department, University of California, Berkeley, CA 94720-1776, smith@cs.berkeley.edu

- Is there significant enough difference between different applications that a DVS algorithm should handle different applications differently?

In this paper, we use several months' worth of trace data from each of eight users running Windows NT/2000. Analyzing these traces provides us the answers to these questions and gives us insights into how to use estimated task information in a DVS algorithm on a PC-like device. Many of these insights are applicable to other operating systems besides Windows NT/2000, and our methodology could be used in future to evaluate the answers to our questions on smaller computing devices such as personal digital assistants.

The paper is structured as follows. Section 2 gives background and related work. Section 3 describes the traces we collected and how we process them to infer when tasks begin and end. Section 4 analyzes the workloads in many different ways to answer the above questions and thereby make recommendations for designing DVS algorithms. Section 5 describes avenues for future work. Finally, Section 6 concludes.

2 Background and Related Work

2.1 Dynamic voltage scaling

In CMOS circuits, the dominant component of power consumption is proportional to $V^2 f$, where V is the voltage and f is the frequency [WE93, p. 235]. Thus, energy per cycle is proportional to V^2. At a given voltage, the maximum frequency at which the CPU can safely run decreases roughly linearly with decreasing voltage. Thus, the system can reduce processor energy consumption by reducing the voltage, but this necessitates running at a slower speed.

2.2 DVS algorithms

Weiser et al. [WWDS94] and Chan et al. [CGW95] proposed the first DVS algorithms. Each of these algorithms divides time into intervals of a fixed length, e.g., 10 ms. At the beginning of each interval, each predicts the CPU utilization of the upcoming interval based on observations of the CPU utilization of previous intervals, then sets the speed for that interval based on this prediction. We call such algorithms *interval-based*.

These authors evaluated their algorithms by energy savings and by how often they completed the work introduced within each interval by the end of that interval. Later research, including Pering et al.'s [PBB98] and Grunwald et al.'s [GLF$^+$00], pointed out that the deadlines of system tasks do not in general correspond to the arbitrarily chosen boundaries between intervals. Therefore, they evaluated these strategies by energy savings and by how often they met task deadlines.

Other researchers have suggested abandoning interval-based strategies entirely in favor of task-based strategies. Such strategies take task information, especially task deadlines, into account in choosing what speed to use at any given time. Pillai et al. [PS01] developed a real-time dynamic voltage scheduler for embedded operating systems that takes information about ongoing periodic tasks and converts it into an energy-efficient speed schedule. Pouwelse et al. [PLS01] modified a video player to predict its CPU requirements for each frame and to use this information to determine what CPU speed it requires for each frame.

Flautner et al. [FRM01] proposed deriving task information automatically from applications that do not provide it to the system. They consider two types of tasks: interactive and periodic. Interactive tasks are triggered by the user initiating an action, typically via a user interface event. Periodic tasks are triggered by a periodic event; they suggest considering an event periodic if the lengths of intervals between the last n events have a small variance. They also presented a heuristic for inferring when a task completes, as follows. The thread that receives the triggering event forms the initial *thread set* for the event. When a member of this set communicates with another thread, that other thread becomes part of the set. The task is considered complete when, for each thread in the set, that thread is not executing, data it has written have been consumed, and it is blocked but not on I/O.

Lorch et al. [LS01] investigated how task information could be used to further reduce the energy consumption of DVS algorithms. They pointed out that a DVS algorithm essentially determines a schedule describing how speed will vary with time as a task runs; two schedules are *performance equivalent* if they either both meet the task's deadline or both miss it by the same amount. They showed that as long as two schedules have the same average pre-deadline speed (i.e., they allocate the same number of CPU cycles prior to the deadline), and as long as they have identical post-deadline parts, those schedules will give the same effective performance no matter how much work a task requires. Thus, one can get the same performance as any existing DVS algorithm by using different, yet performance equivalent, speed schedules; these new schedules may even consume less energy.

They then demonstrated that the theoretically optimal schedule for a task depends on the probability distribution of that task's work requirement. They gave a method for producing an optimal speed schedule from an estimate of this distribution; they call this method PACE (Processor Acceleration to Conserve Energy). The authors suggest estimating the distribution from a sample of recent similar tasks' work requirements. However, they do not describe how the system should identify tasks as being similar.

2.3 User interface workload analysis

Many researchers have used interactive workloads to analyze interactive performance on desktop machines.

Endo et al. [EWCS96] suggested that latency of event handling, not throughput, was the most important performance factor for modern interactive systems. They suggested evaluating the cost of a certain response time in terms of some threshold time below which the user would be unable to distinguish differences in response time. They also suggested interposing the Windows message handling system to infer when interactive events occur and how long the system takes to handle them. They used this approach to compare the performance of various versions of Windows on some application microbenchmarks. We propose using such an interposition scheme to infer the occurrence of user interface events.

Later, in [ES00], Endo et al. used similar techniques to design a collection of tools for monitoring interactive performance. One of these tools runs in the background as a user uses a machine and logs information about ongoing tasks in a small ring buffer. Whenever the user encounters sluggish performance, he can use a hotkey combination to flush this buffer to disk. Later, one can use other tools to analyze the traces collected in this way to understand the source of such performance problems.

Lee et al. [LCB+98] traced Windows NT applications running interactive workloads to compare their use of architectural features to that of SPEC95 workloads. Each such trace was short, consisting of a few million instructions generated by a script that performs some sequence of operations.

Zhou et al. [ZS00] collected traces of many PC users running Windows 95 in everyday use, and reported analyses such as busy period lengths, file system operation frequencies, and system call frequencies.

Flautner et al. [FURM00] used interactive workloads to evaluate how much thread-level parallelism was in modern programs. They used this information to evaluate the effectiveness of using multiprocessing to improve interactive performance. Each workload was run once by a single user, and the six workloads consisted of running Acrobat Reader, Framemaker, Ghostview, GIMP, Netscape, and Xemacs. Later, in [FRM01], they used these workloads to analyze the effectiveness of their methods for estimating task information and for incorporating this information into DVS strategies. This paper is in many ways an extension of this work, as we seek to answer more questions about inferring task information than can be answered from such short workloads.

3 Methodology

3.1 Users

To answer our questions, we traced people using their desktop PC's in normal operation. We used VTrace, a Windows NT/2000 tracer that runs in the background on users' machines [LS00]. It collects time-stamped records describing events related to processes, threads, messages, disk operations, network operations, the keyboard, and the mouse. To limit trace volume, 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 paper, we only use data collected while full tracing was on.

We used traces from several users in this paper. Table 1 contains data about these users' machines as well as summary information about the workloads traced from these users. 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 police captain. He uses his machine primarily for groupware and office suite applications.

User 7 is a software developer in Korea. He uses his machine primarily for software development, web browsing, and mail.

User 8 is a crime laboratory director. He uses his machine primarily for groupware and office suite applications.

3.2 Tasks

Many of the questions we consider concern *tasks*. Generally, people loosely define a task as a sequence of operations serving some end goal. However, we need a way to more precisely define this term and to determine when such a task begins and ends in a trace. Our approach is similar to that of Flautner et al. [FRM01].

When a thread is notified of something that can cause it to begin working on something new, we call this an *event* for that thread. The time a thread spends running between two consecutive events for it we call the *response* to the first event. Possible events are:

- a thread receives a message,
- a thread successfully completes a wait on a waitable object such as a timer or semaphore,
- a thread is notified of an incoming network packet for it, [1]
- a thread starts,
- a thread times out after an unsuccessful wait,
- a thread begins an asynchronous procedure call (APC), i.e., a function call made asynchronously to the thread and invoked via a software interrupt, and
- VTrace begins a session of full logging.

This last type of event requires some explanation. When VTrace begins a session of full logging, one or more threads may be running or ready to run. However, we cannot know what events these threads are responding to, as they occurred when full logging was not on. We use "VTrace begins a session of full logging" as a proxy event for all such unknown events.

If, during a response, a thread causes another event to occur, that event is said to be *dependent* on the event that generated the response. For example, if a thread responding to a user interface event signals an object another thread is waiting for, the event of that other thread completing the wait on that object is considered dependent on the user interface event. As an exception, if a response causes a time-delayed event, e.g., by requesting that a timer message be delivered at some future time, then we do not consider the time-delayed event to be dependent on the event that generated the response. The reason for this is that if a process is willing to let time pass before the work continues, we assume this work is not a critical part of the response and is instead associated with the future time-delayed event.

We define a *trigger event* as one that is not dependent on any other event; such an event is the root of a tree of events in which each event has children corresponding to its dependent events. Finally, we define a *task* as the set of responses to all events in such a tree, and consider the task to be *triggered* by the root trigger event.

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

[1] Identifying when a thread is notified of a packet arrival is not always straightforward, so we use the following heuristic approach. When a packet arrives, we note what process most recently sent a packet on the same connection. The next time a thread of that process successfully completes a wait on an unidentified object, we assume it is notified of that network packet. We devised this heuristic by observing in many traces this general pattern between network arrivals and completions of associated waits.

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	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 1: Trace information for all users

about tasks without unnecessarily intrusive tracing techniques. It is possible 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.

3.3 More definitions

We consider a *user interface event* to occur when a thread receives a message representing either a keystroke (i.e., a key press or release), a mouse movement[2], or a mouse click. A *user interface task* is a task triggered by a user interface event. Note that such a task includes all work the system does in response to the event, not simply the time to handle the user interface device interrupt. For example, if the event is the delivery of a message indicating that the user hit the enter key in a spreadsheet, the triggered task includes the time to perform any spreadsheet calculations triggered by that key press.

User interface events can be further divided into *categories*. For instance, "spacebar press" and "number-key release" are categories of keystroke, and "left (mouse button) down" and "right (mouse button) up" are categories of mouse click. There are many different keystroke categories since a keystroke can be a key press or a key release, it can involve various different keys, and it can occur while various modifiers (the Control, Shift, and Alt keys) are or are not held down.

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

3.4 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, simply aggregating all traces together would cause results to inordinately reflect users with greater tracing time and/or more activity. Because we want results to reflect a broad set of users, not just those with a large amount of activity or those who are willing to be traced for longer periods of time, 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 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 paper.

[2]The system samples the mouse position at some fixed rate, and will generate a message only if the position changes between two consecutive samples. It sends at most one message per sample period, no matter how much the mouse moved during the sampling period.

4 Analyses

In this section, we analyze the trace characteristics to answer the questions we posed earlier. Most of these analyses do not require making any assumptions about CPU characteristics, but some of them require simulating operation on a particular CPU. In these cases, 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 [WWDS94], 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 speed similar to that of the users traced in these workloads. Machines currently for sale, of course, are much faster.

4.1 How much of the CPU's time is spent on user interface events?

First, we consider how much time the CPU spends on user interface events in comparison to other trigger event types. Table 2 shows how much time the CPU spends due to the various types of trigger event. We find that the CPU time due to user interface events 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. An application typically uses timer messages to establish a periodic operation; for example, if it must take some action every 50 ms, it arranges for the delivery of a timer message every 50 ms. 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 spends substantial time (43.0%) on tasks that were already running when VTrace began a session of logging. This suggests a lot of time running long-running processes, and looking at the particular applications responsible, we see they tend to be server processes. Such server workloads are not characteristic of what we would expect on laptops, so we view this value as an

User	User interface message	Timer message	Other message	Timer object	Other waitable 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 2: This table shows the percent of CPU time triggered by each event type. 100% corresponds to the total time the CPU spends running threads other than the idle thread; we assume that the CPU halts while the idle thread is running. The event types are (a) a thread receives a user interface message; (b) a thread receives a timer message; (c) a thread receives a non-user-interface, non-timer message; (d) a thread successfully completes a wait on a timer object; (e) a thread successfully completes a wait on a non-timer object; (f) a thread is notified of an incoming network packet for it; (g) a thread starts; (h) VTrace begins a session of full logging; (i) a thread times out after an unsuccessful wait; and (j) a thread begins an asynchronous procedure call.

aberration not reflective of typical portable computers.

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 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. The most likely 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 4.2, 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 the CPU spends only about 20.3% of its time 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. We conclude that detection of tasks merely by inference from user interface events will reflect only some of the work the CPU must do; a mechanism such as that in [FRM01] for detecting periodic tasks may help infer deadlines for at least some of the remaining work.

4.2 Can we detect task completion efficiently?

There are two important reasons for determining when tasks complete. First, when all tasks are complete, speed is no longer a priority. The system can stop increasing speed and in fact can reduce the speed to the minimum available. Second, to estimate task work distribution, we need to know how long past tasks took, and this requires knowing when they completed.

In Section 3, we explained our methodology for estimating when a user interface task is complete; it is similar to that of Flautner et al. [FRM01]. Unfortunately, Flautner et al.'s approach is quite complex for use in a real on-line algorithm. It requires modifying or interposing many system calls to keep track of thread communications and how they block, and 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.

We propose the following simplified approach. We consider a user interface task 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; in particular, this approach does not require any tracking of inter-thread communication. One problem is that it occasionally misidentifies a task as complete when it is not. Another apparent problem 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 automatically accounts for unrelated work that nevertheless delays the completion of the task. The amount of such unrelated work for this task is a reasonable predictor of what it will be for future tasks, so it is good to account for such sources of delay when estimating the future work requirements for similar tasks.

To evaluate the potential inaccuracy in our method, we determined from our traces what percentage of user interface tasks continue past the point when the system goes idle with no I/O ongoing, and what percentage of them continue past the point when the next user interface event occurs. Table 3 contains these results.

Breaking down this information by application (see [Lor01] for full details), we find that for two applications, exceed and java, a substantial number of user interface tasks go beyond the next idle time and the next user interface task; almost all 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, they may use locks, and our analysis 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 analysis without considering object signaling to cause event dependency;

User	% of user interface tasks continuing past...		
	system idle	next UI event	either boundary
1	10.9%	20.6%	20.8%
2	2.8%	3.1%	3.1%
3	0.3%	0.7%	0.8%
4	2.1%	5.7%	5.9%
5	2.2%	3.2%	3.3%
6	1.4%	1.9%	2.0%
7	2.6%	5.3%	5.3%
8	0.5%	0.8%	0.9%
Without exceed or java...			
1	0.6%	1.3%	1.4%
2	1.4%	1.6%	1.6%
4	0.8%	1.5%	1.6%
With exceed and java, but without considering object signaling to cause event dependence...			
1	0.2%	0.6%	0.6%
2	0.7%	0.8%	0.8%
4	0.4%	1.0%	1.1%

Table 3: For each user, how often the system remains working on a user interface event past the time the system goes idle with no I/O requests and/or past the time the next user interface event is delivered to the same application. This occurs often for users 1, 2, and 4; we present auxiliary results showing this is because these users use exceed and java. Furthermore, we show results suggesting that this phenomenon occurs because of object signaling by those two applications.

User	Key press/release			Mouse move		
	disk	*network*	**either**	disk	*network*	**either**
1	0.3%	*5.0%*	**5.2%**	0.2%	*1.1%*	**1.2%**
2	0.7%	*5.4%*	**5.9%**	0.2%	*0.4%*	**0.6%**
3	0.7%	*0.7%*	**1.4%**	0.3%	*0.2%*	**0.4%**
4	0.1%	*5.4%*	**5.5%**	0.1%	*1.1%*	**1.2%**
5	0.2%	*0.1%*	**0.3%**	0.1%	*0.1%*	**0.2%**
6	0.4%	*0.2%*	**0.6%**	0.2%	*0.1%*	**0.3%**
7	2.6%	*0.4%*	**2.7%**	0.5%	*0.1%*	**0.5%**
8	0.8%	*2.5%*	**3.1%**	0.3%	*0.1%*	**0.4%**
Avg	0.7%	*2.5%*	**3.1%**	0.2%	*0.4%*	**0.6%**

User	Mouse click			All user interface events		
	disk	*network*	**either**	disk	*network*	**either**
1	8.3%	*9.3%*	**15.5%**	0.3%	*3.2%*	**3.5%**
2	5.0%	*2.7%*	**6.8%**	0.5%	*1.1%*	**1.4%**
3	12.5%	*2.9%*	**14.5%**	0.5%	*0.3%*	**0.7%**
4	6.1%	*10.0%*	**14.6%**	0.3%	*2.2%*	**2.4%**
5	3.1%	*2.7%*	**5.1%**	0.2%	*0.2%*	**0.3%**
6	10.0%	*5.1%*	**13.3%**	0.5%	*0.2%*	**0.6%**
7	10.0%	*1.5%*	**10.5%**	1.0%	*0.2%*	**1.0%**
8	17.2%	*7.9%*	**18.1%**	0.7%	*0.3%*	**0.8%**
Avg	9.0%	*5.3%*	**12.3%**	0.5%	*1.0%*	**1.3%**

Table 4: For each user, and for various types of user interface events, the percentage of those events that wait for disk I/O and/or the network

the results are in Table 3. 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 the odd results for those applications.

Ignoring these two applications, we find that few tasks continue past system idle, no more than 2.6%. Also, few tasks continue past the next user interface event, no more than 5.3%. 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.

4.3 How often do user interface tasks wait for I/O?

An important concern for DVS algorithms is how often tasks wait for I/O, because I/O power and time do not scale as the CPU voltage and speed are scaled. Table 4 shows what percentage of user interface tasks require I/O of two kinds: disk and network. For breakdowns of this information by application, see [Lor01].

We find that 0.3–3.5% of user interface tasks wait for I/O, with an average of 1.3%. Restricting consideration to just the disk, only 0.2–1.0% of user interface tasks wait for disk I/O, with an average of 0.5%. This may be the most relevant figure, since computers running on a battery tend not to be connected to a network.

Part of the reason for infrequent I/O is that mouse movements are most common but seldom require I/O. In contrast, 5.1–18.1% of all mouse clicks, with an average of 12.3%, wait for I/O. For this reason, DVS algorithms planning schedules for mouse click events may need to provide extra slack for I/O time. The rate of I/O among keystroke tasks 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.

Different applications have different fractions of their tasks waiting for I/O. For example, for user #4, ssh requires network I/O for 29.1% of all its keystroke events, and for user #2, starcraft requires network I/O for 20.5% of its keystroke events. Thus, it may be worthwhile for a DVS algorithm to keep track of which applications require substantial I/O and treat them specially.

Developing algorithms for dealing with tasks requiring I/O is beyond the scope of this paper, so in subsequent analyses we will restrict consideration to those events that trigger no I/O.

4.4 How fast should we run user interface tasks?

Generally, one can tune any DVS algorithm to provide varying levels of performance and energy savings. An important "knob" in such tuning is the choice of average pre-deadline speed, or, equivalently, the choice of how much computation to perform before the deadline. This quantity, by itself, completely determines which tasks meet their deadlines: a task meets its deadline if and only if its work requirement does not exceed the deadline times the average pre-deadline speed. Therefore, in this section we can ignore differences in DVS algorithms to focus solely on the question of what average pre-deadline speed will give good deadline performance.

We seek a pre-deadline speed such that running at that speed satisfies a large percentage of task deadlines and running at higher speeds does not substantially increase this percentage. We do this

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 5: Results from using suggested DVS algorithm on mouse movements

by examining the cumulative distribution function of task work requirements on a logarithmic scale, as in Figure 1. Our 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. Dividing this point by the deadline gives a pre-deadline speed above which there is not much improvement in deadline performance for a given increase in speed.

A glance at Figure 1 suggests that we should use a different average pre-deadline speed for mouse movements, mouse clicks, and keystrokes due to their different work requirements. We now consider what average speed seems useful for each class of events.

For mouse movements, we assume a soft deadline of 50 ms, following [ES00]. Considering the trend toward faster processors, we will assume conservatively that future processors are unlikely to have available speeds lower than 200 MHz. At this minimum speed, we complete 10 Mc (10 million CPU cycles) within 50 ms. For all users, this is above the 99th percentile, meaning this strategy would complete over 99% of tasks within the deadline. If we want to make 99.5% of all deadlines instead of 99%, we need 3.5–13.3 Mc, and we still can achieve this goal with the minimum speed as long as it is at least 266 MHz. We therefore recommend that mouse movements be scheduled with a pre-deadline speed equal to the minimum available.

To see the potential energy savings of this approach, we conducted a simulation of (only) mouse movement task processing assuming the CPU has speed range 200–600 MHz. Table 5 gives results from using the minimum speed as the pre-deadline speed and, following [LS01], the maximum speed as the post-deadline speed. We see that we can save substantial energy, on average 77.5%, by using the minimum speed as the pre-deadline speed.

Next, we consider keystrokes. We use a soft deadline of 50 ms for each task, following [Shn98]. Looking at the graphs, 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 keystroke tasks, but three of our users would suffer significant changes in deadlines made and thus presumably notice the increased response time. This suggests that there is no single average pre-deadline speed that will work well for all users for keystroke tasks. It is thus likely 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. 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 consider mouse clicks. We assume a 50 ms deadline, conservatively following [Shn98]. With mouse clicks, we find general agreement among all users: the CDF slope is relatively high at and well beyond 10 Mc. 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. A good reason not to use the maximum is that 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.

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. Mouse movements require the minimum available speed, keystrokes some intermediate speed, and mouse clicks a high speed.

4.5 Which DVS algorithm should we use for user interface events?

Lorch et al. [LS01] demonstrated that given an average pre-deadline speed and given knowledge of the probability distribution of a task's work requirements, there is a single optimal schedule that minimizes energy consumption; this the PACE schedule. However, if the designers of a DVS algorithm do not want to add the complexity of estimating the probability distribution and computing the optimal schedule, they must choose a heuristic approximation to the optimal schedule. In this section, we compare three simple DVS algorithms to determine how well they save energy relative to each other, and then we compare them to the optimal schedule computed using PACE. For this, we use simulations assuming the CPU with speed range 200–600 MHz. To compare them fairly, we use the same pre-deadline speed for all algorithms: 400 MHz for keystroke tasks and 500 MHz for mouse click tasks. We noted earlier that mouse movement events should always use the minimum available speed, so the question of which DVS algorithm to use for them is moot. We therefore ignore mouse movement events in our simulations.

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 100 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^{TM} [Kla00].
- **Past/Peg.** The pre-deadline speed is constant at 200 MHz for the first interval, then is pegged to 600 MHz. Interval length is chosen to achieve the desired average pre-deadline speed. This models the algorithm suggested in [GLF$^+$00].
- **PACE-NoClassify.** The pre-deadline speed schedule is computed by PACE using an estimate of task work distribution

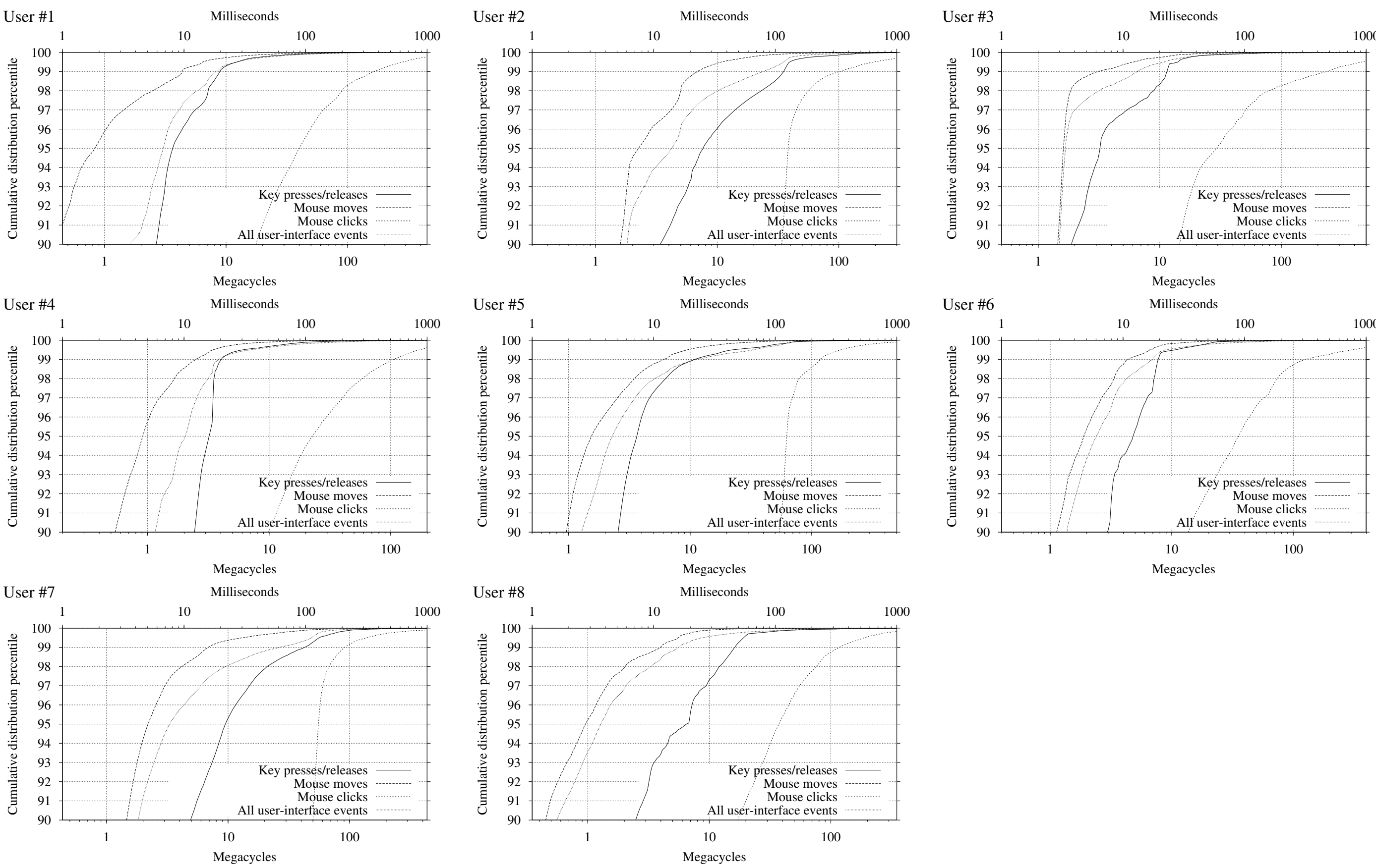


Figure 1: The cumulative distribution function of CPU time required by various user interface event types for each user, but only above the 90th percentile

derived from the most recent tasks, regardless of what category they are or what application they belong to.

We see the results in Tables 6 and 7. (These tables contain simulations of additional algorithms, which we will discuss later.)

We find that, for keystrokes, there is a nearly universal pattern among all users. Flat is always worst, followed typically by Past/Peg, followed by Stepped, followed by PACE. On average, Past/Peg reduces energy consumption by 22.8% relative to Flat, Stepped reduces energy consumption by 10.5% relative to Past/Peg, and PACE reduces energy consumption by 5.0% relative to Stepped. Thus, PACE is best, but if the complexity of PACE is undesirable, Stepped is best among non-PACE algorithms.

For mouse clicks, there is a different ordering of the algorithms and there is less difference between the algorithms. Here, Past/Peg is always worst, followed by Flat, followed by Stepped, followed by PACE. On average, Flat reduces energy consumption by 0.7% relative to Past/Peg, Stepped reduces energy consumption by 3.4% relative to Flat, and PACE reduces energy consumption by 1.5% relative to Stepped. Thus, PACE is also best for mouse clicks, but for these tasks it may not be sufficiently better than Stepped to make up for the complexity of its implementation.

4.6 Do different categories of user interface event have different CPU requirements?

Computing a schedule with PACE requires estimating the probability distribution of task work. One estimates this distribution from a sample of past tasks' work requirements; only recent tasks should be considered because, in general, distributions are non-stationary [LS01]. To better estimate the probability distribution, Lorch et al. suggest dividing tasks into groups of similar tasks with similar distributions. This way, one estimates a task's work requirement using only similar tasks' work requirements. However, they do not provide a general method for dividing tasks this way.

The question of how to divide tasks into groups of similar ones is complex. If the groups are too large, they may include too many different tasks with significantly different probability distributions, making the estimates less accurate. If the groups are too small, there may be few recent tasks in each group from which to estimate a distribution, meaning that estimates may be made from old and therefore less relevant information.

Thus, in this section, we consider whether different user interface event categories have significantly different processing requirements. To the extent they do, 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.

We will only consider the significance of differences between mean task run times. It may be that two event categories have similar means but significantly different distributions; if so, we will not detect such differences here. (In [Lor01], we examine differences between distributions, but for brevity we omit this material here.) It may also be that two means are significantly different from a statistical standpoint, but the difference is small enough to be practically meaningless. We cannot address this issue without making assumptions about CPU characteristics, so we leave this issue for Section 4.8.

Given a pair of event categories, we use a *t*-test [Jai91] to see if they have significantly different mean task lengths. This test considers the set of task lengths for each event category. It determines how much variance each set has, and from this the likelihood that the difference in the two means is not due to just this variance. 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 variance is too high.

4.6.1 Key categories

First, we determine whether there are significant processing time differences between key presses and key releases of the same key by the same user. Examining the results, presented in detail in [Lor01], we find that by far the most common case, across all keys and users, is for key press processing time to be significantly different than key release processing time. Furthermore, key presses usually take longer to process than key releases. This is not surprising, since most applications perform the action triggered by a key as soon as the key is pressed, not after the key is released.

Next, we decide whether there are significant differences between key presses of different key categories within a single application and a single user. We divided keystroke operations into several categories in a way that seemed to us to put similar keys in the same category and different keys in different categories. These categories include letter, shift-letter, ctrl-letter, spacebar, number, shift-number, miscellaneous punctuation, backspace, tab, enter, escape, home, end, del, arrow, F-key, page up/down, ctrl-F-key, ctrl-spacebar, alt-letter, modified-arrow (e.g., ctrl-arrow), Eastern-language special character, modified Eastern-language special character, and modifiers alone.[3] For space reasons, we only present the conclusions from testing for significant differences between key categories here; for the complete results, see [Lor01].

Examining the results, we find that most event category pairs show significant difference from each other, even when we require a 99% level of confidence for significance. Also, there is no pair of event categories that consistently shows no significant difference from each other from application to application. However, 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.

The situation is different for key release events. We find that for key releases 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.

Thus, it is likely better to separate key press events into many small groups than to separate them into few large groups when estimating key press task work length. For key releases, using fewer, larger groups may be better.

[3] We have only listed the top 24 here; the others all together only accounted for 0.14–1.13% of keystroke events with an average of 0.65% and only 0.41–1.94% of keystroke handling time with an average of 1.26% [Lor01].

User	No DVS		Flat		Past/Peg	Stepped	PACE-NoClassify	PACE-NoApps	PACE-Classify
1	4.577 mJ	*(99.86%)*	2.366 mJ	*(99.77%)*	1.630 mJ	1.491 mJ	1.394 mJ	1.375 mJ	1.364 mJ
2	11.621 mJ	*(98.52%)*	6.904 mJ	*(97.74%)*	6.575 mJ	5.880 mJ	5.548 mJ	5.504 mJ	5.452 mJ
3	5.973 mJ	*(99.88%)*	2.806 mJ	*(99.80%)*	1.673 mJ	1.415 mJ	1.328 mJ	1.314 mJ	1.299 mJ
4	4.978 mJ	*(99.93%)*	2.300 mJ	*(99.89%)*	0.874 mJ	0.900 mJ	0.822 mJ	0.817 mJ	0.813 mJ
5	8.380 mJ	*(99.58%)*	4.349 mJ	*(99.45%)*	2.640 mJ	2.506 mJ	2.441 mJ	2.443 mJ	2.432 mJ
6	5.550 mJ	*(99.94%)*	2.546 mJ	*(99.83%)*	1.310 mJ	1.194 mJ	1.112 mJ	1.119 mJ	1.101 mJ
7	13.515 mJ	*(98.56%)*	7.859 mJ	*(97.88%)*	7.396 mJ	6.557 mJ	6.283 mJ	6.182 mJ	6.169 mJ
8	6.685 mJ	*(99.80%)*	3.508 mJ	*(99.61%)*	3.112 mJ	2.621 mJ	2.502 mJ	2.513 mJ	2.478 mJ
Avg	7.660 mJ	*(99.51%)*	4.080 mJ	*(99.25%)*	3.151 mJ	2.820 mJ	2.679 mJ	2.658 mJ	2.638 mJ

Table 6: Average per-task energy consumption for various DVS algorithms operating on various users' keystroke events. Parenthetical information shows percent of deadlines made. All algorithms shown here other than "No DVS" use an average pre-deadline speed of 400 MHz and thus have the same performance including percent of deadlnes made.

User	No DVS		Flat		Past/Peg	Stepped	PACE-NoClassify	PACE-NoApps	PACE-Classify
1	54.22 mJ	*(93.50%)*	47.54 mJ	*(92.55%)*	48.34 mJ	46.59 mJ	45.97 mJ	45.93 mJ	45.79 mJ
2	48.56 mJ	*(95.05%)*	41.89 mJ	*(93.98%)*	42.37 mJ	40.42 mJ	39.62 mJ	39.52 mJ	39.41 mJ
3	85.12 mJ	*(94.19%)*	77.61 mJ	*(93.42%)*	78.11 mJ	75.96 mJ	75.32 mJ	75.18 mJ	75.08 mJ
4	32.96 mJ	*(96.03%)*	27.70 mJ	*(95.35%)*	26.98 mJ	25.69 mJ	25.45 mJ	25.39 mJ	25.32 mJ
5	57.12 mJ	*(90.93%)*	48.85 mJ	*(89.94%)*	49.69 mJ	47.55 mJ	46.66 mJ	46.53 mJ	46.36 mJ
6	54.66 mJ	*(94.10%)*	48.48 mJ	*(93.31%)*	48.67 mJ	47.00 mJ	46.49 mJ	46.43 mJ	46.26 mJ
7	41.52 mJ	*(93.61%)*	33.93 mJ	*(92.51%)*	34.30 mJ	31.92 mJ	31.16 mJ	31.06 mJ	30.95 mJ
8	49.62 mJ	*(93.55%)*	42.42 mJ	*(92.42%)*	42.60 mJ	40.63 mJ	39.85 mJ	39.79 mJ	39.70 mJ
Avg	52.97 mJ	*(93.87%)*	46.05 mJ	*(92.94%)*	46.38 mJ	44.47 mJ	43.81 mJ	43.73 mJ	43.61 mJ

Table 7: Average per-task energy consumption for various DVS algorithms operating on various users' mouse click events. Parenthetical information shows percent of deadlines made. All algorithms shown here other than "No DVS" use an average pre-deadline speed of 500 MHz and thus have the same performance including percent of deadlnes made.

4.6.2 Mouse click categories

We next consider whether mouse click categories are different from each other. We examine only left-down, left-up, left-double-click, right-down, and right-up, since the remaining types of mouse click account for few of the click events and little of the click handling time: 0.0–0.9% of all click events with an average of 0.4%, and 0.0–0.2% of all click handling time with an average of 0.1%.

We consider six questions:

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

See [Lor01] for the results of t-tests addressing these questions; for space reasons, we merely summarize those results here.

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

Another reason we see no difference in the case of iexplore is that we do not consider tasks that wait for I/O. Thus, if the page fetch requires network or disk I/O, we ignore the task in this analysis. This means that most mouse clicks we analyze trigger simple operations such as fetching cached pages, selecting menu items, and scrolling scroll bars.

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.

4.7 Do different applications have different CPU requirements?

The next question we ask is whether different applications exhibit different task work requirements for the same event type or category. We would generally expect this, except that Windows provides default message handlers for applications to use for messages not needing special treatment, so applications using the same default message handler might exhibit similar behavior for certain user interface message types and categories.

Detailed tables with our results are in [Lor01]. The conclusion we draw from these results is that different applications have very different CPU requirements, even when handling the same type of user interface event. This holds for mouse movements, keystrokes, and mouse clicks, despite the fact that the operating system provides default handlers for many of these events. This result for mouse clicks is especially notable because mouse click times have high variance, and it requires either a great deal of data or a large difference in effect for a statistical test to distinguish between two highly variable data sets.

4.8 Should PACE separate tasks by category and application?

Merely demonstrating a significant difference among tasks of different categories and different applications is not sufficient to show that separating tasks by category and application is a good idea for PACE. It may be that the difference is small enough that PACE would actually do worse by keeping tasks of different categories and/or applications 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. It may be then that using smaller sets of tasks, even if it leads to greater similarity within sets, can lead to poorer estimates of task work. (More categories may also result in more overhead.)

To determine whether the significant differences we have seen translate into actual improvement for PACE from using category information, we need to assume certain CPU characteristics. Therefore, we conducted trace-driven simulations assuming a CPU with speed range 200–600 MHz. We compare the following three algorithms, each of which uses PACE to compute the pre-deadline speed schedule using an estimate of task work distribution:

- **PACE-NoClassify.** Derives probability distribution from the most recent tasks, regardless of their category or application.
- **PACE-NoApps.** Derives probability distribution from the most recent tasks of the same category, regardless of their application.
- **PACE-Classify.** Derives probability distribution from the most recent tasks of the same category and same application.

The results are in Tables 6 and 7.

For keystrokes, 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. This demonstrates it is worthwhile to separate keystroke tasks according to both category and application.

For mouse clicks, PACE-Classify reduces energy consumption by 0.5% relative to PACE-NoClassify, indicating that it is not detrimental to separate mouse clicks by category and by application, and indeed it is marginally useful to do so. On the other hand, PACE-NoApps, which separates mouse click events into categories but ignores application information, always does worse than PACE-Classify; we lose about 60% of the effectiveness of categorizing by ignoring application information. This shows that it is worthwhile, albeit marginally, to separate mouse click tasks according to both category and application.

5 Future Work

5.1 Other operating systems

Our results are based on traces of Windows NT/2000, but our questions are applicable to any operating system. Windows is not alone in its ability to allow monitoring of user interface tasks; in particular, Flautner et al. [FRM01] demonstrated it is possible in Linux.

Therefore, an important avenue of future work is investigating the answers to our questions on other operating systems. Some of our results would likely not change on a different operating system, as long as the application workload was similar. For instance, intuitively, the operating system is irrelevant to the question of whether there is significant difference between tasks triggered by different categories of user interface event. However, some results, such as the applicability of our heuristic to determine when user interface tasks end, would likely be highly dependent on the user interface model the system specifies.

5.2 Other mobile devices

Since we answered our questions about user interface events and DVS using traces of PC's, our results are most applicable to laptops and similar devices such as tablet PC's. However, our questions will soon be important on smaller computing devices such as handheld personal digital assistants. Manufacturers will soon release DVS-capable chips for these devices [Int00], and these computing devices also typically have user interface driven workloads. Indeed, our questions are likely to be even more important for them than for laptops since on such devices usually almost all work is done in response to user requests. This is because there is limited battery energy to use for constant background tasks and limited network connectivity that would lead to tasks triggered by packet arrivals. Future research is needed to determine how best to use user interface information to schedule voltage and speed on such devices. We believe similar methodology to that used in this paper could be used for such work.

5.3 Implementation

We have described a heuristic for evaluating when tasks begin and end, prescribed methods for classifying them into groups of similar tasks, and described the best ways to use this information to schedule speed and voltage. Important future work is to implement these ideas on a real system capable of DVS and evaluate how well our answers translate into actual battery lifetime improvements.

6 Conclusions

In this paper, we analyzed months-long traces of user operations to answer questions about using user interface events to estimate task information for DVS algorithms.

We found that, on average, we can attribute only 20.3% of CPU time to processing user interface events. Thus, task information collected from user interface events gives an incomplete picture of the usage of the CPU, and other methods for detecting task information should be used. A promising avenue is detecting tasks and deadlines from timer events, since 35.1% of CPU time is spent responding to such events.

Since most techniques for inferring the completion of user interface tasks are complex and cause high overhead, we evaluated the possibility of considering a user interface task complete when the system is idle with no I/O or when the application receives its next user interface event. We found that, excepting two applications for which this heuristic seems to work badly, this technique is rather accurate, prematurely estimating the termination of fewer than 0.8–5.3% of all tasks. More research is needed to determine how to estimate task termination for the aberrant applications.

When a task waits for I/O, a DVS algorithm must treat it specially, so we analyzed how frequently user interface tasks wait for I/O. We found that I/O waits 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 disconnected from any network, the percentage is only 0.2–1.0%. 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 account for I/O time when scheduling the CPU speed during mouse click processing.

We then examined task work distributions to see how DVS algorithms should choose average pre-deadline speeds. We found that different event types should use different approaches to choosing such speeds. Mouse movements generally require just the minimum available speed. Indeed, one simulation shows that using such a speed makes 99.6–99.9% of all possible deadlines while reducing energy consumption by 68.3–84.0% with an average of 77.5%. 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.

Next, we performed experiments to compare DVS algorithms. We found that among non-PACE algorithms, Stepped, which gradually increases speed as a task progresses, does better than Flat and Past/Peg. But, PACE is even better than Stepped: for keystrokes, PACE is a 5.0% improvement over Stepped; for mouse clicks, PACE is a 1.5% improvement over Stepped.

To evaluate whether PACE should separate tasks into categories and estimate the probability of a task's work only from tasks of the same category, we next evaluated whether different event categories trigger tasks with significantly different mean work requirements. We found, for key presses, 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 found that each pair of categories we considered shows significant differences at the 99% level for at least some applications. We concluded that category differences often lead to significant differences between tasks.

We also evaluated whether different applications show significant differences in mean task work. We found that, for all three user interface event types, applications frequently show differences large enough to permit 99% confidence in their significance.

Finally, we performed simulations to see whether the above two results translate into actual energy savings, i.e., to see whether PACE actually saves energy by separating tasks by category and application. We found that for both keystrokes and mouse clicks, PACE gets better, not worse, when it considers tasks of different applications and different categories separately. The improvement is 1.5% for keystrokes and 0.5% for mouse clicks. This validates our conclusions about the differences between applications and categories.

In conclusion, the main fact we discovered from our traces is that there are significant differences between tasks arising in different circumstances. Differences in the user, the application, the user interface event type, and even the category of event 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 by monitoring details about the user interface tasks presented to applications.

References

[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.

[ES00] Yasuhiro Endo and Margo Seltzer. Improving interactive performance using TIPME. In *Proceedings of the 2000 ACM SIGMETRICS Conference*, pages 240–251, June 2000.

[EWCS96] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. Using latency to evaluate interactive system performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–199, 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.

[FURM00] Krisztián Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge. Thread-level parallelism and interactive performance of desktop applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 129–138, November 2000.

[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.

[Int00] Intel Corporation. Intel XScale™ microarchitecture product brief. On the World Wide Web at http://developer.intel.com/design/intelxscale/xscaleproductbriefweb.pdf, September 2000.

[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.

[Kla00] Alexander Klaiber. The technology behind Crusoe™ processors. White paper, Transmeta Corporation, January 2000.

[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.

[LCB+98] Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 27–38, June 1998.

[Lor01] Jacob R. Lorch. *Operating Systems Techniques for Reducing Processor Energy Consumption*. PhD thesis, Computer Science Division, EECS Department, University of California at Berkeley, 2001.

[LS00] Jacob R. Lorch and Alan Jay Smith. The VTrace tool: building a system tracer for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, October 2000.

[LS01] Jacob R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the 2001 ACM SIGMETRICS Conference*, pages 50–61, June 2001.

[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.

[PLS01] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the Seventh ACM International Conference on Mobile Computing and Networking (MOBICOM 2001)*, July 2001.

[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.

[Shn98] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, 1998.

[WE93] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, MA, 1993.

[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.

[ZS00] Min Zhou and Alan Jay Smith. Tracing Windows 95. *Journal of Microprocessors and Microsystems*, 24(7):333–347, November 2000.