Using User Interface Event Information in Dynamic Voltage Scaling Algorithms[*]

Jacob R. Lorch
Microsoft Research
1 Microsoft Way
Redmond, WA 98052
lorch@microsoft.com

Alan Jay Smith
University of California, Berkeley
EECS Department, CS Division
Berkeley, CA 94720-1776
smith@eecs.berkeley.edu

Abstract

Increasingly, mobile computers use dynamic voltage scaling (DVS) to reduce CPU voltage and speed and thereby increase battery life. To determine how to change voltage and speed when responding to user interface events, we analyze traces of real user workloads. We evaluate a new heuristic for inferring when user interface tasks complete and find it is more efficient and nearly as effective as other approaches. We compare DVS algorithms and find that for a given performance level, the PACE algorithm uses the least energy and the Stepped algorithm uses the second least. We find that different types of user interface event (mouse movements, mouse clicks, and keystrokes) trigger tasks with significantly different CPU use, suggesting one should use different speeds for different event types. We also find differences in CPU use between categories of the same event type, e.g., between pressing spacebar and pressing enter, and between events of different applications. Thus, it is better to predict task CPU use based solely on tasks of the same category and application. However, energy savings from such improved predictions are small.

Keywords—Workload characterization and generation, modeling and simulation, response time, dynamic voltage scaling, energy management, power management.

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 change CPU speed without rebooting to save energy while running at low speeds. To take advantage of this, a DVS algorithm must know when it is worthwhile to trade off speed for energy savings.

Research suggests that DVS algorithms should consider *task information*, i.e., what tasks the system is working on, how much CPU time those tasks use, and the deadlines for those tasks [4, 10, 13, 14]. Unfortunately, most modern systems lack an interface allowing applications to specify such task information. Even if such an interface were added, existing applications could not use it, and writers of many new applications would be unwilling or unable to give it accurate information. Thus, we [10] and others [4] 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 below 50–100 ms [15]. Thus, a DVS algorithm can infer a task with deadline 50–100 ms when a user interface event arrives. The algorithm must also guess when the task completes, using a heuristic such as Flautner et al.'s [4].

Researchers have studied DVS algorithms using short benchmarks, each featuring a user using an application in a specified manner. However, we feel that to design DVS algorithms applicable to any user running any application, it is beneficial to study how users act "in the wild" and thereby answer questions benchmarks cannot answer. So, we collected several months' worth of trace data from each of eight users running Windows NT/2000, then used them to answer the following six questions.

1. What fraction of tasks result from user interface events? On average, only 5.6%–43.7% of CPU time is directly caused by user interface events. Thus, a DVS algorithm must consider other CPU use triggers, such as timers.

2. Is there an efficient heuristic to find when user interface tasks end? We propose considering a user interface task complete when all threads are blocked or when the next user interface event occurs. This technique requires far less intrusive monitoring than Flautner et al.'s [4] and our mea-

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

surements show it is nearly as accurate.

3. How often does handling user interface events require waiting for I/O? This is relevant because I/O time does not scale with CPU speed. We find that 1.3% of these tasks wait for I/O, and different event types incur I/O at different rates.

4. How quickly should we run the CPU to achieve good response time goals for user interface tasks? We find that the minimum CPU speed is sufficient to achieve this for tasks triggered by mouse movements. Keystroke tasks require more speed, and mouse clicks require the most.

5. Which DVS algorithms work best for user interface tasks? The PACE algorithm [10], which optimally adapts the CPU speed according to the expected distribution of the next CPU task, saves the most energy. The Stepped algorithm, which always uses the same steadily increasing CPU speed schedule, performs nearly as well.

6. Is there significant enough difference between categories of user interface event (e.g., between pressing spacebar and enter), or between user interface events of different applications, that a DVS algorithm should handle them differently? There are significant differences in the CPU use of tasks that differ in these ways. However, these differences are small, so DVS algorithms should only consider them if they can do so easily and efficiently.

To our knowledge, we are the first to answer questions 1 and 6, and the first to answer the others using months of traces rather than short traces or artificial benchmarks.

This paper has the following structure. Section 2 gives background and related work. Section 3 describes the traces we collected and our methodology for processing 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. Finally, Section 5 concludes.

2 Background and Related Work

In CMOS circuits, the dominant component of power consumption is proportional to $V^2 f$, where V is voltage and f is frequency [18, p. 235]. Thus, energy per cycle is proportional to V^2. The maximum safe CPU frequency decreases roughly linearly with decreasing voltage. Thus, the system can reduce processor energy consumption by reducing the voltage, but this requires running more slowly.

Weiser et al. [17] and Chan et al. [1] proposed the first DVS algorithms. Each divides time into fixed-length intervals. It predicts the CPU utilization of the next interval based on observations of the CPU utilization of previous intervals, then sets the speed based on that prediction.

Later researchers pointed out the flaws inherent in such interval-based strategies, e.g., that the arbitrarily chosen interval boundaries do not correspond to real deadlines [5, 12]. Thus, recent proposed strategies have been task-based, i.e., they use task information, especially task deadlines. A deadline may be *hard*, meaning it must be made, or it may be *soft*, meaning the task may miss it but doing so impacts perceived performance negatively. Pillai et al. [13] developed a real-time DVS scheduler for embedded operating systems that derives an energy-efficient speed schedule from information about ongoing periodic tasks. Pouwelse et al. [14] modified a video player to set the CPU speed for each frame after predicting each frame's CPU requirements.

Flautner et al. [4] proposed deriving task information automatically from applications. They considered two task types: interactive and periodic. Interactive tasks are triggered by a user-initiated action, typically a user interface event. Periodic tasks are triggered by a periodic event; an event is considered periodic if the times between the last n events have a small variance. They also presented the following heuristic for inferring a task's completion. The thread receiving 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 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.

In earlier work [10], we demonstrated that the theoretically optimal speed schedule for a task depends on the probability distribution of that task's CPU requirement and desired performance. Desired performance is expressed as a required number of CPU cycles before the deadline, or, equivalently, an average pre-deadline speed. We gave a method for producing an optimal speed schedule given an estimate of the task work distribution; we called this method PACE (Processor Acceleration to Conserve Energy). We suggested estimating the distribution from a sample of recent similar tasks' CPU use, but we did not describe how the system should identify tasks as being similar.

For certain tasks not triggered by user interface events, individual task deadlines are less important than overall task completion rate. For example, media players typically use buffers, so different frames may take different amounts of time as long as the aggregate processing rate is sufficient. For workloads like this, methods such as Simunic et al.'s [16] are more appropriate than those we consider here.

3 Methodology

3.1 Traces

We used VTrace [9] to trace people in normal operation on their desktop PC's running Windows NT or 2000. VTrace collects time-stamped records describing various interesting system and application events. To limit trace volume, VTrace only collects the full set of events for 90 minutes at a time, then pauses for 2 hours. Also, it stops

User	1	2	3	4	5	6	7	8
Trace period	8 months	7 months	4 months	15 months	3 months	19 months	2 months	9 months
Time traced	435.8 hr	504.3 hr	83.0 hr	212.2 hr	134.9 hr	202.6 hr	106.9 hr	215.1 hr
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 3	Pentium 2	Pentium 3	Pentium Pro	Pentium 3	Pentium 2	Celeron	Pentium 2
Memory size	128 MB	unreported	96 MB	128 MB	256 MB	128 MB	256 MB	64 MB
Windows OS	NT 4 SP 6	NT 4 SP 4	NT 4	NT 4 SP 3	2000 SP 1	NT 4 SP 4	2000 SP 1	NT 4 SP 4

Table 1: Trace information for all users. Note that "time traced" indicates time full tracing was on.

full event tracing when the user is idle for 10 minutes. In this paper, we only use data collected during full tracing.

We used traces from eight users. Table 1 contains data about these users' machines as well as summary information about the workloads traced from these users. The self-reported characteristics of the users are as follows.

User 1, a computer science graduate student, mainly runs an X server, but also uses his machine for mail, web browsing, software development, and office applications. User 2, another CS graduate student, uses his machine primarily for mail, web browsing, software development, and office applications. User 3, the CTO of a computing-related company, uses his machine for system administration, office applications, web browsing, and mail. User 4, a university system administrator, uses his machine for networking and system administration tasks; he primarily runs an X server, mail client, web browser, and Windows NT system administration tools. User 5 did not report his profession; judging from his applications, he seems to use his machine largely for recreation. User 6, a police captain, uses his machine primarily for groupware and office suite applications. User 7, a software developer in Korea, uses his machine primarily for software development, web browsing, and mail. User 8, a crime laboratory director, uses his machine primarily for groupware and office suite applications.

3.2 Tasks

Many of the questions we consider concern *tasks*. Usually, people loosely define a task as a sequence of operations serving some end goal. However, we need to more precisely define a task so we can determine when one begins and ends in a trace. Our approach is similar to Flautner et al.'s [4].

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 of its consecutive events 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 it has an incoming network packet,
- 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 tracing. (When this happens, threads may be running or ready to run. We cannot know what events these threads are responding to, as they occurred when full tracing was not on. This event type is a proxy for such unknown events.)

If, during a response, a thread causes another event, we say that event is *dependent* on the event that generated the response. For example, if a thread responding to a message starts another thread, the event of that thread starting is considered dependent on the message. As an exception, if a response causes a time-delayed event, e.g., by starting a timer, we do not consider the time-delayed event to be dependent on the event that generated the response. Our rationale is that if a process lets time pass before work continues, we assume this work is not a critical part of the response.

We call an event that is not dependent on any other a *trigger event*. Such an event is the root of a tree in which each event has its dependent events as children. 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. We need them to make reasonable inferences about tasks without unnecessarily intrusive tracing techniques. It is possible we will miss the true trigger event for a task, and it is possible 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, or a mouse click. A *user interface task* is a task triggered by a user interface event. Note that such a task includes all time the system spends responding to the event, not just the time to handle the device interrupt. For example, if the event is due to the user hitting enter in a spreadsheet, the task includes the time to perform any consequent spreadsheet calculations.

User interface events are divided into *types* based on the Windows message type associated with the event [11]. Types include key press, key release, right mouse button down, left mouse button double-click, and others. Events are further divided into *categories* based on the particular key pressed or released, the modifiers held down during event delivery, and what type of window component a mouse click was for (e.g., the border, the menu, the maximize button, etc.). For instance, "Ctrl-F4 pressed" and "enter key released" are categories of keystroke tasks.

We define an *application* as a set of processes with the same name, ignoring extensions such as .exe or .bat.

3.4 Aggregation

When presenting an average across all users, we will scale all users' results to the same level of activity. In this way, the results will reflect a broad set of users, not just those with a large amount of activity or willing to be traced for more time. For example, if there were two users, one with 4 million keystrokes taking an average of 5 ms to process and one with 1 million keystrokes taking an average of 4 ms, we would report the average keystroke processing time as 4.5 ms, not 4.8 ms.

4 Analyses

In this section, we analyze the trace characteristics to answer the questions we posed in Section 1. Most of our analyses are independent of CPU characteristics, but some require simulating a particular CPU. In these cases, we assume the CPU can run between 200 MHz and 600 MHz, and that power is proportional to speed cubed [17] and equal to 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. Table 2 shows the non-idle CPU time triggered by the various types of trigger event. We see that CPU time due to user interface events ranges from 5.6%–43.7%, with an average of 20.3%. This is curiously low, as most Windows applications are user interface applications, and most apparent application work 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. CPU time triggered by timer messages is 6.8–68.1%, on average 35.1%. An application typically uses such messages for a periodic operation. Some of this activity has important implied deadlines, such as media playback, and some does not, such as blinking the cursor.

User 7 spends substantial time (43.0%) on tasks already running when VTrace began a tracing session. This suggests a lot of time in long-running processes, and looking at the application names we see they tend to be server processes. We do not expect such server workloads on laptops, so this value does not reflect typical portable computers.

The largest remaining component of CPU time, accounting for 21.9% of it on average, is triggered by threads completing a wait on a non-timer object that VTrace did not see signaled by a thread. In other words, it is spent working on tasks whose purpose VTrace could not determine. Most likely, a thread caused this event in some way VTrace could not detect. For example, when a thread posts an entry to a shared queue, this implicitly signals the queue object to wake any waiting thread. In Section 4.2, we will consider an alternate approach to tracking event duration that can account for such implicit event signaling.

In conclusion, we see the CPU spends only about 20.3% of its time responding to user interface events. One reason this figure is low is limitations of our methodology, which cannot identify the cause of 21.9% of CPU time. Another reason is server activity that laptops typically lack. Besides user interface events, timer events also trigger substantial CPU time; these events can also be a source of tasks with deadlines. We conclude that tasks detected only via inference from user interface events will reflect only some of the CPU's work; a mechanism such as Flautner et al.'s [4] for detecting periodic tasks may help infer deadlines for some of the remaining work.

4.2 Can we detect task completion efficiently?

In this subsection, we will explain and test an efficient approach for detecting task completion. Such detection is important for two reasons. First, when all tasks are complete, there is no urgency, so the system can use the minimum CPU speed. Second, to estimate task work distribution we need to know how long past tasks took, which requires knowing when they completed.

In Section 3, we described our method for estimating when a task is complete; it is similar to Flautner et al.'s [4]. Unfortunately, this approach is not well suited to a real online algorithm. It requires modifying or interposing many

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: Non-idle CPU time triggered by each event type. See Section 3.2 for event type descriptions.

system calls to keep track of thread communication and how threads block; these modifications can create high system overhead. Also, it cannot be complete, since there are ways for threads to communicate with each other that cannot be efficiently tracked, e.g., writing to shared memory.

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 implementation; in particular, it requires no 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 it considers as part of a task all processing done by unrelated threads, 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 CPU needs of similar tasks.

To evaluate this method's potential inaccuracy, we determined how many user interface tasks continue past the point when the system goes idle with no I/O ongoing, and how many continue past the point when the next user interface event occurs. Table 3 contains these results.

Breaking down this information by application (see [8] for details), we see that two applications, exceed and java, have a substantial number of user interface tasks that go beyond the next idle time or the next user interface task. However, almost all other applications do not show this behavior. We hypothesize that this is because the outlier applications signal objects that cause 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. We support this hypothesis by repeating the analysis without considering object signaling to cause event dependency (see bottom of Table 3). Here, we find applications have a much lower number of tasks that continue past the next idle and user interface event arrival time.

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: User interface tasks continuing past system-idle or next user interface event

Ignoring the two outlier 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%. Furthermore, there is substantial overlap between these two types of persistent tasks, so the total number of both types is never more than 5.3%. It is likely that many of these tasks only appear long due to our base heuristic failing to detect their true end times, so if our heuristic were better we would see fewer tasks extending beyond either boundary.

Thus, our simplified approach to identifying when tasks

User	Key press/release			Mouse move			Mouse click			All user interface events		
	disk	*network*	**either**	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%**	0.3%	*3.2%*	**3.5%**
2	0.7%	*5.4%*	**5.9%**	0.2%	*0.4%*	**0.6%**	5.0%	*2.7%*	**6.8%**	0.5%	*1.1%*	**1.4%**
3	0.7%	*0.7%*	**1.4%**	0.3%	*0.2%*	**0.4%**	12.5%	*2.9%*	**14.5%**	0.5%	*0.3%*	**0.7%**
4	0.1%	*5.4%*	**5.5%**	0.1%	*1.1%*	**1.2%**	6.1%	*10.0%*	**14.6%**	0.3%	*2.2%*	**2.4%**
5	0.2%	*0.1%*	**0.3%**	0.1%	*0.1%*	**0.2%**	3.1%	*2.7%*	**5.1%**	0.2%	*0.2%*	**0.3%**
6	0.4%	*0.2%*	**0.6%**	0.2%	*0.1%*	**0.3%**	10.0%	*5.1%*	**13.3%**	0.5%	*0.2%*	**0.6%**
7	2.6%	*0.4%*	**2.7%**	0.5%	*0.1%*	**0.5%**	10.0%	*1.5%*	**10.5%**	1.0%	*0.2%*	**1.0%**
8	0.8%	*2.5%*	**3.1%**	0.3%	*0.1%*	**0.4%**	17.2%	*7.9%*	**18.1%**	0.7%	*0.3%*	**0.8%**
Avg	0.7%	*2.5%*	**3.1%**	0.2%	*0.4%*	**0.6%**	9.0%	*5.3%*	**12.3%**	0.5%	*1.0%*	**1.3%**

Table 4: Percent of user interface tasks requiring waiting for I/O

end will likely work well for most applications. However, for a small set of applications it may produce wrong results, considering tasks 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, since I/O power and time do not scale with CPU voltage and speed. Table 4 shows how many user interface tasks require I/O of two kinds: disk and network. For breakdowns by application, see [8]. We find that 0.3–3.5% of user interface tasks wait for I/O, with an average of 1.3%. Just looking at the disk, only 0.2–1.0% of user interface tasks wait for I/O, with an average of 0.5%.

Part of the reason for infrequent I/O is that mouse movements are frequent 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. Thus, 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%.

Some applications use I/O far more often than the average. 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 determine which applications require substantial I/O and treat them specially.

Developing algorithms to deal with tasks requiring I/O is beyond the scope of this paper, so in subsequent analyses we restrict consideration to events that do not wait for 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. Average pre-deadline speed is an important “knob” in such tuning, since it determines which deadlines are met: a task misses its deadline if the CPU cycles it needs exceed the pre-deadline CPU cycles. Thus, the question of what average pre-deadline speed gives good performance applies to any DVS algorithm.

We seek a pre-deadline speed satisfying a large percentage of task deadlines, such that running at higher speeds does not substantially increase this percentage. For this, we examine the cumulative distribution function of task CPU needs on a logarithmic scale, as in Figure 1. Here, the slope at any point reflects how much the deadline performance improves for a constant speed increase factor. Our approach for picking a good average pre-deadline speed is to find a point above which the slope is low; often, this appears as a “knee” in the curve. Dividing this point by the deadline gives a pre-deadline speed above which a given speed increase does not improve performance much.

Figure 1 suggests we should use a different average pre-deadline speed for mouse movements, mouse clicks, and keystrokes due to their different CPU needs. We now consider what average speed seems best for each event type.

First, we consider mouse movements. Here, the 99.5th percentile of CPU use is 3.5–13.3 Mc (million cycles), depending on user. Assuming a soft deadline of 50 ms, as Endo et al. [3] suggest, we see that 266 MHz is sufficient for all users to complete 99.5% of tasks within this deadline. Even at 200 MHz, all users complete 99% of tasks within the deadline. Future CPU’s are unlikely to have speeds lower than 200 MHz, so we recommend that mouse movements simply use the minimum pre-deadline speed available. To see the potential energy savings of this approach, we simulated mouse movement task processing assuming a CPU with speed range 200–600 MHz. We found we could save substantial energy, 68.3–84.0% with an average of 77.5%, by using the minimum pre-deadline speed.

Next, we consider keystrokes, for which we assume a soft deadline of 50 ms, following Shneiderman [15]. We find an interesting dichotomy among users. Users 1, 3, 4, 5, and 6 show relatively low CDF slopes beyond about 10.5 Mc, but users 2, 7, and 8 show much steeper CDF slopes above 10.5 Mc. In other words, five users would

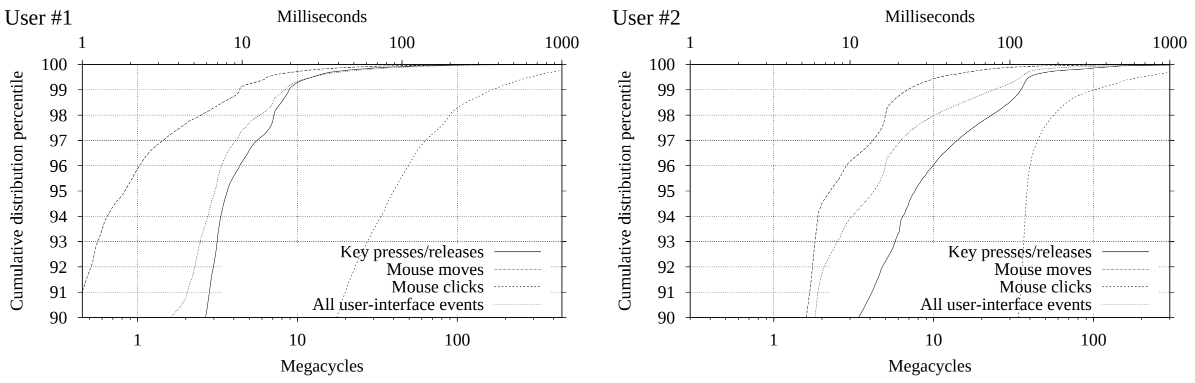

Figure 1: The cumulative distribution function of CPU time required by various user interface event types. For space reasons, only users 1 and 2 are shown, and only those parts of the graphs above the 90th percentile. For the rest, see [8, pp. 180–181].

probably not mind a pre-deadline speed of 210 MHz, but three users would likely notice increased response time at that speed. This suggests that no single average pre-deadline speed will work well for all users, so a DVS algorithm should monitor deadlines and dynamically adjust the pre-deadline speed to achieve reasonable performance.

Finally, we consider mouse clicks. For these tasks, we assume a 50 ms deadline, conservatively following Shneiderman [15]. All users show a relatively high CDF slope at and well beyond 10 Mc. In other words, increasing the CPU speed substantially reduces deadlines missed and thus improves user-perceived response time. Thus, it seems that the average pre-deadline speed for these tasks should be high, though it probably can be below 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 save subtantial energy, as it means the task may start at a low speed that may be sufficient to complete the entire task.

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

In earlier work [10], we showed how to compute an optimal speed schedule, given an average pre-deadline speed and the probability distribution of a task's CPU needs; we called this the PACE schedule. However, estimating probability distribution and computing the PACE schedule is complex, so in some cases it would be better to use a simpler DVS algorithm that saves nearly as much energy. In this section, we compare the energy savings for three simple DVS algorithms and for PACE. For this, we use simulations assuming the CPU with speed range 200–600 MHz. To make comparisons fair, we use the same pre-deadline speed for all algorithms: 400 MHz for keystroke tasks and 500 MHz for mouse click tasks. We ignore mouse movement events since, as we noted earlier, they should use the minimum speed regardless of DVS algorithm.

The three non-PACE algorithms we consider are:

- **Flat.** The pre-deadline speed is constant.
- **Past/Peg.** The pre-deadline speed is 200 MHz for the first interval, then goes up to 600 MHz thereafter. Interval length is chosen to achieve the desired average pre-deadline speed. This models the algorithm suggested by Grunwald et al. [5].
- **Stepped.** The pre-deadline speed begins at 200 MHz and goes up 100 MHz after each interval. Interval length is chosen to achieve the desired average pre-deadline speed. This models algorithms such as that in Transmeta's LongRun^{TM} [7].

Results are in Table 5. These include four different variants of PACE; we will discuss what these variants are later, in subsection 4.6.3. For now, we consider only the best PACE variant, PACE-CA.

For keystrokes, we find a nearly universal pattern among users. PACE is always best, Flat is always worst, and for all but one user Stepped is better than Past/Peg. On average, Past/Peg reduces energy consumption by 30.4% relative to Flat, Stepped reduces it by 9.1% relative to Past/Peg, and PACE reduces it by 7.3% 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 among, and less difference between, the algorithms. Here, PACE is always best, followed by Stepped, and for all but one user Flat is better than Past/Peg. On average, Flat reduces energy consumption by 0.5% relative to Past/Peg, Stepped reduces it by 3.8% relative to Flat, and PACE reduces it by 2.8%

User	Task	No DVS		Flat		P/P	Step	PACE-N	PACE-A	PACE-C	PACE-CA
1	Key	4.577	*(99.86%)*	2.366	*(99.77%)*	1.630	1.491	1.394	1.393	1.367	1.350
2	Key	11.621	*(98.52%)*	6.904	*(97.74%)*	6.575	5.880	5.548	5.522	5.506	5.442
3	Key	5.973	*(99.88%)*	2.806	*(99.80%)*	1.673	1.415	1.328	1.320	1.310	1.289
4	Key	4.978	*(99.93%)*	2.300	*(99.89%)*	0.874	0.900	0.822	0.820	0.818	0.812
5	Key	8.380	*(99.58%)*	4.349	*(99.45%)*	2.640	2.506	2.441	2.438	2.441	2.426
6	Key	5.550	*(99.94%)*	2.546	*(99.83%)*	1.310	1.194	1.112	1.110	1.118	1.101
7	Key	13.515	*(98.56%)*	7.859	*(97.88%)*	7.396	6.557	6.283	6.276	6.179	6.158
8	Key	6.685	*(99.80%)*	3.508	*(99.61%)*	3.112	2.621	2.502	2.499	2.511	2.478
Avg	Key	7.660	*(99.51%)*	4.080	*(99.25%)*	3.151	2.820	2.679	2.672	2.656	2.632
1	Click	54.22	*(93.50%)*	47.54	*(92.55%)*	48.34	46.59	45.97	45.90	45.59	45.42
2	Click	48.56	*(95.05%)*	41.89	*(93.98%)*	42.37	40.42	39.62	39.54	39.39	39.28
3	Click	85.12	*(94.19%)*	77.61	*(93.42%)*	78.11	75.96	75.32	75.28	74.42	74.27
4	Click	32.96	*(96.03%)*	27.70	*(95.35%)*	26.98	25.69	25.45	25.41	24.80	24.73
5	Click	57.12	*(90.93%)*	48.85	*(89.94%)*	49.69	47.55	46.66	46.59	46.32	46.17
6	Click	54.66	*(94.10%)*	48.48	*(93.31%)*	48.67	47.00	46.49	46.39	46.35	46.11
7	Click	41.52	*(93.61%)*	33.93	*(92.51%)*	34.30	31.92	31.16	31.09	30.93	30.85
8	Click	49.62	*(93.55%)*	42.42	*(92.42%)*	42.60	40.63	39.85	39.82	39.54	39.42
Avg	Click	52.97	*(93.87%)*	46.05	*(92.94%)*	46.38	44.47	43.81	43.75	43.41	43.28

Table 5: Average per-task energy consumption in mJ (and, in parentheses, percent of deadlines made) for various DVS algorithms operating on various task traces

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 its complexity.

It is interesting that Stepped saves almost as much as PACE, since PACE adapts the speed schedule to CPU usage, while Stepped uses the same schedule for every task. This suggests that this particular schedule, i.e., a steadily increasing CPU speed, is well suited to the CPU usage patterns seen in most real user interface tasks.

4.6 Should a DVS algorithm distinguish between user interface events from different categories and applications?

To compute a schedule, PACE must estimate the probability distribution of a task's CPU needs. It makes this estimate using a sample of past tasks' CPU needs, weighting recent tasks more heavily. In earlier work, we suggested dividing tasks into groups of similar tasks and keeping a separate sample for each group; this way, the estimate for a task is based only on similar tasks and thus should be more accurate [10]. However, we did not provide or evaluate methods for dividing tasks this way. In this section, we do so.

Dividing tasks into groups of similar ones is difficult. If groups are too large, they can have too many tasks with significantly different work distributions, decreasing estimation accuracy. If groups are too small, there may be few recent tasks in each group, so estimates may be made from old and therefore less relevant information.

4.6.1 Does distinguishing tasks by category and application improve prediction of task length?

First, we see whether distinguishing tasks by category and application improves or worsens predictions of task length, since we expect a sampling method that improves prediction of task length to also improve prediction of overall task distribution. Later, in subsection 4.6.3, we will validate this expectation by evaluating how much energy these predictors allow PACE to save.

We consider four ways to predict task length: no classification (N), classification by application (A), classification by category (C), and classification by category and application (CA). N considers any two tasks similar; A considers two tasks similar if they have the same application; C considers them similar if they have the same category; CA considers them similar if they have the same category and application. Given n earlier similar tasks, the ith most recent of which has length L_i, each predictor predicts the next task will have length $\frac{\sum_{i=1}^{n} \alpha^i L_i}{\sum_{i=1}^{n} \alpha^i}$, where $\alpha = 0.95$. Earlier research showed this value of α worked well [10].

We compare two predictors using *paired observations* [6, p.209]. For each task, we compute the difference between the absolute prediction error for one predictor and that for the other predictor. If the mean of these differences over all tasks is less than zero and the 99% confidence interval about this mean does not include zero, we say the first predictor is significantly better than the other.

Figure 2 shows mean absolute predictor error for

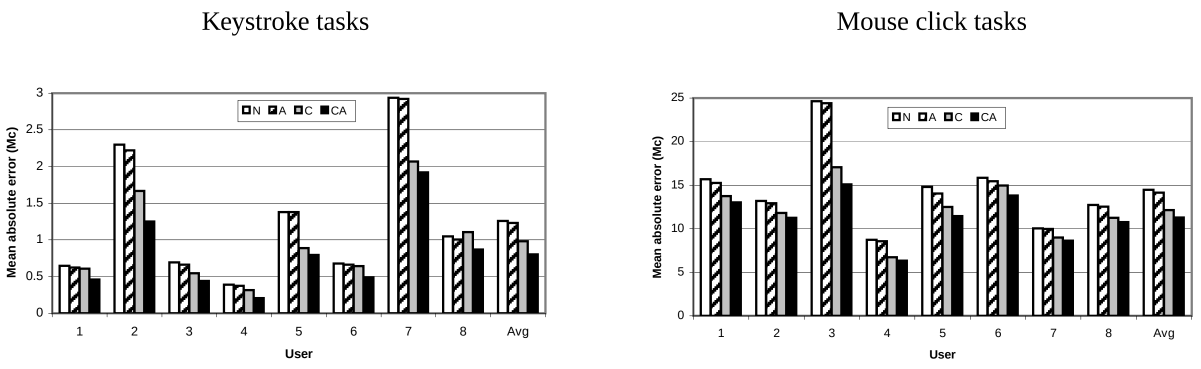

Figure 2: For each user trace, the mean absolute error of each predictor

keystroke and mouse click tasks for each user trace and each predictor. In general, CA has the least error, C the second least, A the third least, and N the highest error. This order also holds for *median* predictor error, not shown here for space reasons. Even restricting ourselves just to what we can say with 99% confidence, CA is still significantly better than each other predictor for each trace.

So, separating tasks by category and application significantly improves task length prediction. We explore whether it improves PACE's energy savings in subsection 4.6.3.

4.6.2 Are some task categories so similar they should be considered equivalent?

Some user interface event categories are so similar that we might expect them to trigger tasks of similar lengths. For example, we might expect most applications to use the same amount of CPU time to process a plus keypress as a minus keypress. If this is so, we can cluster multiple such categories into the same group and thereby improve predictors by having them treat tasks of the same category group as equivalent. In this subsection, we discuss and evaluate two methods for performing such clustering.

Our first method is based on intuition about which categories are similar. We cluster keys into these groups: letter, shift-letter, ctrl-letter, space, number, shift-number, misc. punctuation, backspace, tab, enter, escape, home, end, del, arrow, F-key, page up/down, ctrl-F-key, ctrl-space, alt-letter, modified-arrow (e.g., ctrl-arrow), Eastern-language character, modified Eastern-language character, modifiers alone, and other. We cluster mouse clicks into: left button down, left button up, left button double-click, right button down, right button up, and other.

Our second method is a *global greedy agglomerative algorithm*, as defined by Cutting et al. [2]. We begin with a collection of groups of categories, each group containing a single category. We then randomly pick two groups and see if combining them into one group improves the total absolute prediction error. If it does so for each user's trace, we combine the groups. We repeat this step until we have tried combining every pair of groups remaining. At the end, each group is a group of categories we will consider equivalent.

We found that, in almost all cases, simply treating every category as different is better than either clustering approach. We conclude that clustering categories into groups in either manner is not worthwhile.

4.6.3 Should PACE separate tasks by category and application?

Having shown that separating tasks by category and application improves task length prediction, we now examine whether and to what extent such separation improves PACE. To simulate PACE's performance, we conducted trace-driven simulations assuming a CPU with speed range 200–600 MHz. We compared the following four variants of PACE. **PACE-N** computes the current task's schedule using a probability distribution derived from all recent tasks, regardless of their category or application. **PACE-A** uses only recent tasks of the same application as the current task. **PACE-C** uses only recent tasks of the same category as the current task. **PACE-CA** uses only recent tasks of the same category and application as the current task.

Results are in Table 5. We find that PACE-CA beats PACE-N, PACE-A, and PACE-C for every user and for both keystrokes and mouse clicks. These results validate our hypothesis that separation to improve prediction of task length will also improve prediction of task length distribution.

However, PACE derives only small energy savings by separating tasks by category and application. The average energy reduction for PACE-CA relative to PACE-N is only 1.7% for keystroke tasks and 1.3% for mouse click tasks. We conclude that separation is worthwhile, but only barely so. If such separation incurs significant complexity or overhead, the system designer should avoid it.

5 Conclusions

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

We found that processing user interface events uses only 5.6%–43.7% of CPU time, so DVS algorithms should look at other sources of tasks. Timer events are promising, as their processing accounts for 6.8–68.1% of CPU time.

We described and evaluated a new heuristic for determining when a user interface task completes. It is much easier to implement than existing heuristics, and we found that, excepting two applications, it is nearly as accurate, prematurely estimating completion for only 0.8–5.3% of tasks.

Since I/O wait time is important to DVS algorithms, we analyzed how frequently user interface tasks wait for I/O. This occurs in only 0.3–3.5% of user interface tasks for the various users. Mouse clicks require I/O far more often, 5.1–15.5% of the time, so DVS algorithms should probably account for I/O 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 need different speeds. Mouse movements need only the minimum speed, and simulations showed that this speed makes over 99% of all deadlines while reducing energy consumption by 68.3–84.0% with an average of 77.5%. Keystrokes can require more speed, though the specific speed is different for different users. Mouse clicks require an even higher pre-deadline speed for reasonable response time.

Next, we compared DVS algorithms. We found that Stepped, which gradually increases speed as a task progresses, does better than Flat and Past/Peg. PACE is even better than Stepped: 7.3% better for keystrokes and 2.8% for mouse clicks. Since Stepped saves almost as much energy as the near-optimal PACE, its schedule seems well suited to user interface tasks in real workloads.

Finally, we found that separating tasks by both category and application provides a better prediction of task length than performing no separation or separating only by category or only by application. Furthermore, simulations showed that PACE actually saves energy by performing such separation, improving 1.7% for keystrokes and 1.3% for mouse clicks. However, the small effect size suggests that a DVS algorithm should do this only if it would incur minimal complexity and overhead.

References

[1] E. Chan, K. Govil, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st ACM International Conf. on Mobile Computing and Networking (MOBICOM 95)*, pages 13–25, November 1995.

[2] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. Scatter/Gather: a cluster-based approach to browsing large document collections. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 318–329, June 1992.

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

[4] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th ACM International Conf. on Mobile Computing and Networking (MOBICOM 2001)*, July 2001.

[5] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[6] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., New York, NY, 1991.

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

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

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

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

[11] Microsoft. *Platform SDK Documentation*, 2002.

[12] T. Pering, T. Burd, and R. 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.

[13] P. Pillai and K. 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.

[14] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th ACM International Conf. on Mobile Computing and Networking (MOBICOM 2001)*, July 2001.

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

[16] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. D. Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th Design Automation Conference*, pages 524–529, June 2001.

[17] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.

[18] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, MA, 1993.