PACE: A New Approach to Dynamic Voltage Scaling

Jacob R. Lorch, *Member*, *IEEE*, and Alan Jay Smith, *Fellow*, *IEEE*

Abstract—By dynamically varying CPU speed and voltage, it is possible to save significant amounts of energy while still meeting prespecified soft or hard deadlines for tasks; numerous algorithms have been published with this goal. We show that it is possible to modify any voltage scaling algorithm to minimize energy use without affecting perceived performance and present a formula to do so optimally. Because this formula specifies increased speed as the task progresses, we call this approach PACE (Processor Acceleration to Conserve Energy). This optimal formula depends on the probability distribution of the task's work requirement and requires that the speed be varied continuously. We therefore present methods for estimating the task work distribution and evaluate how effective they are on a variety of real workloads. We also show how to approximate the optimal continuous schedule with one that changes speed a limited number of times. Using these methods, we find we can apply PACE practically and efficiently. Furthermore, PACE is extremely effective: Simulations using real workloads and the standard model for energy consumption as a function of voltage show that PACE can reduce the CPU energy consumption of existing algorithms by up to 49.5 percent, with an average of 20.6 percent, without any effect on perceived performance. The consequent PACE-modified algorithms reduce CPU energy consumption by an average of 65.4 percent relative to no dynamic voltage scaling, as opposed to only 54.3 percent without PACE.

Index Terms—Dynamic voltage scaling, energy management, power management, optimization algorithm.

1 Introduction

The growing popularity of mobile computing devices has made energy management important for modern systems. Designers of laptops, ultraportables, and personal digital assistants must ensure those devices deliver reasonable battery life. A relatively recent energy-saving technology is *dynamic voltage scaling* (DVS), which allows software to dynamically alter the voltage of the processor. Various chip makers, including Transmeta, AMD, and Intel, have recently announced and sold processors with this feature.

Reducing CPU voltage can reduce CPU energy consumption substantially since energy consumption is proportional to the square of the voltage ($E \propto V^2$) [18]. Performance, however, suffers; the highest speed at which the processor will run correctly drops as the voltage decreases. According to theoretical models, the maximum speed should drop roughly proportionally to the voltage ($s \propto V$) [17]. Voltage scaling processors introduced to date have not fully followed this model, but they still require lower speeds at lower voltages [10].

There are two factors that limit the utility of trading lower performance for energy savings. First, a user wants the performance for which he paid. Second, other computer components, such as the disk, display, and backlight, also consume power. If they stay on longer because the CPU runs more slowly, the overall effect can be worse performance and *increased* energy consumption. Thus, a voltage scaling algorithm should generally reduce the voltage only when it will not noticeably affect performance.

A natural way to express this goal is to consider the computer's activity to consist of a set of tasks, each of which has a soft or hard deadline. If the deadline is hard, the task must complete by then; if the deadline is soft, the task should have a high probability of completing by then, but this probability does not have to be 1.0. For example, user interface studies have shown that user think time is unaffected by response time as long as response time is under 50-100 ms [14], so it is reasonable to consider the soft deadline for handling a user interface event to be 50 ms. As another example, multimedia programs that operate on real-time streams or that have limited buffering need to complete processing a frame in time equal to the inverse of the display rate. When goals can be codified this way, the job of a dynamic voltage scaling algorithm is to run the CPU just fast enough to meet the deadline requirements.

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

If a task's CPU requirement is known, the system can minimize energy consumption by running the CPU at a constant speed just fast enough to finish the task by its deadline [17]. Previously published algorithms have been implicitly based on the belief, therefore, that, even when a task's CPU requirement is unknown, a desirable (i.e., energy-minimizing) schedule is one that minimizes the

- *J.R. Lorch is with Microsoft Research, 1 Microsoft Way, Redmond, WA 98052. E-mail: lorch at microsoft.com.*
- *A.J. Smith is with the Electrical Engineering and Computer Science Department, Computer Science Division, University of California, Berkeley, Berkeley, CA 94720-1776. E-mail: smith at eecs.berkeley.edu.*

Manuscript received 2 Oct. 2001; revised 28 Feb. 2003; accepted 21 Nov. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115127.

frequency of changes in speed and voltage. In fact, when the task's CPU requirement is known only probabilistically, we will show that a constant speed is *not* optimal. The expected energy consumption is in fact minimized by gradually *increasing* speed as the task progresses, i.e., as we discover the task requires more work. We therefore call our approach for improving algorithms in this way PACE: Processor Acceleration to Conserve Energy.

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

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

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

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

The rest of this paper is structured as follows: Section 2 discusses related work, including algorithms other authors have proposed for dynamic voltage scaling. Section 3 presents our model of the problem of dynamic voltage scaling and introduces the terminology for the remainder of the paper. Section 4 describes how we suggest improving existing dynamic voltage scaling algorithms. Among other suggestions, it describes our optimal formula for speed scheduling using PACE, as well as methods for practically implementing it. Section 5 describes how algorithms differ even after modification by PACE and how we can choose between them. Section 6 presents and discusses results of simulating our suggested algorithm improvements. Finally, Section 7 concludes. Section 8 describes supplemental material for this paper which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm; see also [9].

Although we explain terms in this paper when we first present them, the reader may find it helpful to refer to Table 1 as needed. It summarizes terms used in this paper, giving their definitions and their abbreviations.

2 Related Work

Weiser et al. [17] observed that doing a fixed amount of work in fixed time with the least energy requires a constant CPU speed. They also noted, as we did earlier, that completing work in a timely manner is important. Based on this, they recommended *interval-based* DVS algorithms. Such algorithms divide time into fixed-length intervals and set the speed in each interval so that not much work remains uncompleted when the interval ends. Chan et al. [4] refined these ideas by separating out the two parts of an algorithm: *prediction* and *speed-setting*. When a new interval begins, the prediction part predicts the interval's CPU utilization, i.e., what fraction of the interval the CPU will be nonidle. Then, the speed-setting part uses this information to set the speed for the interval.

Researchers have proposed several methods for the prediction part, including the following: Weiser et al.'s [17] **Past** method predicts the utilization will be the same as the last interval's. Chan et al.'s [4] **LongShort** method averages the 12 most recent intervals' utilizations, weighting the most recent three of these three times more than the other nine. Chan et al.'s [4] **Flat**-u method always makes the same prediction, namely, that the utilization will be the method parameter u.

Researchers have also proposed methods for the speed-setting part, including the following: Weiser et al.'s [17] method, which we call **Weiser-style**, works as follows: If the upcoming interval's utilization is predicted to be high (more than 70 percent), it increases the speed by 20 percent of the maximum speed. If the upcoming interval's utilization is predicted to be low (less than 50 percent), it decreases the speed by $60 - x$ percent of the maximum speed, where x is the upcoming interval's predicted utilization as a percentage. Chan et al.'s [4] method, which we call **Chan-style**, sets the speed for the upcoming interval just high enough to complete the predicted work. In other words, it multiplies the maximum speed by the utilization to get the speed for the upcoming interval. Finally, Grunwald et al.'s [5] **Peg** method either sets the speed to the minimum possible, keeps it the same, or sets it to the maximum possible, depending on whether the predicted utilization is below 93 percent, between 93 percent and 98 percent, or above 98 percent. They determined the figures 93 percent and 98 percent empirically.

Using interval boundaries as deadlines is somewhat arbitrary. If a task arrives shortly before an interval boundary, there is no reason it must complete by the end of that interval. Furthermore, without discernible deadlines, there is no reason to complete any given task by a certain

TABLE 1
Terms Used in This Paper, along with Their Abbreviations and Definitions

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

time.[1] Pering et al. [11], recognizing this, suggested considering deadlines when evaluating DVS algorithms. They suggested that, when measuring the performance of an algorithm, one should consider a task that completes before its deadline to have effectively completed at its deadline.

Grunwald et al. [5] considered deadlines when they compared several of the algorithms described above (as well as others we have not listed) by implementing them on a real system. They decided that, although none of them are very good, Past/Peg is the best since it never misses any deadlines for the workload they considered, yet still saves a small but significant amount of energy.

1. Without deadlines, it is best to simply measure the average number of nonidle cycles per second and run the CPU at that speed. Transmeta's LongRun™ system uses an approach of that type [6].

Many researchers have explored DVS in real-time environments, where tasks must complete by certain deadlines or their results are useless [2], [12], [13]. In some real-time scenarios, such as processing network packets, the deadline is soft since the system can recover from failed tasks [13]. In other scenarios, the deadline is hard, so the system scheduler must know and use the task's worst-case execution time (WCET) to ensure the CPU runs fast enough to make the deadline [2], [12].

Pillai and Shin [12] and Aydin et al. [2] have independently proposed DVS algorithms for achieving high energy savings in such real-time environments by using the following three-stage scheduling process: First, the scheduler computes a static optimal schedule assuming each task requires its entire WCET. Second, when a task completes having used less than its WCET, the scheduler uses the

slack to create a new schedule for the remaining tasks since they can now run more slowly and still make their deadlines. Third, if a task is predicted to be unlikely to require its WCET, that task is run at a slower speed than the schedule requires. This way, if the task requires much less than its WCET, it will save a lot of energy and, even if it requires more time, it can still make its deadline as long as the scheduler increases the speed soon enough in the course of the task's execution. These researchers found that the utility of such aggressive speed reduction depends on the probability distribution of the task's actual CPU requirement and proposed ad hoc methods for accounting for this. In this paper, we will show that there is an analytically optimal method for accounting for the probability distribution of task requirement when choosing how slowly to start a task and how to increase the CPU speed as it progresses. However, our method currently can only schedule one task at a time; future work includes extending our techniques to deal with simultaneous scheduling of multiple tasks, a typical feature of real-time scheduling.

For some workloads, the goal is not to have each task make its individual deadline, but to ensure that the CPU speed matches the rate at which work is introduced into the system. An example of a workload emphasizing rate matching is media playback; media players typically use buffers, which allow different frames to take different amounts of time to process as long as the aggregate rate of processing is sufficient. For buffered workloads like this, DVS algorithms such as those of Simunic et al. [16] are more appropriate than those we consider here. These algorithms use a stochastic model of the task queuing system to ensure the CPU completes tasks at the required rate.

3 Model

3.1 Processor

We model the processor as follows: The processor can attain any speed between some minimum $s_{\min}$ cycles/sec and maximum $s_{\max}$ cycles/sec, inclusive. At each speed, there is a minimum feasible voltage; we assume it uses that voltage. Thus, CPU power consumption varies with speed.

We assume power consumption is a strictly convex and increasing function of speed and energy consumption per cycle is an increasing function of speed. We define $\mathrm{P}(s)$ to be the power in watts at speed s and define $\mathrm{E}(s)$ to be the energy consumption in joules per cycle at speed s. Note that $\mathrm{E}(s) = \mathrm{P}(s)/s$.

We use the glyphs P and E to distinguish them from the glyphs P and E typically used for power and energy. Our definitions make clear the dependence of power and energy on the speed used, while traditional notation does not. In typical notation, $P(t)$ is the power consumed at time t and $E = \int_a^b P(t)\,dt$ is the energy consumed during some time interval $[a, b]$.

3.2 CPU Scheduling

Now, we describe our model of DVS algorithms. For the purposes of this paper, we restrict ourselves to algorithms for scheduling a single task at a time. It is future work to extend our algorithm to simultaneous scheduling of multiple tasks.

A single-task DVS scheduling algorithm must decide how quickly to run a task as that task progresses. This task has some *work requirement* or, simply, *work* (W) that is the number of CPU cycles it will take to complete. The task also has some *deadline* (D) that is the number of seconds in which the algorithm should try to complete the task once the task is dispatched. The number of seconds the task actually takes, given the algorithm's CPU speed choices, is its *completion time*. A task's *effective completion time* is the maximum of its completion time and its deadline; this reflects the fact that, if a task completes by its deadline, it may as well have completed at its deadline. The task's *delay* is the number of seconds the task takes beyond its deadline, i.e., its effective completion time minus its deadline. If the task does not make its deadline, that means it still has work to do. We call the number of cycles still left to do upon reaching the deadline the *excess*.

When a task arrives, an algorithm must decide how fast to run the CPU to complete it. In general, the algorithm may choose to vary the CPU speed as the task progresses; for instance, it might choose to run the CPU at 300 MHz for the first 10 ms, then 400 MHz for any remaining time. Thus, the algorithm is actually choosing the speed as a function of time. We call this function the *speed schedule* and denote it by f: $\mathrm{f}(t)$ is the speed, in cycles per second, that the algorithm will run the CPU after the task has run for t seconds.

A realizable algorithm cannot know the task's work requirement until the task completes. Since it gains no information about the task's work requirement as the task progresses except to know that the task has not yet completed, it could, in theory, compute the entire schedule when the task arrives. In practice, an implementation of the algorithm may not compute $\mathrm{f}(10)$ until 10 seconds actually pass, but we consider that $\mathrm{f}(10)$ is nevertheless defined as soon as the task arrives. Indeed, even if the task completes after 5 seconds, $\mathrm{f}(10)$ is still defined: It is the speed at which the algorithm *would have* run the CPU if the task had gone longer than 10 seconds.

We can think of a speed schedule as consisting of two parts, the *predeadline part* and the *postdeadline part*. The former is the part of f that describes what happens before the task reaches its deadline (when $t \le D$) and the latter describes what happens after the task misses its deadline (when $t > D$). A speed schedule has a certain number of *predeadline cycles* (PDC), the number of CPU cycles it can perform before the deadline. This value is determined by the predeadline part since $\mathrm{PDC} = \int_0^D \mathrm{f}(t)\,dt$. The PDC is important because the task will miss its deadline if and only if its work requirement exceeds the predeadline cycles of the schedule (i.e., if $W > \mathrm{PDC}$). In particular, if the deadline is hard, the speed schedule must always have $\mathrm{PDC} \ge \mathrm{WCET}$, where WCET is the worst-case execution time expressed in cycles; this ensures that the task completes by the deadline and that the postdeadline part is irrelevant.

We say that two speed schedules are *performance equivalent* if, no matter what the task's work requirement, it will have the same effective completion time no matter which of the two schedules is used. We call two DVS algorithms performance equivalent if they always yield

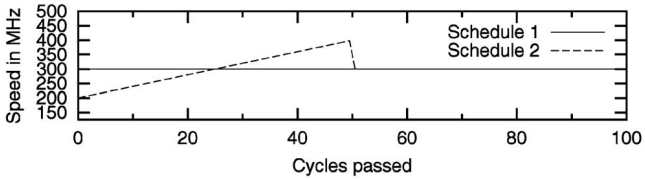

Fig. 1. This graph shows two performance equivalent speed schedules. They have equal predeadline cycles (they both accomplish 15 Mc by the deadline, 50 ms) and they have identical postdeadline parts.

performance equivalent speed schedules. We make the following important observation:

> If two speed schedules accomplish an equal number of predeadline cycles and have identical postdeadline parts, then they are performance equivalent.

Fig. 1 illustrates an example of this. To see that this observation is true, consider two cases. First, suppose the task requires no more work than the predeadline cycles the schedules share. In this case, both schedules make the task complete by the deadline, so both yield an effective completion time of D. Next, suppose the task requires more work than the predeadline cycles. Then, both schedules leave the task the same excess to do after the deadline: $W - \mathrm{PDC}$. Since the schedules have identical postdeadline parts and both have the same excess to do in that part, both will complete the task at the same time.

This observation is the key to the PACE approach. It modifies algorithms without changing their predeadline cycles or their postdeadline parts. Thus, it keeps performance the same. However, by strategically choosing the speed schedule for the predeadline part, it can make the expected energy consumption lower than the original algorithm.

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

3.3 Performance Metrics

In some parts of this paper, we will need to compare the performance of algorithms that are not performance equivalent. For this, we will need performance metrics. One such metric is the *fraction of deadlines made* (FDM), the fraction of all tasks in a workload that meet their deadlines. A problem with this metric is that sometimes a workload contains *impossible tasks*: Tasks so long they could not meet their deadlines even at the maximum CPU speed (i.e., having $W > s_{\max} D$). Then, the maximum achievable FDM is not 1, so the goodness of a value of this metric is unclear. Therefore, we instead use the *fraction of possible deadlines made* (FPDM), the fraction of all tasks with possible deadlines in the workload that make those deadlines.

We also need a metric expressing how undesirable it is to miss the deadline by various amounts. For example, for a user interface task, we want a metric of the user's "impatience function," i.e., how undesirable he finds missing the deadline by various amounts. Several such penalty functions could be justified. For instance, we could treat all delay as equally bad or we could have the first few milliseconds of delay past the deadline incur little penalty while later milliseconds of delay incur greater penalties. This penalty could be some quadratic or cubic function of delay to represent increasing user frustration as deadlines are missed by greater amounts.

We choose to use the linear metric Pering et al. [11] suggest, which they call *clipped delay*. This is the sum of the effective completion times of all the workload's tasks. They justify a linear metric for the following reasons. First, it theoretically reflects the total response time as perceived by the user. Second, since user-perceived delays generally lead to longer operating times, the metric also reflects the time other power-consuming components, such as the backlight, must stay on to support the tasks' operations and, thus, the energy consumed by those components. A similar metric, which we prefer because its magnitude is unaffected by the deadline and the number of tasks, is the *average delay* (AvgDelay). This is simply the average of the delays of all tasks in the workload. Since the average delay is a linear transformation of the clipped delay, using it yields equivalent comparisons between algorithms.

4 Improving Algorithms with PACE

We now discuss our techniques for improving scaling algorithms to reduce their energy consumption without worsening their performance.

4.1 Theoretical Optimal Formula

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

The traditional basis for dynamic speed scheduling techniques has been keeping speed constant, so it may seem that the optimal schedule would be to run at a constant speed. However, as we will now show, the ideal speed schedule is actually a changing speed. An intuitive explanation is that, if the task work requirement is unknown, it may be high or low. It is worthwhile to run slowly at first because the task may require little work and thus end before we get to the point where we increase the speed and, thus, the power consumption. Consider an example: Suppose a task with hard deadline 50 ms takes 5 Mc (megacycles) 75 percent of the time and 10 Mc 25 percent of the time. Suppose further that CPU power is $50\mathrm{nW} \cdot x^3$ when the speed is x MHz. The ideal constant speed is 200 MHz, the slowest speed that will always meet the deadline; this consumes

$$(25\mathrm{ms})(200)^3(50\mathrm{nW}) + (25\%)(25\mathrm{ms})(200)^3(50\mathrm{nW}) = 12.5\mathrm{mJ}$$

on average. An alternate, variable speed schedule is 163 MHz for the first 30.675 ms, then 259 MHz for any remaining time; this consumes

$$(30.675\text{ms})(163)^3(50\text{nW}) + (25\%)(19.325\text{ms})(259)^3(50\text{nW}) = 10.84\text{mJ}$$

on average, an energy savings of 13.3 percent.

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

We are trying to minimize the expected energy consumption of the predeadline part of the algorithm, while keeping the predeadline cycles the same. The expected energy consumption of a schedule s is equal to $\int_0^{\mathrm{PDC}} F^c(w)\mathrm{E}[\mathrm{s}(w)]\,dw$ by the following reasoning: Consider the dw cycles of work after the first w; if dw is small, the speed over this period is approximately constant at $\mathrm{s}(w)$. The energy consumption per cycle is $\mathrm{E}[s(w)]$ and the number of cycles is dw, so the energy consumption is $\mathrm{E}[s(w)]\,dw$. The probability that this work actually ever gets done is $F^c(w)$.

We call a schedule s *valid* if it stays within the allowed CPU speed range and we call it *equivalent* if it has the same number of predeadline cycles as the original algorithm. We call a valid equivalent schedule $\mathrm{s}_{\mathrm{opt}}$ *optimal* if no other valid schedule with at least as many predeadline cycles as the original algorithm has lower expected predeadline energy. We seek such an optimal schedule.

In this section, we describe the optimal formula without proof of its optimality; such proofs are presented in supplemental material for this paper. In those proofs, we present the continuous improvement lemma, which shows that, when you hold some integral $\int_a^b g(w)\,dw$ constant and try to minimize some function $\int_a^b \psi_w[g(w)]\,dw$, the solution is to keep $\psi'_w[g(w)]$ as constant as possible. To ensure the proper predeadline cycles, we require $\int_0^{\mathrm{PDC}} \frac{1}{s(w)}\,dw = D$; to minimize energy consumption, we want to minimize $\int_0^{\mathrm{PDC}} F^c(w)\mathrm{E}[s(w)]\,dw$. So, in our scenario, $g(w) = 1/s(w)$ and $\psi_w(x) = F^c(w)\mathrm{E}(1/x)$, giving

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

Thus, the solution is to keep

$$\psi'_w[g(w)] = -F^c(w)[s(w)]^2\mathrm{E}'[s(w)]$$

as constant as possible. Essentially, then, we want $[s(w)]^2\mathrm{E}'[s(w)] = K/F^c(w)$ for some constant K, but we must be careful not to allow speeds outside of the valid range.

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

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

where E' is the derivative of the energy function E. Next, find a $K > 0$ such that

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

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

In other words, one can form an optimal schedule by making the predeadline part of $\mathrm{s}_{\mathrm{opt}}(w)$ equal to $\Theta^{-1}[K/F^c(w)]$ bounded to between $s_{\min}$ and $s_{\max}$, where $\Theta(s) = s^2\mathrm{E}'(s)$. One chooses this K such that the schedule achieves the proper number of predeadline cycles. If S and F^c both have analytic forms, it may be possible to compute K via a closed-form deterministic algorithm. Otherwise, one can do a binary search for K over its possible range $(0, s_{\max}^2\mathrm{E}'(s_{\max}))$, stopping the search when the PDC of the resulting solution is acceptably close to the desired PDC.

In the proof, we show that Θ^{-1} is an increasing function; therefore, since $F^c(w)$ decreases as w increases, this schedule speeds up the CPU as the task progresses, as suggested earlier.

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

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

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

So, we want the speed for the first 5 Mc to be proportional to $1^{-1/3} = 1$ and the speed for the next 5 Mc to be proportional to $0.25^{-1/3} = 1.587$. To make the deadline, the constant of proportionality, $c = (2C/K)^{-1/3}$ must satisfy

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

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

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

4.2 Piecewise Constant Speed Schedules

The optimal formula gives a continuous speed schedule, which may be unreasonable if software must notify the hardware each time it wants to change the speed. In practice, we should probably change speed for a task no more than some reasonable number of times N. So, we want a schedule with a limited number of *transition points*, points

where the speed may change. We denote the jth transition point by w_j. Note that we are using as transition points values of w where $\mathrm{s}(w)$ changes, not points in time where $\mathrm{f}(t)$ changes. The latter is more natural, but the former makes optimization easier.

Given fixed transition points $w_0, w_1, w_2, \ldots, w_N$ such that $w_0 = 0$ and $w_N = \mathrm{PDC}$, one can construct a speed schedule that minimizes expected energy consumption as follows. (We prove this construction works in our proofs; see Section 8.) For all $i \in \{1, 2, \ldots, N\}$, define

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

Next, find a $K > 0$ such that

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

Then, use the speed schedule

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

Essentially, this uses a speed for each interval as if the tail distribution function were constant over that interval, set equal to the average of the true tail distribution function over that interval.

As before, one chooses the constant of proportionality K so that the schedule achieves the proper number of predeadline cycles. If S has an analytic form, it may be possible to compute K via a closed-form deterministic algorithm. Otherwise, one can do a binary search for K over its possible range $(0, s_{\max}^2 \mathrm{E}'(s_{\max}))$, stopping the search when the PDC of the resulting solution is acceptably close to the desired PDC.

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

A problem with this is that, as the sequence $\{q_j\}$ increases, the q_j values get close together and it wastes our limited supply of speeds to use them. Thus, we terminate this sequence near $q_j = 0.95$ and pick further values of q_j so that they uniformly partition the remaining range. More precisely, we set $J = N - 3$ and $Q = 0.95$. We set $q_J = Q$, then compute c by solving the equation $Q = 1 - c^{-J}$. For each $1 \le j \le J$, we set $q_j = 1 - c^{-j}$; for each $j > J$, we set $q_j = Q + (j - J)\frac{0.995 - Q}{N - J}$. Fig. 2 illustrates how this works. The choices of Q and J are arbitrary, but we showed in other work [8] that there is virtually no change with different choices as long as Q is somewhere between 0.85 and 0.99 and as long as $N - J$ is between 3 and 9.

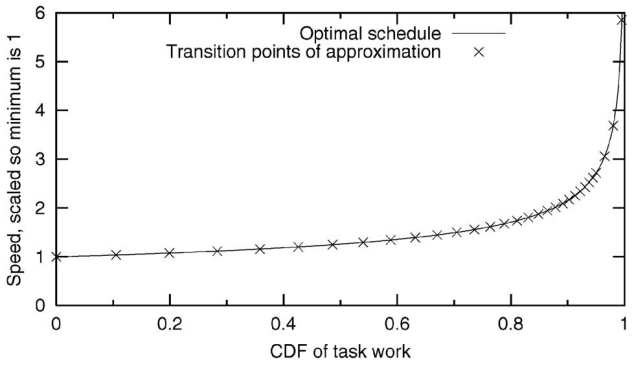

Fig. 2. This graph shows the transition points we use in our piecewise constant approximation to the optimal schedule. Transition points are described by their CDFs. CDF values less than the knee are spaced so that consecutive speeds vary by a constant factor; the few CDF values above the knee are spaced uniformly.

In other work [8], we examined the effect of using different numbers of transitions N. We found that the principle of diminishing returns applies: Increasing the number of transitions becomes less and less worthwhile as the number of transitions increases. Using 10 transitions yields energy consumption within 1.2 percent of the minimum. Using 20 transitions reduces the maximum penalty to 0.27 percent and using 30 transitions reduces it to 0.1 percent. For this paper, we will use $N = 30$. Note that using more transitions can be detrimental if the CPU must halt for a significant amount of time each time the speed changes; fortunately, processors will soon not have to do so [3].

To implement a piecewise constant speed schedule, software must be able to interrupt the task at predetermined intervals to change the CPU speed. If the hardware can be programmed to cause an interrupt at a given cycle count, the algorithm can use this feature. If the system has a high-frequency hardware timer, the algorithm can use that. Another method is to use soft timers, an operating system facility suggested by Aron and Druschel [1] that lets events be scheduled for the next time one can be performed cheaply, such as when a system call begins or a hardware interrupt occurs. This could only work if these events occur sufficiently frequently. A better way to implement speed schedules would be to implement them in hardware. For instance, the processor could accept as input not just a speed at which to run but a full schedule.

4.3 Distribution Estimation Methods

Implementing PACE requires some way to estimate the probability distribution of the current task's work requirement. Usually, an application will not provide information about this distribution, so we must model the distribution by sampling the work requirements of similar recent tasks. We assume tasks can be classified into types so that tasks of the same type have similar work requirements and that we can keep separate samples for each type. For instance, we can keep one sample of Microsoft Word tasks triggered by letter keypresses, another sample of Microsoft Excel tasks triggered by releasing the left mouse button, etc.

There are two general ways to estimate the distribution from a sample: parametric and nonparametric [15]. Parametric methods assume the distribution belongs to a certain family of distributions (e.g., normal distributions) and estimates the parameters that fully specify a member of that family (e.g., the mean and standard deviation of a normal distribution). Nonparametric methods make no such assumption, letting the sample "speak for itself" in describing the entire distribution.

5 Choosing a Base Algorithm

When PACE modifies an algorithm, it leaves two aspects of that base algorithm intact: what PDC it uses for each task and what postdeadline schedule it uses for each task. Thus, different base algorithms will still have different performance even after both are improved with PACE. In this section, we discuss how to choose among base algorithms.

Note that this is largely irrelevant for real-time systems. There, tasks can never go beyond their deadlines, so schedules do not need postdeadline parts. Also, in many real-time environments, deadlines are hard, so there is no choice in PDC—it must be the worst-case cycle requirement.

5.1 Choosing a Postdeadline Part

First, we consider what the base algorithm for postdeadline scheduling should be. Unlike earlier, we will not be creating a performance equivalent algorithm, so we need a performance metric. Since the postdeadline part has no influence on the fraction of deadlines made, we use average delay. We thus want an algorithm that consumes the least energy for a given average delay.

Let $\mathrm{TotalExcess}$ be the total excess of all tasks in the workload. Note that this is determined by the predeadline part; we cannot change it in the postdeadline part. To achieve a certain average delay $\mathrm{AvgDelay}$, we have to perform these $\mathrm{TotalExcess}$ cycles in total time equal to $n \cdot \mathrm{AvgDelay}$, where n is the number of tasks. As Weiser et al. [17] point out, the way to do this with minimum energy is to run at constant speed equal to $\mathrm{TotalExcess}/(n \cdot \mathrm{AvgDelay})$. Another way to look at this is that, if we use a fixed, constant speed after the deadline, we are assured that the energy consumption we achieve is the minimum possible for the achieved average delay. Therefore, we propose that a scheduling algorithm pick a fixed speed to use for all its postdeadline schedules. Many existing algorithms already do this, either because they always use a fixed speed or because they increase speed as average recent utilization increases and thus achieve the maximum CPU speed by the time a task reaches its deadline.

We must now determine what fixed speed to use in the postdeadline part. One possibility is to simply use the maximum available speed. The rationale for this is that, once a task has missed its deadline, the effective completion time depends directly on the actual completion time, so the only way to minimize effective delay is to run at the maximum speed possible. However, it may be that some situations warrant a lower fixed speed in the postdeadline part than the maximum available speed. As evidence, consider the fact that laptop computers using Intel's SpeedStep™ technology permanently lower the speed when the user is running off of battery power. This suggests that, in limited-energy scenarios, the user may be content with a lower maximum speed than the theoretical maximum speed of the processor.

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

5.2 Choosing PDC for Each Task

Since PACE does not change PDC, this value is entirely determined by the choice of base algorithm. In this section, we consider the optimal way for a base algorithm to compute PDC given that we will modify that base algorithm using PACE. In other words, given some target fraction of deadlines to make, TFDM, we would like to compute the optimal PDC for each task. This constraint is interesting because it is a single constraint on all tasks in the workload rather than one constraint per task. Therefore, we have great freedom in choosing the PDC values; for instance, we might decrease the PDC of one task, thereby increasing its probability of missing its deadline, and make up for that by increasing the PDC of another task, thereby decreasing its probability of missing its deadline.

We can describe the optimization problem mathematically as follows: Suppose we want to optimize the aggregate performance and energy consumption of a certain number n of consecutive tasks. If we denote the distribution of task i by F_i, we want to choose a PDC_i for each task i to minimize the expected energy consumption, which is proportional to

$$\sum_{i=0}^{n}\left[\int_0^{\mathrm{PDC}_i} F_i^c(w)\mathrm{E}(\mathrm{S}[F^c(w)/K(F_i,\mathrm{PDC}_i)])\,dw + \int_{\mathrm{PDC}_i}^{\infty} F_i^c(w)s_{\max}^2\,dw\right],$$

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

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

Here, we use $K(F_i, \mathrm{PDC}_i)$ to denote the PACE scaling factor K that depends on F_i and PDC_i.

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

These problems also make it impossible to analytically determine how well a given algorithm for computing PDC

will work from a standpoint of energy consumption versus performance. Therefore, we must rely on empirical, rather than analytic, methods to compare such algorithms. We consider several methods for computing PDC and present results comparing them in Section 6.2.2. Most of these algorithms are simply previously published DVS algorithms; in other words, we compute the PDC for each task by computing the PDC of the schedule that the previously published algorithm would generate.

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

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

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

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

6 Results

In this section, we present results of experiments we performed. First, we examine how effective PACE is at reducing the energy consumption of existing algorithms. Second, we evaluate various techniques for computing predeadline cycles.

For our simulations, we use the following six workloads: Word is a certain traced user's word processor usage, Groupwise is another user's traced groupware usage, and Excel is another user's traced spreadsheet usage. Low-Level is all keyboard activity by another user as seen by hardware observing only keypress events and commands to idle the processor. MPEG-One is playback of one video clip and MPEG-Many is playback of seven consecutive clips. For more details about these workloads, see our previous work using them [8], [9] or the supplemental materials associated with this paper.

In our simulations, we assume the minimum speed is 100 MHz and the maximum speed is 500 MHz. This maximum CPU speed is comparable to the (various different) speeds of the systems traced, although significantly lower than systems typical when we completed this research. We assume that power consumption is proportional to the cube of the speed, with peak power consumption of 3 W at 500 MHz. We assume the CPU speed can be instructed to change to a specific speed at a specific time, but cannot be instructed to change according to a continuous function of time.

For our distribution estimations, we use methods found to work well in our previous study [8]. The "Aged-0.95" sampling method uses a sample consisting of all past similar tasks, with the $k\mathrm{th}$ most recent having weight $(0.95)^k$. The kernel density estimation method is a nonparametric method for distribution estimation [15]. The gamma distribution estimation method is a parametric method assuming the distribution is gamma. In previous work [8], we found it to save almost as much energy as kernel density estimation while requiring substantially less computation.

6.1 Improving Existing Algorithms with PACE

In Section 4, we described how PACE can replace the predeadline part of an existing scheduling algorithm with a schedule that has lower expected energy consumption. In this section, we simulate this as follows: First, we simulate an existing algorithm. Then, we modify that algorithm so that it uses PACE to recompute the predeadline part of its schedule. Since the two algorithms are performance equivalent, we compare them solely on the basis of predeadline energy consumption; all other metrics are always identical.

For these simulations, we use four standard interval-based algorithms, each with an interval length of 10 ms. Each consists of a prediction and speed-setting method, as

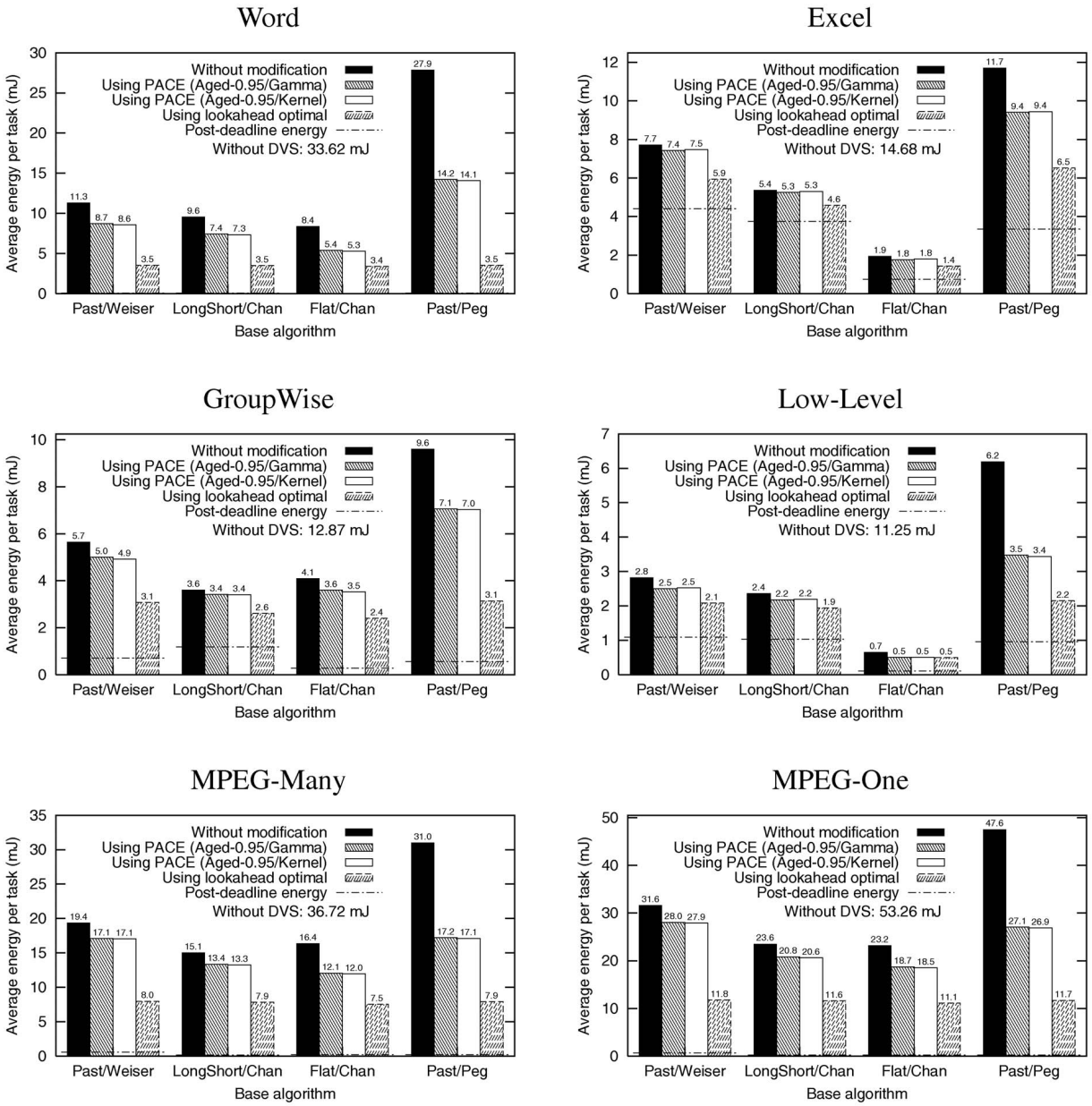

Fig. 3. These graphs show the effect of modifying existing algorithms with PACE to get performance equivalent, but lower-energy, algorithms. Since PACE only modifies the predeadline part of the algorithm, it keeps the postdeadline energy the same; this energy is shown with horizontal lines.

described in Section 2; we name each algorithm after those methods. The four methods we use are:

- **Past/Weiser-style.** This is a realizable version of the algorithm Weiser et al. proposed [17]. By "realizable" we mean it requires no oracle providing knowledge of the future.
- **LongShort/Chan-style.** This is a realizable version of one of the best algorithms Chan et al. proposed [4].
- **Flat/Chan-style.** This is a realizable version of another of the best algorithms Chan et al. proposed [4]. It runs the CPU at a constant speed, so it is similar to Transmeta's LongRun™ in steady state. We choose the speed so that the FPDM is at least 98 percent (99 percent for the Low-Level workload).
- **Past/Peg.** This is the algorithm Grunwald et al. favored [5].

Fig. 3 shows the effect of using PACE to modify these existing algorithms. We evaluate the effect of two versions of PACE, both using the Aged-0.95 sampling method: one uses the gamma model, which is easier to implement, and one uses the kernel density estimation method, which produces better results. Both versions of PACE reduce the CPU energy consumption of every workload and every existing algorithm. PACE using a gamma model reduces the CPU energy consumption of existing algorithms by 2.4-49.0 percent with an average reduction of 20.3 percent. PACE using the kernel density estimation method reduces the CPU energy consumption of existing algorithms by 1.4-49.5 percent with an average reduction of 20.6 percent.

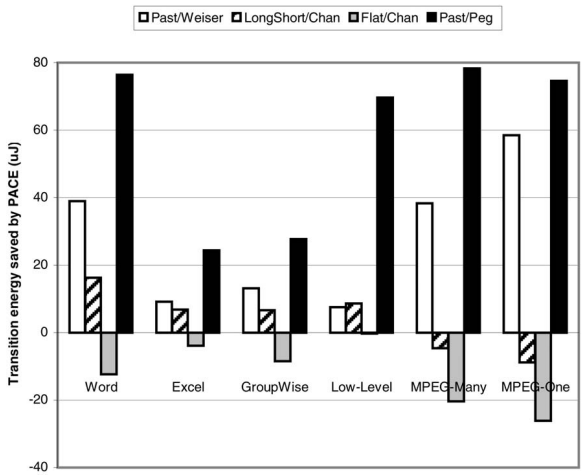

Fig. 4. This graph shows, for various workloads, the average predeadline transition energy saved per task by modifying various algorithms with PACE. For these figures, we assume the maximum voltage is 3.3 V and the voltage transition energy is $10\ \mu F$ times the difference in the square of the voltage.

The 1.4 percent value is lower than the 2.4 percent value because Excel, the workload that gains the least benefit from PACE, happens to also be the only workload for which the gamma model sometimes outperforms the kernel density estimation method. Excel gains less benefit from PACE than other workloads because it consumes a lot of postdeadline energy and PACE has no effect on postdeadline schedules. Interestingly, the existing algorithm most improved with PACE is Past/Peg, the one favored by the most recent comparison of existing algorithms [5]. Past/Peg was favored in that work because it misses fewer deadlines than other algorithms; unfortunately, this requires higher energy consumption, as Fig. 3 shows.

Another way to examine the results is to consider them relative to how much energy would be consumed in the absence of DVS. Without PACE, existing algorithms use DVS to reduce CPU energy consumption by 10.7-94.1 percent with an average of 54.3 percent. With PACE using a gamma model, the CPU energy savings increase to 35.9-95.5 percent with an average of 65.2 percent. With PACE using the kernel density method, the CPU energy savings increase to 35.6-95.5 percent with an average of 65.4 percent. Thus, on average, if a CPU consumes 100 J without DVS, existing DVS algorithms allow it to only consume 46 J; PACE reduces that figure even further to 35 J. Given these figures, if the CPU accounted for 33 percent of total energy consumption in a portable computer without DVS [7], existing DVS algorithms would increase its battery lifetime by about 22 percent; with PACE, the battery lifetime improvement would be about 28 percent.

Our processor model assumes that CPU speed and voltage transitions incur no time or energy overhead. The assumption of no time overhead is reasonable as future processors should be able to run during a voltage transition [3]. However, voltage transitions do consume energy. For example, if the maximum voltage is 3.3 V and the voltage regulator uses a $100\ \mu$F capacitor, then the transition energy is $10\ \mu$F times the difference in the square of the voltage during a task, according to Burd and Brodersen [3]. (With a larger capacitor, the transition energy would be proportionally greater.) Fig. 4 shows, for various algorithms, how much using PACE increases or decreases the energy consumed during speed transitions, assuming the above equation holds. If the algorithm uses more energy to change speeds than it does when modified by PACE, this number is positive; otherwise, it is negative. We see that PACE decreases the overall transition energy for most of the workloads and algorithms; the ones for which it increases transition energy are Flat, which never incurs any transition energy since it never changes speed, and LongShort/Chan, which uses more transition energy when modified by PACE for the MPEG-Many and MPEG-One workloads. It is not surprising that PACE often uses less transition energy for the following reason: PACE uses a low initial speed for long enough to ensure most tasks do not ever need a high speed; this has the additional effect that large voltage transitions are often not necessary. Note, however, that, in all cases, the transition energy change introduced by PACE, whether positive or negative, is always substantially smaller than the average CPU energy per task and, thus, is unlikely to be significant.

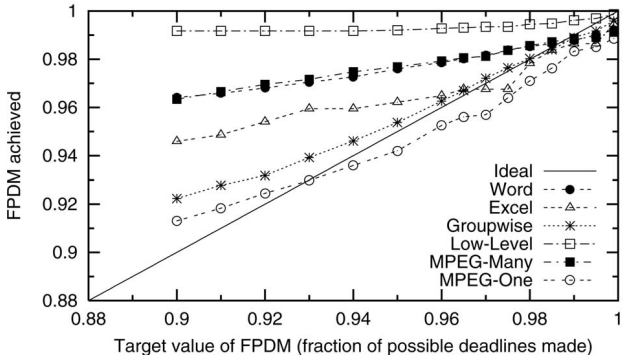

Fig. 5. This graph shows the FPDM achieved when various target levels of FPDM_T are sought using the Aged-0.95/Kernel method.

Fig. 3 also shows the results that could be obtained by the lookahead optimal strategy, a strategy that can see into the future and know exactly what the next task's work requirement is. These results are not attainable in practice on these workloads, but they provide a lower bound on the results that can be attained. We see that the PACE-modified algorithms consume significantly more energy than the lookahead optimal results, illustrating that knowledge of the distribution of task work is much less useful than knowledge of actual task work. In certain real-time environments, a system might have knowledge of actual task work and using that data would substantially reduce energy consumption.

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

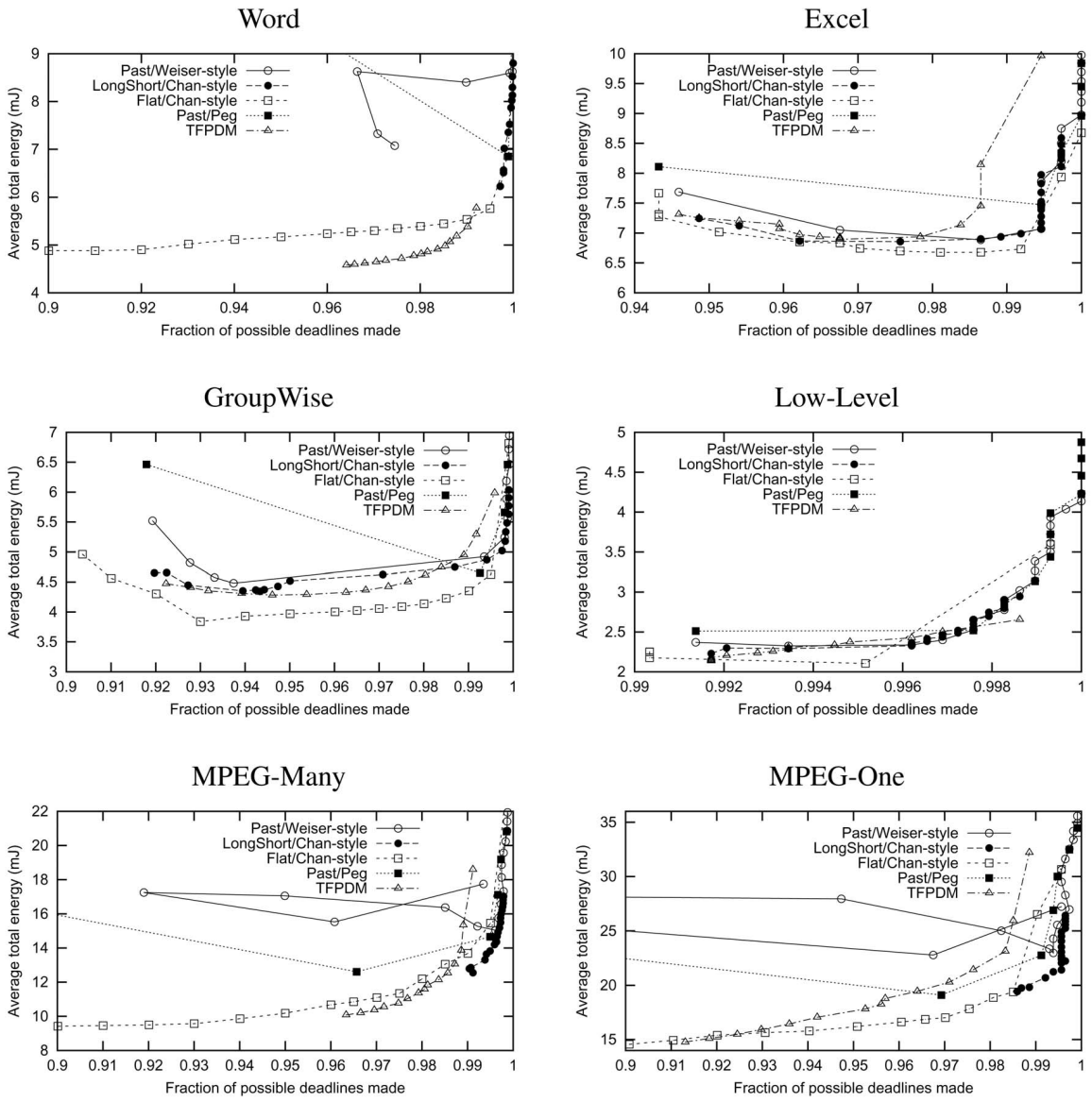

Fig. 6. These graphs compare existing algorithms when the predeadline part is adjusted by PACE (Aged-0.95/Kernel) and the postdeadline part is made to always use the maximum CPU speed. We generate each line's points by simulating an existing algorithm for various parameter values. Monotonic changes in parameter values do not have monotonic effects on FPDM, so the lines do not depict functions.

6.2 Computing Predeadline Cycles

6.2.1 Targeting FPDM

An approach we suggested for setting PDC was to set it to a certain quantile of the task work distribution, to achieve a certain target fraction of possible deadlines made TFPDM. Fig. 5 shows the result of using the Aged-0.95 sampling method and kernel distribution estimation method to estimate this quantile and use it as PDC. We see that increasing the target TFPDM increases FPDM achieved for all workloads. However, there is often substantial error between the target FPDM and the actual FPDM. There are a few reasons for this. First, the model does not work very well at describing the tail of the distribution, so computed quantiles are inaccurate. Second, for some workloads, some values of FPDM are simply too low to be achievable. For example, 94.3 percent of tasks in Excel are shorter than $s_{\min}D$, so FPDM cannot be less than 0.943. The other example is Low-Level, for which 93.4 percent of tasks are shorter than $s_{\min}D$.

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

6.2.2 Which PDC Computation Method Uses the Least Energy?

In Section 5.2, we discussed why we need to use empirical rather than analytical methods to compare the performance of different algorithms for computing PDC. In this section, we perform such empirical tests. Fig. 6 plots the average task energy consumption as a function of fraction of possible deadlines made for each of the workloads. Fig. 7 shows these plots averaged over all workloads. Note that energy generally goes up with increased fraction of deadlines made, but not

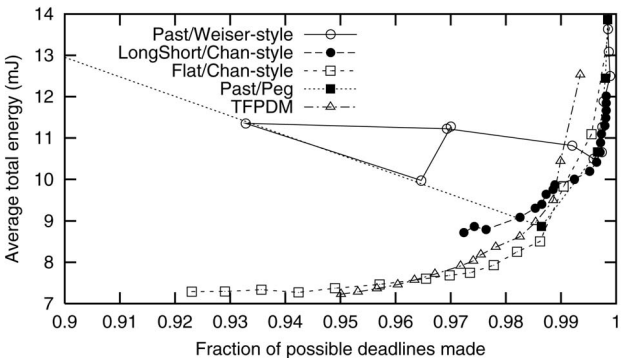

Fig. 7. This graph shows the results in Fig. 6 averaged over all workloads. It shows how existing algorithms compare to each other when their predeadline parts are adjusted by PACE and their postdeadline parts are changed to always use the maximum CPU speed. It graphs average total energy per task as a function of fraction of possible deadlines made.

always; it is possible to decrease energy consumption while making more deadlines by making the deadlines for a different set of tasks, e.g., a shorter set of tasks. The LongShort/Chan-style, Past/Past, and Past/Peg lines show the effect of varying interval length; the Flat/Chan-style line shows the effect of varying the constant predicted utilization (and, thus, the constant PDC); the TFPDM line shows the effect of varying TFPDM. We see that Flat/Chan-style, the only global strategy we considered, most commonly gives the lowest energy consumption for a given fraction of possible deadlines made. This suggests that global strategies tend to do better than local ones. Among the local algorithms, LongShort/Chan-style does best, achieving reasonable energy savings for a given fraction of possible deadlines made.

7 Conclusions

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

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

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

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

8 Supplemental Material

Due to space constraints, we present certain related material as supplemental material which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm. This material includes proofs for this paper's mathematical claims, an approach to computing PDC efficiently, workload descriptions, and avenues for future work. Much of this supplemental material also appears in a technical report [9].

Acknowledgments

The authors thank Michael Jordan for his very helpful suggestions regarding statistical methods, namely, the use of the gamma distribution and kernel density estimation. They also offer great thanks to the anonymous users of VTrace whose traces enabled them to create the workloads for this paper. This material is based upon work supported by the State of California MICRO program, AT&T Laboratories, Cisco Systems, Fujitsu Microelectronics, IBM Corporation, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation, and Veritas Software.

References

[1] M. Aron and P. Druschel, "Soft Timers: Efficient Microsecond Software Timer Support for Network Processing," *Proc. 17th ACM Symp. Operating Systems Principles (SOSP),* pp. 232-246, Dec. 1999.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems," *Proc. 2001 Real-Time Systems Symp. (RTSS),* pp. 95-105, Dec. 2001.

[3] T. Burd and R.W. Brodersen, "Design Issues for Dynamic Voltage Scaling," *Proc. 2000 Int'l Symp. Low Power Electronics and Design,* pp. 9-14, July 2000.

[4] E. Chan, K. Govil, and H. Wasserman, "Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU," *Proc. First ACM Int'l Conf. Mobile Computing and Networking (Mobicom '95),* pp. 13-25, Nov. 1995.

[5] D. Grunwald, P. Levis, K.I. Farkas, C.B. Morrey III, and M. Neufeld, "Policies for Dynamic Clock Scheduling," *Proc. Fourth Symp. Operating Systems Design and Implementation,* Oct. 2000.

[6] A. Klaiber, "The Technology behind Crusoe™ Processors," white paper, Transmeta Corp., Jan. 2000.
[7] J.R. Lorch and A.J. Smith, "Energy Consumption of Apple Macintosh Computers," *IEEE Micro,* vol. 18, no. 6, pp. 54-63, Nov./Dec. 1998.
[8] J.R. Lorch and A.J. Smith, "Improving Dynamic Voltage Scaling Algorithms with PACE," *Proc. 2001 ACM SIGMETRICS Conf.,* pp. 50-61, June 2001.
[9] J.R. Lorch and A.J. Smith, "PACE: A New Approach to Dynamic Voltage Scaling," Technical Report UCB/CSD-01-1136, Computer Science Division, Electrical Engineering and Computer Science Dept., Univ. of California at Berkeley, Mar. 2001.
[10] J.R. Lorch and A.J. Smith, "Operating System Modifications for Task-Based Speed and Voltage Scheduling," *Proc. First Int'l Conf. Mobile Systems, Applications, and Services (Mobisys 2003),* May 2003.
[11] T. Pering, T. Burd, and R.W. Brodersen, "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms," *Proc. 1998 Int'l Symp. Low Power Electronics and Design,* pp. 76-81, Aug. 1998.
[12] P. Pillai and K.G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP),* pp. 89-102, Oct. 2001.
[13] V. Raghunathan, P. Spanos, and M.B. Srivastava, "Adaptive Power-Fidelity in Energy-Aware Wireless Embedded Systems," *Proc. 2001 Real-Time Systems Symp. (RTSS),* pp. 106-115, Dec. 2001.
[14] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Reading, Mass.: Addison-Wesley, 1998.
[15] B.W. Silverman, *Density Estimation for Statistics and Data Analysis.* London: Chapman and Hall, 1986.
[16] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G.D. Micheli, "Dynamic Voltage Scaling and Power Management for Portable Systems," *Proc. 38th Design Automation Conf.,* pp. 524-529, June 2001.
[17] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for Reduced CPU Energy," *Proc. First Symp. Operating Systems Design and Implementation,* pp. 13-23, Nov. 1994.
[18] N.H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design.* Reading, Mass.: Addison-Wesley, 1993.

Jacob R. Lorch received BS degrees in computer science and mathematics from Michigan State University, then the MS and PhD degrees in computer science from the University of California, Berkeley. He is currently a researcher in the Systems and Networking Group at Microsoft Research in Redmond, Washington. His research interests include distributed systems, fault tolerance, file systems, operating systems, and energy management. He is a member of the ACM and the IEEE.

Alan Jay Smith received the BS degree in electrical engineering from the Massachusetts Institute of Technology, Cambridge, and the MS and PhD degrees in computer science from Stanford University, California. He was a US National Science Foundation Graduate Fellow. He is currently a professor in the Computer Science Division of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, where he has been a member of the faculty since 1974; he was vice-chairman of the Electrical Engineering and Computer Science department from July 1982 to June 1984. His research interests include the analysis and modeling of computer systems and devices, computer architecture, and operating systems. He has published a large number of research papers, including one which won the IEEE Best Paper Award for the best paper in the *IEEE Transactions on Computers* in 1979. He also consults widely with computer and electronics companies. He is a fellow of the IEEE, a fellow of the ACM, a fellow of the American Association for the Advancement of Science, a member of IFIP Working Group 7.3, the Computer Measurement Group, Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. In 2003, he received the A.A. Michelson Award from the Computer Measurement Group (CMG). The award is given as a lifetime achievement award for making significant, lasting contributions to the field of computer measurement and performance. He was on the Board of Directors (1993-2003), and was chairman (1991-1993) of the ACM Special Interest Group on Computer Architecture (SIGARCH), was chairman (1983-1987) of the ACM Special Interest Group on Operating Systems (SIGOPS), was on the Board of Directors (1985-1989) of the ACM Special Interest Group on Measurement and Evaluation (SIGMETRICS), was an ACM National Lecturer (1985-1986) and an IEEE Distinguished Visitor (1986-1987), was an associate editor of the *ACM Transactions on Computer Systems* (*TOCS*) (1982-1993), is a subject area editor of the *Journal of Parallel and Distributed Computing*, and is on the editorial board of the *Journal of Microprocessors and Microsystems*. He was program chairman for the Sigmetrics '89/Performance '89 Conference, program cochair for the Second (1990) Sixth (1994), and Ninth (1997) Hot Chips Conferences, and has served on numerous program committees.