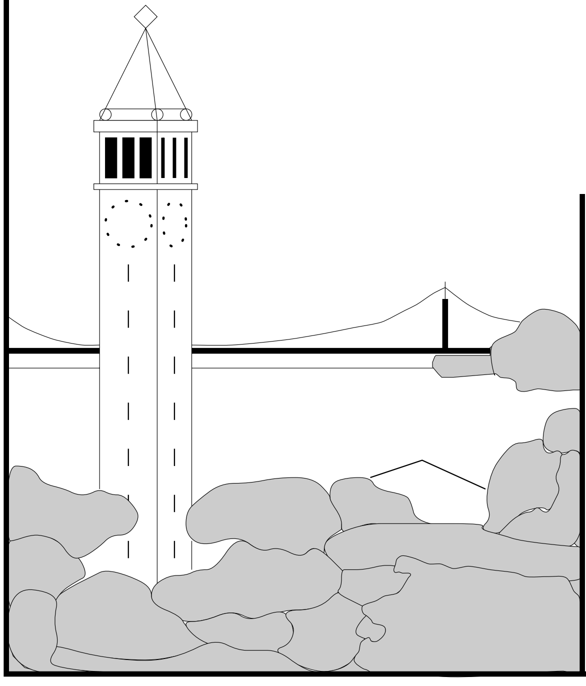
# PACE: A New Approach to Dynamic Voltage Scaling

*Jacob R. Lorch and Alan Jay Smith*



**Report No. UCB/CSD-01-1136**

March 2001

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

# PACE: A New Approach to Dynamic Voltage Scaling[*]

Jacob R. Lorch Alan Jay Smith
Computer Science Division (EECS)
University of California
Berkeley, California 94720

March 6, 2001

**Abstract**

This paper addresses algorithms for dynamically scaling CPU speed and voltage in order to save energy. Such scaling is useful and effective when the user will perceive the same performance, despite a slower CPU, as long as the task completes by some (soft) deadline. We show that it is possible to modify any scaling algorithm to minimize energy use without affecting perceived performance, and present a formula to do so. 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 show that PACE can reduce the CPU energy consumption of classic algorithms by up to 49.5%, with an average of 20.6%, without any effect on perceived performance.

## 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*, 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 ($E \propto V^2$). Performance, however, suffers; typically, over the range of allowed voltages, the highest speed at which the processor will run correctly drops approximately proportionally to the voltage ($s \propto V$). Thus, energy consumption is proportional to the square of the speed ($E \propto s^2$), so a CPU can save substantial energy by running more slowly. For instance, running at half speed means the CPU can run at half voltage and will consume only 1/4 the energy to run for the same number of cycles.

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,

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

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 deadline. (We call the deadline soft because the task should have a high probability of completing within this amount of time, but this probability does not have to be 1.0.) 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 [Shn98], so it is reasonable to consider the deadline for handling a user interface event to be 50 ms. As another example, multimedia programs that operate on real-time streams or that have limited buffering need to complete processing a frame in time equal to the inverse of the display rate. When goals can be codified this way, the job of a dynamic voltage scaling algorithm is to run the CPU just fast enough to meet the deadline with high probability.

The key property of a deadline is that as long as a task completes by its deadline, its actual completion time does not matter. This means that if we run the task more slowly, but it still completes by its deadline, performance is unaffected. The primary goal of 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 [WWDS94]. Previously published algorithms have been implicitly based on the belief, therefore, that a desirable (energy minimizing) schedule is one that minimizes the 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 continuous function of time, but a practical implementation may not be able to change CPU speed continuously.

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

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

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

The rest of this paper is structured as follows. §2 discusses related work, including algorithms other authors have proposed for dynamic voltage scaling. §3 presents our model of the problem of dynamic voltage scaling, and introduces the terminology for the remainder of the paper. §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. §5 describes how algorithms differ even after modification by PACE and how we can choose between them. §6 describes the workloads we use for analyzing the performance and energy consumption of algorithms. §7 presents and discusses results of simulating our suggested algorithm improvements. §8 discusses avenues for future work by outlining how our approach might be adapted to address issues such as I/O time and threshold voltage considerations. Finally, §9 concludes.

Although terms defined in this paper are explained when first presented, the reader may find it helpful to refer to Table 1 as needed. This table summarizes terms used in this paper, giving their definitions and their abbreviations.

## 2 Related Work

Weiser et al. [WWDS94] observed that to do a fixed amount of work in a fixed time with least energy, the CPU should run at constant speed. They also noted, as we did earlier, that completing work in a timely manner is important. Based on this, they recommended *interval-based* algorithms for dynamic voltage scaling. 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. [CGW95] 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 how busy the CPU will be during the interval, and the speed-setting part uses this information to set the speed for the interval. They measure how busy the CPU is via the *utilization*, the fraction of the interval the CPU spends non-idle.

Several authors, including Pering et al. [PBB98] and Grunwald et al. [GLF$^+$00], have pointed out that the algorithms proposed by Weiser et al. and Chan et al. are not practically realizable. The proposals assume an algorithm can know how much work would have been done in previous intervals if speed scaling were not used. However, if a task has not finished because the CPU is running slowly, an algorithm has no way of knowing how much work remains to be done on that task. Thus, the algorithm cannot estimate when the task would have completed were speed scheduling never used. In light of this, Pering et al. and Grunwald et al. have proposed practical versions of Weiser et al.'s and Chan et al.'s algorithms. Prediction methods they suggest include:

- **Past.** This method assumes the upcoming interval's utilization will be the same as the last interval's utilization.
- **Aged-$a$.** This method averages all past intervals' utilizations to predict the upcoming interval's utilization. More recent intervals are considered more reliable than older intervals. So, the $k$th most recent interval's utilization is weighted by $a^k$ in the average, where $a \leq 1$ is some constant.
- **LongShort.** This method averages the 12 most recent intervals' utilizations to predict the upcoming interval's utilization. The three most recent of these are weighted 3 times more than the other nine.
- **Flat-$u$.** This method always makes the same prediction for the upcoming interval's utilization. It predicts it to be some constant $u$, chosen based on policy.

Speed-setting methods they suggest include:

| **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. Generally, a deadline will be *soft*, meaning some tasks may miss their deadlines. The key property of a deadline is that as long as a task completes by its deadline, its actual completion time does not matter. |
| Effective completion time | The completion time of a task, or its deadline, whichever is greater. This measure reflects the fact that as long as a task completes by its deadline, its actual completion time does not matter. |
| Delay | The number of seconds a task takes beyond its deadline. |
| Average delay (AvgDelay) | The average delay of all tasks in a workload. |
| Excess | The number of cycles of work a task still has left to do after its deadline has passed. |
| Maximum speed ($M$) | The maximum speed, in cycles per second, at which the CPU can run (at maximum voltage). |
| Minimum speed ($m$) | 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. $f(t)$ is the speed to use after the task has run for $t$ seconds. $s(w)$ is the speed to use after the task has completed $w$ cycles of work. |
| Transition point ($w_i$) | A point at which a practical speed schedule changes from one speed to another. |
| 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. |
| Parametric method | A method for estimating a probability distribution from a sample by assuming the distribution belongs to some family of distributions (e.g., normal) and estimating the parameters of that distribution (e.g., the mean). |
| Nonparametric method | A method for estimating a probability distribution from a sample without assuming any given distribution type. Such a method thus lets the data "speak for themselves." |
| Kernel density estimation | A nonparametric method that builds up a probability distribution by adding up lots of little distributions, each centered on one of the sample points. |
| Bandwidth ($h$) | The width of each little distribution in kernel density estimation. |

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

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

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

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

Dividing time into intervals and using those boundaries as deadlines is somewhat arbitrary. If a task arrives shortly before an interval boundary, there is no particular reason it must complete by the end of that interval. Furthermore, without discernible deadlines, there is no particular reason to complete any given task by a certain time.[1] Pering et al. [PBB98], recognizing this, suggested taking deadlines into account when evaluating dynamic voltage scaling 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. [GLF$^+$00] 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.

# 3 Model

## 3.1 CPU scheduling

We model the CPU as follows. At minimum voltage, it runs at some minimum speed $m$ cycles/sec; at maximum voltage, it runs at some maximum speed $M$ cycles/sec. It can attain any speed between these values. CPU energy consumption per cycle is proportional to the square of the speed.

Now, we describe our model of scaling algorithms. Such an algorithm decides 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. 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 in order 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

[1] Without deadlines, it is best to simply measure the average number of non-idle cycles per second and run the CPU at that speed. Transmeta's LongRun$^{\text{TM}}$ system does something like this [Kla00].

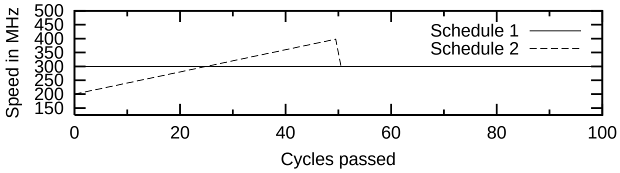


Figure 1: This graph shows two performance equivalent speed schedules. They have equal pre-deadline cycles (they both accomplish 15 Mc by the deadline, 50 ms) and they have identical post-deadline parts.

algorithm is actually choosing the speed as a function of time. We call this function the *speed schedule*, and denote it by $f$: $f(t)$ is the speed, in cycles per second, that the algorithm will run the CPU after the task has run for $t$ seconds.

A practical algorithm cannot know the task's work requirement until the task completes. Since it gains no information about the task's work requirement as the task progresses, it could, in theory, compute the entire schedule when the task arrives. In practice, an implementation of the algorithm may not compute $f(10)$ until 10 seconds actually pass, but we consider that $f(10)$ is nevertheless defined as soon as the task arrives. Indeed, even if the task completes after 5 seconds, $f(10)$ 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 *pre-deadline part* and the *post-deadline part*. The former is the part of $f$ that describes what happens before the task reaches its deadline (when $t \leq D$), and the latter describes what happens after the task misses its deadline (when $t > D$). A speed schedule has a certain number of *pre-deadline cycles* (PDC), the number of CPU cycles it can perform before the deadline. This value is determined by the pre-deadline part, since $\mathsf{PDC} = \int_0^D f(t)\,dt$. The PDC is important because the task will miss its deadline if and only if its work requirement exceeds the pre-deadline cycles of the schedule (i.e., if $W > \mathsf{PDC}$).

We say that two possible 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 schedule is used. We call two algorithms performance equivalent if they always have performance equivalent speed schedules. We make the following important observation:

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

Figure 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 pre-deadline 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 pre-deadline cycles. Then, both schedules leave the task the same excess to do after the deadline: $W - \mathsf{PDC}$. Since the schedules have identical post-deadline parts, and both have the same excess to do in that part, both will complete the task at the same time.

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

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

### 3.2 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 $W > MD$). 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.

It is also useful to have a metric that reflects 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 and greater penalties. For example, the 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. [PBB98] suggested, 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

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

The traditional basis for dynamic speed scheduling techniques has been keeping speed constant, so it may seem that the optimal schedule would be to run at a constant speed. However, surprisingly, the ideal speed schedule is actually a changing speed. 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 deadline 50 ms takes 5 Mc (megacycles) 75% of the time and 10 Mc 25% of the time. Suppose further that CPU power is $50 \text{ nW} \cdot x^3$ when the speed is $x$ MHz. The ideal constant speed is 200 MHz, the slowest speed that will always meet the deadline; this consumes

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

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

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

on average, an energy savings of 13.3%.

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

We are trying to minimize the expected energy consumption of the pre-deadline part of the algorithm, while keeping the pre-deadline cycles the same. The expected energy consumption is equal to $C \int_0^{\mathsf{PDC}} F^c(w)[s(w)]^2\, dw$, where $C$ is the constant of proportionality between energy and speed squared, 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 $s(w)$. The energy consumption per cycle is $C[s(w)]^2$, and the number of cycles is $dw$, so the energy consumption is $C[s(w)]^2\, dw$. The probability that this work actually ever gets done is $F^c(w)$.

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

One can construct an optimal valid schedule as follows. (We prove this construction works in Appendix A.) Define

$$\Phi(\sigma, w) = \begin{cases} m & \text{if } F^c(w) \geq (\sigma/m)^3 \\ M & \text{if } F^c(w) \leq (\sigma/M)^3 \\ \sigma[F^c(w)]^{-1/3} & \text{otherwise.} \end{cases}$$

Find an $S_0 > 0$ such that

$$\int_0^{\mathsf{PDC}} \frac{1}{\Phi(S_0, w)}\, dw = D.$$

Then, use the speed schedule $s(w) = \Phi(S_0, w)$ for $0 \leq w \leq \mathsf{PDC}$. In other words, one can form an optimal schedule by making $s(w)$ proportional to $[F^c(w)]^{-1/3}$ for the pre-deadline part; one chooses the constant of proportionality $S_0$ between $s(w)$ and $[F^c(w)]^{-1/3}$ so that the schedule achieves the proper number of pre-deadline cycles. Note that, since $F^c(w)$ decreases as $w$ increases, this schedule speeds up the CPU as the task progresses, as suggested earlier.

Given any scheduling algorithm, it is worthwhile to replace its pre-deadline part with this optimal formula. 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 CPU each time it wants to change the speed. In practice, we should probably change speed for a task no more than some reasonable number of times $n$. So, we want a schedule with a limited number of *transition points*, points where the speed may change. We denote the $j$th transition point by $w_j$. Note that we are using as transition points values of $w$ where $s(w)$ changes, not points in time where $f(t)$ changes. The latter is more natural, but the former makes optimization easier.

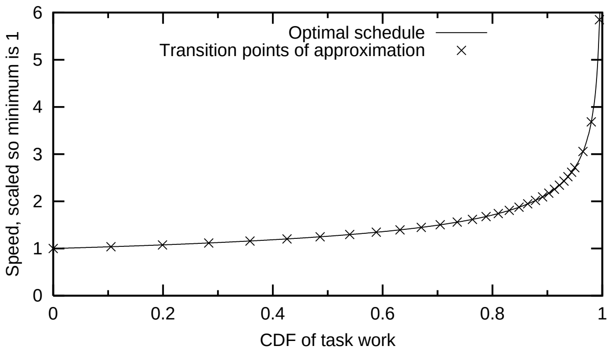


Figure 2: This graph shows the transition points we use in our piecewise constant approximation to the optimal schedule. Transition points are described by their CDF's. CDF values less than the knee are spaced so that consecutive speeds vary by a constant factor; the few CDF values above the knee are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

Given fixed transition points $w_0, w_1, w_2, \ldots, w_n$ such that $w_0 = 0$ and $w_n = \mathsf{PDC}$, one can construct a speed schedule that minimizes expected energy consumption as follows. (We prove this construction works in Appendix A.) For all $i \in \{1, 2, \ldots, n\}$, define

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

Also, for all $\sigma > 0$ and $i \in \{1, 2, \ldots, n\}$, define

$$\Phi(\sigma, i) = \begin{cases} m & \text{if } H_i \geq (\sigma/m)^3 \\ M & \text{if } H_i \leq (\sigma/M)^3 \\ \sigma H_i^{-1/3} & \text{otherwise.} \end{cases}$$

Find an $S_0 > 0$ such that

$$\sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\Phi(S_0, i)} = D.$$

Then, use the speed schedule

$$\hat{s}(w) = \begin{cases} \Phi(S_0, 1) & \text{if } w = 0 \\ \Phi(S_0, i) & \text{if } w_{i-1} < w \leq w_i \text{ for some } i \in \{1, 2, \ldots, n\}. \end{cases}$$

Essentially, this uses a speed for each interval proportional to the $-1/3$ power of the average value of $F^c(w)$ over that interval. As before, one chooses the constant of proportionality $S_0$ so that the schedule achieves the proper number of pre-deadline cycles.

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 $q_j = 1 - c^{-3j}$ for some constant $c$. Then, $F^c$ at the $q_j$th quantile of $F$ equals $c^{-3j}$. If we use these quantiles as transition points, then $[F^c(w)]^{-1/3}$, and thus the optimal speed, never varies by more than a factor of $c$ between any two consecutive transition points.

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

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

### 4.3 Sampling methods

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

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

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

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

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

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

## 4.4 Distribution estimation methods

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

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

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

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

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

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

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

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

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

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

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

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

The theoretical optimal bandwidth is

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

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

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

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

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

# 5 Choosing a Base Algorithm

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

## 5.1 Choosing a post-deadline part

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

Let TotalExcess be the total excess of all tasks in the workload. Note that this is determined by the pre-deadline part; we cannot change it in the post-deadline part. In order to achieve a certain average delay AvgDelay, we have to perform these TotalExcess cycles in total time equal to $n \cdot \mathsf{AvgDelay}$ where $n$ is the number of tasks. As Weiser et al. [WWDS94] point out, the way to do this with minimum energy is to run at constant speed equal to $\mathsf{TotalExcess}/(n \cdot \mathsf{AvgDelay})$. Another way to look at this is that if we use a fixed, constant speed after the deadline, we are assured that the energy consumption we achieve is the minimum possible for the achieved average delay. Therefore, we propose that a scheduling algorithm pick a fixed speed to use for all its post-deadline schedules. Many classic 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 post-deadline part. We observe that usually, other components like the backlight will be running and consuming power, and delay past the deadline will generally cause these components to consume more energy. Running at a speed $s$ makes CPU energy consumption proportional to $s^2$ but makes energy consumption of those other components proportional to $1/s$ as they stay on longer. We therefore believe it is best to always use the maximum speed once a task misses its deadline, as many classic algorithms generally do anyway. This minimizes delay, generally at some energy cost, but not at substantial energy cost considering that other components' power consumption would mitigate the effect of lower speeds.[2]

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

## 5.2 Choosing PDC for each task

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

[2] If $\rho$ is the ratio of the power consumption of these components to the power consumption of the CPU at its maximum speed, then running at speed $s$ causes total energy consumption to be proportional to $(s/M)^2 + \rho M/s$. For $s < M(\rho/2)^{1/3}$, the derivative of this value is negative; in other words, we get better energy consumption *and* better performance by increasing the speed. Therefore, there is no reason to use a speed less than $\max\{M, M(\rho/2)^{1/3}\}$.

We can describe the optimization problem mathematically as follows. Suppose there are $n$ tasks. If we denote the distribution of task $i$ by $F_i$, we want to choose a $\mathsf{PDC}_i$ for each task $i$ in order to minimize the expected energy consumption, which is proportional to

$$\sum_{i=0}^{n} \left[ \int_0^{\mathsf{PDC}_i} F_i^c(w) [\Phi(S_0(F_i, \mathsf{PDC}_i), w)]^2 \, dw + \int_{\mathsf{PDC}_i}^{\infty} F_i^c(w) M^2 \, dw \right],$$

subject to the constraint that we must expect to make a fraction $\mathsf{TFDM}$ of the deadlines:

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

Here, we use $S_0(F_i, \mathsf{PDC}_i)$ to denote the PACE scaling factor $S_0$ that is dependent on $F_i$ and on $\mathsf{PDC}_i$.

Unfortunately, we cannot solve this optimization problem, for two reasons. First, the complex dependence of $S_0$ on $\mathsf{PDC}_i$ 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 §7.5.3. Most of these algorithms are simply previously published DVS algorithms; in other words, we compute the PDC for each task by computing the PDC of the schedule that the previously published algorithm would generate.

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

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

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

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

### 5.3 Computing $\mathsf{PDC}$ efficiently

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

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

$$\mathsf{PDC} = \begin{cases} m(10 \text{ ms}) + M(D - 10 \text{ ms}) & \text{if } W_{\text{prev}} \leq m(10 \text{ ms}) + M(D - 20 \text{ ms}) + 0.7M(10 \text{ ms}) \\ MD & \text{otherwise,} \end{cases}$$

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

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

LongShort/Chan-style begins a task by running fast enough to complete its expected work by the deadline. Because this work estimate is weighted heavily toward recent values, it is usually approximately equal to the last task's work requirement, $W_{\text{prev}}$. As the task progresses, utilization goes up, and the algorithm ramps up speed toward the maximum $M$. If the initial speed were used for the entire task, the pre-deadline cycles would obviously be $W_{\text{prev}}$; if the latter speed were used for the entire task, the pre-deadline cycles would be $MD$. Depending on the rate at which speed is ramped up, which depends on the interval length used, the pre-deadline cycles actually achieved is somewhere between $W_{\text{prev}}$ and $MD$. We have found that we can reasonably approximate the pre-deadline cycles by using a weighted average of $W_{\text{prev}}$ and $MD$; the appropriate weighting varies with interval length. We will show that for an interval length of 10 ms, a reasonable approximation to $\mathsf{PDC}$ is $0.55W_{\text{prev}} + 0.45MD$.

## 6 Workloads

We evaluate these algorithms using six workloads. We derived several of these workloads from Windows NT and Windows 2000 traces obtained using VTrace, a tracer described in [LS00]. This tracer collects timestamped records describing events related to processes, threads, messages, disk operations, network operations, the keyboard, the mouse, and the cursor. From this data, we deduce how much work is done due to a user interface event as follows: we assume that a thread is working on such an event from the time it receives the message describing that event until the time it either performs a wait for a new event or requests and receives a message describing a different event. Furthermore, if the thread sends a message to another thread while working on such an event, we assume that any work done due to that sent message is actually done due to the original event. (For example, while processing a key-down message, a thread may post a character-pressed message to its message queue. Obviously, any work done to process the character-pressed message is really work done to process the key-down message.) Similarly, if a thread working on an event signals another thread, we assume that the signalled thread begins working on the same event.

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

Each workload is defined by a class of events, such as letter keypresses in Microsoft Word. The workload consists of the series of tasks triggered by all such events that occurred during the tracing period. Each such task is described by the amount of CPU processing in the task triggered by that event. In other words, each task of each workload is roughly of the same type; by separating different task types into different workloads, we model the effect of keeping separate samples for different task types, as described in §4.3. A full machine workload would consist of many of these kinds of workloads, interleaved. Since our approach operates independently on each different task type, we can correctly simulate it by considering each task type in isolation. Since task work distributions should be different for different task types, the approach of separating the task types and estimating work requirements based on task type should yield higher energy savings than using the combined work distribution.

We discard any task that blocked on any I/O, e.g., to a disk or network device. We do this because when a task blocks for I/O, it should use a different algorithm that takes I/O time into account, and such algorithms are beyond the scope of this paper. §8.2 discusses this avenue for future work further. Also, I/O generally occurs in only a small fraction of the tasks, so leaving them out should not significantly influence the results.

For our simulations, we assume the minimum speed 100 MHz, the maximum speed is 500 MHz, and the peak CPU power consumption is 3 W. The maximum CPU speed is comparable to the (various different) speeds of the systems traced.

## 6.1 Word processor typing

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

## 6.2 Groupware

Groupware, i.e., software that enables and enhances communication with others, is already an important application on the desktop, and will become more important in portable computers as users demand greater wireless networking functionality. Thus, we include a workload using a common groupware product, Novell's GroupWise. This workload uses releases of the left mouse button as its class of events. We derived

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

Table 2: Animations used in the MPEG workloads

this workload from traces. VTrace collected these traces on a 350 MHz Pentium II with 64 MB of memory running Windows NT 4.0 used by a crime laboratory director in the Michigan State Police. The workload is from 6.5 months of traces. This workload is interactive, so we use a 50 ms deadline for each task.

## 6.3 Spreadsheet

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

## 6.4 Video playback

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

The animations we use are all works taken from the BMRC FTP site where we got the MPEG decoder. Table 2 gives names and descriptions for these videos. One workload, which we call MPEG-One, consists only of the Red’s Nightmare animation. The other workload, which we call MPEG-Many, consists of all seven video clips, one played after the other. We made the measurements of CPU time on a 450 MHz Pentium III with 128 MB of memory running RedHat Linux 6.1. Assuming a typical rate of 25 frames per second, we assign a deadline of 40 ms to each frame.

## 6.5 Low-level workload

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

We derive this workload from VTrace, but in a different manner than the others. We define a task as the time between a keypress and the next time either the CPU becomes idle or there is another keypress. For the keypress time, we use the actual time the keyboard device was invoked, not the time the keypress message was delivered to an application. To determine when the CPU is idle, we use the time that the idle

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

Table 3: This table shows the statistical characteristics of the workloads' task work requirements. These work requirements are in Kc. The coefficient of skewness is the third central moment divided by the cube of the sample standard deviation.

thread is running. (In battery-powered systems, the idle thread typically halts the CPU, so hardware can deduce when this thread is active.) If any disk operations are ongoing when the thread goes idle, we throw out any keypress currently being worked on, for two reasons. First, we do not know if the I/O is part of the processing of the keypress, so we cannot determine whether the processing is over or simply waiting for I/O. Second, as stated before, we are not including tasks that perform I/O in our workloads since we are only considering algorithms for dealing with tasks that perform no I/O.

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

### 6.6 Statistical characteristics

Table 3 lists the statistical characteristics of the six workloads' task work requirements. Figure 3 shows the overall cumulative distribution function for each workload's task work requirements. The number of events in some workloads may appear small considering the trace is several months long. This is due to two factors: the user typically did not use each application every day during those months, and the workload only consists of a particular event type which may occur infrequently during normal application use. The standard deviations are moderate because all tasks in each workload are the same general type.

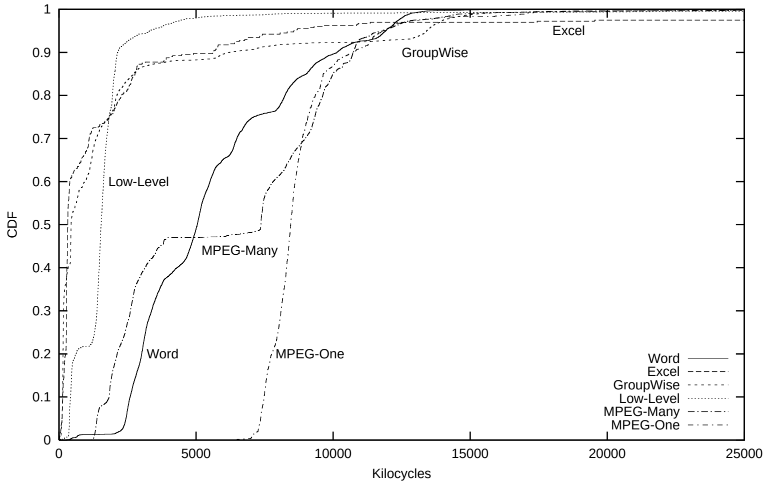


Figure 3: Overall CDF's of each workload's task work requirements

# 7 Results

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

## 7.1 Modeling task work distributions

In this section, we determine how best to practically estimate the probability distribution of task work requirements. We want to know which methods are general enough to work well for a variety of workloads, so we evaluate them all using simulations with our six different workloads. To determine how effective a method is at describing the distribution of task work requirements, we use a pragmatic approach: we use the method to implement PACE and simulate how much energy consumption results. We consider a method better if it produces lower pre-deadline energy consumption. No other metric is relevant, since PACE does not change any other metric.

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

### 7.1.1 Which sampling method to use

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

First, we consider what sample size to use for the sampling methods that use only recent data, Recent-$k$ and LongShort-$k$. Figures 4 and 5 show the outcome of using different sample sizes for different workloads.

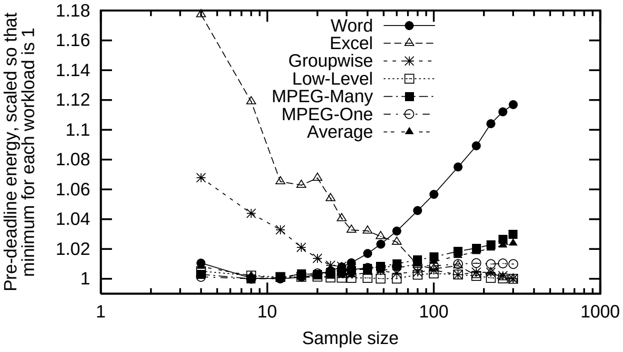


Figure 4: A comparison of the effect of various sample sizes $k$ on energy consumption when PACE uses the Recent-$k$ sampling method

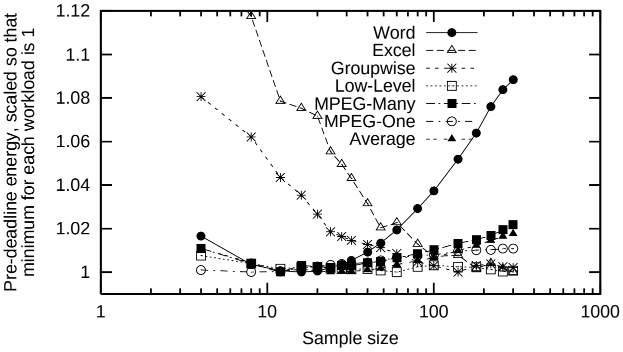


Figure 5: A comparison of the effect of various sample sizes $k$ on energy consumption when PACE uses the LongShort-$k$ sampling method

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

We now consider what aging factor $a$ to use for the Aged-$a$ sampling method. Figure 6 shows the outcome of using different aging factors for different workloads. Just as with sample sizes, some workloads do best with high aging factors while others do best with low aging factors. Not surprisingly, the workloads that do best with high aging factors are the same ones that do best with high sample sizes. This is expected, because a high aging factor makes sample values age more slowly, so that PACE effectively uses more old values. We feel a reasonable aging factor is 0.95. With this aging factor, each workload besides Excel has energy within 1.3% of what it would be using the best aging factor for that workload, and Excel has energy within 2.8% of the best possible.

Next, we determine which sampling method works best. Figure 7 compares the Future, Recent-28, LongShort-28, and Aged-0.95 sampling methods for the six workloads. We do not evaluate the All sampling method, since it is equivalent to Recent-$\infty$, and we have already shown that the Recent method works quite badly for some workloads with large sample sizes. The first thing we observe is that the Future method

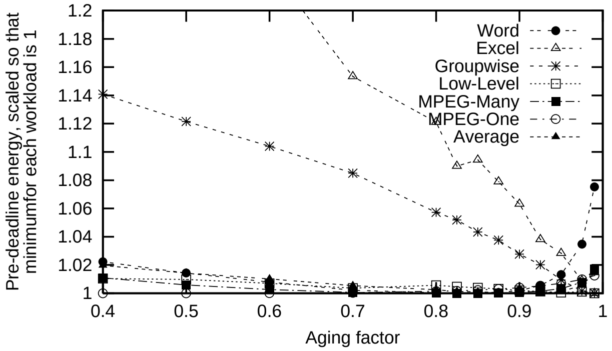


Figure 6: A comparison of the effect of various aging factors $a$ on energy consumption when PACE uses the Aged-$a$ sampling method

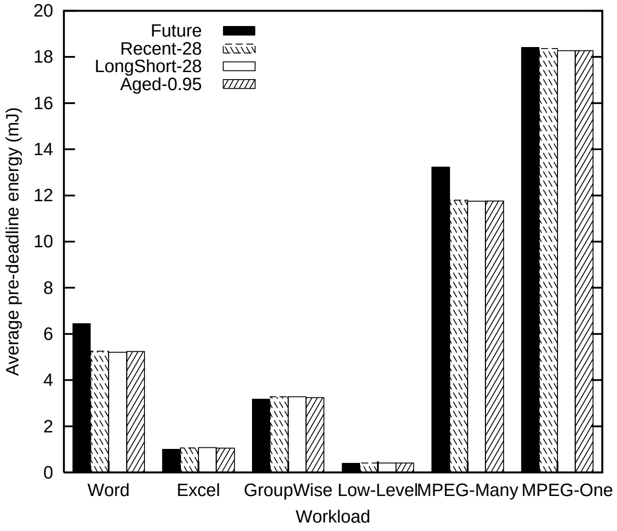


Figure 7: A comparison of the effect of various PACE sampling methods on energy consumption

sometimes uses less energy than the other methods, but sometimes uses more. This indicates that even if complete information about the full distribution of task work is available, it can still be better to use recent information to predict the distribution of the next task work. Presumably, this is because these distributions are non-stationary, so recent information is a better predictor than global information. This is heartening, since the Future method requires knowledge of the future and will thus usually be impossible to implement. The next thing to observe is that the remaining three methods have virtually identical energy consumption. Generally, Recent-28 consumes the most, LongShort-28 the next most, and Aged-0.95 the least. However, the difference between any two is never more than 2.2%. We conclude that, all things being equal, it is better to use the Aged-0.95 sampling method. However, if one of the three methods is substantially easier to implement, one should consider using the simplest implementation, since this sacrifices little energy.

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

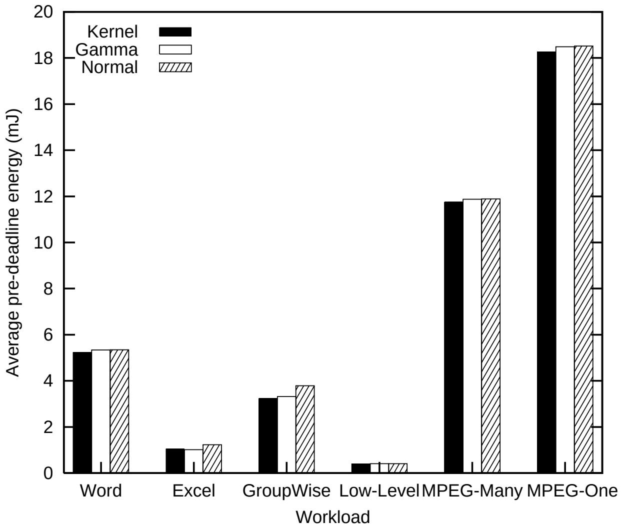


Figure 8: A comparison of the effect of various PACE distribution estimation methods on energy consumption

other methods work reasonably and may be good choices if they are easier to implement.

### 7.1.2 Which distribution estimation method to use

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

To test whether a model truly fits a set of data, one can use that model to estimate the CDF at each data point, and test whether the set of CDF's is distributed uniformly over the interval $(0, 1)$. For this uniformity test, Rayner and Best [RB89] recommend Neyman's $\Psi_4^2$ test. Applying this test to any of our workloads, using any of our sampling methods, the test reveals an extremely small probability that the data fit either the normal or gamma model. Fortunately, the key issue is not the accuracy with which we can approximate the distribution of the task work. The key issue is the extent to which a statistically unacceptable model of this distribution produces a suboptimal solution to the energy minimization problem.[3]

Therefore, the more important question to ask is how effectively PACE can use each model to approximate the optimal schedule. We thus simulate using each model along with the Aged-0.95 sampling method for each workload. We use the same PDC values that we did in the last section. Figure 8 shows the results of these simulations. For almost all workloads, the kernel density model is best, followed by the gamma model, followed by the normal model. In all cases, the gamma model consumes no more than 2.3% more energy than the kernel density model. We conclude that, all things being equal, one should use the kernel density estimation method. However, if this method is too complex to implement, the gamma model can achieve respectably similar results.

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

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

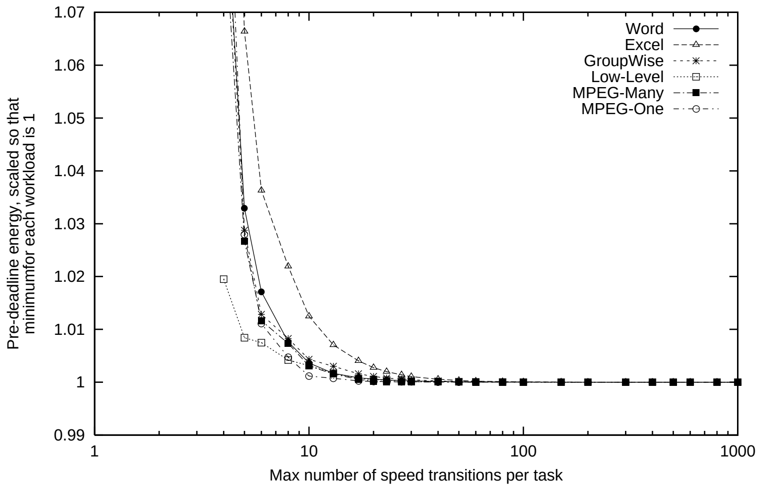


Figure 9: A comparison of the effect on energy consumption of using different numbers of speed transitions to approximate the continuous schedule

## 7.2 Piecewise constant approximations to PACE's optimal formula

In §4.2, we discussed how PACE can approximate the optimal, continuous schedule using a piecewise constant schedule with a limited number of transitions. In this section, we determine empirically the best ways to choose the speed transition points for the schedule.

Figure 9 shows the effect of using different numbers of transitions. We see that the principle of diminishing returns applies: increasing the number of transitions becomes less and less worthwhile as the number of transitions increases. Using 10 transitions yields energy consumption always within 1.2% of the minimum. Using 20 transitions reduces the maximum penalty to 0.27%, and using 30 transitions reduces it to 0.1%. All the results in this paper use a maximum number of transitions of 30. However, even if a practical implementation requires no more than 10 transitions be used for each task, this should not be a problem for PACE; such a practical consideration would increase energy consumption by no more than 1.2%.

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

## 7.3 Theoretical effect of approximations

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

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

### 7.4 Improving classic algorithms with PACE

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

For these simulations, we use four standard interval-based algorithms, each with an interval length of 10 ms. Each consists of a prediction and speed-setting method, as described in §2; we name each algorithm after those methods. The four methods we use are:

- **Past/Weiser-style.** This is a practical version of the algorithm Weiser et al. proposed [WWDS94].
- **LongShort/Chan-style.** This is a practical version of one of the best algorithms Chan et al. proposed [CGW95].
- **Flat/Chan-style.** This is a practical version of another of the best algorithms Chan et al. proposed [CGW95]. It runs the CPU at a constant speed, so it is similar to Transmeta's LongRun$^{\text{TM}}$ in steady state. We choose the speed so that the FPDM is at least 98% (99% for the Low-Level workload).
- **Past/Peg.** This is the algorithm Grunwald et al. favored [GLF$^+$00].

Figure 10 shows the effect of using PACE to modify these classic 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 classic algorithm. PACE using a gamma model reduces the CPU energy consumption of classic algorithms by 2.4–49.0% with an average reduction of 20.3%. PACE using the kernel density estimation method reduces the CPU energy consumption of classic algorithms by 1.4–49.5% with an average reduction of 20.6%. The 1.4% value is lower than the 2.4% value because Excel, the workload that gains the least benefit from PACE, happens also to be the only workload for which the gamma model sometimes outperforms the kernel density estimation method. Excel gains less benefit from PACE than other workloads because it consumes a lot of post-deadline energy, and PACE has no effect on post-deadline schedules. Interestingly, the classic algorithm most improved with PACE is Past/Peg, the one favored by the most recent comparison of classic algorithms [GLF$^+$00]. Past/Peg was favored in that work because it misses fewer deadlines than other algorithms; unfortunately, this requires higher energy consumption, as Figure 10 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, classic algorithms use DVS to reduce CPU energy consumption by 10.7–94.1% with an average of 54.3%. With PACE using a gamma model, the CPU energy savings increase to 35.9–95.5% with an average of 65.2%. With PACE using the kernel density method, the CPU energy savings increase to 35.6–95.5% with an average of 65.4%. Thus, on average, if a CPU consumes 100 J without DVS, classic DVS algorithms allow it to only consume 46 J; PACE reduces that figure even further to 35 J. Given these figures, if the CPU accounted for 33% of total energy consumption in a portable computer without DVS [LS98], classic DVS algorithms would increase its battery lifetime by about 22%; with PACE, the battery lifetime improvement would be about 28%.

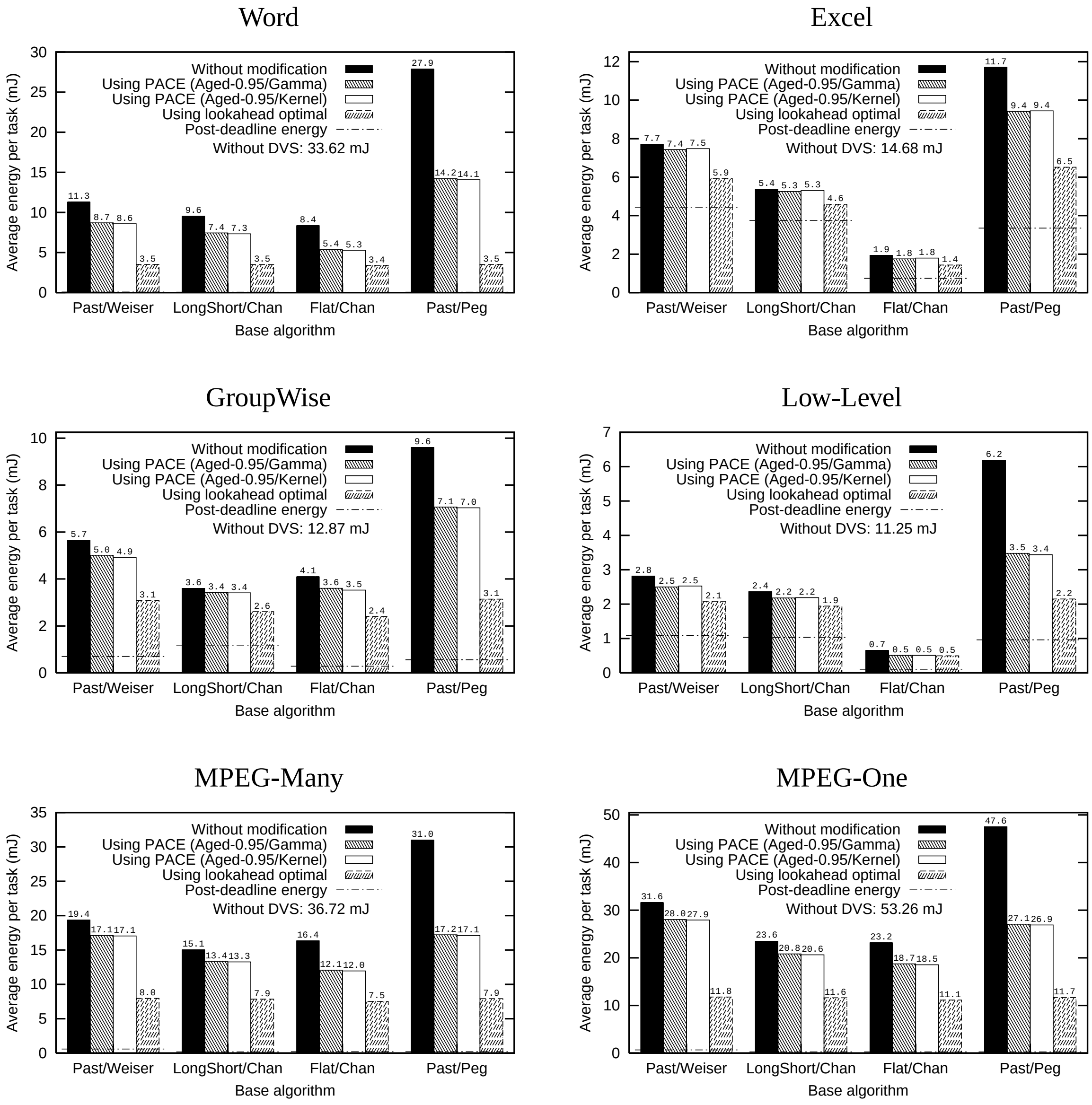


Figure 10: These graphs show the effect of modifying classic algorithms with PACE to get performance equivalent, but lower-energy, algorithms. Results are shown for each workload. Since PACE only modifies the pre-deadline part of the algorithm, it keeps the post-deadline energy the same; this energy is shown with horizontal lines. The Lookahead Optimal results are obtained by giving PACE perfect knowledge of the current task time; these figures are not attainable by a realizable algorithm, but serve as a lower bound on what can be attained by a performance equivalent algorithm. To give perspective on the effectiveness of dynamic voltage scaling at reducing energy consumption, each workload is labeled with the average energy consumption per task if DVS is not used at all.

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

Table 4: This table shows the resulting differences in various metrics when we approximate the PDC values of the Past/Peg and LongShort/Chan-style classic algorithms.

Figure 10 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 classic algorithms. The high energy savings is all the more exciting because PACE by definition has absolutely no effect on performance.

## 7.5 Computing pre-deadline cycles

### 7.5.1 Approximations to classic algorithms

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

Table 4 shows the results of these simulations. Note that results are not shown for the method that approximates the Flat-0.6/Chan-style algorithm, because we know this approximation to be exact. We see that our approximations for Past/Peg and LongShort/Chan-style are almost always within 5% of the classic algorithms on all metrics, with one exception. That exception is GroupWise, where our approximation produced an average delay over 29% less than the classic algorithm. Since this produces much better energy savings and delay than the original algorithm, while only decreasing FPDM by 0.17%, this should not be a problem.

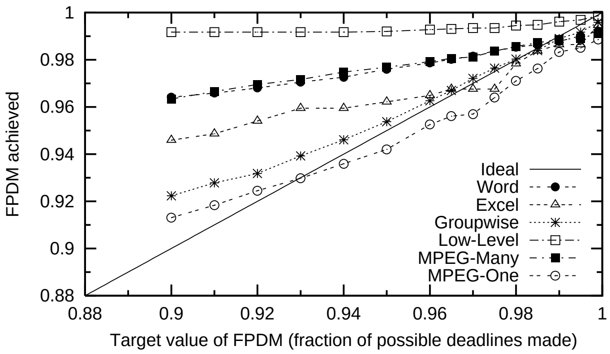


Figure 11: This graph shows the FPDM achieved when various target levels of FPDM$_T$ are sought using the Aged-0.95/Kernel method.

### 7.5.2 Targetting FPDM

Another approach we suggested for setting PDC was to set it to a certain quantile of the task work distribution, in order to achieve a certain target fraction of possible deadlines made TFPDM. Figure 11 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 chosen quantile sometimes is lower than $mD$, the minimum work that can be achieved by the deadline; since the PDC must be at least $mD$, we wind up using too high a quantile and thus achieve too high an FPDM. Second, the model does not work very well at describing the tail of the distribution, so computed quantiles are inaccurate. Third, for some workloads, some values of FPDM are simply too low to be achievable. For example, 94.3% of tasks in Excel are shorter than $mD$, so FPDM cannot be less than 0.943. The other example is Low-Level, for which 93.4% of tasks are shorter than $mD$.

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.

### 7.5.3 Which PDC computation method uses the least energy?

In §5.2, we discussed why we need to use empirical rather than analytical methods to compare the performance of different algorithms for computing PDC. In this section, we perform such empirical tests. Figure 12 plots the average task energy consumption as a function of fraction of possible deadlines made for each of the workloads. 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.

## 7.6 Overhead analysis

Although PACE reduces the energy consumption of algorithms, it also increases their complexity. Thus, it makes the CPU spend more time, and thus more energy, computing speed schedules. We can evaluate this

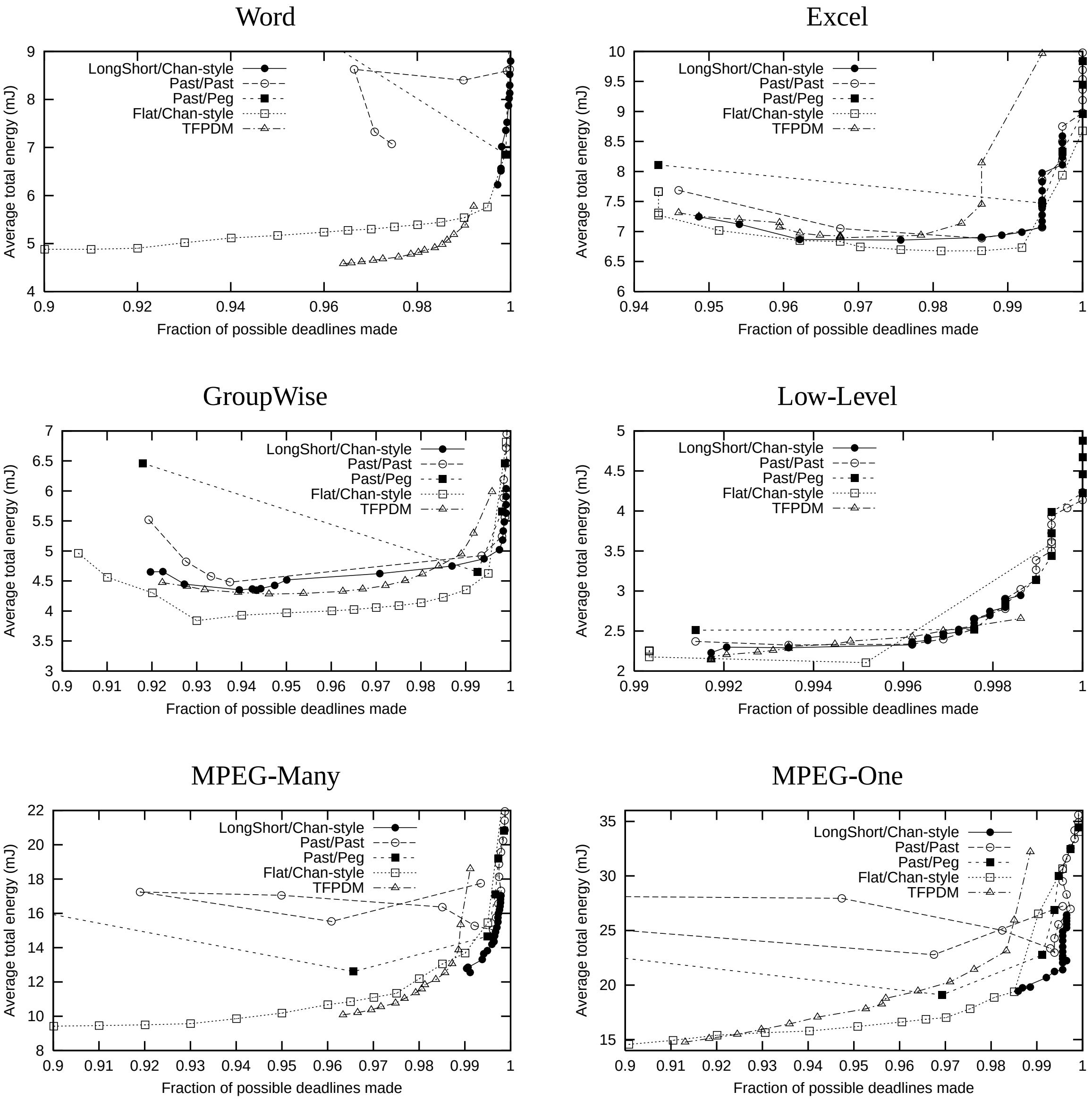


Figure 12: These graphs compare classic algorithms when the pre-deadline part is adjusted by PACE (Aged-0.95/Kernel) and the post-deadline part is changed to always use the maximum CPU speed. We generate each line by choosing a classic algorithm and simulating it for various parameter values. Each point reflects the energy consumed and the fraction of deadlines made for one such simulation, so each line graphs how well the algorithm trades off deadline completions for energy.

| Algorithm | Word | Excel | GroupWise | Low-Level | MPEG-Many | MPEG-One |
|---|---|---|---|---|---|---|
| Aged-0.95/Gamma | 31 $\mu$s<br>(0.05%) | 27 $\mu$s<br>(0.08%) | 30 $\mu$s<br>(0.10%) | 35 $\mu$s<br>(0.25%) | 29 $\mu$s<br>(0.06%) | 28 $\mu$s<br>(0.04%) |
| Recent-28/Kernel | 68 $\mu$s<br>(0.11%) | 70 $\mu$s<br>(0.20%) | 77 $\mu$s<br>(0.25%) | 73 $\mu$s<br>(0.51%) | 63 $\mu$s<br>(0.14%) | 62 $\mu$s<br>(0.08%) |

Table 5: This table shows the average time a 450 MHz Pentium III takes per task to execute variants of the PACE algorithm. The numbers in parentheses indicate the energy overhead: how much energy the simulated CPU would consume to perform the PACE speed schedule computations, as a percentage of the energy it would consume just to execute the workload tasks.

overhead by simulating how much time and energy PACE-modified algorithms would consume to compute schedules.

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

The resultant times are shown in Table 5. This table shows, for each workload and each algorithm, the average time per task to compute a speed schedule on a 450 MHz Pentium III with 128 MB of memory running RedHat Linux 6.2. The table also shows how much energy the simulated CPU would consume to perform this computation, as a percentage of the energy consumed to perform the tasks of the workload. We see from this table that we can implement these algorithms with minimal overhead. The Aged-0.95/Gamma algorithm is more efficient than the Recent-28/Kernel algorithm, but even the Recent-28/Kernel algorithm imposes overhead of at most 77 $\mu$s per task and at most a 0.51% increase in energy consumption. The time overhead is small in all cases, and considering that the computation can be done at the end of each task in anticipation of the next task, it should not in general delay the completion of any task.

# 8 Future Work

## 8.1 Nonlinear relationship between speed and voltage

We stated earlier that the maximum speed permissible at a certain voltage is roughly proportional to that voltage ($s \propto V$). A more accurate formula is $s = k(V - V_{th})^2/V$ where $k$ is some constant of proportionality and $V_{th}$ is the threshold voltage [CSB92]. So, instead of $E \propto s^2$, as we were assuming in our proof of optimality, we have

$$E \propto \left( V_{th} + \frac{s}{2k} + \sqrt{\frac{V_{th}s}{k} + \left(\frac{s}{2k}\right)^2} \right)^2 .$$

This formula is complicated, but we can generally approximate it with a simpler one. For example, consider a CPU with threshold voltage 0.7 V, voltage range 1.1–1.8 V, speed range 100–500 MHz, and maximum power consumption 3 W. Then, Figure 13 shows that an approximation of the form $E = as^2 + b$ is still reasonably close, suggesting our optimal formula may yield reasonable results. (Minimizing

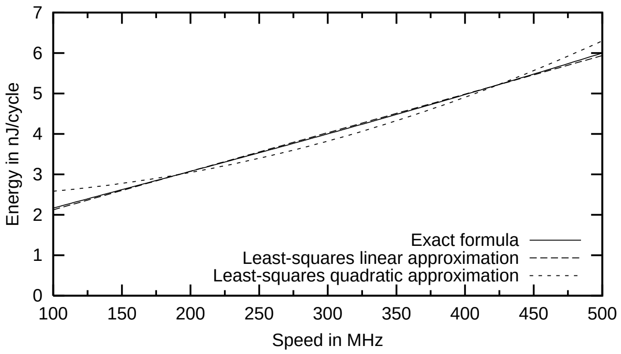


Figure 13: This graph of energy versus speed for a hypothetical processor shows that the exact formula, which takes the threshold voltage into account, is better approximated by a linear approximation $E = as + b$ than by a quadratic one $E = as^2 + b$.

$\int_0^{\mathsf{PDC}} F^c(w)(a[s(w)]^2 + b)\ dw$ is equivalent to minimizing $\int_0^{\mathsf{PDC}} F^c(w)[s(w)]^2\ dw$, so the extra $b$ term does not affect the optimal formula.) However, Figure 13 shows that an even better approximation is the linear formula $E = as + b$. Using this approximation, the optimization problem changes: we must then minimize $\int_0^{\mathsf{PDC}} F^c(w)s(w)\ dw$. This changes the optimal speed schedule in a simple way: it makes the power of $F^c(w)$ change from $-1/3$ to $-1/2$.

In future work we plan to determine the effect of changing our algorithms to use this different power of $F^c(w)$ in the case that the threshold voltage effect is significant and the energy vs. speed curve is better approximated by a linear curve. We believe that the best algorithms will remain the best algorithms even in this case.

Another cause of nonlinearity in the relationship between speed and voltage is memory effects, as observed by Martin et al. [MS99]. When memory speed does not scale precisely with CPU speed, the rate at which work is completed may be nonlinearly related to voltage for some voltage ranges. In future work, we should explore the effect of this nonlinearity on solutions to the energy optimization problem.

## 8.2 I/O

I/O time does not change when CPU speed changes. So, for tasks that perform synchronous I/O, a speed scheduling algorithm must take this into account. The task must complete its CPU work *and* its synchronous I/O within the deadline. Another way to look at this is to consider the deadline for the CPU part of the task to be reduced by the amount of time spent performing I/O. In other words, we can model I/O by having the deadline change dynamically: any time spent performing I/O reduces the deadline for the CPU work. In future work, we hope to develop a reasonable algorithm for scheduling tasks that takes into account such dynamic deadlines. The algorithm must anticipate and consider the probability that a task will perform I/O in the future and that the deadline will be reduced accordingly. It must also incorporate an algorithm for recomputing the schedule for a task whenever it completes an I/O and thus has had its deadline reduced.

## 8.3 Multiple deadlines

We have assumed that once a task misses its deadline, there are no further constraints on its completion time, other than a general desire to minimize delay. However, some tasks may have multiple deadlines. For instance, we may want a task to complete within 50 ms with probability 95%, within 100 ms with probability 99%, within 500 ms with probability 99.9%, etc. There is an optimal formula for scheduling a task with multiple deadlines, but it is beyond the scope of this paper.

In this work, we do not address the case in which multiple tasks must be scheduled at once. If only one of these tasks has a deadline, then it may be reasonable to model all the other tasks as requiring some fixed amount of CPU usage before the deadline. Then, we can add this amount to the work requirement of the task with a deadline to get the aggregate work requirement. In other words, we simply shift the probability distribution function of work to the right by an amount equal to the expected number of cycles for other tasks.

If multiple tasks have deadlines, the optimal schedule depends on the joint probability function of the two tasks' work requirements, which is likely too complicated to use in practice. We must therefore, in future work, develop heuristics for properly scheduling two or more tasks simultaneously. Note, however, that most mobile computers have limited resources and only one user, so we feel such computers will rarely be working on more than one task with a deadline at a time.

### 8.4 Limited set of valid speeds

Some processors may offer only a fixed, limited set of valid speeds. Currently, we have no way to adjust the optimal formula to take such limitations into account. Intuitively, rounding to the nearest available speed should work reasonably well. However, it will not necessarily give the optimal solution. Investigating the effect of different rounding methods, and perhaps approaches other than rounding, is future work.

### 8.5 Overhead of changing speed and voltage

In this paper, we have assumed that CPU speed and voltage transitions consume no time or energy. However, in reality, this is not the case. According to Burd et al. [BB00], changing between two levels takes time roughly proportional to the voltage differential and energy roughly proportional to the difference between the squares of the voltages. So, if a schedule only increases speed as time progresses, the total transition time and energy depend only on the initial and final voltages, and not on the number of transitions. However, these are only approximations, and they do not completely account for per-transition costs. For example, there may be a delay every time the speed changes in order to stabilize the clock and synchronize the CPU and bus clocks. Such per-transition costs are especially noticeable on modern architectures, since DVS is a relatively young technology and designers have not spent great effort to keep such transition costs low. In future work, we will address the issue of how PACE should take into account such actual transition costs. We expect that PACE will work well even under these conditions, especially for future architectures that will have very low transition time and energy.

### 8.6 Task type groupings

We mentioned the desirability of grouping tasks by type, and keeping separate samples for each type. This way, we only use a task's work requirement to model the work requirements of similar tasks. In future work, we plan to investigate what groupings work best, and how effective different grouping methods are.

## 9 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 it is possible to change how an algorithm schedules tasks in a way that has no effect on performance but can reduce energy consumption. Furthermore, we developed an optimal formula for scheduling tasks with minimal energy consumption.

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

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

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

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

## 10 Acknowledgments

We thank Michael Jordan for his extremely helpful suggestions regarding statistical methods, namely the use of the gamma distribution and kernel density estimation. We also thank Ronald Wolff for his suggestions. Finally, we offer great thanks to the anonymous users of VTrace whose traces enabled us to create the workloads for this paper.

## A PACE Optimal Formula and Supporting Proofs

We address the following optimization problem. We are given the tail distribution of task work $F^c$, a deadline $D$, and minimum and maximum CPU speeds $m$ and $M$. We assume $D > 0$ and $M > m > 0$. We are also given the pre-deadline cycles $\mathsf{PDC}$ of some speed schedule. Thus, $mD \leq \mathsf{PDC} \leq MD$. We say a schedule $s$ is *valid* if

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

and if $m \leq s(w) \leq M$ for all $0 \leq w \leq \mathsf{PDC}$. We say a valid schedule $\hat{s}$ is an *optimal valid schedule* if, for any valid schedule $s$,

$$\int_0^{\mathsf{PDC}} F^c(w)[s(w)]^2\, dw \geq \int_0^{\mathsf{PDC}} F^c(w)[\hat{s}(w)]^2\, dw.$$

We seek an optimal valid schedule.

We assume that $F^c(mD) > 0$. If this were not the case, then the task could never require more than $mD$ cycles, and the best schedule would be obvious: always run at speed $m$. Such a schedule would always consume the least power possible, and would never miss the deadline.

## A.1 Supporting notation and lemmas

We will first introduce some notation, then prove some important lemmas: the proportional improvement lemma and the continuous proportional improvement lemma.

**Notation.** We define the notation $x_1 \rightleftharpoons x_2 \rightleftharpoons \cdots \rightleftharpoons x_n$ to mean that either $x_1 \le x_2 \le \cdots \le x_n$ or $x_1 \ge x_2 \ge \cdots \ge x_n$.

**Proportional Improvement Lemma. If** $n \ge 0$**;** $\quad p \ge 1$**;** $\quad \sum_{i=1}^n y_i \le \sum_{i=1}^n z_i$**; and,** $\forall i \in \{1, 2, \ldots, n\}$**,** $x_i \ge 0$**,** $\quad y_i > 0$**,** $\quad z_i > 0$**, and** $y_i \rightleftharpoons z_i \rightleftharpoons cx_i$**; then** $\sum_{i=1}^n x_i^p y_i^{1-p} \ge \sum_{i=1}^n x_i^p z_i^{1-p}$**.**

**Proof.** We prove this by induction on $n > 1$, since it is trivial for $n = 0$ and $n = 1$.

First, we prove the lemma for $n = 2$.

Without loss of generality, $y_1 \le z_1$. $y_2 \le z_2$ is a trivial case, so assume $y_2 > z_2$. Now, we cannot have $y_1 = z_1$, since this combined with $y_2 > z_2$ would contradict $y_1 + y_2 \le z_1 + z_2$. So, $y_1 < z_1$.

Since $y_1 < z_1$ and $y_1 \rightleftharpoons z_1 \rightleftharpoons cx_1$, we have $y_1 < z_1 \le cx_1$. Thus, $cx_1 > 0$, and we can conclude that $c > 0$ and $x_1/z_1 \ge x_1/(cx_1) = 1/c$.

Since $y_2 > z_2$ and $y_2 \rightleftharpoons z_2 \rightleftharpoons cx_2$, we have $y_2 > z_2 \ge cx_2$. If $x_2 = 0$, then $x_2/z_2 = 0 \le 1/c$; if $x_2 > 0$, then $x_2/z_2 \le x_2/(cx_2) = 1/c$. So, in all cases, $x_2/z_2 \le 1/c$.

Since $x_1/z_1 \ge 1/c$ and $x_2/z_2 \le 1/c$, we have $x_1/z_1 \ge x_2/z_2$.

Define $\gamma = z_1 + z_2 - y_1$ and $\delta = z_1 - y_1 = \gamma - z_2$. Note that $\delta > 0$ because $z_1 > y_1$. For $0 \le u \le \delta$, define

$$Z(u) = x_1^p (y_1 + u)^{1-p} + x_2^p (\gamma - u)^{1-p}.$$

$Z(u)$ is well-defined over this interval because $\gamma - u \ge \gamma - \delta = z_2 > 0$ over this interval. Its derivative over this interval is

$$\frac{dZ}{du} = (1-p)\left[x_1^p (y_1+u)^{-p} - x_2^p (\gamma - u)^{-p}\right] = (p-1)\left[\left(\frac{x_2}{\gamma - u}\right)^p - \left(\frac{x_1}{y_1 + u}\right)^p\right].$$

Since $p \ge 1$, $\frac{dZ}{du} \le 0$ as long as $x_2/(\gamma - u) \le x_1/(y_1 + u)$. Now, for all $0 \le u \le \delta$,

$$\frac{x_2}{\gamma - u} \le \frac{x_2}{\gamma - \delta} = \frac{x_2}{z_2} \le \frac{x_1}{z_1} = \frac{x_1}{y_1 + \delta} \le \frac{x_1}{y_1 + u},$$

so we have $\frac{dZ}{du} \le 0$ over this entire interval. So, $Z(u)$ is nonincreasing there, which means $Z(0) \ge Z(\delta)$. In addition, since $y_1 + y_2 \le z_1 + z_2$, we have $y_2 \le z_1 + z_2 - y_1 = \gamma$. We can use these two results, $Z(0) \ge Z(\delta)$ and $y_2 \le \gamma$, to observe that

$$\sum_{i=1}^n x_i^p y_i^{1-p} = x_1^p y_1^{1-p} + x_2^p y_2^{1-p} \ge x_1^p y_1^{1-p} + x_2^p \gamma^{1-p} = Z(0) \ge Z(\delta) = x_1^p z_1^{1-p} + x_2^p z_2^{1-p} = \sum_{i=1}^n x_i^p z_i^{1-p}.$$

This completes the proof for the case $n = 2$.

Now, supposing the hypothesis holds for $2 \le n < N$, we show it holds for $n = N$. $y_i \le z_i$ for all $i$ is a trivial case, so assume there is some $j$ such that $y_j > z_j$. Then, $\exists k$ such that $y_k < z_k$. Without loss of generality, $N-1$ and $N$ form such a pair $\{j,k\}$ (not necessarily in that order) with $|z_N - y_N| \leq |z_{N-1} - y_{N-1}|$.

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

Thus, in either case, we have $y_{N-1} \rightleftharpoons z_{N-1} + z_N - y_N \rightleftharpoons z_{N-1}$. This fact, combined with $y_{N-1} \rightleftharpoons z_{N-1} \rightleftharpoons cx_{N-1}$, gives $y_{N-1} \rightleftharpoons z_{N-1} + z_N - y_N \rightleftharpoons z_{N-1} \rightleftharpoons cx_{N-1}$. It also shows that $z_{N-1} + z_N - y_N > 0$, since $y_{N-1} > 0$ and $z_{N-1} > 0$.

Define $\bar{z}_i$ for $i \in \{1, 2, \ldots, N-2\}$ by $\bar{z}_i = z_i$ and define $\bar{z}_{N-1} = z_{N-1} + z_N - y_N$. Then, we have

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

Clearly, $y_i \rightleftharpoons \bar{z}_i \rightleftharpoons cx_i$ and $\bar{z}_i > 0$ for $i \in \{1, 2, \ldots, N-2\}$, since for such $i$ we have $\bar{z}_i = z_i$. Furthermore, for $i = N-1$, we also have $y_i \rightleftharpoons \bar{z}_i \rightleftharpoons cx_i$ and $\bar{z}_i > 0$, since we showed earlier that $y_{N-1} \rightleftharpoons z_{N-1} + z_N - y_N \rightleftharpoons cx_{N-1}$ and $z_{N-1} + z_N - y_N > 0$. By the inductive hypothesis, the lemma holds for $n = N-1$. Thus, we can apply it to the sequences $\{x_1, x_2, \ldots, x_{N-1}\}$, $\{y_1, y_2, \ldots, y_{N-1}\}$, and $\{\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_{N-1}\}$ to get $\sum_{i=1}^{N-1} x_i^p y_i^{1-p} \geq \sum_{i=1}^{N-1} x_i^p \bar{z}^{1-p}$. In other words,

$$\sum_{i=1}^{N-1} x_i^p y_i^{1-p} \geq x_{N-1}^p (z_{N-1} + z_N - y_N)^{1-p} + \sum_{i=1}^{N-2} x_i^p z_i^{1-p}. \tag{1}$$

Now, define $\dot{x}_1 = x_{N-1}$; $\dot{x}_2 = x_N$; $\dot{y}_1 = z_{N-1} + z_N - y_N$; $\dot{y}_2 = y_N$; $\dot{z}_1 = z_{N-1}$; and $\dot{z}_2 = z_N$. Then, we have $\sum_{i=1}^{2} \dot{y}_i = \sum_{i=1}^{2} \dot{z}_i$ and, for all $i \in \{1, 2\}$, $\dot{x}_i \geq 0$, $\dot{y}_i > 0$, $\dot{z}_i > 0$, and $\dot{y}_i \rightleftharpoons \dot{z}_i \rightleftharpoons c\dot{x}_i$. By the inductive hypothesis, the lemma holds for $n = 2$, so we can apply it to the sequences $\{\dot{x}_1, \dot{x}_2\}$, $\{\dot{y}_1, \dot{y}_2\}$, and $\{\dot{z}_1, \dot{z}_2\}$ to get $\sum_{i=1}^{2} \dot{x}_i^p \dot{y}_i^{1-p} \geq \sum_{i=1}^{2} \dot{x}_i^p \dot{z}_i^{1-p}$. In other words,

$$x_{N-1}^p (z_{N-1} + z_N - y_N)^{1-p} + x_N^p y_N^{1-p} \geq x_{N-1}^p z_{N-1}^{1-p} + x_N^p z_N^{1-p}.$$

Combining this with (1), we get

$$\sum_{i=1}^{N-1} x_i^p y_i^{1-p} \geq x_{N-1}^p z_{N-1}^{1-p} + x_N^p z_N^{1-p} - x_N^p y_N^{1-p} + \sum_{i=1}^{N-2} x_i^p z_i^{1-p},$$

which directly leads to $\sum_{i=1}^{N} x_i^p y_i^{1-p} \geq \sum_{i=1}^{N} x_i^p z_i^{1-p}$. This completes the induction step, and thus the proof. □

**Continuous Proportional Improvement Lemma. If** $a \leq b$**;** $p \geq 1$**;** $c > 0$**;** $L > 0$**;** $U > 0$**;** $\int_a^b g(u)\,du \leq \int_a^b h(u)\,du$**; and,** $\forall u \in [a,b]$**,** $\psi(u) \in [0,U]$**,** $g(u) \geq L$**,** $h(u) \geq L$**, and** $g(u) \rightleftharpoons h(u) \rightleftharpoons c\psi(u)$**; then** $\int_a^b \psi(u)^p g(u)^{1-p}\,du \geq \int_a^b \psi(u)^p h(u)^{1-p}\,du$.

**Proof.** Since $\psi(u)^p g(u)^{1-p}$ and $\psi(u)^p h(u)^{1-p}$ are both nonnegative and bounded above by $U^p L^{1-p}$ for $a \leq u \leq b$, the integrals $\int_a^b \psi(u)^p g(u)^{1-p}\,du$ and $\int_a^b \psi(u)^p h(u)^{1-p}\,du$ exist.

If $a = b$, then $\int_a^b \psi(u)^p g(u)^{1-p}\,du = 0 = \int_a^b \psi(u)^p h(u)^{1-p}\,du$, so the result trivially holds. If $p = 1$, then $\int_a^b \psi(u)^p g(u)^{1-p}\,du = \int_a^b \psi(u)\,du = \int_a^b \psi(u)^p h(u)^{1-p}\,du$, so the result trivially holds. Therefore, from this point on, we assume $a < b$ and $p > 1$.

Define $\gamma$ by

$$\gamma = \int_a^b \psi(u)^p [h(u)^{1-p} - g(u)^{1-p}]\,du.$$

Suppose, for the purpose of proof by contradiction, that $\gamma > 0$. By the definition of the integral, this means that there exists some $\delta_1 > 0$ such that

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

Define $\epsilon = (\gamma/2)(c/2)^p$. Since $p > 1$, we have $\epsilon > 0$. By the definition of the integral, there exists some $\delta_2 > 0$ such that

$$v_0 = a; \quad v_n = b; \quad \text{and, } \forall i \in \{1, 2, \ldots, n\}, \quad 0 < v_i - v_{i-1} < \delta_2 \text{ and } v_{i-1} \le u_i \le v_i \Rightarrow \\ \left| \int_a^b [h(u) - g(u)]\,du - \sum_{i=1}^{n} [h(u_i) - g(u_i)](v_i - v_{i-1}) \right| < \epsilon. \quad (3)$$

Let $\delta = \max\{\delta_1, \delta_2\} > 0$. Consider any sequences $\{v_0, v_1, \ldots, v_n\}$ and $\{u_1, u_2, \ldots, u_n\}$ such that $v_0 = a, \quad v_n = b$, and, for all $i \in \{1, 2, \ldots, n\}, \quad 0 < v_i - v_{i-1} < \delta$ and $v_{i-1} \le u_i \le v_i$.

For $i \in \{1, \ldots, n\}$, define $x_i = \psi(u_i)(v_i - v_{i-1}), \quad y_i = g(u_i)(v_i - v_{i-1})$, and $z_i = h(u_i)(v_i - v_{i-1})$. Also, define $x_{n+1} = 2\epsilon/c, \quad y_{n+1} = \epsilon$, and $z_{n+1} = 2\epsilon$. This way,

$$\begin{aligned} \sum_{i=1}^{n+1} (z_i - y_i) &= (2\epsilon - \epsilon) + \sum_{i=1}^{n} [h(u_i) - g(u_i)](v_i - v_{i-1}) \\ &> \epsilon + \int_a^b [h(u) - g(u)]\,du - \epsilon && \text{by (3)} \\ &\ge 0. && \text{because } \int_a^b g(u)\,du \le \int_a^b h(u)\,du \end{aligned}$$

We thus see that $\sum_{i=1}^{n+1} y_i \le \sum_{i=1}^{n+1} z_i$. For $i \in \{1, 2, \ldots, n+1\}$, it's clear that $x_i \ge 0, \quad y_i > 0$, and $z_i > 0$. For $i \in \{1, \ldots, n\}, \quad g(u_i) \rightleftharpoons h(u_i) \rightleftharpoons c\psi(u_i)$, so we also have $y_i \rightleftharpoons z_i \rightleftharpoons cx_i$. For $i = n+1, \quad z_i = cx_i$, so we have $y_i \rightleftharpoons z_i \rightleftharpoons cx_i$ in this case as well. Applying the proportional improvement lemma to the sequences $\{x_1, x_2, \ldots, x_{n+1}\}$, $\{y_1, y_2, \ldots, y_{n+1}\}$, and $\{z_1, z_2, \ldots, z_{n+1}\}$, we get $\sum_{i=1}^{n+1} x_i^p y_i^{1-p} \ge \sum_{i=1}^{n+1} x_i^p z_i^{1-p}$. In other words,

$$(2\epsilon/c)^p \epsilon^{1-p} + \sum_{i=1}^{n} \psi(u_i)^p g(u_i)^{1-p}(v_i - v_{i-1}) \ge (2\epsilon/c)^p (2\epsilon)^{1-p} + \sum_{i=1}^{n} \psi(u_i)^p h(u_i)^{1-p}(v_i - v_{i-1}).$$

In yet other words,

$$\sum_{i=1}^{n} \psi(u_i)^p [h(u_i)^{1-p} - g(u_i)^{1-p}](v_i - v_{i-1}) \le (2\epsilon/c)^p [\epsilon^{1-p} - (2\epsilon)^{1-p}] < \epsilon(2/c)^p = \gamma/2.$$

This contradicts (2). So, our supposition must be false, and we must have $\gamma \le 0$. In other words, $\int_a^b \psi(u)^p [h(u)^{1-p} - g(u)^{1-p}]\,du \le 0$, which leads directly to the desired result, $\int_a^b \psi(u)^p g(u)^{1-p}\,du \ge \int_a^b \psi(u)^p h(u)^{1-p}\,du$. □

### A.2 Optimal formula

We will now show how to construct an optimal valid schedule. Then, we will prove that this construction works.

**Optimal Schedule Theorem. For $\sigma > 0$, define**

$$\Phi(\sigma, w) = \begin{cases} m & \textbf{if } F^c(w) \geq (\sigma/m)^3 \\ M & \textbf{if } F^c(w) \leq (\sigma/M)^3 \\ \sigma[F^c(w)]^{-1/3} & \textbf{otherwise.} \end{cases}$$

**If $F^c(mD) > 0$, then there exists an $S_0 > 0$ such that**

$$\int_0^{\mathsf{PDC}} \frac{1}{\Phi(S_0, w)}\, dw = D.$$

**Furthermore, given such an $S_0$, if we define $\hat{s}(w) = \Phi(S_0, w)$ then $\hat{s}$ is an optimal valid schedule.**

**Proof.** Note first that for any $\sigma > 0$ and any $w$, $\Phi(\sigma, w) \in [m, M]$. If $F^c(w) \geq (\sigma/m)^3$ or $F^c(w) \leq (\sigma/M)^3$, this is clear, and if $(\sigma/M)^3 < F^c(w) < (\sigma/m)^3$, it holds because $M > \sigma[F^c(w)]^{-1/3} > m$.

Also, note that for any $\sigma_1$, $\sigma_2$, and $w$ such that $\sigma_1 \geq \sigma_2 > 0$, we have $\Phi(\sigma_1, w) \geq \Phi(\sigma_2, w)$. This is easily verified by checking the three cases in the definition of $\Phi$.

For $\sigma > 0$, define

$$Z(\sigma) = \int_0^{\mathsf{PDC}} \frac{1}{\Phi(\sigma, w)}\, dw.$$

This is well-defined, since $\frac{1}{\Phi(\sigma,w)}$ is nonnegative and bounded above over the interval of integration. Define $Y(x) = Z(1/x)$ for $x > 0$; this is also well-defined.

Next, define $\psi(w) = [F^c(w)]^{1/3}$. $F^c$ must be nonincreasing since it is a tail distribution function. Thus,

$$\int_0^{\mathsf{PDC}} \psi(w)\, dw = \int_0^{\mathsf{PDC}} [F^c(w)]^{1/3}\, dw \geq \int_0^{mD} [F^c(w)]^{1/3}\, dw \geq \\ \int_0^{mD} [F^c(mD)]^{1/3}\, dw = mD[F^c(mD)]^{1/3}. \quad (4)$$

By assumption, $F^c(mD) > 0$, so

$$\int_0^{\mathsf{PDC}} \psi(w)\, dw > 0. \quad (5)$$

Now, we want to show that

$$u > 0, v > 0 \quad \Rightarrow \quad \left| \frac{1}{\Phi(1/u, w)} - \frac{1}{\Phi(1/v, w)} \right| \leq |u\psi(w) - v\psi(w)|. \quad (6)$$

If $F^c(w) \geq (\sigma/m)^3$, then the left side of this equation is $|1/m - 1/m| = 0$ and the right side is nonnegative, so it holds. If $F^c(w) \leq (\sigma/M)^3$, then the left side of this equation is $|1/M - 1/M| = 0$ and the right side is nonnegative, so it holds. If $(\sigma/M)^3 < F^c(w) < (\sigma/m)^3$, then both sides of the equation are equal to $\left|u[F^c(w)]^{1/3} - v[F^c(w)]^{1/3}\right|$, so it holds. We have covered all the cases, so we have proven (6).

Suppose $\delta > 0$. Let

$$\epsilon = \frac{\delta}{\int_0^{\mathsf{PDC}} \psi(w)\, dw}.$$

From (5), we know $\epsilon$ is well-defined and $\epsilon > 0$. Furthermore, for any $u > 0$ and $v > 0$ satisfying $|u-v| < \epsilon$, we can use (6) to show the following:

$$\begin{aligned}
|Y(u) - Y(v)| &= \left| \int_0^{\mathsf{PDC}} \left[ \frac{1}{\Phi(1/u, w)} - \frac{1}{\Phi(1/v, w)} \right] dw \right| \\
&\leq \int_0^{\mathsf{PDC}} \left| \frac{1}{\Phi(1/u, w)} - \frac{1}{\Phi(1/v, w)} \right| dw \\
&\leq \int_0^{\mathsf{PDC}} |u\psi(w) - v\psi(w)| \; dw \\
&< \epsilon \int_0^{\mathsf{PDC}} \psi(w) \, dw \\
&= \delta.
\end{aligned}$$

We have shown that for all $\delta > 0$, there exists an $\epsilon > 0$ such that $u > 0, \quad v > 0, \quad |u - v| < \epsilon \Rightarrow |Y(u) - Y(v)| < \delta$. In other words, we have shown that $Y(x)$ is continuous over $x > 0$.

Now, $Z(\sigma)$ is the composition of $Y(x)$ and the reciprocal function $r(\sigma) = 1/\sigma$. The latter function is continuous over $\sigma > 0$ and has range $(0, \infty)$. $Y(x)$ is continuous there, so $Z(\sigma)$ is continuous over $\sigma > 0$.

$F^c$ is a tail distribution function. So, for all $w$, $F^c(w) \leq 1 = (M/M)^3$ and thus $\Phi(M, w) = M$. Consequently,

$$Z(M) = \int_0^{\mathsf{PDC}} \frac{1}{\Phi(M, w)} \, dw = \frac{\mathsf{PDC}}{M} \leq D.$$

$F^c$ is nonincreasing, so for all $w \leq mD, \quad F^c(w) \geq F^c(mD) = (m\psi(mD)/m)^3$, so $\Phi(m\psi(mD), w) = m$. From this, we can reason

$$Z(m\psi(mD)) = \int_0^{\mathsf{PDC}} \frac{1}{\Phi(m\psi(mD), w)} \, dw \geq \int_0^{mD} \frac{1}{\Phi(m\psi(mD), w)} \, dw = \frac{mD}{m} = D.$$

We thus see that $Z(M) \leq D$ and $Z(m\psi(mD)) \geq D$. Since $Z(\sigma)$ is continuous over $\sigma > 0$, and $M$ and $m\psi(mD)$ are both positive, there must exist some $S_0 > 0$ such that $Z(S_0) = D$. This means

$$\int_0^{\mathsf{PDC}} \frac{1}{\Phi(S_0, w)} \, dw = D,$$

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

Now, suppose we have such an $S_0 > 0$, and define $\hat{s}(w) = \Phi(S_0, w)$. We need to show $\hat{s}$ is an optimal valid schedule. We already showed that the range of $\Phi$ is $[m, M]$, so clearly $m \leq \hat{s}(w) \leq M$ for all $0 \leq w \leq \mathsf{PDC}$.

Consider any valid schedule $s$. Define $g(w) = 1/s(w)$ and $h(w) = 1/\hat{s}(w)$ for all $0 \leq w \leq \mathsf{PDC}$. Also, let $a = 0, \quad b = \mathsf{PDC}, \quad p = 3, \quad c = 1/S_0, \quad L = 1/M$, and $U = 1$.

Then, $a \leq b; \quad c > 0; \quad L > 0; \quad U > 0$; and, $\forall u \in [a, b], \quad \psi(u) \in [0, U], \quad g(u) \geq L$, and $h(u) \geq L$.

Also,

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

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

Now, consider any $w$.

- If $F^c(w) \geq (S_0/m)^3$, then $\hat{s}(w) = m$ and $\psi(w)/S_0 \geq 1/m$, so $1/s(w) \leq 1/m = 1/\hat{s}(w) \leq \psi(w)/S_0$.

- If $F^c(w) \leq (S_0/M)^3$, then $\hat{s}(w) = M$ and $\psi(w)/S_0 \leq 1/M$, so $1/s(w) \geq 1/M = 1/\hat{s}(w) \geq \psi(w)/S_0$.
- Otherwise, $\hat{s}(w) = S_0[F^c(w)]^{-1/3}$, so $1/\hat{s}(w) = \psi(w)/S_0$.

In any case, we see that $1/s(w) \rightleftharpoons 1/\hat{s}(w) \rightleftharpoons \psi(w)/S_0$, so $g(w) \rightleftharpoons h(w) \rightleftharpoons c\psi(w)$.

We now know enough to apply the continuous proportional improvement lemma using the functions $\psi$, $g$, and $h$. This gives $\int_0^{\mathsf{PDC}} \psi(w)^3 g(w)^{-2}\, dw \geq \int_0^{\mathsf{PDC}} \psi(w)^3 h(w)^{-2}\, dw$, which is equivalent to

$$\int_0^{\mathsf{PDC}} F^c(w)[s(w)]^2\, dw \geq \int_0^{\mathsf{PDC}} F^c(w)[\hat{s}(w)]^2\, dw.$$

We have thus shown all that is needed to prove that $\hat{s}$ is an optimal valid schedule. □

## A.3 Piecewise constant optimal formula

We now consider how to create a schedule if we are restricted in the points at which we can change speed. We assume, for the purposes of this section, that we are given a strictly increasing sequence $\{w_0, w_1, \ldots, w_n\}$ such that $w_0 = 0$ and $w_n = \mathsf{PDC}$. We say a function $s$ is *properly-divided* if it is constant over each interval $(w_{i-1}, w_i]$, i.e., if, for all $i \in \{1, 2, \ldots, n\}$ and any $w_{i-1} < w \leq w_i, \quad s(w) = w_i$. We say a function $\hat{s}$ is an *optimal properly-divided valid schedule* if it is properly-divided and valid and if, for any properly-divided valid function $s$,

$$\int_0^{\mathsf{PDC}} F^c(w)[s(w)]^2\, dw \geq \int_0^{\mathsf{PDC}} F^c(w)[\hat{s}(w)]^2\, dw.$$

We will now show how to construct an optimal properly-divided valid schedule. Then, we will prove that this construction works.

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

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

**Also, for all $\sigma > 0$ and $i \in \{1, 2, \ldots, n\}$, define**

$$\Phi(\sigma, i) = \begin{cases} m & \textbf{if } H_i \geq (\sigma/m)^3 \\ M & \textbf{if } H_i \leq (\sigma/M)^3 \\ \sigma H_i^{-1/3} & \textbf{otherwise}. \end{cases}$$

**If $F^c(mD) > 0$, then there exists an $S_0 > 0$ such that**

$$\sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\Phi(S_0, i)} = D.$$

**Furthermore, given such an $S_0$, if we define $\hat{s}(w)$ by**

$$\hat{s}(w) = \begin{cases} \Phi(S_0, 1) & \textbf{if } w = 0 \\ \Phi(S_0, i) & \textbf{if } w_{i-1} < w \leq w_i \textbf{ for some } i \in \{1, 2, \ldots, n\}, \end{cases}$$

**then $\hat{s}$ is an optimal properly-divided valid schedule.**

**Proof.** Note first that for any $\sigma > 0$ and any $i \in \{1, 2, \ldots, n\}$, $\Phi(\sigma, i) \in [m, M]$. This is because if $(\sigma/M)^3 < H_i < (\sigma/m)^3$, then $M > \sigma H_i^{-1/3} > m$, and because in other cases for $H_i$, $\Phi(\sigma, i) = m$ or $M$.

Consider the function $\Phi(\sigma, i)$ for any $i \in \{1, 2, \ldots, n\}$. It is continuous over $0 < \sigma < mH_i^{1/3}$, since it is equal to $m$ there. It is continuous over $mH_i^{1/3} < \sigma < MH_i^{1/3}$, since it is equal to $\sigma H_i^{-1/3}$ there. It is continuous over $\sigma > MH_i^{1/3}$, since it is equal to $M$ there. Consider any $\delta > 0$ and let $\epsilon = \min\{\delta H_i^{1/3}, (M - m)H_i^{1/3}\} > 0$. Then,

$$\begin{aligned}
|u - mH_i^{1/3}| < \epsilon \Rightarrow u \le mH_i^{1/3} \quad &\textbf{or} \quad mH_i^{1/3} < u < mH_i^{1/3} + \epsilon \le MH_i^{1/3} \\
\Rightarrow \Phi(u, i) = m \quad &\textbf{or} \quad |\Phi(u, i) - m| = |uH_i^{-1/3} - m| = uH_i^{-1/3} - m < \epsilon H_i^{-1/3} \\
\Rightarrow |\Phi(u, i) - \Phi(mH_i^{1/3}, i)| &= |\Phi(u, i) - m| < \delta
\end{aligned}$$

and

$$\begin{aligned}
|u - MH_i^{1/3}| < \epsilon \Rightarrow u \ge MH_i^{1/3} \quad &\textbf{or} \quad MH_i^{1/3} > u > MH_i^{1/3} - \epsilon \ge mH_i^{1/3} \\
\Rightarrow \Phi(u, i) = M \quad &\textbf{or} \quad |\Phi(u, i) - M| = |uH_i^{-1/3} - M| = M - uH_i^{-1/3} < \epsilon H_i^{-1/3} \\
\Rightarrow |\Phi(u, i) - \Phi(MH_i^{1/3}, i)| &= |\Phi(u, i) - M| < \delta.
\end{aligned}$$

So, $\Phi(\sigma, i)$ is also continuous at $\sigma = mH_i^{1/3}$ and $\sigma = MH_i^{1/3}$. We conclude that $\Phi(\sigma, i)$ is continuous over $\sigma > 0$ for any $i \in \{1, 2, \ldots, n\}$.

Define

$$Z(\sigma) = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\Phi(\sigma, i)}$$

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

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

$$H_i = \frac{\int_{w_{i-1}}^{w_i} F^c(w)\, dw}{w_i - w_{i-1}} \le \frac{\int_{w_{i-1}}^{w_i} dw}{w_i - w_{i-1}} = \frac{w_i - w_{i-1}}{w_i - w_{i-1}} = 1 = (M/M)^3.$$

So, $\Phi(M, i) = M$, and we have

$$Z(M) = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\Phi(M, i)} = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{M} = \frac{\mathsf{PDC}}{M} \le D.$$

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

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

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

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

So, for all $i \in \{1, 2, \ldots, j\}, \quad H_i \geq H_j = (mH_j^{1/3}/m)^3$, meaning $\Phi(mH_j^{1/3}, i) = m$. Thus,

$$Z(mH_j^{1/3}) = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\Phi(mH_j^{1/3}, i)} \geq \sum_{i=1}^{j} \frac{w_i - w_{i-1}}{\Phi(mH_j^{1/3}, i)} = \sum_{i=1}^{j} \frac{w_i - w_{i-1}}{m} = \frac{w_j}{m} \geq \frac{mD}{m} = D.$$

We thus see that $Z(M) \leq D$ and $Z(mH_j^{1/3}) \geq D$. Since $Z(\sigma)$ is continuous over $\sigma > 0$, and $M$ and $mH_j^{1/3}$ are both positive, there must exist some $S_0 > 0$ such that $Z(S_0) = D$. This means

$$\sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\Phi(S_0, i)}\, dw = D,$$

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

Now, suppose we have such an $S_0 > 0$, and define $\hat{s}(w)$ by

$$\hat{s}(w) = \begin{cases} \Phi(S_0, 1) & \text{if } w = 0 \\ \Phi(S_0, i) & \text{if } w_{i-1} < w \leq w_i \text{ for some } i \in \{1, 2, \ldots, n\}. \end{cases}$$

We already showed that the range of $\Phi$ is $[m, M]$, so obviously $m \leq \hat{s}(w) \leq M$ for all $0 \leq w \leq \mathsf{PDC}$. Adding to this the fact that

$$\int_0^{\mathsf{PDC}} \frac{1}{\hat{s}(w)}\, dw = \sum_{i=1}^{n} \int_{w_{i-1}}^{w_i} \frac{1}{\Phi(S_0, i)}\, dw = \sum_{i=1}^{n} \frac{w_i - w_{i-1}}{\Phi(S_0, i)}\, dw = D,$$

we see $\hat{s}$ is valid. Also, it is clear from inspection that $\hat{s}$ is properly-divided.

Consider any properly-divided valid schedule $s$. For each $i \in \{1, 2, \ldots, n\}$, define

$$x_i = H_i^{1/3}(w_i - w_{i-1}), \qquad y_i = \frac{w_i - w_{i-1}}{s(w_i)}, \qquad \text{and } z_i = \frac{w_i - w_{i-1}}{\hat{s}(w_i)}.$$

Also, let $p = 3$ and $c = 1/S_0$.

Then, each $x_i \geq 0, \quad y_i > 0$, and $z_i > 0$. Also, since $s$ is valid, it must satisfy $\int_0^{\mathsf{PDC}} \frac{1}{s(w)}\, dw = D$. Thus,

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

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

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

- If $H_i \geq (S_0/m)^3$, then $\hat{s}(w_i) = m$ and $H_i^{1/3}/S_0 \geq 1/m$, so $1/s(w_i) \leq 1/m = 1/\hat{s}(w_i) \leq H_i^{1/3}/S_0$.
- If $H_i \leq (S_0/M)^3$, then $\hat{s}(w_i) = M$ and $H_i^{1/3}/S_0 \leq 1/M$, so $1/s(w_i) \geq 1/M = 1/\hat{s}(w_i) \geq H_i^{1/3}/S_0$.
- Otherwise, $\hat{s}(w_i) = S_0 H_i^{-1/3}$, so $1/\hat{s}(w) = H_i^{1/3}/S_0$.

In any case, we see that $1/s(w) \rightleftharpoons 1/\hat{s}(w) \rightleftharpoons H_i^{1/3}/S_0$. Multiplying by $(w_i - w_{i-1})$, we see $y_i \rightleftharpoons z_i \rightleftharpoons cx_i$.

We now know enough to apply the proportional improvement lemma using the sequences $\{x_1, x_2, \ldots, x_n\}$, $\{y_1, y_2, \ldots, y_n\}$, and $\{z_1, z_2, \ldots, z_n\}$. This gives $\sum_{i=1}^n x_i^3 y_i^{-2} \geq \sum_{i=1}^n x_i^3 z_i^{-2}$. In other words,

$$\sum_{i=1}^n H_i(w_i - w_{i-1})[s(w_i)]^2 \geq \sum_{i=1}^n H_i(w_i - w_{i-1})[\hat{s}(w_i)]^2.$$

Substituting the definition of $H_i$, we get

$$\sum_{i=1}^n \int_{w_{i-1}}^{w_i} F^c(w)[s(w_i)]^2 \, dw \geq \sum_{i=1}^n \int_{w_{i-1}}^{w_i} F^c(w)[\hat{s}(w_i)]^2 \, dw.$$

Since $s$ and $\hat{s}$ are properly-divided, we can rewrite this as

$$\int_0^{\mathsf{PDC}} F^c(w)[s(w)]^2 \, dw \geq \int_0^{\mathsf{PDC}} F^c(w)[\hat{s}(w)]^2 \, dw.$$

This completes the demonstration that $\hat{s}$ is an optimal properly-divided valid schedule. □

# References

[AD99] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 232–246, December 1999.

[BB00] Tom Burd and Robert W. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 9–14, July 2000.

[CGW95] Edwin Chan, Kinshuk Govil, and Hal Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM 95)*, pages 13–25, November 1995.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[FS99] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.

[GD60] J. Greenwood and D. Durand. Aids for fitting the gamma distribution. *Technometrics*, 2(1):55–65, January 1960.

[GLF$^+$00] Dirk Grunwald, Philip Levis, Keith I. Farkas, Charles B. Morrey III, and Michael Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., New York, NY, 1991.

[JK70] Norman L. Johnson and Samuel Kotz. *Continuous Univariate Distributions - I: Distributions in Statistics*. John Wiley & Sons, Inc., New York, NY, 1970.

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

[LS98] Jacob R. Lorch and Alan Jay Smith. Energy consumption of Apple Macintosh computers. *IEEE Micro*, 18(6):54–63, November/December 1998.

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

[MS99] Thomas L. Martin and Daniel P. Siewiorek. The impact of battery capacity and memory bandwidth on CPU speed-setting: a case study. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 200–205, August 1999.

[PBB98] Trevor Pering, Tom Burd, and Robert W. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[Pre92] William H. Press. *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press, Cambridge, MA, 1992.

[PSR93] Ketan Patel, Brian Smith, and Lawrence Rowe. Performance of a software MPEG video decoder. In *Proceedings of the First ACM International Conference on Multimedia*, pages 75–82, August 1993.

[RB89] J. C. W. Rayner and D. J. Best. *Smooth Tests of Goodness of Fit*. Oxford University Press, New York, NY, 1989.

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

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, England, 1986.

[WWDS94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.