Improving Dynamic Voltage Scaling Algorithms with PACE[*]

Jacob R. Lorch and Alan Jay Smith
Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720-1776
{lorch,smith}@cs.berkeley.edu

ABSTRACT

This paper addresses algorithms for dynamically varying (scaling) CPU speed and voltage in order to save energy. Such scaling is useful and effective when it is immaterial when a task completes, as long as it meets some deadline. We show how to modify any scaling algorithm to keep performance the same but minimize expected energy consumption. We refer to our approach as PACE (Processor Acceleration to Conserve Energy) since the resulting schedule increases speed as the task progresses. Since PACE depends on the probability distribution of the task's work requirement, we present methods for estimating this distribution and evaluate these methods on a variety of real workloads. We also show how to approximate the optimal schedule with one that changes speed a limited number of times. Using PACE causes very little additional overhead, and yields substantial reductions in CPU energy consumption. Simulations using real workloads show it reduces the CPU energy consumption of previously published algorithms by up to 49.5%, with an average of 20.6%, without any effect on performance.

1. INTRODUCTION

The growing popularity of mobile computing devices has made energy management important for modern systems, because users of these devices want long battery lifetimes. A relatively recent energy-saving technology is *dynamic voltage scaling* (DVS), which allows software to dynamically vary 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. Performance suffers, however: over the range of allowed voltages, the highest frequency at which the CPU will run correctly drops approximately proportionally to the voltage ($f \propto V$). Since the main component of power consumption is proportional to $V^2 f$, and energy per cycle is power divided by frequency, energy consumption is proportional to frequency squared ($E \propto f^2$). So a CPU can save substantial energy by running more slowly; e.g., it can run at half speed and thereby use 1/4 the energy to run for the same number of cycles.

Two factors limit the utility of trading performance for energy savings. First, a user wants the performance for which he paid. Second, other components, such as the disk and backlight, also consume power [12]. If they stay on longer because the CPU runs more slowly, the overall effect can be worse performance and *increased* energy consumption. Thus, one should reduce the voltage only when it will not noticeably affect performance.

A natural way to express this goal is to assign a soft deadline to each of the computer's tasks. (We call a deadline *soft* when a task should, but does not have to, complete by this time.) For example, user interface studies have shown that response times under 50–100 ms do not affect user think time [21]; we can thus make 50 ms the deadline for handling a user interface event. Also, multimedia operations with limited buffering, e.g. on real-time streams, need to complete processing a frame in time equal to one over the display rate, and there is no need for any earlier completion. When goals can be codified this way, the job of a DVS algorithm is to run the CPU just fast enough to meet the deadline with high probability.

Our soft deadline's key property is that if the task completes by then, its actual completion time does not matter. Thus, if we run the task more slowly, but it still completes by its deadline, performance is the same. Our primary goal is to improve DVS algorithms so that performance remains the same but energy consumption goes down.

Current DVS algorithms incorrectly assume that a constant speed consumes minimal energy even when task work requirements are unknown. But, we will show that in this common case expected energy consumption is in fact minimized by *increasing* speed as the task progresses. We therefore call our approach for improving algorithms PACE: Processor Acceleration to Conserve Energy.

We will give a formula for a speed schedule that minimizes expected energy consumption without changing performance. But, there are two problems with using this formula in practice. First, it depends on the probability distribution of a task's work requirement. Second, the schedule gives speed as a continuous function of time but real CPU's cannot change speed continuously.

To solve the first problem, we must estimate the distribution of task work from the requirements of previous, similar tasks. We describe and compare various methods for this and find some general and practical methods that work well on a variety of real workloads. For the second problem, we present and test heuristics for approximating the schedule with a piecewise constant one.

Using trace-driven simulations of real workloads, we show that our improvements significantly reduce the energy consumption of previously published algorithms without changing their performance. We also show that our approach is practical and efficient.

[*]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.

Note that PACE is not a complete DVS algorithm by itself; it is a method for *improving* such an algorithm. For example, it does not change characteristics of the algorithm that affect performance. So, we compare certain algorithms to show which ones work best when modified by PACE. For reasons we discuss, we do some such comparisons empirically rather than analytically.

This paper is organized as follows. Section 2 discusses related work, including DVS algorithms others have proposed. Section 3 presents our model of the DVS problem and introduces useful terminology. Section 4 describes how to improve algorithms with PACE. Section 5 discusses what algorithms work best when modified by PACE. Section 6 describes the workloads we use for analyzing algorithms' energy consumption and performance. Section 7 presents these analyses and discusses results. Section 8 suggests possibilities for future work. Finally, section 9 concludes.

Although we explain terms when we first present them, the reader may find Table 1, which summarizes these terms, helpful.

2. RELATED WORK

Researchers have studied CPU scheduling for decades. One important result is that if a set of tasks has feasible deadlines, scheduling them in increasing deadline order will always make all the deadlines [11]. Another useful result, described by Błażewicz et al. [2, pp. 346–350], is that when the rate of consumption of some resource is a convex function of CPU speed, an ideal schedule will run each task at a constant speed. Yao et al. [24] observe that with DVS, power consumption is a convex function of CPU speed. They show how to compute an optimal speed-setting policy by constructing an earliest-deadline-first schedule, and then choosing the minimal possible speed for each task that will still make the deadlines.

However, one can only compute such optimal schedules if the tasks' CPU requirements are known in advance, and task requirements in most systems are unpredictable random variables; see, e.g., [20]. For this reason, most research on scheduling for DVS has focused on heuristics for estimating CPU requirements and attempting to keep CPU speed as constant as possible.

Weiser et al. [23] recommended *interval-based* algorithms for DVS. These divide time into fixed-length intervals and set each interval's speed so that most work is completed by the interval's end. Chan et al. [4] refined these ideas by separating out an algorithm's two parts: *prediction* and *speed-setting*. When an interval begins, the prediction part predicts how busy the CPU will be during the interval (i.e., how much work there will be to do), and the speed-setting part uses this information to set the speed. 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. [17] and Grunwald et al. [7], have shown that Weiser et al. and Chan et al.'s algorithms are impractical because they require knowledge of the future. However, they have proposed practical versions of these algorithms. Prediction methods they suggest include:

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

Speed-setting methods they suggest include:

- **Weiser-style.** If the utilization prediction x is high ($> 70\%$), increase the speed by 20% of the maximum speed. If the utilization prediction is low ($< 50\%$), decrease the speed by $60 - x\%$ of the maximum speed.
- **Peg.** If the utilization prediction is high ($> 98\%$), set the speed to its maximum. If the utilization prediction is low ($< 93\%$), decrease the speed to its minimum positive value.
- **Chan-style.** Set the speed for the upcoming interval just high enough to complete the predicted work. In other words, multiply the maximum speed by the utilization to get the speed.

We refer to previously published algorithms by concatenating the names of their methods. For example, the Flat/Chan-style algorithm uses the Flat prediction method and the Chan-style speed-setting method.

Note that dividing time into intervals and using those boundaries as deadlines is somewhat arbitrary. For example, if a task arrives near the end of an interval, it does not really have to complete by the end of that interval. Furthermore, without deadlines, there is no particular reason to complete any given task by a certain time; it is best to simply measure the average number of non-idle cycles per second and run the CPU at that speed. (Transmeta's LongRun^{TM} system does something like this [10].) Pering et al., recognizing this, suggested considering deadlines when evaluating DVS algorithms [17]. To do so, they suggest considering a task that completes before its deadline to effectively complete at its deadline.

Grunwald et al. [7] considered deadlines when they compared several of the algorithms described above (as well as others not listed here) by implementing them on a real system. They decided that although none of them are very good, Past/Peg is the best: it never misses any deadlines for the workload they considered, yet still saves a small but significant amount of energy.

3. MODEL

In our model of the CPU, voltage can change continuously over some range. Over this range, CPU speed increases continuously between some minimum and maximum speeds. We assume CPU energy consumption per cycle is proportional to the speed squared.

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

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

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

Term (and abbreviation)	Definition
Work requirement / work (W)	The number of CPU cycles a task requires.
Completion time	The number of seconds a task takes to complete.
Deadline (D)	The number of seconds a task has to complete. Generally, a deadline will be *soft*, meaning some tasks may miss their deadlines. The key property of a deadline is that as long as a task completes by its deadline, its actual completion time does not matter.
Effective completion time	The completion time of a task, or its deadline, whichever is greater. This measure reflects the fact that as long as a task completes by its deadline, its actual completion time does not matter.
Delay	The number of seconds a task takes beyond its deadline.
Excess	The number of cycles of work a task still has left to do after its deadline has passed.
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 CPU speed will vary as a task runs. $f(t)$ is the speed after the task has run for t seconds. $s(w)$ is the speed after the task has completed w cycles of work.
Transition point	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 requirement.
Parametric method	A way to estimate 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 way to estimate a probability distribution from a sample without assuming any distribution type. It thus lets the data "speak for themselves."
Kernel density estimation	A nonparametric method that builds up a probability distribution by adding up 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

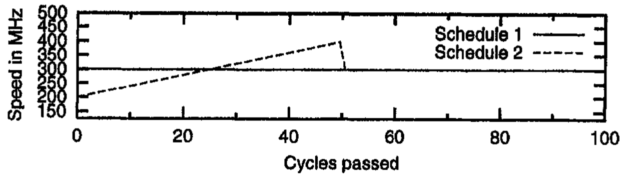

Figure 1: This graph shows two performance equivalent speed schedules with deadline 50 ms. Their pre-deadline cycles are equal (15 Mc) and their post-deadline parts are identical.

the task misses its deadline (when $t > D$). A speed schedule has a certain number of *pre-deadline cycles* (PDC), the number of cycles it can perform before the deadline. Note that PDC $= \int_0^D f(t)\,dt$.

We say that two speed schedules are *performance equivalent* if, no matter what a task's work requirement, it will have the same effective completion time under both schedules. We call two algorithms performance equivalent if they always have performance equivalent speed schedules. We make the following important observation: ***If two speed schedules have equal pre-deadline cycles and identical post-deadline parts, then they are performance equivalent.*** Figure 1 illustrates two such schedules.

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

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

It is often useful to consider the speed schedule to be a function of work completed instead of a function of time. So, we will sometimes describe the schedule with a function s, where $s(w)$ is the speed to use after the task has completed w cycles of work. f and s are just different expressions of the same function; it is straightforward to convert a schedule from one functional form to the other.

4. IMPROVING DVS ALGORITHMS

4.1 Theoretical optimal formula

In this section, we present a formula for the optimal (energy minimizing) speed schedule that is performance equivalent to that of a previously known algorithm.

As previously noted, when we know the task's work requirement, the optimal algorithm uses a constant speed. When we know only the distribution of this work, however, the optimal schedule uses a variable speed. An intuitive explanation is that if the task work is unknown, it may be high or low. It is best to run slowly at first, because the task may require little work and thus end before we must increase the speed and thus the power consumption. For example, suppose a task with a deadline of 50 ms needs 5 megacycles (Mc) 75% of the time and 10 Mc 25% of the time. Suppose further that CPU power is 50 nW $\cdot\, x^3$ when the speed is x MHz. The ideal constant speed is 200 MHz, the slowest that will always meet the deadline; this consumes 12.5 mJ on average.[1] An alternate, variable speed schedule is 163 MHz for the first 30.675 ms, then 259 MHz for any remaining time; this consumes 10.84 mJ on average,[2] 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: $F^c(w) = 1 - F(w)$.

We are trying to minimize the expected energy consumption of the pre-deadline part of the algorithm,[3] subject to the con-

[1] $(25\text{ms})(200)^3(50\text{nW}) + (25\%)(25\text{ms})(200)^3(50\text{nW}) = 12.5\text{mJ}$.

[2] $(30.675\text{ms})(163)^3(50\text{nW}) + (25\%)(19.325\text{ms})(259)^3(50\text{nW}) = 10.84\text{mJ}$.

[3] The expected energy consumption is $k \int_0^{\mathsf{PDC}} F^c(w)[s(w)]^2\,dw$, where k 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 $k[s(w)]^2$, and the number of cycles is dw, so the energy consumption is $k[s(w)]^2\,dw$. The probability that this work actually ever gets done is $F^c(w)$.

straint that the pre-deadline cycles must be the same as the PDC of the original algorithm. In other words, we want to minimize $\int_0^{\text{PDC}} F^c(w)[s(w)]^2\,dw$ subject to the constraint $\int_0^{\text{PDC}} \frac{1}{s(w)}\,dw = D$. Algebraically, this is equivalent to minimizing

$$\int_0^{\text{PDC}} \left([F^c(w)]^{-1/3}/s(w)\right)^{-2} \cdot [F^c(w)]^{1/3}\,dw$$

subject to

$$\int_0^{\text{PDC}} \left([F^c(w)^{-1/3}/s(w)\right) \cdot [F^c(w)]^{1/3}\,dw = D.$$

In other words, we are given the weighted sum of the values $[F^c(w)]^{-1/3}/s(w)$ and we want to minimize the weighted sum of their -2nd powers. By Jensen's inequality, since the -2nd power function is concave up, this minimization occurs when all the values are the same. In other words, we want $[F^c(w)]^{-1/3}/s(w)$ to be as constant as possible. We achieve this by making $s(w)$ be the valid speed closest to $C[F^c(w)]^{-1/3}$, where C is a constant chosen to satisfy the deadline constraint. For a full proof that this works, see [14]. Since $F^c(w)$ decreases as w increases, this schedule speeds up the CPU as the task progresses, as noted 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 schedule is a continuous function, which is impractical to implement precisely since software must issue a command each time it wants to change the speed. In practice, we want a schedule with a limited number of *transition points*, points where the speed may change. We specify transition points by 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.

Given fixed transition points, we can construct a speed schedule that minimizes expected energy consumption, as follows. In the interval between any two transition points, we use the valid speed closest to $C(F^c_{\text{avg}})^{-1/3}$, where F^c_{avg} is the average value of F^c over that interval. As before, C is constant over the entire schedule; we choose a value for it that meets the deadline constraint. The rationale is similar to that for the continuous optimal speed schedule; for a full proof that this works, see [14].

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_jth 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 this may result in an excessive number of speed changes. 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 $q_j = Q + (j - J)\frac{0.995-Q}{N-J}$.

To implement a piecewise-constant speed schedule, software must interrupt the task at predetermined intervals to change the CPU speed. A CPU cycle counter or clock timer could generate such interrupts. Alternately, software could use soft timers, an operating system facility suggested by Aron et al. [1] that lets one schedule events 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 CPU could accept commands not just to change speed immediately but also to establish a speed schedule for the next few milliseconds. Alternately, the CPU itself could implement the DVS algorithm, so software would not have to spend time communicating schedule information to hardware.

4.3 Sampling methods

To implement PACE, we must estimate the probability distribution of the current task's work requirement. It is rare to have this information a priori; usually, we must estimate the distribution from a sample of work requirements of similar recent tasks. We consider the following sampling methods.

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

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

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

Fortunately, all we need to compute these two numbers are n, ω, the weighted sum, and the weighted sum of squares. For each of our sampling methods, there exists a simple algorithm to update these four quantities, and thus the sample mean and variance, in $O(1)$ time whenever a new sample value arrives.

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. We may express this distribution as a CDF or as a set of quantiles. There are two general ways to estimate a distribution from a sample: parametric and nonparametric. Parametric methods assume the distribution belongs to a given 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.

Note that the criterion for the desirability of an estimation method is not the goodness of fit, but rather the extent to which the use of that method leads to lower energy consumption. In particular, task run times are well known to be highly skewed, but we are more interested in modeling the portion of the task run time prior to the deadline than the portion after it.

Gamma. The first method we consider is the parametric method assuming a gamma distribution. This distribution is commonly used to model service times [8, p. 490], and we will show later that it works well. The gamma distribution has range $x \geq 0$. It has two parameters: the shape α and the scale β. The probability density function is $p(x) = x^{\alpha-1}e^{-x/\beta} / \beta^{\alpha}\Gamma(\alpha)$. Reasonable estimators for the model parameters are $\hat{\alpha} = \hat{\mu}^2/\hat{\sigma}^2$ and $\hat{\beta} = \hat{\sigma}^2/\hat{\mu}$ [8]. Maximum likelihood estimators also exist, but we do not use them, since (a) we cannot compute them 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 [9, 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 methods in [18], but we avoid those methods when possible since they are computationally expensive.

Normal. The second 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, since normal distributions are shaped similarly to gamma distributions in some cases and are far easier to model. The normal distribution has only two parameters: the mean μ and the standard deviation σ, whose unbiased estimators are $\hat{\mu}$ and $\hat{\sigma}$. (The maximum likelihood estimator for $\hat{\sigma}$ leaves out the $n/(n-1)$, but we have found it does slightly worse for our purposes.) Furthermore, since the normal distribution $N(\mu, \sigma)$ is a simple linear transformation of the unit normal distribution $N(0, 1)$, one can easily compute quantiles and CDF values using lookup tables.

Pareto. A method we considered and rejected is the parametric method assuming a Pareto distribution. This model is appealing because it is heavy-tailed and other researchers have found task times to be heavy-tailed (highly skewed); see, e.g., [20]. However, we found this model to fit our distributions very poorly, so we consider it no further in this paper. In any event, modeling tails accurately is not a high priority, since the tail of the distribution only affects the speed used near or after the deadline, and most tasks will complete before then.

Kernel density estimation. The nonparametric method we consider is kernel density estimation, a popular nonparametric method [22]. 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 [22, 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 thus chosen the triangular kernel: $K(t) = \max\{1-|t|, 0\}$, which is simpler to implement than most.

We can compute the theoretical optimal bandwidth from p'', the second derivative of the true probability density, using $\left(\int t^2K(t)\ dt\right)^{-\frac{2}{5}} \left(\int K(t)^2\ dt\right)^{\frac{1}{5}} \left(\int p''(x)^2\ dx\right)^{-\frac{1}{5}} n^{-\frac{1}{5}}$. For the triangular kernel, $\int t^2K(t)\ dt = \frac{1}{6}$ 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. Assuming a normal distribution with parameters $\hat{\mu}$ and $\hat{\sigma}$ makes 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.

Note that the range of the kernel density estimate may extend below 0. We use reflection [22, pp. 29–31] to avoid this. This method adds to the sample the set of values $\{-X_i\}$, each weighted ω_i, making the sample size $2n$. It then computes the probability density $\hat{p}_{\text{adj}}(x)$ using this adjusted sample, and sets $\hat{p}(x) = 2\hat{p}_{\text{adj}}(x)$ for $x \geq 0$, $\hat{p}(x) = 0$ otherwise.

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.

5.1 Choosing a post-deadline part

First we consider what the base algorithm for post-deadline scheduling should be. To compare such algorithms, we need a performance metric that takes into account the user's "impatience function", i.e., how undesirable he finds missing the deadline by various amounts. We choose to use Pering et al.'s suggested metric, the *clipped delay*, which is the sum of all tasks' effective completion times [17]. Our goal is to find an algorithm that consumes the least possible energy for a given clipped delay.

Let TotalExcess be the total excess (the amount of task work left after the deadline) of all tasks in the workload. Note that the pre-deadline part determines this; we cannot change it in the post-deadline part. A clipped delay value corresponds to some total amount of delay T past all deadlines, so to achieve a given clipped delay all we must do is perform the total excess in some given time T. The minimal-energy solution to this single constraint is to use the constant speed TotalExcess$/T$. Another way to look at this is that if we use a fixed, constant speed after the deadline, we assure that the energy consumption we achieve is the minimum possible for the clipped delay we achieve. Therefore, we propose picking a fixed speed to use for all post-deadline parts. Many previously published 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 after the deadline. Usually, other components like the backlight will be running and consuming power, and delay past the deadline can cause these components to consume more energy. Using a CPU frequency f makes CPU energy consumption proportional to f^2 but makes energy consumption of those other components proportional to $1/f$ as they may stay on longer. We therefore choose to always use the maximum speed once a task misses its deadline, as many previously published algorithms generally do anyway. This minimizes delay, generally at some energy cost, but not necessarily at substantial energy cost considering that other components' power consumption would mitigate the effect of lower speeds.

Another approach is to choose a target average delay, predict the average excess, and use the ratio of these as the speed. However, we have found this to be impractical, since two factors make predicting average excess difficult. First, excess is nonzero only rarely, since most tasks meet their deadlines. Therefore, samples of excess will tend to be small until many tasks have occurred, and even then most sample values will be quite old. 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

We have shown how to improve a DVS algorithm by changing its pre-deadline and post-deadline parts. Thus, the only remaining influence the base DVS algorithm has on the final schedule is its choice of pre-deadline cycles (PDC). We now consider how to choose PDC for each task in order to minimize energy consumption for a given fraction of deadlines made. This constraint is interesting because it is only one constraint on all tasks rather than one constraint per task. That is, we need to meet a given fraction of all deadlines, but not necessarily meet each deadline with the same probability. Thus, even if the task work distribution were known and stationary, the optimal solution might not be to use the same PDC for all tasks. (This is a property of the Flat/Chan-style algorithm, which uses a fixed speed and thus has the same PDC for all tasks: the speed times the deadline.)

Unfortunately, we cannot solve this optimization problem for two reasons. First, the complex dependence of the speed schedule on the PDC we choose makes choosing an optimal set of PDC values intractable. Second, even if there were an analytical solution, it would depend on all of the work distributions. Therefore, we would need a model of the distribution of distributions, and we know no reasonable way to model this.

Depending on the distribution of distributions, different approaches to choosing PDC will work better or worse than others. Therefore we must rely on empirical rather than analytic methods to decide which algorithms work best when modified by PACE. We present such results in §7.7.

One interesting distinction between base DVS algorithms is that for some, such as LongShort/Chan-style, PDC is dependent on the current task work distribution, while for others, such as Flat/Chan-style, it is not. (LongShort/Chan-style uses a speed proportional to recent utilization, so its PDC is higher when recent tasks have been long; Flat/Chan-style has a constant PDC for all tasks.) The former type will tend to miss the deadlines of tasks whose work requirements are local maxima, so we call these *local* algorithms. The latter type will tend to miss the deadlines of the longest tasks in the whole workload, so we call them *global*. When the distribution is nonstationary, as is usual, local approaches will tend to miss a different set of tasks' deadlines than global ones. We cannot analytically determine whether local approaches have lower energy consumption for a given fraction of deadlines made than global ones, or even whether one local approach is better than another. Therefore, we rely on empirical data to compare them.

6. WORKLOADS

We evaluate these algorithms using six workloads. We derived most workloads from traces of users performing their normal business on desktop machines running Windows NT or Windows 2000. VTrace, a tracer described in [13], generated these traces. The traces contain timestamped records describing events related to processes, threads, messages, disk operations, network operations, the keyboard, and the mouse. We deduce what work is done due to a user interface event as follows: we assume that a thread is working on such an event from the time it receives the message describing that event until the time it either performs a wait for a new event or requests and receives a message describing a different event. Furthermore, if the thread sends a message or signal to another thread while working on such an event, we assume that work done due to that message or signal is done due to the original event.

To reduce the amount of data VTrace collects, it only collects the full set of events it can for sessions lasting 90 minutes at a time, after which it pauses for two hours. In our analyses here, any trace longer than 90 minutes only represents the 40% of the time VTrace actually traced its full set of events.

We define each workload by a class of events, such as letter key-presses in Microsoft Word. The workload consists of the set of tasks triggered by all such events. In other words, each task of each workload is roughly of the same type; by separating different task types into different workloads, we model the effect of keeping separate samples for different task types, as described in section 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.

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. Section 8.2 discusses this avenue for future work. Furthermore, 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 is 100 MHz, the maximum speed is 500 MHz, and the peak CPU power consumption is 3 W. Most currently shipping machines are faster, but 500 MHz is representative of the traced machines.

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 3.4 months of traces VTrace collected on a 450 MHz Pentium III computer with 128 MB of memory running Windows NT 4.0. The first author, a computer science graduate student, used this computer. This workload is interactive, so we use a 50 ms deadline for each task.

6.2 Groupware

Software that enables and enhances communication with others, i.e., groupware, is important on the desktop, and will be more important in portable computers as they become more connected. So we include a workload using a common groupware product, Novell's GroupWise. This workload uses left mouse button releases as its class of events. We derived this workload from 6.5 months of traces VTrace collected on a 350 MHz Pentium II computer with 64 MB of memory running Windows NT 4.0. A crime laboratory director in the Michigan State Police used this computer. This workload is interactive, so we use a 50 ms deadline for each task.

6.3 Spreadsheet

Spreadsheets are also common 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 3 months of traces VTrace collected on a 500 MHz Pentium III computer with 96 MB of memory running Windows NT 4.0. The chief technical officer of a computing-related company used this computer. This workload

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

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 [6]. 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) [16]. 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.

We obtained animations to use from the same BMRC FTP site as 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 computer 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

Some system designers may want to implement a scheduling algorithm without instrumenting the operating system, relying only on information hardware can observe. One of our workloads represents such a scenario.

We derive this workload from VTrace traces as follows. A task begins when the keyboard device generates a keypress signal, and ends the next time either the CPU becomes idle or there is another keypress. To determine when the CPU is idle, we use the time that the idle thread is running. (In battery-powered systems, the idle thread typically halts the CPU, so hardware can deduce when this thread is running.) If a disk operation is ongoing when the CPU goes idle, we throw out any keypress being worked on, for two reasons. First, we cannot know if the I/O is part of this task, so we cannot know whether the task is over or simply waiting for I/O. Second, as stated before, we are ignoring tasks that perform I/O since we are only considering algorithms for tasks with no I/O.

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

7. RESULTS

7.1 Modeling task work distributions

In this section, we determine how best to practically estimate the probability distribution of tasks' 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 tasks' 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 by definition cannot change performance.

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

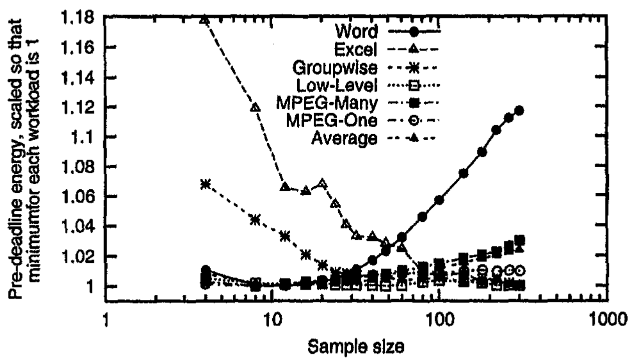

Figure 2: A comparison of the effect of various sample sizes k on energy consumption when PACE uses the Recent-k sampling method

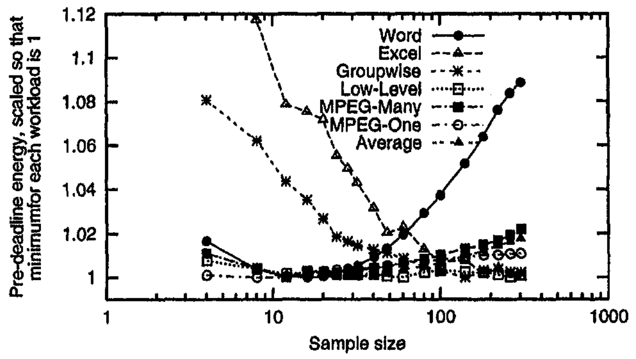

Figure 3: A comparison of the effect of various sample sizes k on energy consumption when PACE uses the LongShort-k sampling method

7.2 Which sampling method to use

We now compare the sampling methods to determine the best ones to use with PACE. For these comparisons, we assume that we estimate distributions using kernel density estimation.

First, we consider what sample size to use for the sampling methods that use only recent data, Recent-k and LongShort-k. Figures 2 and 3 show the outcome of using different sample sizes for different workloads. 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 distribu-

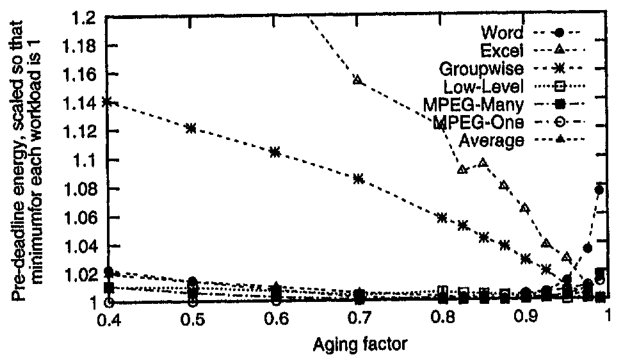

Figure 4: A comparison of the effect of various aging factors a on energy consumption when PACE uses the Aged-a sampling method

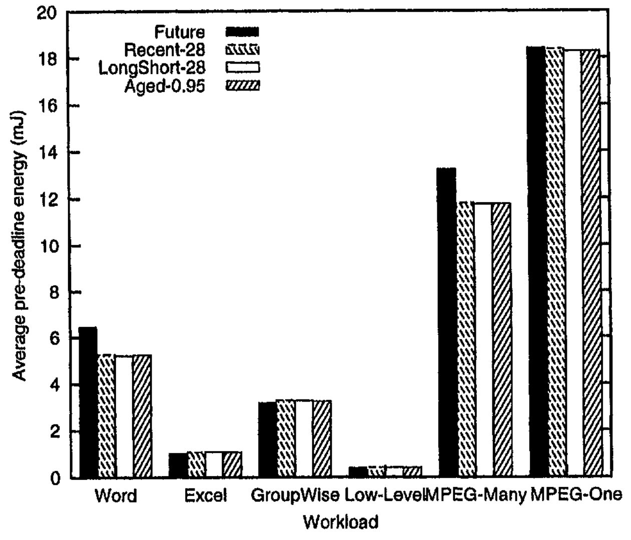

Figure 5: A comparison of the effect of various PACE sampling methods on energy consumption

tions, 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 that 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 very 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 4 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. Based on the graphs, we feel a reasonable aging factor is 0.95. With this value, 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 5 compares the Future, Recent-28, LongShort-28, and Aged-0.95 sampling methods for the six workloads. We do not evaluate the All sampling method, since it is equivalent to Recent-∞, and we have already shown that the Recent method works quite badly for some workloads with large sample sizes. The first thing we observe is that the Future method sometimes uses less energy than the other methods, but sometimes uses much more. This indicates that even if complete information about the full distribution of task work is available, it is often better to use recent information to predict the distribution of the next task work. Presumably, this is because these distributions are nonstationary, so recent information is a better predictor than global information. The next thing to observe is that the remaining three methods have virtually identical energy consumption. In general, 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 Aged-0.95 generally produces the best results, but other methods work reasonably and may be good choices if they are easier to implement.

7.3 Which distribution estimator 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 [19] recommend Neyman's Ψ_4^2 test. Applying this test to any of our workloads, using any of our sampling methods, the test reveals an extremely small probability that the data fit either the normal or gamma model. Fortunately, the key issue is not the accuracy with which we can approximate the distribution of the task work. The key issue is the extent to which a statistically unacceptable model of this distribution produces a suboptimal solution to the energy minimization problem.

Therefore, the more important question to ask is how effectively PACE can use each model (kernel density, normal, or gamma) 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 6 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 reasonably close results.

7.4 Choosing transition points

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

Figure 7 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

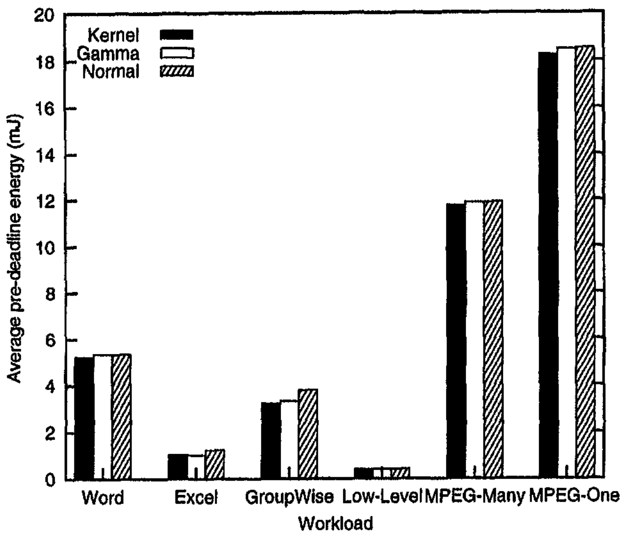

Figure 6: A comparison of the effect of various PACE distribution estimation methods on energy consumption

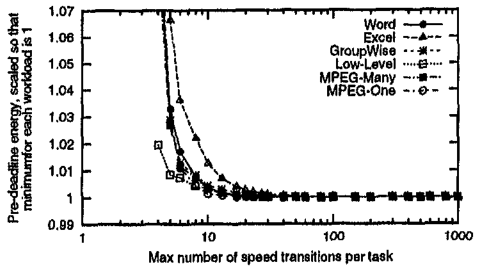

Figure 7: A comparison of the effect on energy consumption of using different numbers of speed transitions to approximate the continuous schedule

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 $Q = 0.95$.

7.5 Effect of approximations

To evaluate the effect of our approximations to the theoretically optimal formula for PACE, we must know what that optimal formula is. For real workloads, this is impossible, since we cannot know the underlying distribution of each task's work. (We can know the overall distribution of task work, but as workloads can be nonstationary we cannot know the distribution for any given task.) Thus, in this section, we use a synthetic workload generated from a known probability distribution, 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 with 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%.

7.6 Improving algorithms with PACE

In section 4, we described how PACE can replace the pre-deadline part of a standard scheduling algorithm with a schedule that has lower expected energy consumption. Here, we simulate this as follows. First, we simulate a previously published algorithm. Then, we modify the algorithm so that it uses PACE to re-compute 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 previously published interval-based algorithms, each with an interval length of 10 ms. The four methods we use are:

- **Past/Weiser-style.** This is a practical version of Weiser et al.'s algorithm [23].
- **LongShort/Chan-style.** This is a practical version of one of the best algorithms Chan et al. proposed [4].
- **Flat/Chan-style.** This is a practical version of another of the best algorithms Chan et al. proposed [4]. It uses a fixed speed, so it is similar to Transmeta's LongRun^{TM} in steady state [10]. We choose the speed so that at least 98% (99% for the Low-Level workload) of all tasks that can make their deadlines do so.
- **Past/Peg.** Grunwald et al. [7] favored this algorithm.

Figure 8 shows the effect of using PACE to modify these 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 algorithm. PACE using a gamma model reduces the CPU energy consumption of algorithms by 2.4–49.0%; the average reduction over all workloads and all algorithms is 20.3%. PACE using the kernel density estimation method reduces the CPU energy consumption of 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 algorithm most improved with PACE is Past/Peg, the one favored by the most recent comparison of DVS algorithms [7]. Past/Peg was favored in that work because it misses fewer deadlines than other algorithms; unfortunately, this requires higher energy consumption, as Figure 8 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, previously published algorithms use DVS to

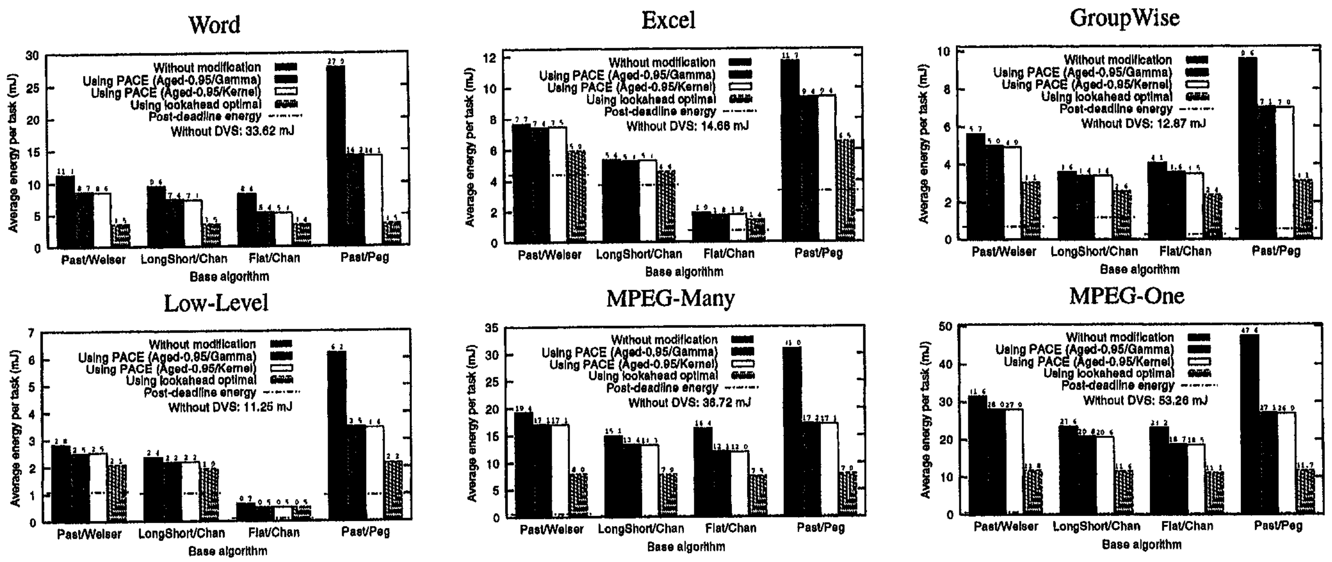

Figure 8: These graphs show the effect of modifying algorithms with PACE to get performance equivalent, but lower-energy, algorithms. Horizontal lines show the post-deadline energy, which none of our modifications change. The lookahead optimal results use knowledge of the current task's work requirement and thus cannot be practically implemented; however, they serve as a lower bound on what can be attained by a performance equivalent algorithm.

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, previously published DVS algorithms allow it to consume only 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 [12], previously published DVS algorithms would increase its battery lifetime by about 22%; with PACE, the battery lifetime improvement would be about 28%.

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 algorithms. The high energy savings is especially exciting because PACE by definition has no effect on performance.

7.7 Which base algorithm to use

As discussed in section 5.2, different algorithms compute PDC in different ways, so they still differ even when modified by PACE. Lacking a model of this effect, we must rely on empirical results to find which algorithm performs best when we modify its pre-deadline part with PACE and its post-deadline part to always use the maximum speed. For space reasons, we leave out plots showing the average task energy consumption as a function of fraction of possible deadlines made for each of the workloads and each of the base algorithms; interested readers can find these plots in [14]. They show that Flat/Chan-style, the only global algorithm we considered, most often gives the lowest energy consumption for a given number of deadlines made. This suggests that global algorithms tend to do better than local ones. Among the local algorithms, LongShort/Chan-style does best, achieving reasonable energy savings for a given number of deadlines made.

7.8 Overhead analysis

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

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

Table 3 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 the energy the simulated CPU would consume to perform this computation, as a percentage of the energy consumed

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

Table 3: This table shows the average time per task for a 450 MHz Pentium III to execute variants of the PACE algorithm. Energy overhead values are in parentheses; they show how much energy the simulated CPU would consume to compute the PACE speed schedules, as a percentage of the energy it would consume just to execute the workload tasks.

to perform the tasks of the workload. We see 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 μs per task and at most 0.51% energy consumption. The time overhead is small compared to the deadline 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 only delay the completion of a task when there is no idle time before the next task starts.

8. FUTURE WORK

8.1 Nonlinear speed-voltage relationship

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 [5]. So, instead of $E \propto f^2$, as we were assuming in our proof of optimality, we have $E \propto \left(V_{th} + \frac{f}{2k} + \sqrt{\frac{V_{th}f}{k} + \left(\frac{f}{2k}\right)^2}\right)^2$.

This formula is complicated, but we can generally approximate it with a simpler one of the form $E = as^2 + b$. In this case, the optimal solution is the same, since the extra b term does not affect it. However, often a formula of the form $E = as + b$ is an even better approximation. Using this approximation changes the optimization problem: we must then minimize $\int_0^{\text{PDC}} F^c(w)s(w)\,dw$. This changes the optimal 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 using this power when the energy vs. speed curve is better approximated by a linear curve. We believe that most of our results will still hold.

Memory effects can also produce a nonlinear speed-voltage relationship, as observed by Martin et al. [15]. When memory speed does not scale precisely with CPU speed, the work completion rate 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

If a task performs synchronous I/O, a speed scheduling algorithm must attempt to have the task complete its CPU work *and* its synchronous I/O within the deadline. One way to do this is to consider the deadline for the CPU part to be reduced by the time spent performing I/O while the CPU waits. In future work, we will test approaches that take such dynamic deadlines into account. Such approaches 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 a wait for I/O and thus has shortened its deadline.

8.3 Overlapping tasks

In future work, we should address scheduling multiple tasks at once. If only one of these tasks has a deadline, it may be reasonable to consider it the "main task" and to model all other tasks as contributing to its work requirement. In other words, we might consider all work done while the main task is active to be part of that task, even though some of this work is unrelated. This way, the main task work distribution will automatically take into account all work the CPU will do before the main task completes.

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 they will not often have two simultaneous tasks with deadlines.

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. [3], 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 DVS algorithms without affecting their performance. We showed that one can change how an algorithm schedules tasks so that performance stays the same but expected energy consumption decreases. Furthermore, we developed an optimal formula for scheduling tasks with minimal energy consumption. Although one cannot implement this formula precisely, we described various methods to approximate it effectively.

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 is probably easier and faster. Practically implementing PACE also involves choosing a limited number of speed transitions. We found heuristics for this 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, previously published 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 previously published algorithms by 1.4–49.5% with an average of 20.6%.

Besides PACE, we made other suggestions for changing DVS algorithms. We recommended using a constant speed, probably the maximum CPU speed possible, for all tasks that have missed their deadlines. Furthermore, among the algorithms we considered, we found that using Flat/Chan-style (i.e., using a fixed PDC for all tasks) with our recommended changes gave the lowest energy consumption for a given number of deadlines made.

We therefore recommend constructing a DVS algorithm as follows. 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 between 40–60% of the number that the maximum speed would accomplish), 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 about statistical methods, including the use of the gamma model and kernel density estimation. We also thank Ronald Wolff for his suggestions. Finally, we offer great thanks to the many VTrace users whose traces yielded the workloads for this paper.

11. REFERENCES

[1] M. Aron and P. 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.

[2] J. Błażewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. *Scheduling Computer and Manufacturing Processes*. Springer-Verlag, Berlin, Germany, 1996.

[3] T. Burd and R. 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.

[4] E. Chan, K. Govil, and H. 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.

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

[6] J. 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.

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

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

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

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

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

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

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

[14] J. R. Lorch and A. J. Smith. PACE: a new approach to dynamic voltage scaling. Technical Report UCB/CSD-01-1136, Computer Science Division, EECS, University of California at Berkeley, March 2001.

[15] T. L. Martin and D. 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.

[16] K. Patel, B. Smith, and L. Rowe. Performance of a software MPEG video decoder. In *Proceedings of the First ACM International Conference on Multimedia*, pages 75–82, August 1993.

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

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

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

[20] S. Sherman, F. Baskett III, and J. C. Browne. Trace-driven modeling and analysis of CPU scheduling in a multiprogramming system. *Communications of the ACM*, 15(12):1063–1069, December 1972.

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

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

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

[24] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE 36th Annual Symposium on Foundations of Computer Science*, pages 374–382, October 1995.