Practical Voltage Scaling for Mobile Multimedia Devices

Wanghong Yuan, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL 61801, USA
{wyuan1, klara}@cs.uiuc.edu

ABSTRACT

This paper presents the design, implementation, and evaluation of a *practical* voltage scaling (PDVS) algorithm for mobile devices primarily running multimedia applications. PDVS seeks to minimize the total energy of the whole device while meeting multimedia timing requirements. To do this, PDVS extends traditional real-time scheduling by deciding *what execution speed* in addition to when to execute what applications. PDVS makes these decisions based on the discrete speed levels of the CPU, the total power of the device at different speeds, and the probability distribution of CPU demand of multimedia applications. We have implemented PDVS in the Linux kernel and evaluated it on an HP laptop. Our experimental results show that PDVS saves energy substantially without affecting multimedia performance. It saves energy by 14.4% to 37.2% compared to scheduling algorithms without voltage scaling and by up to 10.4% compared to previous voltage scaling algorithms that assume an ideal CPU with continuous speeds and cubic power-speed relationship.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms

Algorithms, Design, Experimentation.

Keywords

Power Management, Mobile Computing, Multimedia.

1. INTRODUCTION

Battery-powered mobile devices are becoming increasingly important platforms to run multimedia applications. For example, we can already take and send pictures with a cell phone and watch TV on an iPAQ. These mobile devices need to support multimedia Quality of Service (QoS) and save energy simultaneously. A challenging problem is how to minimize energy consumption while provisioning QoS. This paper addresses this problem via energy-efficient real-time CPU scheduling.

Recently, there has been a lot of related work on energy-efficient scheduling, often by extending real-time scheduling with dynamic frequency/voltage scaling (DVS) [7, 10, 11, 13, 15, 21]. DVS reduces CPU energy by lowering the CPU speed (frequency and voltage) based on application CPU demand. The effectiveness of DVS algorithms is therefore dependent on the prediction of application demand— over-prediction may waste energy, while under-prediction may degrade application performance. It is often difficult to precisely predict the instantaneous demand of multimedia applications, which changes largely due to the variations in the input data (e.g., I, P and B frames).

The probability distribution of CPU demand of multimedia applications, however, is much more stable due to the periodic nature of multimedia processing. Statistical DVS [7, 11, 21] has been therefore proposed to save energy by adjusting the execution speed of each application based on its demand distribution. Previous statistical DVS algorithms, and generally most of previous DVS algorithms, often assume an ideal CPU: (1) the CPU can change speed continuously, (2) the CPU power is dominated by the dynamic power, which is proportional to the speed and square of the voltage, and (3) the voltage is proportional to the speed. That is, a lower speed yields a cubic power reduction and a quadratic energy reduction.

In practice, however, mobile devices often have a *non-ideal* CPU. First, mobile processors (e.g., Intel Pentium-M and AMD Athlon) support a discrete set of speeds, rather than a continuous range. Second, a lower speed does not yield a cubic power reduction, since the static power also has a significant effect and the voltage does not scale linearly to the speed. For example, our measurements on an HP N5470 laptop with an Athlon CPU show that a lower speed saves much less power than the ideal cubic power-speed relationship (Figure 1). When applied to non-ideal CPUs, previous DVS algorithms assuming an ideal CPU may make a wrong decision and hence waste energy.

This paper presents a *practical* voltage scaling (PDVS) algorithm for mobile devices with a non-ideal CPU. PDVS also differs significantly from previous DVS algorithms in that it minimizes the total energy of the whole device, rather than only CPU energy, while meeting multimedia timing requirements. Specifically, PDVS extends traditional real-time scheduling by deciding when to execute what applications as well as *what execution speed*. It makes these decisions by explicitly considering the discrete speed levels, the total device power at different speeds, and the demand distribution of multimedia applications. We show that this scheduling problem is NP-hard and develop a heuristical solution.

We have implemented PDVS as a part of GRACE-OS, which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MM'04, October 10-16, 2004, New York, New York, USA.
Copyright 2004 ACM 1-58113-893-8/04/0010 ...$5.00.

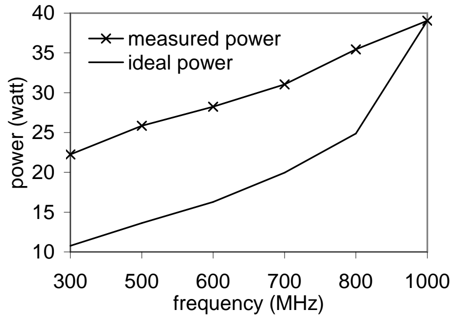

Figure 1: Measured and ideal power on an HP N5470 laptop with an Athlon CPU: The measured power is obtained with an oscilloscope, while the ideal power is calculated by assuming that the power is proportional to the cube of the speed.

estimates the demand distribution of applications via kernel-based profiling [21]. We have evaluated PDVS on an HP N5470 laptop with an Athlon CPU and typical video codecs. Our experimental results show that PDVS saves energy significantly with little impact on multimedia performance. Specifically, PDVS reduces the total energy of the laptop by 14.4% to 37.2% relative to algorithms without DVS and by up to 10.4% relative to previous DVS algorithms assuming an ideal CPU.

The rest of the paper is organized as follows. Section 2 introduces system models. Section 3 introduces the PDVS algorithm. Sections 4 and 5 present the implementation and experimental evaluation, respectively. Section 6 compares PDVS with related work. Finally, Section 7 concludes the paper.

2. SYSTEM MODELS

This section introduces system models for the PDVS algorithm. We first model the non-ideal CPU and multimedia applications, and then briefly describe a soft real-time scheduling algorithm, which is often used to support multimedia quality.

CPU Model. We consider mobile devices with a single adaptive CPU that can operate at multiple speeds, $\{s_1, \cdots, s_m\}$. At a lower speed, the CPU consumes less power. However, a lower speed may increase the power of other resources such as memory. Since our goal is to save the total energy of the whole device, we are more interested in the total power of the device. In general, the relationship between the speed, s, and the total power, $p(s)$, can be obtained via measurements. Figure 1, for example, shows the measured power-speed relationship for the HP laptop.

Task Model. We consider *periodic* multimedia tasks (processes or threads) that release a job (e.g., decodes a video frame) every period. Each job has a soft deadline, typically defined as the end of the period, and consumes CPU cycles. Different jobs of the same task (e.g., decoding I, P, and B frames) may need different amount of cycles. That is, the instantaneous cycle demand for individual jobs changes largely. The probability distribution of cycle demand of a task, however, is much more stable due to the periodic nature of its jobs.

We take the kernel-based profiler in GRACE-OS [21] to estimate the probability distribution of each task. This profiler monitors the cycles usage for each job of the task and estimate the probability that a job demands no more than a certain amount of cycles, i.e.,

$$\mathcal{P}r(X \leq x) \tag{1}$$

where X is the random variable associated with the number of cycles demanded by each job.

Scheduling Algorithm. We use soft real-time scheduling to meet the performance and CPU requirements of multimedia tasks. In particular, the scheduler periodically allocates cycles to each task based on its statistical cycle demand, e.g., the 95th percentile of cycle demand of all its jobs. The statistical demand can be specified based on the task's characteristics (e.g., audio requires higher percentile than video). The purpose of this statistical, rather than the worst-case based, allocation is to improve CPU utilization while providing soft (statistical) performance guarantees.

The scheduler then enforces the allocation through an earliest deadline first (EDF) based algorithm. This algorithm makes admission control to ensure that the total CPU utilization (at the highest CPU speed s_m) of all concurrent tasks is no more than one, i.e.,

$$\sum_{i=1}^{n} \frac{\frac{C_i}{s_m}}{P_i} \leq 1 \tag{2}$$

where there are n tasks and each task demands C_i cycles per period P_i. Among all tasks, the scheduler first executes the task with the earliest deadline and positive allocation. As the task is executed, its cycle allocation is decreased by the number of cycles it consumes. When its allocation is exhausted, the task is preempted to run in best-effort mode for overrun protection.

3. THE PDVS ALGORITHM

The goal of PDVS is to minimize the total energy of the mobile device while providing soft performance guarantees to each multimedia task. To achieve this goal, PDVS extends the above real-time scheduling algorithm by changing the CPU speed of task execution. The purpose is to save energy since the CPU may run slower without affecting application performance.

Intuitively, we could set a uniform speed for all concurrent tasks until the task set changes. Assume there are n tasks and each task is allocated C_i cycles per period P_i. The total CPU demand of all tasks is

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \tag{3}$$

cycles per second (Hz). The uniform speed can be the lowest speed that is no less than the total demand, i.e.,

$$\min \left\{ s : s \in \{s_1, ..., s_m\} \text{ and } s \geq \sum_{i=1}^{n} \frac{C_i}{P_i} \right\} \tag{4}$$

If each task used its allocated cycles *exactly*, this uniform speed would minimize CPU energy due to the convex nature of the speed-power relationship [3, 9].

However, multimedia tasks often change their cycle demand dynamically. In particular, a task may, and often does, complete a job earlier before using up the allocated cycles. For example, if the cycle allocation of a task is based on the 95th percentile of the number of cycles demanded by its jobs, then about 95% of its jobs will complete earlier. Early completion may eventually cause the CPU to be idle, which, in turn, results in energy waste since the device still consumes energy during the idle time.

Therefore, there is a potential to save more energy by avoiding or reducing the idle time. To realize this potential, PDVS dynamically adapts the CPU speed of each job execution in a way that minimizes the total energy consumed during the job execution while bounding the job's execution time. The reason for bounding the execution

time is not to miss the deadline of the job or other jobs executed after the job. In other words, each job should finish within a certain amount of time.

We therefore allocate a time budget for each job. Specifically, if there are n concurrent tasks and each task is allocated C_i cycles per period P_i, then the ith task is allocated

$$T_i = \frac{C_i}{\sum_{i=1}^{n} \frac{C_i}{P_i}} \quad (5)$$

time units per period P_i (i.e. for each of its jobs). That is, we distribute the time among all tasks based on their cycle demands. Intuitively, if there is only a single task, then its time budget equals to its period; if multiple tasks run concurrently, they need to share time with each other and hence get a shorter time budget. By using cycle and time allocation together, we can adapt the execution speed for each job as long as the job can use its allocated cycles within its allocated time.

Now the problem is how to dynamically adapt the execution speed for jobs of each invitational task. Without loss of generality, we consider a specific task with cycle allocation C and time allocation T per period P. In the next two subsections, we formulate the speed adaptation problem and then describe its solution.

3.1 Problem Formulation

The basic idea behind PDVS is to minimize the total energy consumed during each job execution while limiting the job's execution time within its time budget. To do this, PDVS sets a speed for each of the cycles allocated to the job. If a cycles $x, 1 \leq x \leq C$, is executed at speed $s(x)$, its execution time is $\frac{1}{s(x)}$. The energy consumed by the device during this time interval is

$$\frac{1}{s(x)} \times p(s(x)) = \frac{p(s(x))}{s(x)} \quad (6)$$

where $p(s(x))$ is the total power consumed by the whole device at speed $s(x)$. As we discussed in Section 2, multimedia tasks demand cycles statistically. That is, the cycle x is executed with a probability and its *expected energy* is

$$F(x)\frac{p(s(x))}{s(x)} \quad (7)$$

where $F(x)$ is the tail distribution function, i.e.,

$$F(x) = 1 - \mathcal{P}r(X \leq x) \quad (8)$$

By setting a speed for each of the cycles allocated the job, we try to minimize the total energy consumed during the job execution while bounding its execution time. More formally, we formulate the speed adaptation problem as

$$\min \quad \underbrace{\sum_{x=1}^{C} F(x)\frac{p(s(x))}{s(x)}}_{\text{busy energy}} + \underbrace{\left(T - \sum_{x=1}^{C} F(x)\frac{1}{s(x)}\right) p_{idle}}_{\text{idle energy}} \quad (9)$$

$$\text{s.t.} \quad \sum_{x=1}^{C} \frac{1}{s(x)} \leq T \quad (10)$$

$$s(x) \in \{s_1, ..., s_m\}, 1 \leq x \leq C \quad (11)$$

where T is the time budget allocated to the job and p_{idle} is the device power when the CPU is idle at the lowest speed.

In Equation (9), the first part is the energy consumed when executing all allocated cycles; the second part is the energy consumed during the residual time, which equals to the time budget minus the expected execution time of all allocated cycles. During this residual time, the CPU is often idle since the task needs to wait until next job is available[1]. This idle interval is often very short; so we cannot switch the CPU to the lower power *sleep* state due to the switching overhead (which is, e.g., 160 ms for StrongARM SA-1100 [4]). We therefore set the CPU to the lowest speed during the idle interval. Note that Equation (10) bounds the worst-case, rather than the expected, execution time of all allocated cycles.

The above constrained optimization is similar to the energy optimization in previous statistical DVS algorithms [7, 11, 21] in that all of them find a speed for each of the allocated cycles to minimize their total energy. However, PDVS differs substantially from previous statistical DVS algorithms (and generally most of previous DVS algorithms) for two reasons: First, PDVS explicitly considers the discrete set of speeds when optimizing energy. Second, PDVS considers the energy consumed when the CPU is idle and also minimizes the total energy consumed by the whole device, rather than CPU energy only.

3.2 Practical Considerations and Solution

In practice, however, we cannot use the constrained optimization in Equations (9)-(11) directly for two reasons: First, the number of allocated cycles, C, may be very large for multimedia tasks (e.g., in millions). Consequently, the computational overhead for solving the optimization problem may be unacceptably large. Second, we cannot set a speed for individual cycles since each speed change may take tens of microseconds (Section 5.2).

To reduce the cost, we use a piece-wise approximation technique that groups the allocated cycles and sets a same speed within each group. Specifically, we first use a set of points, $b_1, ..., b_{k+1}$, to divide the allocated cycles into k groups, each with size $g_i, 1 \leq i \leq k$. We then find a speed $s(b_i)$ for each group $[b_i, b_{i+1})$. In this way, we rewrite the above constrained optimization as

$$\min \quad \sum_{i=1}^{k} g_i \frac{F(b_i)p(s(b_i))}{s(b_i)} + \left(T - \sum_{i=1}^{k} g_i \frac{F(b_i)}{s(b_i)}\right) p_{idle} \quad (12)$$

$$\text{s.t.} \quad \sum_{i=1}^{k} g_i \frac{1}{s(b_i)} \leq T \quad (13)$$

$$s(b_i) \in \{s_1, ..., s_m\}, 1 \leq i \leq k \quad (14)$$

This new optimization problem (and the original one) is NP hard since one can easily prove that the multi-choice Knapsack problem [16], which is known to be NP hard, is an instance of the optimization problem in Equations (12)-(14). Being NP hard, the optimization problem does not have an optimal yet feasible solution.

PDVS provides a heuristic solution with a dynamic programming algorithm, based on the algorithm proposed by Pisinger [16]. Specifically, we sort the combinations of all speed options for all cycle groups in the non-decreasing order of a slope, which is defined as the energy-to-time ratio by increasing a group's speed to the next higher speed. We initially set all groups to the lowest speed and increase each group's speed by visiting the sorted slope list until we meet the time constraint in Equation (13). The complexity of this algorithm is $O(km \log(km))$, where k is the number of cycle groups and m is the number of speeds. This algorithm is called for each task when the demand distribution or time budget changes. Multimedia tasks often have a stable demand distribution, as shown

[1]Although the EDF algorithm allows other tasks to share the residual time, the CPU may be idle eventually. As a part of future work, we are investigating how PDVS can utilize the residual time; e.g., we can allocate the residual time, ΔT, to the next task and hence relax its time constraint to $\Delta T + T$ in Equation (10).

in our previous work [21]. The time budget changes in coarse time granularity when another task joins or leaves. As a result, the algorithm for calculating the speed schedule is called infrequently.

The output of this algorithm is a speed $s(b_i)$ for each cycle group $i, 1 \le i \le k$. We refer to the pair $(b_i, s(b_i))$ as a speed changing point and the list of pairs

$$(b_1, s(b_1)), (b_2, s(b_2)), \cdots, (b_k, s(b_k)) \quad (15)$$

as the *speed schedule* of the task. The speed schedule is sorted by the increasing order of the cycle. We also combine adjacent points with the same speed. As a result, although a task may have a large number of cycle groups, its speed schedule consists a small number of changing points due to the discrete speed levels.

PDVS stores the speed schedule of a task into its process control block and updates the speed schedule when the task's time budget changes due to the entry or exit of other tasks (Equation (5)). The scheduler uses the speed schedule to adapt the executing speed of the task. Specifically, when the task starts a new job, its execution speed is set to $s(b_1)$; when the cycle usage of the job is b_i cycles, its execution speed is changed to $s(b_i)$. Furthermore, during a context switch, the scheduler stores the execution speed of the switched-out task and sets the execution speed for the switched-in task based on its speed schedule. Figure 2 illustrates the scheduling process.

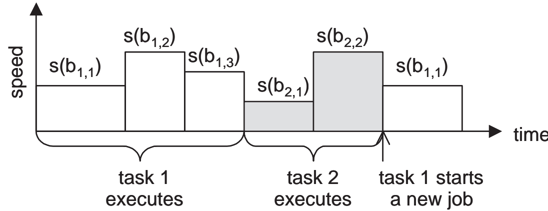

Figure 2: Scheduling of two tasks: The CPU speed changes during the task execution and in a context switch.

4. IMPLEMENTATION

We have implemented PDVS in the Linux kernel[2]. The hardware platform for our implementation is an HP N5470 laptop with a single Athlon CPU [1]. This CPU supports six different speeds: 300, 500, 600, 700, 800, and 1000 MHz. Its speed can be adjusted dynamically under operating system control. The operating system is Red Hat 8.0 with a modified Linux kernel 2.6.5. Figure 3 illustrates the software architecture of the implementation.

First, we add four new system calls (Table 1) to support soft real-time requirements of multimedia tasks (processes or threads). A task tells its demand distribution to the scheduler and makes CPU reservation. The scheduler calculates the speed schedule for each task when a task joins or leaves (i.e., upon the call `EnterSRT` or `ExiSRT`). The task calls `FinishJob` to tell the scheduler that the task has finished a job; so the scheduler can replenish the cycle allocation, set the scheduling priority, and reset the speed schedule for the next job.

Second, we add two new modules in the Linux kernel. The DVS module changes the CPU speed by writing the frequency and voltage to a system register [1]. The soft real-time scheduling module is responsible for allocating and charging cycles, calculating the speed schedule, and setting the real-time priority for individual tasks. The standard Linux scheduler then executes tasks based on their priority and adapts the execution speed if necessary. This is similar to hierarchical real-time scheduling [6, 15, 21]. To improve the scheduling granularity, we add a high resolution timer [2] into the kernel and attach the soft real-time scheduler as the call-back function of this timer. This timer expires and invokes scheduling every millisecond.

[2]The code is available at http://cairo.cs.uiuc.edu/software/grace.tar.gz.

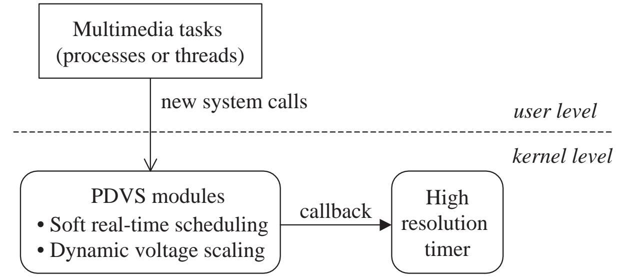

Figure 3: Architecture of PDVS implementation.

Table 1: New system calls for PDVS.

System Call	Description
`SetCycleGrp`	Specify cycle groups and their demand distribution.
`EnterSRT`	Enter soft real-time mode and reserve CPU cycles.
`FinishJob`	Tell the kernel the task has finished a job.
`ExitSRT`	Exit real-time mode and release reservation.

	// sample code
	`SetCycleGrp(groups)`
	`EnterSRT(period, cycle)`
	For each job
	  `DoAJob()` *// e.g., decoding a video frame*
	  `FinishJob()`
	`ExitSRT()`

5. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate PDVS. We first describe the experimental setup and interested metrics, and then report our measurement results.

5.1 Experimental Setup

Our experiments are performed on the HP Pavilion N5470 laptop with 256 MB memory. The experimental multimedia applications are three video codecs and are summarized in Table 2. Both of the `MPGDec` and `H263Dec` call the X library to display the decoded image. We use the priority inheritance protocol [17] to address the dependency between these two decoders and the X server. In particular, the scheduler sets the scheduling priority of the X server to that of a decoder when the decoder calls the X library. The X server then runs immediately at the CPU speed set by the previous decoder. When the X server returns, its priority is reset to best-effort. We have also tried other inputs for the codecs in Table 2 and obtained similar results. In this paper, we do not show these results due to the space limitation.

Table 2: Experimental multimedia applications.

Application	Input stream	Period (ms)
MPGDec	`Starwars.mpg` (3260 frames)	50
H263Enc	`Paris.cif` (1065 pictures)	150
H263Dec	`Paris.263` (1065 frames)	40

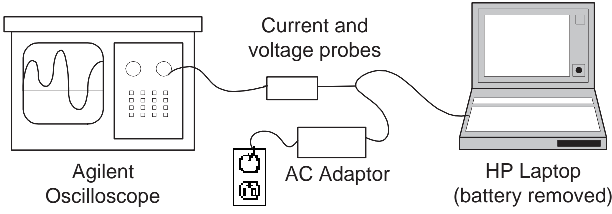

Figure 4: Power measurement with a digital oscilloscope.

To evaluate PDVS, we are interested in three metrics: *overhead*, *energy*, and *deadline miss ratio*. For overhead, we measure the cost for relevant operations such as scheduling. For the energy, we use an Agilent 54621A oscilloscope to measure the power consumed by the HP laptop. Specifically, we remove the battery from the laptop and the current and voltage of the AC power adaptor (Figure 4). The total power of the laptop is the product of the measured current and voltage, and the energy consumption is the integral of the total power over time. For performance support, we measure the deadline miss ratio for each individual task. Intuitively, the lower the miss ratio, the better the multimedia quality.

5.2 Overhead

We first analyze the overhead of PDVS's operations, including speed optimization (i.e., calculating the speed schedule as discussed in Section 3.2), soft real-time scheduling, and speed change. To measure the cost for speed optimization, we always run the CPU at the lowest speed 300 MHz and measure the time elapsed when using the dynamic programming algorithm to calculate the speed schedule for each task. The results (Figure 5) show that this cost is small and negligible relative to multimedia processing. For example, for `H263Enc`, which has 37 cycle groups, the speed schedule calculation takes less than 30 microseconds (μs), while a typical video frame processing takes tens of milliseconds (ms).

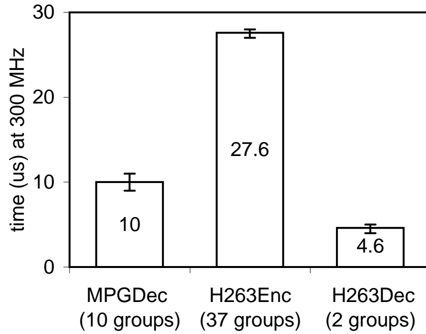

Figure 5: Cost of speed optimization: the bars show the mean of 6 measurements and the error bars show the minimum and maximum of 6 measurements.

To measure the cost of soft real-time scheduling, we run different number of tasks and measure the time elapsed for each invocation of soft real-time scheduling. Figure 6 plots the results. The scheduling cost is very small and below 3 μs, thus negligible during multimedia processing. In terms of relative overhead, the scheduling cost is below 0.3% since the scheduling granularity is 1000 μs.

Finally, we measure the cost for DVS. To do this, we adjust the CPU from one speed to another and measure the time elapsed for each adjustment, during which the CPU cannot perform computation. Figure 7 plots the results. The cost for speed adaptation

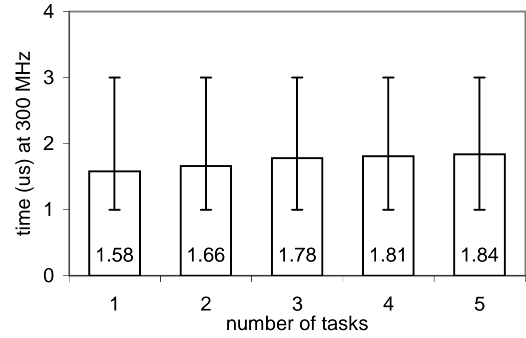

Figure 6: Cost of soft real-time scheduling: the bars show the average of 5,000 measurements and the error bars show the 95% confidence intervals.

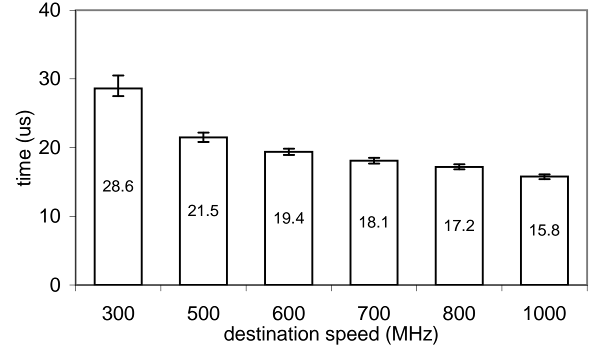

Figure 7: Cost of changing the CPU frequency: the bars show the mean of 12 measurements and the error bars show the minimum and maximum of 12 measurements.

depends on the destination speed and is below 40 μs. This cost is acceptable for the PDVS algorithm to change the speed several (often less than six) times during a job execution, since a job execution often takes tens of milliseconds.

5.3 Benefits of PDVS

To show the benefits of PDVS, we evaluate how much energy it saves without substantially degrading multimedia performance. To do this, we compare PDVS with the following DVS techniques:

- *No DVS*. This is the baseline technique in which the CPU always runs at the highest speed.
- *Uniform DVS*. It sets a uniform speed for all concurrent tasks until the task set changes. This uniform speed is the lowest speed that is greater than or equal to the total CPU demand $\sum_{i=1}^{n} \frac{C_i}{P_i}$, where there are n tasks and each task is allocated C_i cycles per period P_i.
- *Reclamation DVS* [3, 15, 22]. It first sets a uniform speed for all concurrent tasks and lowers the speed when a task completes a job early. Specifically, it sets the speed to $\sum_{i=1}^{n} \frac{C_i^*}{P_i}$, where C_i^* is the number of allocated cycles when the ith task releases a job and is the number of consumed cycles when the task completes a job.
- *Statistical DVS* [7, 11, 21]. It is same as our PDVS algorithm but assumes an ideal CPU. Specifically, it optimizes the execution speed based on the demand distribution of each task.

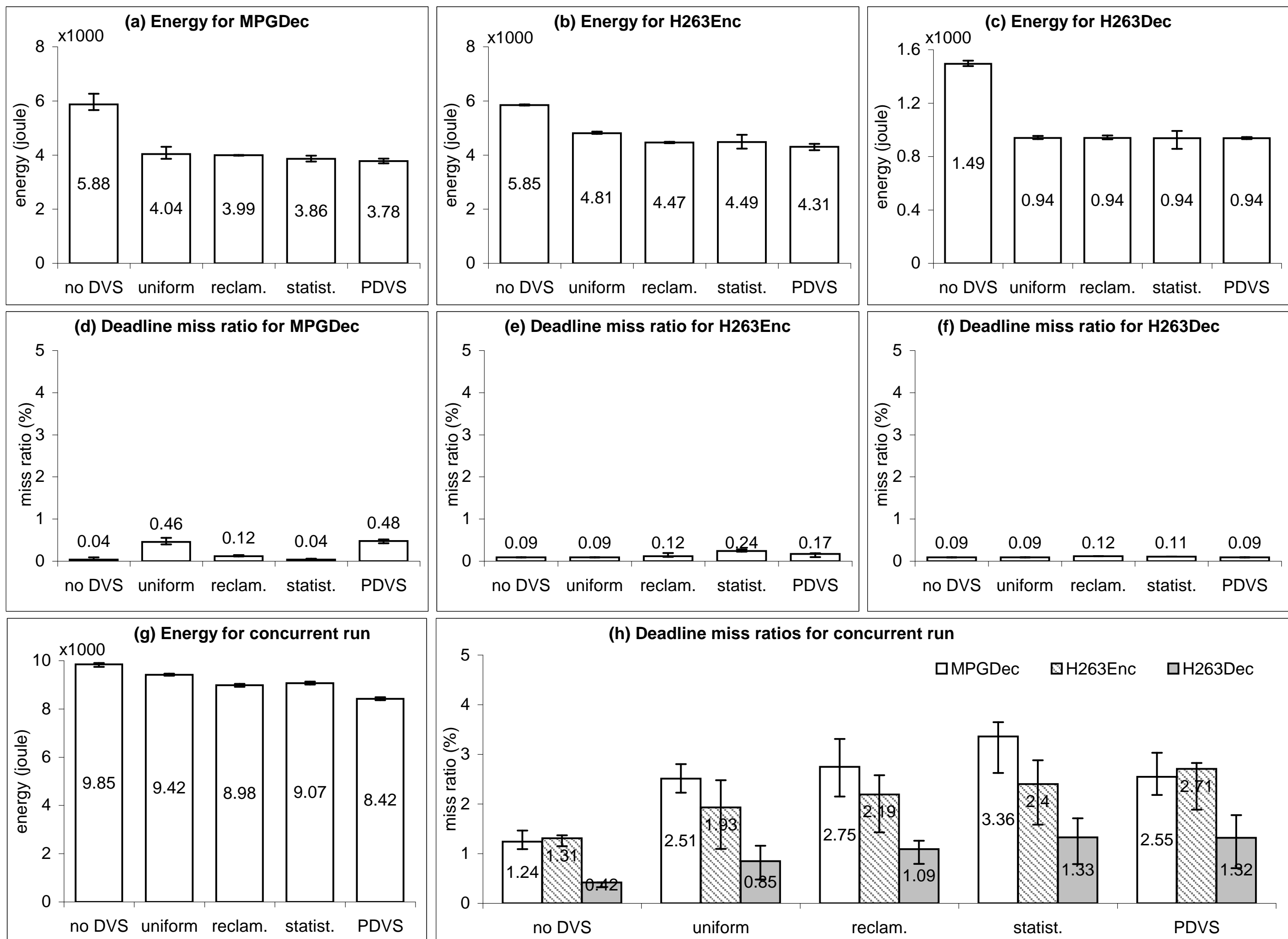

Figure 8: Comparing PDVS with other DVS techniques: the bars show the mean of five measurements and the error bars show the minimum and maximum of the five measurements.

This optimization, however, assumes that the CPU supports a continuous range of speeds and the CPU power is proportional to the cube of the speed. At runtime, the speed calculated in this optimization is rounded to the upper bound of the available speeds.

Like PDVS, all the above techniques are integrated with soft real-time scheduling. We use our previously developed profiler [21] to profile the number of cycles demanded by all jobs of each task offline. Unless specified otherwise, the scheduler allocates cycles to each task based on the 95th percentile of demand cross all jobs of the task. That is, the allocation is sufficient for about 95% of jobs, and the desired deadline miss ratio should be below 5%.

Under each of the DVS techniques, we perform two kinds of experiments, single run and concurrent run. In the single run cases, we run each of `MPGDec`, `H263Enc` and `H263Dec` (Table 2) one at a time. In the concurrent case, we run all three codecs with different starting time; to admit all of them (i.e., their total CPU utilization at the highest speed is no more than one), we let `MPGDec` decode video in gray mode and `H263Enc` encode a frame every 250 ms. In each of the single and concurrent runs, we measure the energy consumed by the laptop and deadline miss ratio for each task. Figure 8 reports these two metrics. We now use these results to evaluate PDVS.

Energy Saving: Compared to the baseline without DVS, all DVS techniques save energy significantly. The reason is that the CPU does not need to always run at the highest speed. Compared to other DVS techniques, PDVS consumes almost the same energy when running the single `H263Dec`. The reason is that `H263Dec` demands only 194 million cycles per second (MHz). To meet this low demand, the CPU can always run at the lowest speed 300 MHz (for PDVS, the speed schedule of `H263Dec` consists of only one speed changing point with speed 300 MHz). Consequently, the energy is already minimized. This indicates that PDVS (and other DVS algorithms as well) is limited by the lower bound of the supported speeds.

In all other cases, PDVS saves more energy than other DVS techniques. In particular, compared to the uniform and reclamation DVS techniques, PDVS reduces the total energy by 2.6% to 10.4%. This clearly demonstrates the benefits of optimizing energy based on the demand distribution of each task. Compared to statistical DVS, PDVS reduces the total energy by 4.0% to 7.0%. This shows the benefits of considering the discrete speed options when optimizing energy.

Another interesting result is that for the single `H263Enc` run and the concurrent run, statistical DVS yields no benefits than the simple uniform DVS technique and even consumes more energy. The reason is that statistical DVS incorrectly assumes an ideal CPU

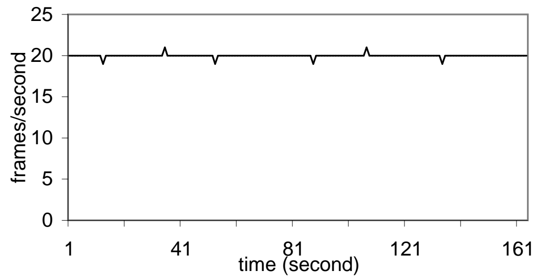

Figure 9: The frame rate of `MPGDec` under PDVS in the concurrent run case.

when optimizing energy. As the assumption does not hold for our HP laptop, statistical DVS ends up consuming more energy. This clearly illustrates the motivation for explicitly considering the non-ideal CPU characteristics when optimizing energy.

Performance Support: PDVS saves energy with little impact on multimedia performance. Recall that each task demands CPU based on its 95th percentile of cycle usage across all jobs. As a result, the expected deadline miss ratio would be around 5%. In the single run cases, PDVS bounds the deadline miss ratio under 0.5%. That is, the single codec meet almost all deadlines. The reason is that it can utilize the unallocated cycles when they overrun (i.e., need more cycles than the allocated). The deadline miss ratios in the concurrent run are much higher than the single runs. The reason is that multiple tasks may overrun simultaneously and hence run in best-effort mode to compete for the CPU. However, the deadline miss ratios are still below 5% and hence meet application performance requirement.

To further evaluate the impact of deadline miss ratio on the user's perceived multimedia quality, we also measure the frame rate in the application level. Figure 9 shows the frame rate of `MPGDec` under PDVS (the frame rates under other DVS techniques are similar) in the concurrent run case. The actual frame rate is very close to the desired frame rate, which is 20 frames per second since the period is 50 milliseconds (Table 2). This implies that PDVS can save energy without affecting the perceptual multimedia quality from the user's point of view.

5.4 Result Summary and Discussions

Overall, our experimental results show that although PDVS employs a heuristic algorithm for the NP-hard energy optimization problem, it saves energy substantially with little impact on multimedia QoS. Compared to systems without DVS, PDVS reduces the total energy of the laptop by 14.4% to 37.2%. Compared to systems that perform DVS based on the prediction of instantaneous cycle demand or assume an ideal CPU, PDVS further reduces the total energy by up to 10.4%.

We also found that the priority inheritance protocol decreases the deadline miss ratio significantly for video decoders that use the X library. Our previous experiments without the protocol missed more than 8.5% of deadlines, especially for the concurrent case. On the other hand, DVS also provides opportunities for soft real-time scheduling. For example, when a task overruns, we can speed up the CPU to allocate the task extra cycles (e.g., in [22]), thus meeting the deadline.

6. RELATED WORK

Energy is a critical resource for battery-powered mobile devices. Recently, there has been a lot of related work on reducing energy for various components such as CPU, network, and memory. In this Section, we focus on the related work on saving CPU energy.

In general, there are two approaches to saving CPU energy: dynamic power management (DPM) [4] and dynamic frequency/ voltage scaling (DVS) [3, 5, 8, 15, 20, 21]. DPM puts the idle CPU into the lower-power sleep state, while DVS lowers the speed and voltage of the active CPU. DPM is not suitable for our targeted multimedia applications since they need to use CPU periodically (e.g., every 30 milliseconds) and consequently the idle interval is often too short to put the CPU into sleep.

DVS slows down the CPU based on application workload, and has been investigated in two major areas, general-purpose systems (GP-DVS) and real-time systems (RT-DVS). GP-DVS algorithms heuristically predict the workload based on average CPU utilization [5, 8, 20]. They cannot be directly applied to multimedia applications since the heuristic prediction may violate multimedia timing constraints.

RT-DVS algorithms, often integrated with real-time scheduling, allocate CPU resource to individual applications based on their worst-case demand and adapt the CPU speed based on the worst-case allocation [3, 14, 15]. Applications may, and often do, complete earlier before using up the worst-case allocation since they change CPU demand dynamically. To handle the runtime variations, some reclamation techniques have been proposed to reclaim the residual allocation to save more energy [3, 15]. These reclamation techniques first run the CPU fast by assuming the worst-case demand, and then slow down the CPU when an application completes earlier. These RT-DVS algorithms do not consider the soft real-time nature and CPU usage patterns of multimedia applications, which provides more opportunities for energy saving. For example, our experimental results in Section 5 imply that the reclamation-based DVS consumes more energy than PDVS when running multimedia applications.

Statistical DVS is an alternative approach to handling runtime variations of CPU demand [7, 11, 21]. Simunic et al. [18] and Sinha et al. [19] proposed algorithms that changes speed for each job of a task based on a stochastic model (e.g., Markov process) of the task's CPU demands. PDVS differs from them in that they changes speed only at the beginning of a job, while PDVS changes speed within a job execution.

Some groups have also investigated on intra-job statistical DVS. Gruian [7] used statistical DVS for hard real-time systems. Lorch and Smith [11] used it to improve GP-DVS algorithms. The basic idea of these statistical DVS algorithms is similar to that in PDVS: minimizing energy by adapting the execution speed based on the demand distribution of applications. However, these previous algorithms assume an ideal CPU that can change speed continuously and consumes power cubically to the speed. PDVS differs from them (and generally most of previous DVS algorithm) substantially by explicitly considering non-ideal CPUs commonly used in mobile devices. PDVS is built on our previous work on GRACE-OS [21], which provides a kernel-based profiling to estimate the probability distribution of cycle demand of applications and develops a statistical DVS for an ideal CPU.

Recently, much research effort has been made on handling the discrete speed options of the CPU. For example, Miyoshi et al. [12] analyzed the runtime effect of DVS empirically and found that different systems have different optimal speed points. This work is complementary to PDVS; PDVS also explicitly considers the total power at different speeds. Other related work includes mapping the calculated speed to the speeds supported by the CPU. A simple approach is to round the calculated speed to the upper bound of the supported speeds [15, 21]. An alternative approach is to emu-

late the calculated speed with two bounding supported speeds [7, 9, 11]. This approach distributes cycles that need to be executed at the calculated speed into two parts, one for the lower bound and the other for the upper bound. This emulation approach has been shown to be effective in simulations. It, however, may potentially result in large overhead when used in real implementations since it changes the speed more frequently.

7. CONCLUSION

This paper presents the design, implementation, and evaluation of *PDVS*, a practical voltage scaling algorithm for battery-powered mobile devices that primarily run multimedia applications. PDVS minimizes the total energy consumed by the whole device while meeting multimedia performance requirements. To do this, PDVS extends traditional real-time scheduling by deciding *what execution speed* in addition to when to execute what applications. PDVS makes these decisions to minimize energy by explicitly considering the discrete speed levels of the CPU, the total power of the device at different speeds, and the probability distribution of cycle demand of multimedia applications.

PDVS has been implemented as a part of GRACE-OS [21]. Our experimental results, based on the real implementation and measurements, show that although PDVS uses a heuristic solution for the NP-hard energy optimization problem, PDVS provides significant benefits for both energy saving and performance support with acceptable overhead. It reduces the total energy of the laptop by 14.4% to 37.2% compared to non-DVS algorithms and by up to 10.4% compared to DVS algorithms assuming an ideal CPU.

As a part of the future work, we are currently integrating other scheduling techniques into PDVS, e.g., sharing budget among tasks to relax the time constraint when optimizing energy. We also plan to investigate the impact of the hints from applications (e.g., updating the demand distribution due to scene changes) on DVS. Finally, we would like to integrate PDVS with energy management techniques for other resources such as wireless network card.

8. ACKNOWLEDGMENTS

We would like to thank the GRACE members, especially Professor Sarita Adve, for their informative discussions on energy saving, anonymous reviewers for their constructive feedback for improving this paper. This work was supported by NSF under CCR 02-05638 and CISE EIA 99-72884. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF or the U.S. government.

9. REFERENCES

[1] AMD. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. http://www.amd.com, Nov. 2001.

[2] G. Anzinger et al. High resolution POSIX timers. http://high-res-timers.sourceforge.net/, 2004.

[3] H. Aydin, R. Melhem, D. Mosse, and P. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proc. of 22nd IEEE Real-Time Systems Symposium*, Dec. 2001.

[4] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3), June 2000.

[5] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[6] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of Symposium on Operating System Design and Implementation*, Oct. 1996.

[7] F. Gruian. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proc. of Intl. Symp. on Low-Power Electronics and Design*, Aug. 2001.

[8] D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proc. of 4th Symposium on Operating System Design and Implementation*, Oct. 2000.

[9] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of Intl. Symp. on Low-Power Electronics and Design*, 1998.

[10] X. Liu, P. Shenoy, and W. Gong. A time series-based approach for power management in mobile processors and disks. In *14th ACM Workshop on Network and Operating System Support for Audio and Video (NOSSDAV)*, June 2004.

[11] J. Lorch and A. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proc. of ACM SIGMETRICS 2001 Conference*, June 2001.

[12] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *16th Annual ACM International Conference on Supercomputing*, June 2002.

[13] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian. Integrated power management for video streaming to mobile devices. In *Proc. of ACM Multimedia*, Nov. 2003.

[14] T. Pering, T.Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proc. of Intl. Symposium on Low Power Electronics and Design*, July 2000.

[15] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of 18th Symposium on Operating Systems Principles*, Oct. 2001.

[16] D. Pisinger. A minimal algorithm for the multiple-choice Knapsack problem. *European Journal of Operational Research, 83*, pages 394–410, 1995.

[17] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchroniztion. *IEEE trans. on computers, 39(9)*, Sept. 1990.

[18] T. Simunic et al. Dynamic voltage scaling and power management for portable systems. In *Proc. of Design Automation Conference*, June 2001.

[19] A. Sinha and A. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proc. of 4th International Conference on VLSI Design*, Jan. 2001.

[20] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. of Symposium on Operating Systems Design and Implementation*, Nov. 1994.

[21] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proc. of 19th Symposium on Operating Systems Principles*, Oct. 2003.

[22] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Proc. of Multimedia Computing and Networking Conf.*, Jan. 2003.