Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems

Wanghong Yuan, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave, Urbana, IL 61801, USA
{wyuan1, klara}@cs.uiuc.edu

ABSTRACT

This paper presents *GRACE-OS*, an energy-efficient soft real-time CPU scheduler for mobile devices that primarily run multimedia applications. The major goal of GRACE-OS is to support application quality of service and save energy. To achieve this goal, GRACE-OS integrates dynamic voltage scaling into soft real-time scheduling and decides how fast to execute applications in addition to when and how long to execute them. GRACE-OS makes such scheduling decisions based on the probability distribution of application cycle demands, and obtains the demand distribution via online profiling and estimation. We have implemented GRACE-OS in the Linux kernel and evaluated it on an HP laptop with a variable-speed CPU and multimedia codecs. Our experimental results show that (1) the demand distribution of the studied codecs is stable or changes smoothly. This stability implies that it is feasible to perform stochastic scheduling and voltage scaling with low overhead; (2) GRACE-OS delivers soft performance guarantees by bounding the deadline miss ratio under application-specific requirements; and (3) GRACE-OS reduces CPU idle time and spends more busy time in lower-power speeds. Our measurement indicates that compared to deterministic scheduling and voltage scaling, GRACE-OS saves energy by 7% to 72% while delivering statistical performance guarantees.

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, ranging from laptops to cellular phones, are becoming important platforms for processing multimedia data such as audio, video, and images. Compared to traditional desktop and server systems, such mobile systems need both to support application quality of service (QoS) requirements and to save the battery energy. The operating systems therefore should manage system resources, such as the CPU, in a QoS-aware and energy-efficient manner.

On the other hand, these mobile systems also offer new opportunities. First, system resources are able to operate at multiple modes, trading off performance for energy. For example, mobile processors on the market today, such as Intel's XSacle, AMD's Athlon, and Transmeta's Crusoe, can change the speed (frequency/voltage) and corresponding power at runtime. Second, multimedia applications present *soft real-time* resource demands. Unlike hard real-time applications, multimedia applications require only statistical performance guarantees (e.g., meeting 96% of deadlines). Unlike best-effort applications, as long as multimedia applications complete a job (e.g., a video frame decoding) by its deadline, the actual completion time does not matter from the QoS perspective. This soft real-time nature results in the possibility of saving energy without substantially affecting application performance.

This paper exploits these opportunities to address the above two challenges, namely, QoS provisioning and energy saving. This work was done as part of the Illinois GRACE cross-layer adaptation framework, where all system layers, including the hardware, operating system, network, and applications, cooperate with each other to optimize application QoS and save energy [1, 36]. In this paper, we discuss the OS resource management of the GRACE framework. In particular, we focus on CPU scheduling and energy saving for stand-alone mobile devices.

Dynamic voltage scaling (DVS) is a common mechanism to save CPU energy [10, 14, 15, 23, 27, 28, 34]. It exploits an important characteristic of CMOS-based processors: the maximum frequency scales almost linearly to the voltage, and the energy consumed per cycle is proportional to the square of the voltage [8]. A lower frequency hence enables a lower voltage and yields a quadratic energy reduction.

The major goal of DVS is to reduce energy by as much as possible without degrading application performance. The effectiveness of DVS techniques is, therefore, dependent on the ability to predict application CPU demands— overestimating them can waste CPU and energy resources, while underestimating them can degrade application performance. In general, there are three prediction approaches: (1) monitoring average CPU utilization at periodic intervals [10, 15, 26, 34], (2) using application worst-case CPU demands [27, 28], and (3) using application runtime CPU usage [14, 23, 28]. The first two approaches, however, are not suitable for multimedia applications due to their highly dynamic demands: the interval-based approach may violate the timing constraints of multimedia applications, while the worst-case-based approach is too conservative for them.

We therefore take the third approach, i.e., runtime-based DVS, and integrate it into soft real-time (SRT) scheduling. SRT scheduling is commonly used to support QoS by combining predictable CPU allocation (e.g., proportional sharing and reservation) and real-time scheduling algorithms (e.g., earliest deadline first) [7, 9, 13, 25, 18, 19, 32]. In our integrated approach, the DVS algorithms are implemented in the CPU scheduler. The enhanced scheduler, called *GRACE-OS*, decides how fast to execute applications in addition to when and how long to execute them. This integration enables the scheduler to make DVS decisions properly, since the scheduler has the full knowledge of system states such as performance requirements and resource usage of applications.

Our goal is to obtain benefits of both SRT and DVS—to maximize the energy saving of DVS, while preserving the performance guarantees of SRT scheduling. To do this, we introduce a *stochastic* property into GRACE-OS. Specifically, the scheduler allocates cycles based on the statistical performance requirements and probability distribution of cycle demands of individual application tasks (processes or threads). For example, if an MPEG decoder requires meeting 96% of deadlines and for a particular input video, 96% of frame decoding demands no more than 9 million cycles, then the scheduler can allocate 9 million cycles per frame to the decoder. Compared to the worst-case-based allocation, this stochastic allocation increases CPU utilization. It also saves energy at the task-set level, since the CPU can run at a minimum speed that meets the aggregate statistical demand of all concurrent tasks. For example, if an MPEG video decoder and an MP3 audio decoder are running concurrently with statistical allocation of 300 and 50 million cycles per second, respectively, then the CPU can slow down to 350 MHz to save energy.

Further, the potential exists to save more energy at the task level. The reason is that a task may, and often does, complete a job before using up its allocated cycles. Such early completion often results in CPU idle time, thus resulting in energy waste. To realize this potential, GRACE-OS finds a *speed schedule* for each task based on the probability distribution of the task's cycle demands. This speed schedule enables each job of the task to start slowly and to accelerate as the job progresses. Consequently, if the job completes early, it can avoid the high speed (high energy consumption) part. This stochastic, intra-job DVS is in sharp contrast to previous DVS approaches that either execute an entire job at a uniform speed or start a job at a higher speed and decelerate upon early completion.

Since the stochastic scheduling and DVS are both dependent on the demand distribution of tasks, we estimate it via online profiling and estimation. We first use a kernel-based profiling technique to monitor the cycle usage of a task by counting the number of cycles during task execution. We then use a simple yet effective histogram technique to estimate the probability distribution of the task's cycle usage. Our estimation approach is distinguished from others (e.g., [14, 23, 32]) in that it can be used online with low overhead. This is important and necessary for live multimedia applications such as video conferencing.

We have implemented GRACE-OS in the Linux kernel, and evaluated it on an HP Pavilion laptop with a variable speed processor and with multimedia applications, including codecs for speech, audio, and video. The experimental results show four interesting findings:

1. Although the studied code applications vary instantaneous CPU demands greatly, the probability distribution of their cycle demands is stable or changes slowly and smoothly. Therefore, GRACE-OS can either estimate the demand distribution from a small part of task execution (e.g., first 100 jobs) or update it infrequently. This stability indicates that it is feasible to perform stochastic scheduling and DVS based on the demand distribution.

2. GRACE-OS delivers soft performance guarantees with stochastic (as opposed to worst case) allocation: it meets almost all deadlines in a lightly loaded environment, and bounds the deadline miss ratio under application-specific requirements (e.g., meeting 96% of deadlines) in a heavily loaded environment.

3. Compared to other systems that perform allocation and DVS deterministically, GRACE-OS reduces CPU idle time and spends more busy time in lower-power speeds, thereby saving energy by 7% to 72%.

4. GRACE-OS incurs acceptable overhead. The cost is 26-38 cycles for the kernel-based online profiling, 0.25-0.8 ms for the histogram-based estimation, 80-800 cycles for SRT scheduling, and 8,000-16,000 cycles for DVS. Further, the intra-job DVS does not change CPU speed frequently. For our studied codes, the average number of speed changes is below 2.14 per job.

The rest of the paper is organized as follows. Section 2 introduces the design and algorithms of GRACE-OS. Sections 3 and 4 present the implementation and experimental evaluation, respectively. Section 5 compares GRACE-OS with related work. Finally, Section 6 summarizes the paper.

2. DESIGN AND ALGORITHMS

We first introduce the application model, on which the stochastic scheduling and DVS are based. We consider *periodic* multimedia tasks (processes or threads) that release a job per period, e.g., decoding a video frame every 30 ms. A job is a basic computation unit with timing constraints and is characterized by a release time, a finishing time, and a soft deadline. The deadline of a job is typically defined as sum of its release time and the period, i.e., the release time of the next job. By *soft* deadline, we mean that a job should, but does not have to, finish by this time. In other words, a job may miss its deadline. Multimedia tasks need to meet some percentage of job deadlines, since they present soft real-time performance requirements.

We use the *statistical performance requirement*, ρ, to denote the probability that a task should meet job deadlines;

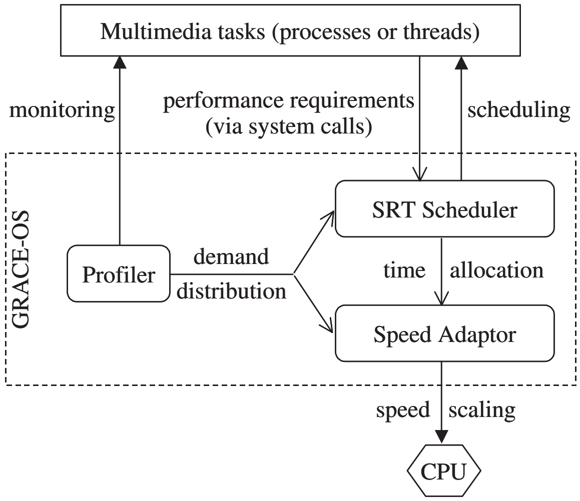

Figure 1: GRACE-OS architecture: the enhanced scheduler performs soft real-time scheduling and DVS based on the probability distribution of cycle demands of each multimedia task.

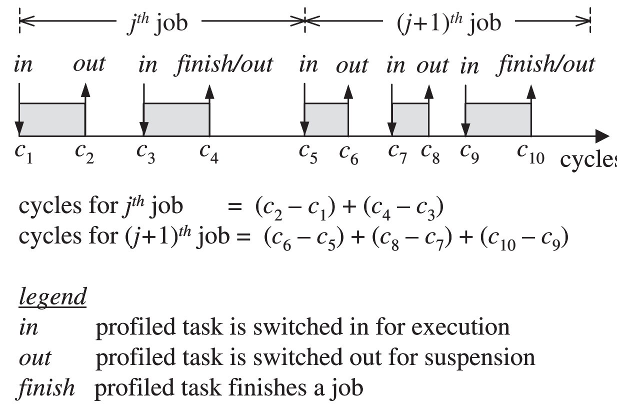

Figure 2: Kernel-based online profiling: monitoring the number of cycles elapsed between each task's switch-in and switch-out in context switches.

e.g., if $\rho = 0.96$, then the task needs to meet 96% of deadlines. In general, application developers or users can specify the parameter ρ, based on application characteristics (e.g., audio streams have a higher ρ than videos) or user preferences (e.g., a user may tolerate some deadline misses when the CPU is overloaded).

Next, we describe the architecture of GRACE-OS and its major algorithms for QoS provisioning and energy saving.

2.1 Overview

Our goal is to reduce CPU energy consumption by as much as possible, while meeting the statistical performance requirements of multimedia tasks. The operating system therefore needs to provide predictable CPU scheduling and speed scaling. To do this, we enhance the CPU scheduler to integrate scheduling and speed scaling. This enhanced scheduler, called GRACE-OS, consists of three major components: a profiler, a SRT scheduler, and a speed adaptor, as shown in Figure 1. The profiler monitors the cycle usage of individual tasks, and automatically derives the probability distribution of their cycle demands from the cycle usage. The SRT scheduler is responsible for allocating cycles to tasks and scheduling them to deliver performance guarantees. It performs soft real-time scheduling based on the statistical performance requirements and demand distribution of each task. The speed adaptor adjusts CPU speed dynamically to save energy. It adapts each task's execution speed based on the task's time allocation, provided by the SRT scheduler, and demand distribution, provided by the profiler.

Operationally, GRACE-OS achieves the energy-efficient stochastic scheduling via an integration of demand estimation, SRT scheduling, and DVS, which are performed by the profiler, SRT scheduler, and speed adaptor, respectively. We describe these operations in the following subsections.

2.2 Online estimation of demand distribution

Predictable scheduling and DVS are both dependent on the prediction of task cycle demands. Hence, the first step in GRACE-OS is to estimate the probability distribution of each task's cycle demands. We estimate the demand distribution rather than the instantaneous demands for two reasons. First, the former is much more stable and hence more predictable than the latter (as demonstrated in Section 4). Second, allocating cycles based on the demand distribution of tasks provides statistical performance guarantees, which is sufficient for our targeted multimedia applications.

Estimating the demand distribution of a task involves two steps: profiling its cycle usage and deriving the probability distribution of usage. Recently, a number of measurement-based profiling mechanisms have been proposed [3, 32, 37]. Profiling can be performed online or off-line. Off-line profiling provides more accurate estimation with the whole trace of CPU usage, but is not applicable to live applications. We therefore take the online profiling approach.

We add a *cycle counter* into the process control block of each task. As a task executes, its cycle counter monitors the number of cycles the task consumes. In particular, this counter measures the number of cycles elapsed between the task's switch-in and switch-out in context switches. The sum of these elapsed cycles during a job execution gives the number of cycles the job uses. Figure 2 illustrates this kernel-based online profiling technique.

Note that multimedia tasks tell the kernel about their jobs via system calls; e.g., when an MPEG decoder finishes a frame decoding, it may call `sleep` to wait for the next frame. Further, when used with resource containers [6], our proposed profiling technique can be more accurate by subtracting cycles consumed by the kernel (e.g., for interrupt handling). We currently do not count these cycles, since they are typically negligible relative to cycles consumed by a multimedia job.

Our proposed profiling technique is distinguished from others [3, 32, 37] for three reasons. First, it profiles during runtime, without requiring an isolated profiling environment (e.g., as in [32]). Second, it is customized for counting job cycles, and is simpler than general profiling systems that assign counts to program functions [3, 37]. Finally, it incurs small overhead, which happens *only* when updating cycle counters before a context switch. There is no additional

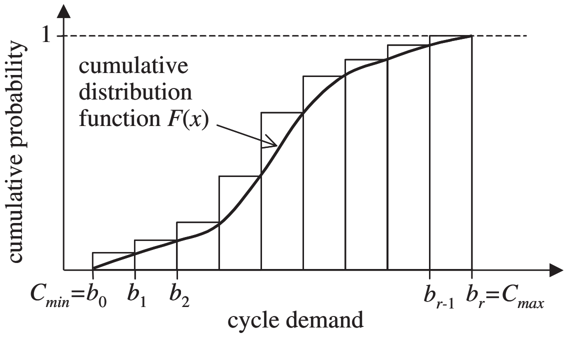

Figure 3: Histogram-based estimation: the histogram approximates the cumulative distribution function of a task's cycle demand.

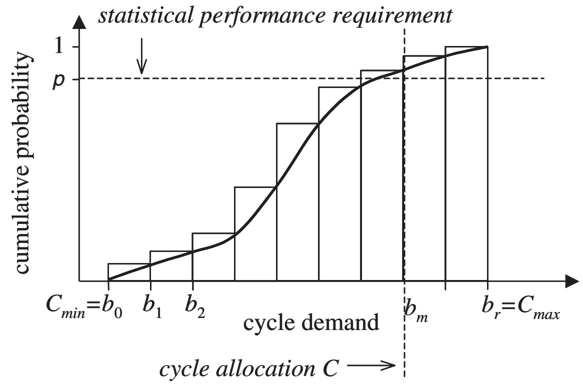

Figure 4: Stochastic cycle allocation: allocating the smallest b_m with $\mathcal{P}[X \leq b_m] \geq \rho$.

overhead, e.g., due to sampling interrupts [3, 37].

Next, we employ a simple yet effective *histogram* technique to estimate the probability distribution of task cycle demands. To do this, we use a profiling window to keep track of the number of cycles consumed by n jobs of the task. The parameter n can either be specified by the application or be set to a default value (e.g., the last 100 jobs). Let C_{min} and C_{max} be the minimum and maximum number of cycles, respectively, in the window. We obtain a histogram from the cycle usage as follows:

1. We use $C_{min} = b_0 < b_1 < \cdots < b_r = C_{max}$ to split the range $[C_{min}, C_{max}]$ into r equal-sized groups. We refer to $\{b_0, b_1, ..., b_r\}$ as the group boundaries.

2. Let n_i be the number of cycle usage that falls into the i^{th} group $(b_{i-1}, b_i]$. The ratio $\frac{n_i}{n}$ represents the probability that the task's cycle demands are in between b_{i-1} and b_i, and $\sum_{j=0}^{i} \frac{n_j}{n}$ represents the probability that the task needs no more than b_i cycles.

3. For each group, we plot a rectangle in the interval $(b_{i-1}, b_i]$ with height $\sum_{j=0}^{i} \frac{n_j}{n}$. All rectangles together form a histogram, as shown in Figure 3.

From a probabilistic point of view, the above histogram of a task approximates the cumulative distribution function of the task's cycle demands, i.e.,

$$F(x) = \mathcal{P}[X \leq x] \tag{1}$$

where X is the random variable of the task's demands. In particular, the rectangle height, $\sum_{j=0}^{i} \frac{n_j}{n}$, of a group $(b_{i-1}, b_i]$ approximates the cumulative distribution at b_i, i.e., the probability that the task demands no more than b_i cycles. In this way, we can estimate the cumulative distribution for the group boundaries of the histogram, i.e., $F(x)$ for $x \in \{b_0, b_1, ..., b_r\}$.

Unlike distribution parameters such as the mean and standard deviation, the above histogram describes the property of the full demand distribution. This property is necessary for stochastic DVS (see Section 2.4). On the other hand, compared to distribution functions such as normal and gamma (e.g., PACE [23]), the histogram-based estimation does not need to configure function parameters off-line. It is also easy to update, with low overhead, when the demand distribution changes, e.g., due to a video scene change.

2.3 Stochastic SRT scheduling

Multimedia tasks present demanding computational requirements that must be met in soft real time (e.g., decoding a frame within a period). To support such timing requirements, the operating system needs to provide soft real-time scheduling, typically in two steps: predictable cycle allocation and enforcement.

The key problem in the first step is deciding the amount of cycles allocated to each task. GRACE-OS takes a stochastic approach to addressing this problem: it decides cycle allocation based on the statistical performance requirements and demand distribution of each task. The purpose of this stochastic allocation is to improve CPU and energy utilization, while delivering statistical performance guarantees.

Specifically, let ρ be the statistical performance requirement of a task— the task needs to meet ρ percentage of deadlines. In other words, each job of the task should meet its deadline with a probability ρ. To support this requirement, the scheduler allocates C cycles to each job of the task, so that the probability that each job requires no more than the allocated C cycles is at least ρ, i.e.,

$$F(C) = \mathcal{P}[X \leq C] \geq \rho \tag{2}$$

To find this parameter C for a task, we search its histogram group boundaries, $\{b_0, b_1, ..., b_r\}$, to find the smallest b_m whose cumulative distribution is at least ρ, i.e., $F(b_m) = \mathcal{P}[X \leq b_m] \geq \rho$. We then use this b_m as the parameter C. Figure 4 illustrates the stochastic allocation process.

After determining the parameter C, we use an earliest deadline first (EDF) based scheduling algorithm to enforce the allocation. This scheduling algorithm allocates each task a budget of C cycles every period. It dispatches tasks based on their deadline and budget— selecting the task with the earliest deadline and positive budget. As the selected task is executed, its budget is decreased by the number of cycles it consumes. If a task overruns (i.e., it does not finish the current job but has used up the budget), the scheduler can either notify it to abort the overrun part, or preempt it to run in best-effort mode. In the latter case, the overrun task is either executed by utilizing unused cycles from other tasks or is blocked until its budget is replenished at the beginning of next period.

2.4 Stochastic DVS

SRT scheduling determines *which task* to execute as well as *when* and *how long* (in number of cycles) to execute it. We next discuss another scheduling dimension— *how fast* to execute a task (i.e., CPU speed scaling). The purpose of the speed scaling is to save energy, while preserving the statistical performance guarantees of SRT scheduling.

The intuitive idea is to assign a uniform speed to execute all tasks until the task set changes. Assume there are n tasks and each is allocated C_i cycles per period P_i. The aggregate CPU demand of the concurrent tasks is

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

cycles per second (MHz). To meet this aggregate demand, the CPU only needs to run at speed $\sum_{i=1}^{n} \frac{C_i}{P_i}$. If each task used exactly its allocated cycles, this uniform speed technique would consume minimum energy due to the convex nature of the CPU speed-power function [5, 17].

However, the cycle demands of multimedia tasks often vary greatly. In particular, a task may, and often does, complete a job before using up its allocated cycles. Such early completion often results in CPU idle time, thereby wasting energy. To save this energy, we need to dynamically adjust CPU speed. In general, there are two dynamic speed scaling approaches: (1) starting a job at the above uniform speed and then decelerating when it completes early, and (2) starting a job at a lower speed and then accelerating as it progresses. The former is conservative by assuming that a job will use its allocated cycles, while the latter is aggressive by assuming that a job will use fewer cycles than allocated. In comparison, the second approach saves more energy for jobs that complete early, because these jobs avoid the high speed (high energy consumption) execution. GRACE-OS takes the second approach, since most multimedia jobs (e.g., 95%) use fewer cycles than allocated (as shown in Section 4.2).

Specifically, we define a *speed schedule* for each task. The speed schedule is a list of scaling points. Each point (x, y) specifies that a job accelerates to the speed y when it uses x cycles. Among points in the list, the larger the cycle number x is, the higher the speed y becomes. The point list is sorted by the ascending order of the cycle number x (and hence speed y). According to this speed schedule, a task always starts a job at the speed of the first scaling point. As the job is executed, the scheduler monitors its cycle usage. If the cycle usage of a job is greater than or equal to the cycle number of the next scaling point, its execution is accelerated to the speed of the next scaling point.

Figure 5-(a) shows an example of a task's speed schedule with four scaling points. Figure 5-(b) shows the corresponding speed scaling for three jobs of the task. Each job starts at speed 100 MHz and accelerates as it progresses. If a job needs fewer cycles, it avoids the high speed execution. For example, the first job requires 1.6×10^6 cycles and thus needs to execute at speed 100 and 120 MHz only.

Next, we discuss how to calculate the speed schedule for each task based on its demand distribution, similar to the stochastic DVS techniques proposed by Lorch and Smith [23] and Gruian [14]. The goal is to minimize the task's energy consumption, while meeting its statistical performance requirements. To do this, we allocate some CPU time to each task as follows. If there are n concurrent tasks and each task is allocated C_i cycles per period P_i, then the scheduler allocates the i^{th} task CPU time

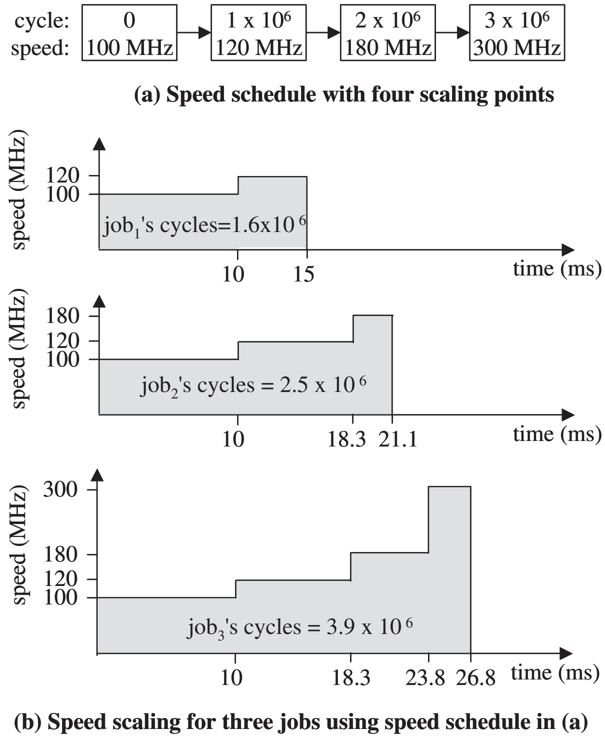

Figure 5: Example of speed schedule and corresponding speed scaling for job execution: each job starts slowly and accelerates as it progresses.

$$T_i = \frac{C_i}{\sum_{i=1}^{n} \frac{C_i}{P_i}} \tag{4}$$

every period P_i. The reason for time allocation (in addition to cycle allocation) is to guarantee that each task executes for up to its allocated cycles within its allocated time, regardless of speed changes. That is, we want to preserve the statistical performance guarantee of SRT scheduling when using DVS to save energy.

The speed schedule construction problem thus becomes, for each task, to find a speed for each of its allocated cycles, such that the total energy consumption of these allocated cycles is minimized while their total execution time is no more than the allocated time. Formally, if a cycle x executes at speed f_x, its execution time is $\frac{1}{f_x}$ and its energy consumption is proportional to f_x^2 [8]. Since a task requires cycles statistically, it uses each of its allocated cycles with a certain probability. Therefore, each allocated cycle x is executed with a certain probability; consequently, its average energy consumption is proportional to

$$(1 - F(x))f_x^2 \tag{5}$$

where $F(x)$ is the cumulative distribution function defined in Equation (1). In this way, constructing the speed schedule for a task is equivalent to:

$$\text{minimize:} \quad \sum_{x=1}^{C} (1 - F(x)) f_x^2 \tag{6}$$

$$\text{subject to:} \quad \sum_{x=1}^{C} \frac{1}{f_x} \le T \tag{7}$$

where C and T are the task's allocated cycles and allocated time per period, respectively.

To solve the above constrained optimization, we need to know the cumulative distribution, $F(x)$, for each allocated cycle. However, our histogram-based estimation provides the cumulative distribution for only the group boundaries of the histogram; i.e., we know $F(x)$ for $x \in \{b_0, b_1, ..., b_m\}$, where $b_m = C$ is the cycle group boundary that is equal to the number of allocated cycles (i.e., the ρ^{th} percentile of the task's cycle demands fall into the first m groups of its histogram, as discussed in Section 2.3).

We therefore use a piece-wise approximation technique to find the speed for the group boundaries and use a uniform speed within each group. That is, we rewrite the above constrained optimization as:

$$\text{minimize:} \quad \sum_{i=0}^{m} s_i \times (1 - F(b_i)) f_{b_i}^2 \tag{8}$$

$$\text{subject to:} \quad \sum_{i=0}^{m} s_i \times \frac{1}{f_{b_i}} \le T \tag{9}$$

where s_i is the size of the i^{th} group, i.e.,

$$s_i = \begin{cases} b_0 & : \quad i = 0 \\ b_i - b_{i-1} & : \quad 0 < i \le m \end{cases} \tag{10}$$

By Jensen's inequality [20], Equation (8) has a lower bound

$$\left(\frac{\sum_{i=0}^{m} \sqrt{s_i^3 (1 - F(b_i))}}{\sum_{i=0}^{m} \frac{s_i}{f_{b_i}}} \right)^2 \ge \left(\frac{\sum_{i=0}^{m} \sqrt{s_i^3 (1 - F(b_i))}}{T} \right)^2 \tag{11}$$

if and only if

$$\forall i \quad f_{b_i} = \frac{\sum_{j=0}^{m} \sqrt{s_j^3 (1 - F(b_j))}}{T \sqrt{s_i (1 - F(b_i))}} \tag{12}$$

Equation (12) gives the speed for each of the group boundaries, i.e., f_x for $x \in \{b_0, b_1, ..., b_m = C\}$. Therefore, we can construct the speed schedule of a task by adding a scaling point for each group boundary. That is, the speed schedule consists of $m + 1$ scaling points. Each point has cycle number b_i and speed $f_{b_i}, 0 \le i \le m$. Since the speed f_{b_i} increases as the cycle number b_i increases, this speed schedule accelerates the CPU as a job progresses.

3. IMPLEMENTATION

We have implemented a prototype of GRACE-OS. The hardware platform for our implementation is an HP Pavilion N5470 laptop with a single AMD Athlon 4 processor [2]. This processor features the `PowerNow` technology, and supports six different frequencies, {300, 500, 600, 700, 800, 1000 MHz}. Further, its frequency and voltage can be adjusted dynamically under operating system control.

The prototype software is implemented as a set of modules and patches that hook into the Linux kernel 2.4.18 (Figure 6). The entire implementation contains 716 lines of C code, including about 30 lines of modification to the standard Linux scheduler files, `sched.h` and `sched.c`. The implementation includes four major issues:

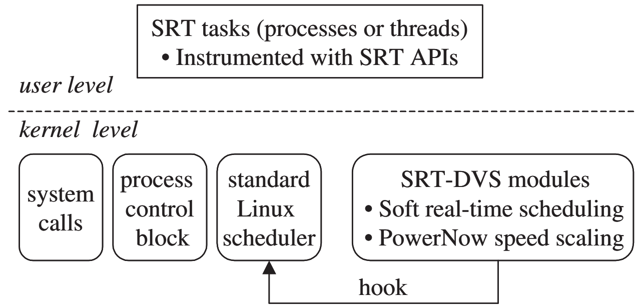

Figure 6: Software architecture of implementation.

Table 1: New system calls for SRT tasks.

API	Description
start_srt	Start real-time mode by specifying period and statistical performance requirement ρ.
exit_srt	Exit real-time mode.
finish_job	Tell the scheduler that the task finishes a job.
set_budget	Set the task's cycle budget.
set_dvspnt	Set a scaling point in the task's speed schedule.

* The last two are used to tell the kernel about the results of the demand estimation, which is moved to the user level.

1. Adding new system calls. We add three new system calls (Table 1) to support soft real-time requirements of multimedia tasks. A task uses `start_srt` to declare itself as a SRT task and to require statistical performance guarantees from the OS. It uses `finish_job` to notify the scheduler that it has finished its current job. Upon the `finish_job` call, the scheduler gets the number of cycles used by the task's current job, and may (re)calculate the task's demand distribution, cycle budget, and speed schedule if necessary.

This calculation, however, requires `double` data type computation (e.g., Equation (12)), which currently is not supported in the Linux kernel modules. To solve this problem, we move the calculation to the user level by intercepting the `finish_job` call. Before returning to the calling task, this call estimates the demand distribution in the user level and uses `set_budget` and `set_dvspnt` to tell the scheduler about the task's budget and speed schedule, respectively. This interception is enabled only for demand estimation, and is disabled otherwise to reduce overhead.

Note that a scaling point's speed, calculated via Equation (12), may not overlap with the speeds supported by the Athlon processor. Hence, we need to convert the calculated speed to the supported one. To do this, we round the speed of each scaling point to the upper bound in the supported speeds, and combine scaling points that have the same upper bound. As a result, a task's speed schedule consists of at most six points, each for one of the supported speeds. This conversion is specific to the processor specification; so there may be different number of points for different processors.

We once considered an alternative approach, often used in simulations, that approximates a calculated speed with two bounding supported speeds [14, 17, 23]. This approach divides cycles that need to be executed at the calculated speed into two parts, one for the lower bound and the other

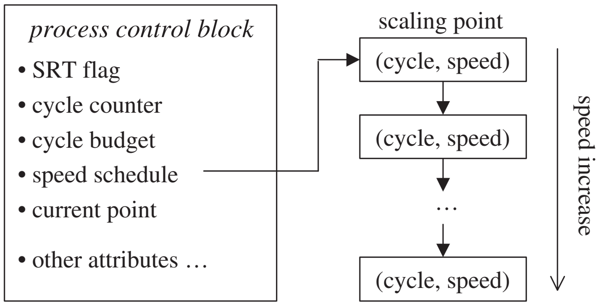

Figure 7: Enhanced process control block.

for the upper bound. This approach provides more accurate approximation. It, however, requires a very fine granularity of speed scaling (about tens of microseconds) due to the cycle division, and may potentially result in large overhead when used in real implementations.

2. Modifying the process control block. We add five new attributes into the process control block (i.e., the `task_struct`), as shown in Figure 7. The first new attribute `SRT flag` indicates if the task is a SRT task. The other four attributes apply to SRT tasks only. `Cycle counter` is used for profiling (Section 2.2). `Cycle budget` stores the number of allocated cycles (Section 2.3). `Speed schedule` is a list of speed scaling points, which define how to accelerate the CPU for a job execution. `Current point` specifies the current execution speed for the task (Section 2.4).

3. Adding SRT and DVS modules. We add two new modules in the kernel, one for soft real-time scheduling and the other for speed scaling. The `PowerNow` module is responsible for setting speed by writing the frequency and voltage to a special register `FidVidCtl` [2, 28, 36]. This module provides a simple, clean interface for speed setting, and is separated from the DVS decision maker (the scheduler in our case). In doing so, we improve the flexibility and reusability of our implementation: we can apply the stochastic scheduling and DVS to other processors by replacing only the speed setting module (e.g., using Transmeta's LongRun [10]).

The SRT scheduling module is hooked into the standard Linux scheduler, rather than replacing the latter. This is similar to hierarchical real-time scheduling [13, 28]. There are two reasons for this: (1) to support the coexistence of real-time and best-effort applications, and (2) to minimize the modification to the OS, thus improving the usability of our implementation. We patch the kernel with the UTIME package [21] and add a periodic, 500 μs resolution UTIME timer into the kernel. The SRT scheduler is attached as the call-back function of the timer and hence is invoked every 500 μs. Like soft-timers [4], UTIME timers achieve high resolution with low overhead by running the hardware timer as an aperiodic device [21]. We use this high resolution timer, rather than the standard Linux kernel timer with 10 ms resolution, because the latter's resolution is too coarse for our SRT scheduling and intra-job DVS.

When the UTIME timer expires, the SRT scheduler is invoked to perform real-time scheduling as follows: (1) it checks the cycle budget of the current task. If the budget is exhausted, it sets the current task's scheduling policy to best-effort mode for overrun protection. (2) It checks all SRT tasks. If a task begins a new period, it replenishes the task's cycle budget and puts the task back to real-time mode. (3) It sets the priority of all real-time tasks based on their deadline— the earlier the deadline, the higher the priority. (4) Finally, it invokes the standard Linux scheduler, which in turn dispatches the real-time task with the earliest deadline.

Table 2: Experimental multimedia applications.

Application	Type	Jobs	Period (ms)
mpgplay	MPEG video decoder	7691	30
madplay	MP3 audio decoder	6118	30
tmn	H263 video encoder	1000	400
tmndec	H263 video decoder	1000	30
toast	GSM speech encoder	11631	25

4. Modifying the standard Linux scheduler. We modify the standard Linux scheduler to add profiling, cycle charging, and DVS. When the `schedule`() function is invoked, if there is no context switch (i.e., the current task is dispatched again), the Linux scheduler may accelerate the speed for the current task based on its speed schedule. If a context switch happens, the Linux scheduler does some housekeeping for the *switched-out* task by (1) increasing its cycle counter by the number of cycles elapsed since its last switch-in, (2) decreasing its cycle budget by the same amount, and (3) advancing its current scaling point if its cycle counter reaches the cycle number of the next scaling point. The Linux scheduler then adjusts the speed for the *switched-in* task based on its current scaling point. This task will execute at the new speed after the context switch.

4. EXPERIMENTAL EVALUATION

Our experiments are performed on the HP Pavilion N5470 laptop with 256MB RAM. The operating system is Red Hat Linux 7.2 with a modified version of Linux kernel 2.4.18, as discussed in Section 3. The experimental applications are codecs for video, audio, and speech. Table 2 summarizes these applications and their inputs. We have also experimented with other inputs for each codec, but do not show results here due to space limitations.

Unless specified otherwise, we repeat each of the following experiments at least eight times, and measure the relevant metrics (such as scheduling cost and CPU energy consumption) in each run. For each metric, we discard the largest and smallest value measured in the multiple runs, and report the average of the remaining ones.

4.1 Overhead

In our first experiments, we evaluate GRACE-OS's overhead by measuring the cost for demand estimation, SRT scheduling, and DVS. We measure cost in cycles, rather than time, since the elapsed time for an operation (e.g., an invocation of SRT scheduling) depends on the speed, while the number of consumed cycles does not change substantially with the speed. We get the number of cycles by reading the timestamp register (a CPU cycle counter) in the kernel, and provide an API to get the cycle count in the user level.

First, we evaluate the cost for the new system calls in Table 1. To do this, we run the *mpgplay* task, and measure the elapsed cycles for each new system call in the *mpgplay* program. Table 3 shows that these system calls take around 1000 cycles (about 3-6 μs at speed 300 MHz). This over-

Table 3: Cost of new system calls (in cycles).

start_srt	exit_srt	finish_job	set_budget	set_dvspnt
1282	1136	925	918	906

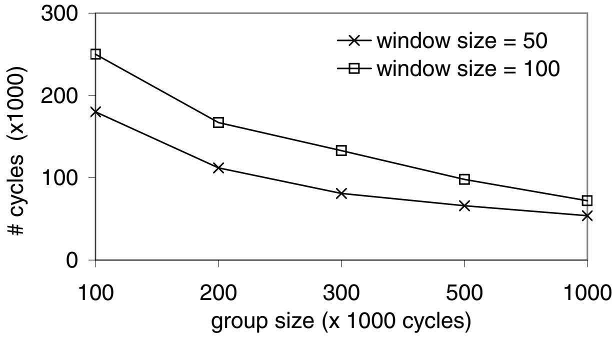

Figure 8: Cost of demand estimation.

head is low and negligible relative to multimedia execution, since our studied codecs consume about 2×10^5 to 2.5×10^8 cycles for a frame processing (i.e., these new system calls cost about 0.0004% to 0.5% of job cycles). Note that the cost for `finish_job` does not include cycles for calculating the demand distribution of the calling task (recall that this calculation is moved to the user level by intercepting `finish_job` when the demand distribution changes).

Second, we measure the cost for demand profiling and estimation. The profiling cost primarily results from the access to the timestamp register (i.e., reading the current number of cycles). It is about 26-38 cycles on the HP laptop. To evaluate the estimation cost, we run the *mpgplay* codec and measure the number of cycles for building histogram and calculating cycle budget and speed schedule. The results (Figure 8) show that the estimation cost is dependent on the size of the profiling window and histogram groups. The cost is in hundreds of thousands of cycles and hence is quite large. It is about 0.1% to 100% of the number of cycles consumed by jobs of our studied codecs (2×10^5 to 2.5×10^8 cycles per job). This means that the online demand estimation cannot happen frequently; otherwise, it will incur unacceptably large overhead. In Section 4.2, we will show that the demand distribution of our studied multimedia tasks is relatively stable; consequently, GRACE-OS only needs to estimate their demand distribution infrequently.

Third, we measure the cost for SRT scheduling. We run one to ten copies of the *toast* codec and measure the number of cycles elapsed during each SRT scheduling. We choose *toast*, since it presents low CPU demand (about 2×10^5 cycles per job) and hence multiple copies can run concurrently without violating the EDF schedulability (i.e., the total CPU utilization is no more than 100%). For each run, we sample the scheduling cost 10,000 times in the kernel. Figure 9 plots the results with 95% confidence intervals. The scheduling cost depends on the number of multimedia tasks since the SRT scheduler needs to check the status of each task (e.g., whether it begins a new period). When tasks do not begin a new period, the cost primarily results from budget charging for the currently running task and status checking for all tasks. It is below 300 cycles for up to ten tasks. With a scheduling granularity of 500 μs, the relative scheduling cost is below 0.06%-0.2%, depending on the CPU speed[1]. When a task begins a new period, the SRT scheduler needs to replenish its cycle budget and change its scheduling parameters. As a result, the scheduling cost becomes larger, but is still small relative to multimedia execution (e.g., no more than 0.4% of job cycles).

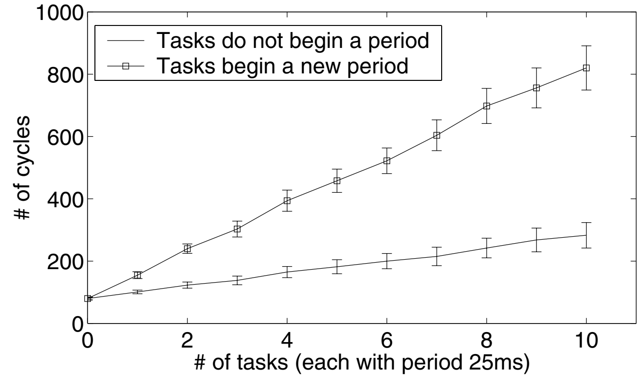

Figure 9: Cost of soft real-time scheduling (with 95% confidence intervals).

Table 4: Cost of speed scaling (in cycles).

		to frequency (MHz)					
		300	500	600	700	800	1000
from frequency (MHz)	300		10391	11332	12380	13490	15454
	500	8232		11510	12580	13700	15697
	600	8345	10637		12667	13779	15894
	700	8474	10721	11634		13851	16015
	800	8652	10837	11706	12813		16152
	1000	9133	11080	11887	12960	14062	

Table 5: Average # of speed changes per job.

single run					concurrent run
mpegplay	madplay	tmn	tmndec	toast	
2.14	0	1.28	0. 94	0	0.8

Finally, we measure the cost for speed scaling. To do this, we adjust the processor from one speed to another one, and measure the number of cycles for each change. The results (Table 4) show that the CPU can change speed within 8,000-16,000 cycles (about 10-50 μs). It means that the speed scaling does not incur large overhead, but should be invoked only infrequently. This is one of the reasons that our implementation uses a 500 μs timer to trigger SRT scheduling and DVS. Note that with the advances in microprocessor design, the speed change overhead will become smaller; e.g., the lpARM processor can change speed in about 1250 cycles and continue operation during the change [27].

We notice that the stochastic intra-job DVS may *potentially* result in frequent speed changes due to the context switches between different tasks and the acceleration of each task's job execution. In practice, however, GRACE-OS does

[1]For every 500 μs, the processor provides 500μs $\times$ 300MHz $= 1.5 \times 10^5$ cycles at speed 300 MHz, and 500μs $\times$ 1000MHz $= 5 \times 10^5$ cycles at 1000MHz.

not change speed frequently for two reasons: (1) jobs often complete early before accelerating, and (2) there are only six speeds available, which means that the CPU speed may remain the same after a context switch. To validate this, we run each of the codecs one at a time and run all of them (except *tmn*) concurrently. During each run, we measure the average number of speed changes per job. The results (Table 5) confirm that the stochastic intra-job DVS does not change speed frequently. In particular, there is almost no speed change during the single run of the *madplay* and *toast* codecs, since the lowest speed, 300 MHz, is sufficient to meet their CPU demands.

4.2 Stability of demand distribution

The stochastic scheduling and speed scaling both depend on the probability distribution of task demands. If a task's demand distribution is stable, the scheduler can estimate it with a small profiling window; otherwise, the scheduler can either estimate the demand distribution with a large profiling window or update it when it changes. Our next experiment examines the stability of the demand distribution. To do this, we profile cycle usage of each codec during various time intervals of its execution (e.g., during the first 50 and 100 jobs), estimate the demand distribution from the cycle usage, and compare the demand distributions of different time intervals. Note that the cycle usage and demand distribution of each codec are both dependent on its inputs. Although we report the results for only the inputs in Table 2, we have also experimented with other inputs for each codec and found similar results.

Figure 10-(a) depicts the cycle usage of the *mpgplay* application for the whole video clip *lovebook.mpg* with frame size 320×240 pixels and 7691 frames. Figure 10-(b) plots its demand distribution for decoding different parts of the video (e.g., the first 50 and 100 frames). The figure shows two important characteristics of the *mpgplay*'s CPU usage. First, its instantaneous cycle demands are bursty and most jobs do not need the worst case cycles; e.g., for the first 100 jobs, the worst-case demand is 9.9×10^6 cycles, but 99% of jobs require less than 9.4×10^6 cycles. This indicates that compared to worst-case-based allocation and speed scaling, stochastic allocation and scaling can improve CPU and energy utilization. For example, the scheduler can improve CPU utilization by 5% when delivering the *mpgplay* codec 99% (as opposed to 100%) deadline guarantees.

Second, *mpgplay*'s instantaneous cycle demands change greatly (up to a factor of three), while its demand distribution is much more stable. For example, the cumulative probability curves for the first 50 jobs, the first 100 jobs, and all 7691 jobs are almost the same. This stability implies that GRACE-OS can perform stochastic scheduling and DVS for *mpgplay* based on a small part of its cycle usage history (e.g., cycle usage of the first 50 jobs).

We next repeat the experiment for other codecs. Figure 11-(a) to (d) plot the demand distribution of the *toast*, *madplay*, *tmn*, and *tmndec* codecs, respectively. The results show that *toast* and *madplay* both present low CPU demands; e.g., the 95^{th} percentile of their jobs need less than 2.3×10^5 and 8.6×10^5 cycles, respectively. Further, the probability distribution of their cycle demands is stable; e.g., the cumulative probability curve for the first 50 jobs is almost the same as that for all jobs.

On the other hand, *tmn* and *tmndec* present high CPU demands; e.g., the 50^{th} percentile of *tmn*'s jobs need more than 2.5×10^8 cycles. Further, their demand distribution changes over time (i.e., for different parts of the input video). The reason is that their input videos have several scene changes and hence require different amount of CPU cycles. Such changes indicate that GRACE-OS needs to dynamically update the demand distribution for *tmn* and *tmndec*. However, the demand distribution of *tmn* and *tmndec* changes in a slow and smooth manner (e.g., there is little variation between the first 50 and 100 jobs). This implies that GRACE-OS only needs to update their demand distribution infrequently (e.g., for every 100 jobs).

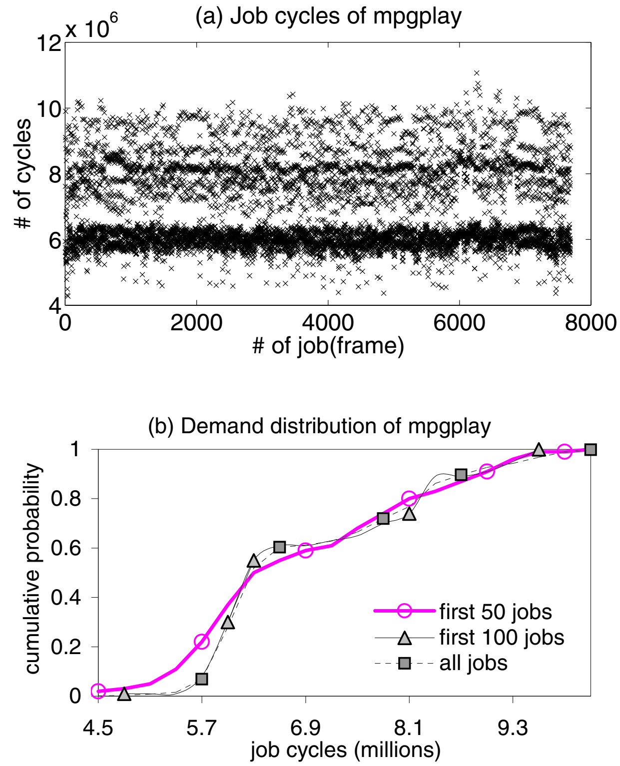

Figure 10: Cycle usage and estimated demand distribution of *mpgplay*: its instantaneous cycle demands change greatly, while its demand distribution is much more stable.

4.3 Efficiency of GRACE-OS

We now evaluate GRACE-OS's efficiency for QoS support and energy saving by comparing it with other schemes that perform allocation and/or DVS deterministically:

- *Worst-uniform* (wrsUni). It allocates cycles based on each task's worst-case demand, and sets a uniform speed that meets the aggregate allocation of the current task-set, i.e., $\sum_{i=1}^{n} \frac{C_i^{ws}}{P_i}$, where there are n tasks and each has period P_i and worst-case demand C_i^{ws}.
- *Worst-reclaim* (wrsRec). It is the same as *wrsUni* except that it reclaims the unused cycles when a task completes a job early. It sets CPU speed to $\sum_{i=1}^{n} \frac{C_i^{act}}{P_i}$,

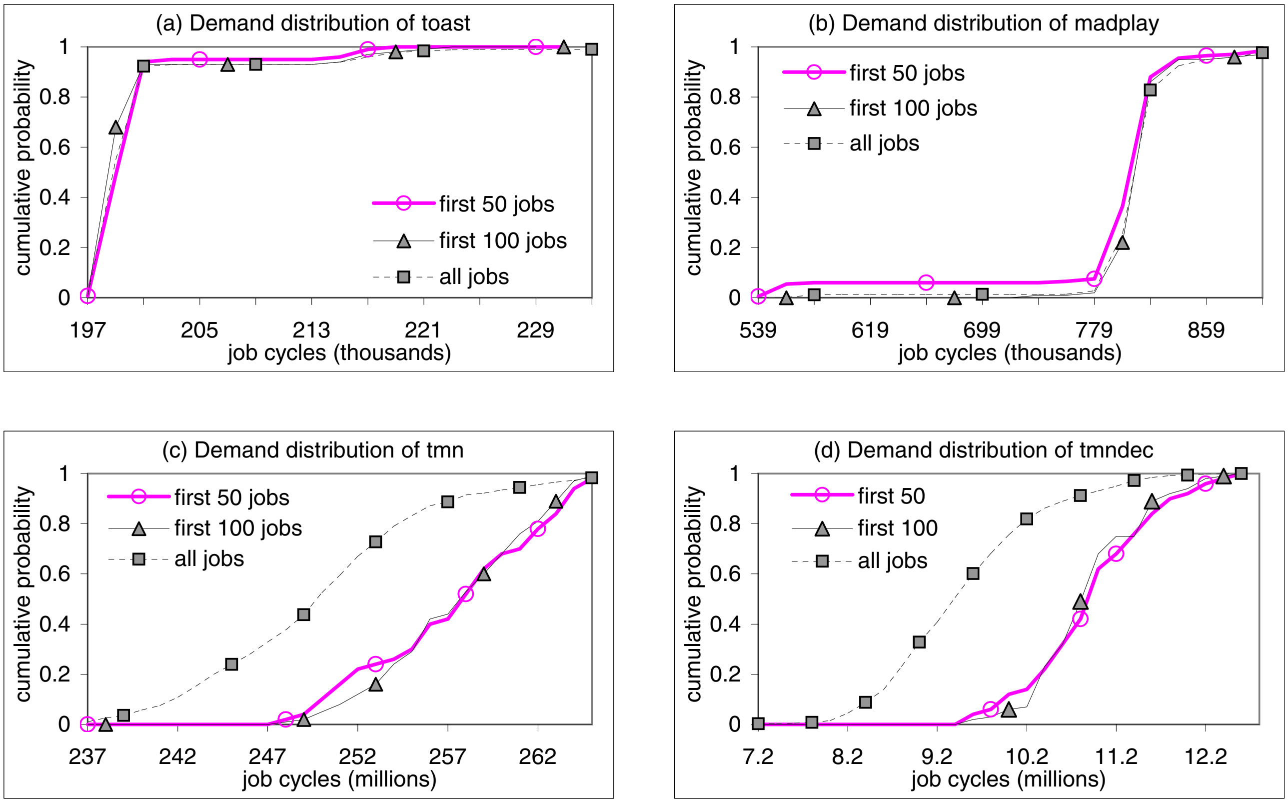

Figure 11: Stability of demand distribution of other codecs: *toast* and *madplay*'s are stable, and *tmn* and *tmndec*'s change slowly and smoothly.

where C_i^{act} equals to the worst-case demand upon a job release and to the number of actually consumed cycles upon a job completion. *WrsRec* represents DVS techniques that first start jobs at high speed and then decelerate upon early completion [5, 28].

- *Worst-stochastic* (wrsSto). It allocates cycles based on each task's worst-case demand, and performs stochastic DVS to adjust the speed for each task based on the task's demand distribution.

- *Stochastic-uniform* (stoUni). It allocates cycles based on each task's statistical demand, and sets a uniform speed that meets the aggregate allocation for the current task-set, i.e., $\sum_{i=1}^{n} \frac{C_i^{st}}{P_i}$, where there are n concurrent tasks and each has period P_i and statistical cycle demand C_i^{st}.

- *Stochastic-reclaim* (stoRec). It is the same as *wrsRec* except that it allocates cycles based on each task's statistical demand and sets CPU speed correspondingly. That is, the parameter C_i^{act} is set to the statistical demand (as opposed to the worst-case demand in *wrsRec*) when the i^{th} task releases a job.

Like GRACE-OS, all the above schemes also use the EDF-based soft real-time scheduling with overrun protection. We run the codecs in Table 2 under each of the above schemes and measure two metrics: *deadline miss ratio* and *CPU energy consumption*. Since we currently do not have power meters like PowerScope [11], we cannot measure the actual energy consumption. Instead, we evaluate CPU energy by measuring the fraction of CPU idle time and the distribution of busy time at each speed. This is similar to the measurement in Vertigo [10]. The correlation between speed levels and energy consumption has been studied in the literature [11, 28]. Intuitively, the CPU saves energy, if it spends more time in lower speeds and has less idle time.

To simplify the comparison among different schemes, we also calculate *normalized energy* as follows. We assume the CPU power is proportional to the cube of the speed, and then normalize the power at the highest speed as one unit; i.e., the CPU power at speed f is $p(f) = \left(\frac{f}{f_{max}}\right)^3$, where f_{max} is the maximum speed. This assumption holds, since the speed is proportional to the voltage (i.e., $f \propto V$) and the dynamic power, which dominates CPU energy consumption, is proportional to the speed and the square of the voltage (i.e., $p(f) \propto fV^2$) [8]. Based on this power normalization, the CPU consumes normalized energy of

$$\int_0^T p(f(t))\, dt = \int_0^T \left(\frac{f(t)}{f_{max}}\right)^3 dt \tag{13}$$

where T is the total execution time and $f(t)$ is the speed at time $t, 0 \le t \le T$. Such a normalized evaluation is also commonly used in previous work [5, 14, 23, 26, 28, 34].

Unless specified otherwise, we set each codec's statistical performance requirement ρ to 0.95 (i.e., the codec task needs to meet 95% of deadlines) and estimate its demand distribution from the cycle usage of its first 100 jobs. The

Table 6: CPU speed distribution for mpgplay.

	percent of CPU busy time at each speed (MHz)						percent of idle time
	300	500	600	700	800	1000	
wrsUni		100%					54.2%
wrsRec	51.6%	48.4%					54.2%
wrsSto	92.3%	7.1%	0.4%	0.2%			29.1%
stoUni		100%					54.2%
stoRec	51.6%	48.4%					54.2%
GRACE-OS	92.3%	7.1%	0.4%	0.2%			29.1%

Table 7: Energy and deadline miss ratio for mpgplay.

	wrsUni	wrsUni	wrsRec	wrsSto	stoUni	GRACE-OS
normalized energy	28.8	17.2	8.2	28.8	17.2	8.2
deadline miss ratio	0.4%	0.3%	0.5%	0.5%	0.6%	0.4%

first 100 jobs of each codec task are executed in best-effort mode and at the highest CPU speed, 1000 MHz. That is, GRACE-OS allocates cycles and changes the speed for each task after the task has finished 100 jobs. We do not count the missed deadlines of the first 100 jobs since they do not have performance guarantees.

Run a single application. We first run the *mpgplay* codec without CPU competition from other applications and measure the speed distribution, normalized energy, and deadline miss ratio (Tables 6 and 7). The results show two important observations. First, GRACE-OS delivers statistical performance guarantees by bounding the deadline miss ratio under $1-\rho = 5\%$. Actually, the deadline miss ratio for all schemes is approximately 0 (the measured values 0.3%-0.6% are primarily due to the uncertainty resulted from several low-level mechanisms such as caching and interrupts). The reason is that when *mpgplay* overruns, it utilizes unallocated cycles, which exist since the CPU has discrete speed options and often runs faster than required.

Second, GRACE-OS spends most CPU busy time (92.3%) in the lowest speed, 300 MHz, and also has less idle time (29.1%). This implies that GRACE-OS slows down the CPU and reduces idle slack. Consequently, it results in a 53.4% to 71.6% reduction of normalized energy. This benefit of energy saving primarily results from stochastic DVS. Stochastic allocation does not contribute to energy saving; e.g., GRACE-OS and *wrsSto* have the same normalized energy. We expect that the reason for this result is the existence of discrete speed options. To verify this, we take a look at *mpgplay*'s speed schedule, and find that it is the same in GRACE-OS and *wrsSto*. As a result, GRACE-OS and *wrsSto* have the same speed scaling, and hence consume the same energy, during the *mpgplay* run.

We then run each of the other codecs one at a time, and measure the above metrics. We find that similar to the *mpgplay* run, deadline miss ratio is negligible for other codecs. Therefore, we focus on energy evaluation. Table 8 shows the speed distribution for *tmn*, *tmndec*, *toast*, and *madplay*. For each DVS method (i.e., uniform, reclaim, and stochastic), we plot worst-case and statistical allocation together (e.g., *wrsSto*/GRACE-OS), since they have the same speed

Table 8: Speed distribution for other codecs.

(a) tmn

	percent of CPU busy time at each speed (MHz)				percent of idle time
	300	600	700	800	
wrsUni/stoUni			100%		10.8%
wrsRec/stoRec		0.9%	99.1%		10.8%
wrsSto/GRACE-OS	11%		88.8%	0.2%	10.6%

(b) tmndec

	percent of CPU busy time at each speed (MHz)			percent of idle time
	300	500	600 - 1000	
wrsUni/stoUni		100%		12.5%
wrsRec/stoRec	0.4%	99.6%		12.4%
wrsSto/GRACE-OS	35.3%	64.7%		9%

(c) toast and madplay

	percent of CPU busy time at each speed (MHz)		percent of CPU idle time	
	300	500 - 1000	toast	madplay
wrsUni/stoUni	100%		97.3%	91%
wrsRec/stoRec	100%		97.3%	91%
wrsSto/GRACE-OS	100%		97.3%	91%

schedule and speed scaling when executing a single task. We notice immediately that for *tmn* and *tmndec*, GRACE-OS reduces the CPU idle time and spends more busy time at lower speeds. This implies that GRACE-OS consumes less energy during the single run of both *tmn* and *tmndec*. The normalized energy is summarized in Figure 12.

On the other hand, for *toast* and *madplay*, all schemes spend their time at the lowest speed, 300 MHz, and have the same fraction of idle time. This indicates that compared to other schemes, GRACE-OS does not save energy for *toast* and *madplay* (Figure 12). The reason is that these two applications present a low CPU demand. Specifically, the stochastic allocation of *toast* and *madplay* is 2.2×10^5 and 8.6×10^5 per period 25 and 30 ms, respectively; i.e., they demand only 8.8 and 28.6 million cycles per second (MHz), respectively. Their low CPU demands mean that their speed schedule has only a single scaling point, associated with the lowest speed, 300 MHz. Consequently, the processor always runs at the lowest speed, thereby resulting in the same speed distribution and energy consumption for all schemes.

We therefore conclude that for a single low-demand application, the effectiveness of GRACE-OS's energy saving is limited by the available discrete speeds. We expect that GRACE-OS can save energy for a single low-demand application if there are more CPU speeds available. This expectation can be validated via trace-based simulation; we leave it for future work.

Run multiple applications concurrently. We next run all codecs concurrently (except *tmn* that demands too many cycles for concurrent execution). Tables 9 and 10 show the

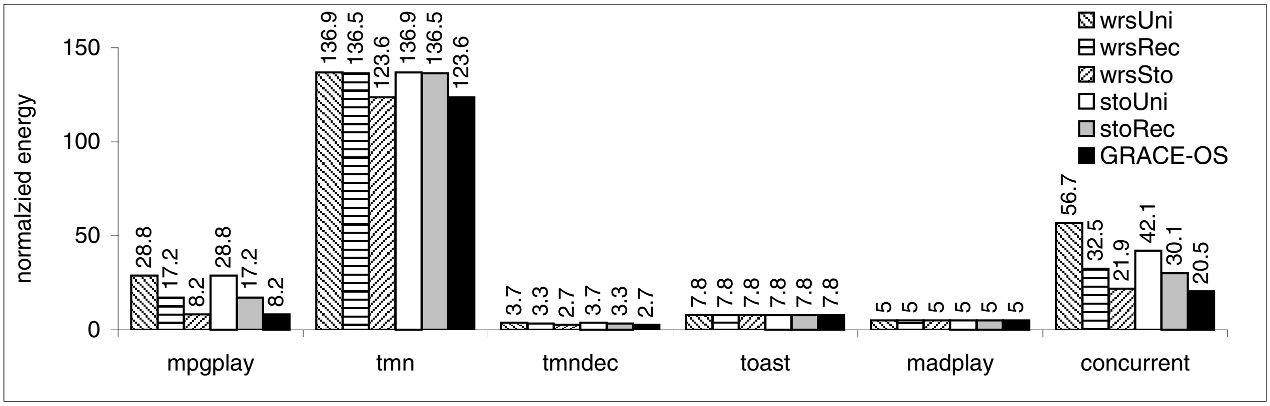

Figure 12: Summary of normalized energy for single and concurrent runs.

Table 9: Speed distribution for concurrent run.

	percent of CPU busy time at each speed (MHz)						percent of idle time
	300	500	600	700	800	1000	
wrsUni	20.6%	72.5%				6.9%	57.4%
wrsRec	52.7%	40.4%	1.8%	1.8%	3%	0.3%	54.3%
wrsSto	64.4%	28.8%	3%	0.7%	3%	0.1%	51.2%
stoUni	20.7%	72.4%			6.9%		56.4%
stoRec	53.2%	40%	1.5%	3.7%	1.6%		53.7%
GRACE-OS	83.8%	9.1%	2.6%	4%	0.3%	0.2%	37.6%

Table 10: Energy and deadline miss ratio for concurrent run.

	wrsUni	wrsUni	wrsRec	wrsSto	stoUni	GRACE-OS
normalized energy	56.7	32.5	21.9	42.1	30.1	20.5
deadline miss ratio	0.3%	0.5%	0.4%	4.8%	5.2%	4.9%

results. We notice that the deadline miss ratio of stochastic allocation schemes is higher than in the single run case (compared to Table 7). The reason is that multiple tasks may compete for the CPU during overruns (recall that an overrun task runs in best-effort mode by utilizing unused cycles, as discussed in Section 2.3). However, GRACE-OS bounds deadline miss ratio under application statistical performance requirements; i.e., the deadline miss ratio 4.9% is below $1 - \rho = 5\%$.

Another obvious result is that GRACE-OS spends more CPU busy time in lower-power speeds and has less idle time than *wrsSto*; e.g., the fraction of CPU busy time at speed 300 MHz of GRACE-OS and *wrsSto* is 83.8% and 64.4%, respectively. This implies that stochastic allocation and stochastic DVS both contribute to energy saving (Table 10). This is different from the single run cases, where GRACE-OS and *wrsSto* consume the same energy (see Tables 6-8). The reason behind this difference is that the integration of stochastic scheduling and DVS yields statistical multiplexing gains for concurrent run; e.g., since the EDF-based scheduling is work-conserving, it enables a task to take advantage of residual budgets from other tasks.

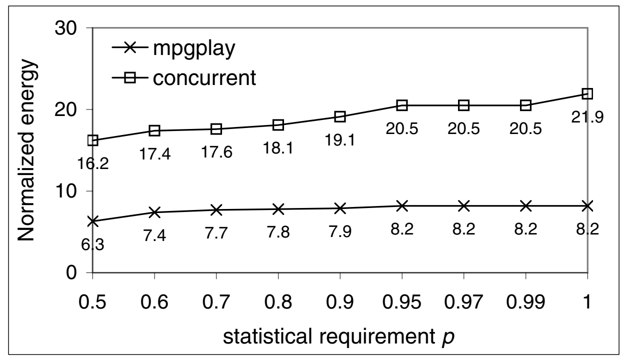

Figure 13: Impact of ρ on normalized energy.

4.4 Impact of ρ and mixed workload

In all the above experiments, we set each task's statistical performance requirement ρ to 0.95, and do not run any best-effort application in the background. Now, we examine the impact of the parameter ρ and mixed workload on GRACE-OS's energy saving.

First, we repeat the above *mpgplay* and concurrent runs, but change the parameter ρ from 0.5 (average-case requirement) to 1.0 (worst-case requirement). The results (Figure 13) show that normalized energy increases as ρ increases from 0.5 to 0.95. The reason is that a higher performance requirement means more CPU allocation, which in turn results higher execution speed.

When ρ changes from 0.95 to 1.0, however, energy consumption is almost the same (the concurrent run consumes more energy with $\rho = 1.0$ as explained above). The reason is that during this interval of ρ, (1) the cycle budget of codecs does not change significantly; e.g., *mpgplay*'s budget changes from 9.3 to 10 millions as ρ increases from 0.95 to 1.0, and (2) the speed schedule of codecs has little difference after rounding up to the available speeds. It implies that the discrete speed options limit GRACE-OS's energy saving.

Next, we analyze the impact of best-effort applications on GRACE-OS's energy saving. To do this, we repeat the above *mpgplay* and concurrent runs, but execute *math*, a CPU-intensive program, in the background. *Math* runs when multimedia applications do not use the CPU, and its execution speed is the same as that of the previously-executed

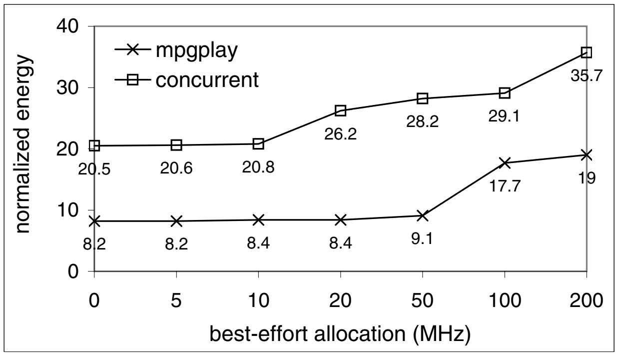

Figure 14: Impact of mixed workload.

multimedia application (i.e., the scheduler does not adjust speed for *math*.)

To protect *math* from starvation, we allocate it some millions of cycles per second (MHz). Figure 14 plots normalized energy for the *mpgplay* and concurrent runs by changing the best-effort allocation from 0 to 200 MHz. The results indicate that the extra best-effort allocation increases energy consumption. The reason is that the extra allocation increases total CPU demand. Consequently, each multimedia application is allocated less time and needs to run faster (see Equation (4) and (12)). Note that for the *mpgplay* run, the extra 5 MHz allocation does not affect the normalized energy, since *mpgplay*'s speed schedule does not change (due to the discrete CPU speed options).

4.5 Summary

Overall, our experimental results show that GRACE-OS provides significant benefits for QoS provisioning and energy saving and incurs acceptable overhead. GRACE-OS meets almost all deadlines in a lightly loaded environment (without CPU competition) and bounds deadline miss ratio under application statistical requirements in a heavily loaded environment. Compared to other systems with deterministic scheduling and DVS, GRACE-OS spends more CPU time in lower-power speeds, thus saving energy. When executing a single high-demand application, stochastic DVS primarily contributes to energy saving, reducing normalized energy by 10%-72%. When executing multiple applications concurrently, stochastic scheduling and DVS both contribute to energy saving, reducing normalized energy by 7%-64%.

We also find that GRACE-OS's efficiency is limited by the discrete speed options, especially when executing a single low-demand application and when changing the statistical performance requirement ρ from 0.95 to 0.99. This is similar to Pillai and Shin's finding [28], which shows through simulations that the speed availability profoundly affects energy saving of real-time DVS algorithms.

5. RELATED WORK

Soft real-time scheduling for QoS support. Recently, a number of soft real-time (SRT) scheduling mechanisms has been proposed to support QoS requirements of multimedia applications. These approaches typically integrate predictable CPU allocation (such as proportional sharing [7, 9, 13, 18, 25] and reservation [19, 29]) and real-time scheduling algorithms (such as EDF and rate monotonic [22]). GRACE-OS distinguishes itself from the above SRT approaches for two reasons: (1) GRACE-OS derives each task's CPU demands via online profiling and estimation, while others typically assume that the CPU demands are known in advance, e.g., via off-line profiling; and (2) GRACE-OS uses SRT scheduling, integrated with DVS in a variable speed context, while others implicitly assume a constant CPU speed. The variable speed context brings challenges to SRT scheduling, e.g., how to enforce budget (share or reservation) when the underlying speed changes.

Several groups have also studied stochastic SRT scheduling for statistical performance guarantees. Gardner [12] proposed a stochastic time demand analysis technique to compute the bound of deadline miss ratio for fixed-priority systems. This computation is based on runtime execution by analyzing the time demands of a task and other tasks with higher priority. In contrast, GRACE-OS aims for dynamic-priority (EDF-based) systems, and delivers statistical guarantees by allocating cycle budget based on the demand distribution of each individual task. Hamann et al. [16] and Wang et al. [33] proposed a scheduling technique to provide statistical guarantees for imprecise computations and differentiated services, respectively. Both approaches assume a predefined stochastic distribution of resource demands. In contrast, GRACE-OS obtains the demand distribution via online profiling and estimation.

More recently, Urgaonkar et al. [32] proposed automatic profiling and overbooking techniques to provide statistical guarantees. Their approach is similar to our stochastic allocation. However, there are two differences: (1) their approach profiles resource busy intervals in an isolated environment using on-off traces, while GRACE-OS profiles the number of cycles consumed by each task at runtime. (2) The overbooking technique aims to support more services in shared hosting platforms, while GRACE-OS aims to save energy in mobile devices.

Dynamic voltage scaling for energy saving. DVS is commonly used to save CPU energy by adjusting the speed based on application workload. Recently, DVS has been investigated in two main areas, general-purpose systems (GP-DVS) and real-time systems (RT-DVS). GP-DVS algorithms heuristically predict the workload based on average CPU utilization [15, 26, 34]. Although they save energy without degrading performance of best-effort applications, they are unsuitable for multimedia applications due to the timing constraint and demand variations of multimedia applications. For example, Grunwald et al. [15] concluded that no heuristic algorithm they examined saves energy without affecting multimedia application performance.

RT-DVS algorithms, often integrated with CPU scheduling, derive workload from worst-case CPU demands of real-time applications [27, 28]. That is, they set CPU speed based on the assumption that applications require worst-case CPU resources. Since an application's demand is not always the worst-case, some reclamation techniques have been proposed to reclaim the unused cycles to save more energy [5, 28]. These reclamation techniques first run CPU fast (assuming the worst-case demand) and then decelerate when a job finishes early.

Stochastic DVS is an alternative approach to handling runtime demand variations [14, 23]. It starts a job slowly and then accelerates as the job progresses. Gruian [14] used stochastic DVS for hard real-time systems, while Lorch and

Smith [23] proposed a technique, called PACE, to improve GP-DVS algorithms. Their basic idea is similar to that in GRACE-OS— finding a speed for each cycle based on the demand distribution of applications.

GRACE-OS differs from the above two stochastic DVS techniques for three reasons. First, GRACE-OS obtains the demand distribution via online profiling and estimation, while the other two either assume a given distribution function or estimate it off-line. Second, GRACE-OS supports multiple tasks by integrating SRT scheduling and DVS. In contrast, PACE supports only a single task and treats concurrent tasks as a joint workload without isolation among them. Although Gruian's approach [14] claims to support concurrent tasks for fixed-priority systems, it is not clear on how it decides the time allocation for multiple tasks. Finally and more importantly, the other two present simulations only, while GRACE-OS implements the stochastic DVS. More recently, Lorch and Smith implemented the PACE algorithm in Windows 2000 [24]. Their implementation, however, does not support soft real-time scheduling.

Recently, some groups proposed a *per-job* stochastic DVS technique [30, 31], which changes speed for each job of a task based on a stochastic model (e.g., Markov process) of the task's CPU demands. This per-job DVS changes speed only when starting a job, while GRACE-OS changes speed within a job execution.

Finally, GRACE-OS is built on our previous work [35], which shows the benefits of integration of soft real-time scheduling and DVS via simulation.

6. CONCLUSION

This paper presents the design, implementation, and evaluation of GRACE-OS, an energy-efficient soft real-time CPU scheduler. GRACE-OS explores an observation that multimedia applications present dynamic cycle demands, and the probability distribution of their cycle demands is relatively stable. This observation provides opportunity both for saving energy and for delivering soft guarantees to multimedia applications. To realize this opportunity, GRACE-OS statistically allocates cycles to individual applications, and executes their allocated cycles at different speeds. It makes such stochastic decisions based on the demand distribution of multimedia applications.

Our experimental results, based on the real implementation, show that GRACE-OS provides significant deadline guarantees and energy saving with acceptable overhead. It bounds deadline miss ratio under application-specific requirements, and saves energy by 7% to 72%. This energy saving primarily results from the stochastic DVS, especially when executing a single application.

Although our current study on GRACE-OS yields strong results, lessons learned motivate the following future work:

1. **Limitation of energy saving due to few speed options in the HP Pavilion laptop**. We expect that GRACE-OS will result in more benefits, if there are more speeds available and frequent speed changes incur low overhead. In general, such expectation can be examined in three ways: (1) using a trace-based simulator to experiment with an ideal processor that supports continuous DVS, (2) converting an optimal speed to two available speeds [14, 17, 23], and (3) applying GRACE-OS to processors that support continuous DVS (e.g., lpARM [27]). We plan to examine the first two approaches.
2. **Limitation of experiments due to lack of diversity of application classes**. GRACE-OS is targeted to periodic multimedia applications whose demand distribution is stable. We expect that GRACE-OS can also benefit best-effort applications and other soft real-time applications such as hosting servers with highly bursty demands [32]. Investigating GRACE-OS's impact on these applications is a part of our future work. We also plan to analyze its impact on the perceptual quality of multimedia and interactive applications.
3. **Limitation of energy evaluation due to lack of actual energy measurement**. GRACE-OS uses a normalized approach for energy evaluation with three assumptions: (1) the CPU power is proportional to the cube of the speed, (2) stochastic scheduling and DVS do not incur energy overhead, and (3) DVS has no impact on energy consumption of other resource components such as memory. In practice, however, these assumptions may be too strong. We therefore plan to purchase and use power meters to measure the actual energy consumption, and analyze GRACE-OS's practical impact, e.g., by comparing the expected and measured energy saving.

7. ACKNOWLEDGMENTS

We would like to thank the GRACE members, especially Professor Sarita Adve, for their informative discussions on energy saving, anonymous reviewers and the shepherd, Professor Kevin Jeffay, for their constructive feedback, Normand Paquin and William Conner for proofreading the final paper, and Padmanabhan Pillai for providing the code to set Athlon CPU speed. 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.

8. REFERENCES

[1] S. Adve and et al. The Illinois GRACE Project: Global Resource Adaptation through CoopEration. In *Proc. of Workshop on Self-Healing, Adaptive and self-MANaged Systems*, June 2002.

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

[3] J. M. Anderson and et al. Continuous profiling: Where have all the cycles gone? In *Proc. of 16th Symposium on Operating Systems Principles*, Oct. 1997.

[4] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. In *Proc. of 17th Symposium on Operating Systems Principles*, Dec. 1999.

[5] 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.

[6] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of 3rd Symp. on Operating System Design and Implementation*, Feb. 1999.

[7] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proc. of 4th Symposium on Operating System Design and Implementation*, Oct. 2000.

[8] A. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27:473–484, Apr. 1992.

[9] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general purpose scheduler. In *Proc. of 17th Symposium on Operating Systems Principles*, Dec. 1999.

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

[11] J. Flinn and M. Satyanarayanan. PowerScope: A tool for proling the energy usage of mobile applications. In *Proc. of 2nd IEEE Workshop on Mobile Computing Systems and Applications*, Feb. 1999.

[12] K. Gardner. Probabilistic analysis and scheduling of critical soft real-time systems. PhD thesis, Depratment of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[13] 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.

[14] 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.

[15] 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.

[16] C. Hamann and et al. Quality-assuring scheduling-using stochastic behavior to improve resource utilization. In *Proc. of 22nd IEEE Real-Time Systems Symposium*, Dec. 2001.

[17] 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.

[18] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, Dec. 1998.

[19] M. Jones, D. Rosu, and M. Rosu. CPU reservations & time constraints: Efficient, predictable scheduling of independent activities. In *Proc. of 16th Symposium on Operating Systems Principles*, Oct. 1997.

[20] S. Krantz, S. Kress, and R. Kress. *Jensen's Inequality*. Birkhauser, 1999.

[21] KURT. UTIME– micro-second resolution timers for linux. http://www.ittc.ku.edu/kurt, Mar. 2003.

[22] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, Jan. 1973.

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

[24] J. Lorch and A. Smith. Operating system modifications for task-based speed and voltage scheduling. In *Proc. of the 1st Intl. Conf. on Mobile Systems, Applications, and Services*, May 2003.

[25] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. of 16th Symposium on Operating Systems Principles*, Oct. 1997.

[26] T. Pering, T.Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. of Intl. Symposium on Low Power Electronics and Design*, June 1998.

[27] 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.

[28] 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.

[29] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proc. of SPIE Multimedia Computing and Networking Conference*, Jan. 1998.

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

[31] 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.

[32] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proc. of 5th Symp. on Operating Systems Design and Implementation*, Dec. 2002.

[33] S. Wang, D. Xuan, R. Bettati, and W. Zhao. Differentiated services with statistical real-time guarantees in static-priority scheduling networks. In *Proc. of 22nd IEEE Real-Time Systems Symposium*, Dec. 2001.

[34] 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.

[35] W. Yuan and K. Nahrstedt. Integration of dynamic voltage scaling and soft real-time scheduling for open mobile systems. In *Proc. of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, May 2002.

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

[37] X. Zhang, Z. Wang, N. Gloy, J. Chen, and M. Smith. System support for automated profiling and optimization. In *Proc. of Symposium on Operating Systems Principles*, Oct. 1997.