Appears in SPIE/ACM Multimedia Computing and Networking Conference (MMCN), 2003

Design and Evaluation of a Cross-Layer Adaptation Framework for Mobile Multimedia Systems

Wanghong Yuan[a], Klara Nahrstedt[a], Sarita V. Adve[a], Douglas L. Jones[b], Robin H. Kravets[a]

[a]Department of Computer Science
[b]Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

ABSTRACT

Mobile multimedia systems must provide application quality of service (QoS) in the presence of dynamically varying and multiple resource constraints (e.g., variations in available CPU time, energy, and bandwidth). Researchers have therefore proposed adaptive systems that can respond to changing resource availability and application demands. All system layers can benefit from adaptation, but fully exploiting these benefits requires a new cross-layer adaptation framework to coordinate the adaptations in the different layers. This paper presents such a framework and its first prototype, called *GRACE-1*. The framework supports application QoS under CPU and energy constraints via coordinated adaptation in the hardware, OS, and application layers. Specifically, *GRACE-1* uses *global* adaptation to handle large and long-term variations, setting application QoS, CPU allocation, and CPU frequency/voltage to qualitatively new levels. In response to small and temporary variations, it uses *local* adaptation within each layer. We have implemented the GRACE-1 prototype on a laptop system with an adaptive processor. Our experimental results show that, compared to previous approaches that exploit adaptation in only some of the layers or in an uncoordinated way, GRACE-1 can provide higher overall system utility in several cases.

Keywords: QoS, power management, adaptation, mobile multimedia

1. INTRODUCTION

Battery-powered mobile devices, ranging from laptops to PDAs, are becoming increasingly important platforms for multimedia applications. Compared to traditional desktop systems, such mobile systems offer both new challenges and new opportunities. The challenges lie in the demanding and dynamic application QoS requirements coupled with multiple, varying system resource constraints (e.g., CPU time, energy, and network bandwidth). New opportunities arise because system resources and applications can *adapt* to changes in the system environment or applications. For example, hardware components are being designed to switch among multiple operating modes, trading off performance and energy consumption. Similarly, multimedia applications can gracefully adapt their output quality under QoS violations.

We believe that such adaptations are important in all layers of the mobile end-system, and it is important to develop a new cross-layer adaptation framework that can coordinate adaptations among multiple layers. The Illinois GRACE (Global Resource Adaptation through CoopEration) project is developing such a cross-layer adaptation framework.[1] In this framework, all layers cooperate to reach a globally optimal solution, i.e., maximize system-wide utility while meeting all resource constraints. This paper presents the first prototype of such a framework, GRACE-1, that coordinates CPU frequency/voltage adaptation in hardware, soft real-time scheduling adaptation in the OS, and QoS adaptation in applications. GRACE-1 seeks to maximize overall system utility while meeting application QoS requirements with the available battery energy.

Recently, there have been numerous research contributions on QoS or energy aware adaptation in the hardware, OS, and application layers.* Hardware layer adaptation, such as dynamic voltage scaling (DVS),[2–4]

Further author information: Send correspondence to grace@cs.uiuc.edu.

*There have been many related contributions on QoS and energy adaptations in the network layer as well. However, currently, we concentrate on cross-layer adaptation only within a stand-alone mobile system.

dynamically reconfigures hardware components to save energy while providing the requested resource. OS layer adaptation changes resource allocation[5, 6] and/or scheduling policies[7] to stabilize application quality under variations in resource usage or availability. Application layer adaptation trades off quality of applications for their resource and energy usage.[8–12]

Most of the above adaptation techniques either focus on adapting a single layer at a time, or utilize the OS to facilitate application adaptation. None of them, however, consider a coordinated cross-layer adaptation in all three layers. Combining these adaptation techniques, as they exist in individual layers, is not straightforward due to at least four reasons. First, adaptation behaviors in different layers may be in conflict. For example, the processor may slow down to save energy and the application may increase its CPU demand due to media changes at the same time. Second, independent adaptations in various layers may cause large fluctuations of application output quality, which is annoying to the end user. Third, various runtime scenarios have different adaptation objectives, e.g., maximize application quality when resources are sufficient, or minimize energy consumption when the battery runs low. Finally, adaptations in individual layers have different runtime costs and impact. Hence, if all layers deploy adaptive techniques, we need to consider their integration and coordination. The goal of GRACE-1 is to integrate adaptations in the hardware, OS, and application layers through coordination, so that application QoS and energy requirements are satisfied.

The GRACE framework takes a hierarchical approach to coordinating adaptations.[1] Specifically, it employs a combination of global and local adaptations, where the former is less frequent and more expensive than the latter. *Global* coordination is triggered in response to large and long-term variations. In this case, a central coordinator globally coordinates all layers to establish the QoS level and resource allocation for each application. Once the global decision is made, *local* lower-overhead adaptations react to small and temporary variations. In GRACE-1, the global coordinator assigns the application QoS level, the CPU allocation, and the CPU frequency/voltage; the local adaptations include local adjustments of CPU frequency and allocation.

It is important to stress that GRACE is a generic cross-layer adaptation framework, where many components and parameters at each layer can adapt.[1] The key contribution of this paper is in the presentation of the first prototype, GRACE-1, and a concrete case study of a system with coordinated cross-layer adaptation. In particular, we present the design and evaluation of a system that integrates and coordinates (1) adaptive multimedia player applications in the application layer, (2) adaptive frequency-aware soft real-time scheduling in the OS layer, and (3) the adaptive CPU frequency/voltage mechanism in the hardware layer. Our experiments show that, for several cases, GRACE-1 can provide higher system utility than approaches that exploit adaptation in fewer layers or in an uncoordinated way.

The rest of the paper is organized as follows. Section 2 introduces the system models used by GRACE-1. Section 3 presents the GRACE-1 framework. Section 4 presents the implementation and experimental evaluation. Section 5 describes related work. Finally, Section 6 summarizes the paper.

2. SYSTEM MODELS

GRACE-1 uses different adaptive models in the hardware, application, and OS layers, which react to various adaptation triggers in a single mobile node.

2.1. Processor frequency/voltage adaptation model

We consider mobile devices with a single processor that has an *adjustable* frequency (speed) set $F_{cpu} = \{f_1, ..., f_{max} | f_1 < \cdots < f_{max}\}$, where f_k, $1 \leq k \leq max$, denotes the k^{th} frequency. A lower frequency enables operating at a lower supply voltage, thereby saving power and energy. The average processor power (energy consumption rate) at frequency f, $p(f) \propto C_L \times f \times V_f^2$, where C_L is the effective load capacitance and V_f is the voltage at the frequency f.[13] [†] Table 1 shows the supported frequencies and the corresponding voltages and relative power of an AMD Athlon processor as an example.[15]

[†] In general, the effective load capacitance, C_L, is a function of switching activity and can vary among tasks and among jobs of the same task. For several multimedia applications, however, C_L (and hence $p(f)$) is roughly constant across jobs of the same task.[14] In this work, we assume that it is also constant across all tasks; i.e., average power is only a function of the frequency. In the future, profiling could be used to measure the actual power for higher accuracy.

Table 1. Frequency, voltage, and relative power of a mobile AMD Athlon 4 CPU.

Frequency (MHz)	300	500	600	700	800	1000
Voltage (Volt)	1.2	1.2	1.25	1.3	1.35	1.4
Relative Power (%)	22.04	36.73	47.83	60.35	74.39	100

2.2. Application quality adaptation model

We consider *periodic* multimedia applications or tasks, where each task consumes CPU time and energy and provides an output quality. *Adaptive tasks* are soft real-time tasks that can operate at multiple QoS levels $\{q_1, ..., q_m\}$.[16, 17] For example, a QoS level q may correspond to a frame rate. We aggregate all best-effort applications into one *logical* adaptive task, denoted by T_0. This logical task delivers either average (in a lightly loaded environment) or no (in a heavily loaded environment) quality guarantees to individual best-effort tasks.

Utility (or benefit), $u(q)$, is a quantitative measure of the output quality from the user's view point.[17–20] An example of utility is the perceptual video quality. The utility function can be either provided by applications or users,[20] or defined by basic quality models.[18] *System utility* is defined as the weighted sum of utilities of all concurrent tasks in the mobile system; i.e., $\sum_{i=0}^{n} w_i u_i$, where n, w_i, and u_i denote the number of tasks and the weight and utility of task i, $0 \leq i \leq n$, respectively. Hence, the overall system utility measures the total output quality of all concurrent applications in the system.

2.3. Cross-layer task model

Each periodic task releases a job (e.g., a frame decoding) every period. The released job has a soft deadline, typically defined as the end of the period. The task then consumes a certain amount of CPU processing time for the job execution. Hence, the task CPU requirement can be characterized by its *period* P and *average processing time* C^{avg}.[‡] The period of a task can be calculated from its application QoS level, q, given by the parameter 'rate,' via the equation $P(q) = \frac{1}{rate}$. The average processing time $C^{avg}(q, f)$ is dependent on the application QoS level q as well as the CPU frequency f. For example, an MPEG decoder at frame rate 25fps has period 40ms. For each period, it may need processing time of 6ms at the frequency 1GHz or 12ms at 500MHz.

Overall, we represent each task as a cross-layer task model with period $P(q)$ and average job processing time $C^{avg}(q, f)$, as shown in Figure 1-(a). Figure 1-(b) illustrates the cross-layer task model as a relationship among task average CPU requirements, task QoS levels, and CPU frequencies. Thus, the OS needs to provide multimedia tasks with predictable CPU resource allocations in the presence of variable CPU frequency. For example, for the MPEG player discussed above, the OS needs to allocate 6ms CPU time every period of 40ms (i.e., 15% CPU time) at the frequency 1GHz, and 12ms every 40ms (i.e., 30% CPU time) at 500MHz. Our underlying CPU scheduling algorithm is EDF-based,[21] and described further in Section 3.3.

Here, we assume that, for a specific QoS level q, the average processing time of a task is inversely proportional to the CPU frequency; i.e., the average CPU *cycles* required by a job, $C^{avg}(q, f) \times f$, is roughly constant. This assumption holds for several multimedia applications and current architectures since time spent on memory stalls is small, and the remaining CPU performance scales with CPU frequency.[3, 22]

2.4. Triggers for adaptation

At runtime, there can be several reasons for a change in resource demands and/or resource availability, serving as triggers for adaptation. These include: (1) changes in manipulated media streams, causing fluctuation of CPU usage; e.g., changes of MPEG frame type (I, P, B) or scene changes; (2) joining or leaving of multimedia tasks, causing changes in total CPU demand; and (3) low energy availability; i.e., when the battery of the mobile device cannot last for the *desired lifetime* at the current rate of power consumption. (The desired lifetime is the time until which the battery needs to last without recharging. This is often known by the user of a mobile device; e.g., the time length of a flight or a lecture presentation.[18, 23]) All the above adaptation triggers represent a change in resource availability and/or demand, but may occur at different time scales. Changes of the first type may

[‡]We use average rather than worst-case CPU time because the latter can be too conservative for multimedia applications. These applications exhibit large execution time variability due to their input dependent nature.

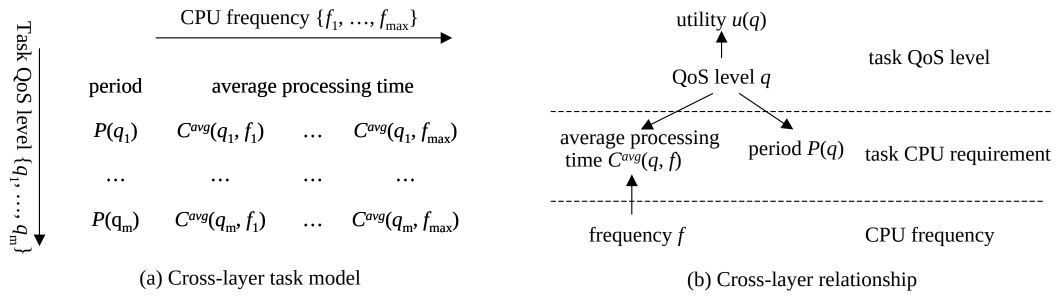


Figure 1. Cross-layer task model.

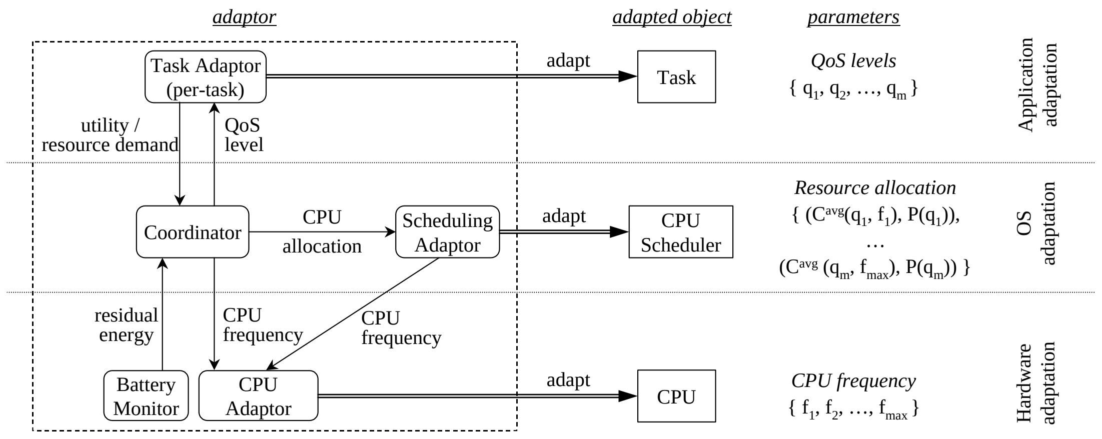


Figure 2. Overview of the GRACE-1 cross-layer adaptation framework.

happen frequently in a short-term (e.g., in tens of milliseconds or per-job, in the case of multiple frame types) or in the medium-term (e.g., in several seconds or across multiple jobs for a scene change). In contrast, changes of the second and third types occur sparsely in long-term intervals (e.g., in minutes, or per-task).

3. CROSS-LAYER ADAPTATION

3.1. Overview

The GRACE-1 framework, shown in Figure 2, consists of five major components: coordinator, scheduling adaptor, battery monitor, CPU adaptor, and a set of task adaptors. The *coordinator* mediates task QoS levels, CPU processing allocations, and CPU frequency, according to task utilities, CPU demands, and energy availability observed by the *battery monitor*. To enable such a cross-layer coordination, the coordinator resides in the OS and has full access to the knowledge of system states (e.g., task resource demands and energy availability). The *task adaptor* is responsible for adjusting its task to the QoS level configured by the coordinator. The *scheduling adaptor* adjusts task CPU allocations. This adaptation enables the scheduler to deliver a soft real-time performance guarantee in a variable frequency context. The *CPU adaptor* dynamically adjusts the CPU frequency to save energy. The CPU frequency is determined either by the coordinator based on the total CPU demands or by the scheduler based on the actual runtime CPU usage.

Operationally, the cross-layer adaptation is achieved through a combination of *global* and *local* adaptations. In response to long-term variations, the global adaptation mediates among all layers, establishing application QoS levels, CPU allocations, and CPU frequency. The local adaptations adjust CPU frequency and allocations to handle temporary, per-job variations in application runtime CPU usage, thereby maintaining application quality. Our prototype does not currently perform local application adaptation since the adaptations currently considered have high overheads that are unsuitable responses for local variations (see Section 4). In general, however, multimedia applications could adapt their QoS locally within an acceptable range of the globally assigned task QoS level, using, for example, rate control or media scaling.[9, 11, 12]

3.2. Global coordination

The coordinator mediates all three layers at global adaptation triggers, including a change in the number of tasks in the system, low energy availability, or long-term and large changes in the CPU demand of an application. The global coordination in GRACE-1 aims to maximize the accumulated system utility for a desired battery lifetime, given an available energy constraint; i.e.,

$$\text{maximize:} \quad \int_{T^{cur}}^{T^{des}} \sum_{i=0}^{n(t)} w_i u_i(q_i) dt \quad \text{(accumulated system utility)} \tag{1}$$

$$\text{subject to:} \quad \forall t \sum_{i=0}^{n(t)} \frac{C_i^{avg}(q_i, f(t))}{P_i(q_i)} \leq 1 \quad \text{(CPU resource constraint)} \tag{2}$$

$$\int_{T^{cur}}^{T^{des}} p(f(t))dt \leq E^{res} \quad \text{(energy constraint)} \tag{3}$$

$$f(t) \in F_{cpu} = \{f_1, ..., f_{max} | f_1 < \cdots < f_{max}\} \tag{4}$$

where T^{cur} is current time, T^{des} is the time until the remaining desired battery lifetime, E^{res} is the residual battery energy, $n(t)$ is the number of tasks at time t, and $f(t)$ is the CPU frequency at t.[§] Equation (2) guarantees the schedulability of the EDF-based soft real-time scheduling algorithm (Section 3.3). Equation (3) ensures that the total energy consumption is not greater than the residual energy.

The above constrained optimization requires future knowledge about applications during the entire desired lifetime. At the current coordination time, such knowledge is generally unavailable since tasks can dynamically join and leave the system. Hence, we propose two heuristic policies, *utility-greedy* and *energy-greedy*, to solve the problem. At any coordination time, the utility-greedy heuristic aims to maximize the *current* system utility, i.e.,

$$\text{maximize:} \quad \sum_{i=0}^{n} w_i u_i(q_i) \quad \text{(current system utility)} \tag{5}$$

$$\text{subject to:} \quad f = \min\{f_j : f_j \in F_{cpu} \text{ and } \sum_{i=0}^{n} \frac{C_i^{avg}(q_i, f_j)}{P_i(q_i)} \leq 1\} \quad \text{(CPU resource constraint)} \tag{6}$$

In contrast, the energy-greedy heuristic seeks to guarantee energy availability for the desired lifetime by assuming that the current applications will stay for the remaining lifetime, and maximizes the system utility given this constraint. Hence, this heuristic is guided by the additional constraint:

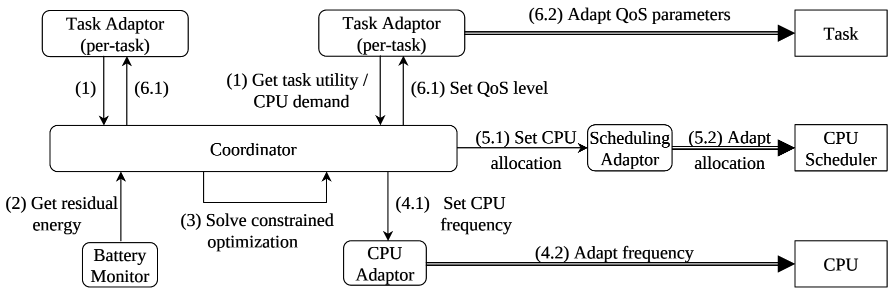
$$p(f) \leq \frac{E^{res}}{T^{des} - T^{cur}} \quad \text{(power constraint)} \tag{7}$$

Although both the utility-greedy and energy-greedy heuristics result in NP-hard problems (reduced to the Knapsack problem), they can be solved efficiently using well-known techniques for constrained optimization problems. In particular, using the dynamic programming algorithm proposed by Pisinger,[24] we can solve the problems with complexity $O(N + M)$, where N is the sum of the number of QoS levels of all current tasks and M is the number of supported CPU frequencies. After solving the utility-greedy or energy-greedy heuristic, the coordinator makes a global decision on task QoS levels, CPU allocations, and the CPU frequency/voltage. It then notifies local adaptors to perform corresponding adaptations, as shown in Figure 3-(a).

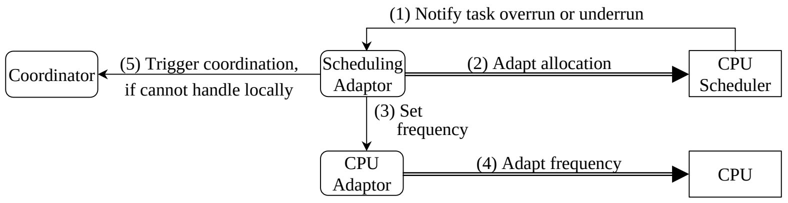
3.3. Local adaptation

Local adaptation handles small and temporary variations in task runtime CPU usage. Before discussing the local adaptation approach, we briefly present GRACE-1's frequency-aware soft real-time CPU scheduling algorithm.[21] The algorithm delivers soft real-time performance guarantees with variable frequency by maintaining a task's

[§]Alternate formulations of the problem are possible (e.g., consideration of the duration of each application), and are part of our ongoing work.



(a) Protocol for global coordination



(b) Protocol for local adaptation

Figure 3. Protocols for global coordination and local adaptation.

CPU *budget* in terms of CPU *cycles* (which are independent of the frequency as discussed in Section 2.3). Specifically, it allocates a budget of $C^{avg}(q, f_{max}) \times f_{max}$ cycles every period $P(q)$ to a task with QoS level q. It then dispatches tasks based on their deadlines and budgets — selecting the task with the earliest deadline and non-exhausted budget. As the task is executed, its budget is decreased by the number of cycles it consumes. Thus, if a task executes for Δt time units at frequency f, its budget is decreased by $\Delta t \times f$. Therefore, at any frequency f, each task is allowed to consume its average processing time $C^{avg}(q, f)$, because $C^{avg}(q, f) \times f = C^{avg}(q, f_{max}) \times f_{max}$. In this way, the algorithm isolates the changes of the underlying CPU frequency from the resource reservation guaranteed to the multimedia applications. To ensure schedulability of all tasks (based on average task demands), the frequency is set such that $\sum \frac{C^{avg}(q_i,f)\times f}{P_i(q_i)} \leq f$, equivalent to Equation (2).

The scheduler allocates CPU budgets according to the average demands of tasks. The actual runtime CPU usage of an individual job of a task may be less than or greater than its allocated budget, incurring an underrun or overrun respectively. When the scheduler detects such a case, it notifies the scheduling adaptor. The scheduling adaptor locally changes the CPU allocation by adding or reclaiming the budget, and requests the CPU adaptor to adjust the CPU frequency to account for the extra allocation or reclamation. Hence, the local adaptations of CPU allocation and frequency are tightly integrated. Figure 3-(b) shows the interactions among GRACE-1 components during the local adaptation.

More specifically, consider a task T_i underruns at time t with a residual budget of b_i cycles. Let t' be T_i's deadline, at which T_i releases a new job. At the current frequency, f_{cur}, the processor provides a total CPU budget (for all tasks) of $f_{cur} \times (t' - t)$ cycles in the interval $[t, t')$. However, the actual total budget demand is $f_{cur} \times (t' - t) - b_i$ cycles because of T_i's underrun. Hence, during the interval $[t, t')$, the processor can reclaim the residual budget by slowing down to a lower frequency $f^- = f_{cur} - \frac{b_i}{t'-t}$ (Figure 4-(a)).

On the other hand, consider a task T_i overruns at time t, and t' is T_i's deadline. The basic idea behind overrun handling is to allocate T_i with an extra budget, so that T_i can finish the overrun job by its deadline. To predict the additional CPU budget required by the overrun job, GRACE-1 uses the heuristic that the overrun cycles for the last job that had an overrun will be required again.[4] Because it has executed a part of the job, the task can provide some useful information (e.g., MPEG frame type) to help the prediction. For a predicted overrun budget of o_i cycles, the total budget demand (across all tasks) over the interval $[t, t')$ is $f_{cur} \times (t' - t) + o_i$ cycles. Thus, in this interval, the processor needs to run at a higher frequency $f^+ = f_{cur} + \frac{o_i}{t'-t}$ (Figure 4-(b)).

If the CPU frequency required to handle the underrun or overrun is not supported, then the scheduler can either (1) tolerate deadline misses or resource waste, or (2) in case of consistent overruns and underruns, it can

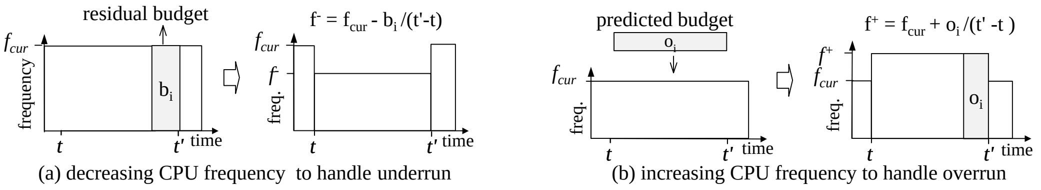

residual budget
f_{cur}
frequency
b_i
t
t'
time
$f^- = f_{cur} - b_i/(t'-t)$
f^-
freq.
(a) decreasing CPU frequency to handle underrun
predicted budget
o_i
$f^+ = f_{cur} + o_i/(t'-t)$
f^+
(b) increasing CPU frequency to handle overrun

Figure 4. Illustration of overrun and underrun handling via CPU frequency adjustment.

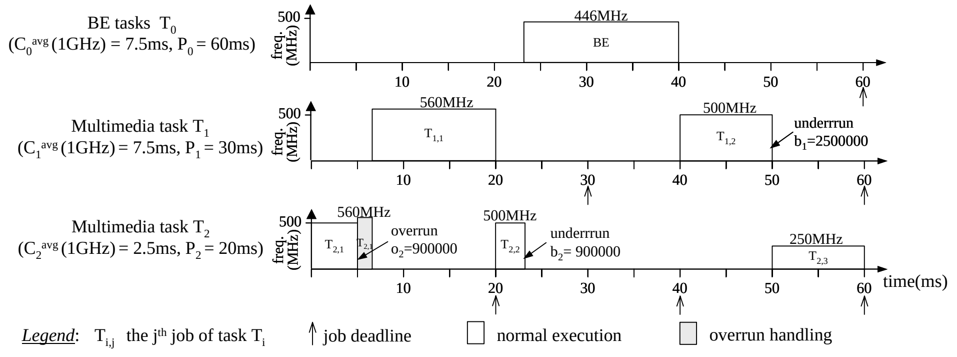


Figure 5. An example of local adaptation to handle runtime CPU usage variations.

adapt the CPU allocation for all later jobs of the task and accordingly adapt the CPU frequency, thus avoiding frequent local adaptations, or (3) in case the overruns and underruns are consistently large, it can trigger a global coordination to adapt the task QoS and CPU demands. Our experiments in Section 4.2 use option (1).

Figure 5 gives an example of local adaptation with best-effort tasks aggregated into T_0, and two multimedia tasks T_1 and T_2. Initially, these tasks start with the globally coordinated CPU budget allocation and CPU frequency of 500 MHz. At time 0, T_2 has the earliest deadline, and thus executes until time 5 when it overruns. The scheduling adaptor predicts the overrun part to be 9×10^5 cycles, so the frequency is increased to $500 + \frac{9\times10^5}{20-5ms} = 560$ MHz. With the extra budget, T_2 continues to execute, and completes its overrun part at time 6.6. The CPU frequency is reset to 500 MHz at time 20, at which T_2 begins a new period. At time 23.2, T_2 underruns with a residual budget of 9×10^5 cycles, and the frequency is decreased to $500 - \frac{9\times10^5}{40-23.2ms} = 446$ MHz. Similarly, when T_1 underruns at time 50 with a residual budget of 2.5×10^6 cycles, the frequency is decreased to 250 MHz.

4. IMPLEMENTATION AND EVALUATION

4.1. Implementation

We have implemented a prototype of the GRACE-1 framework, using the Red Hat Linux OS. The scheduler and CPU adaptor are implemented as loadable kernel modules in Linux kernel 2.4.7. The scheduling adaptor is integrated into the scheduler. The coordinator is implemented as a user-level daemon process. We also modified the experimental application to integrate one task adaptor (Section 4.2). We have not implemented the battery monitor, because we currently do not have power meters such as PowerScope.[23] Instead, we calculate normalized energy consumption based on the relative power in Table 1.

The coordinator communicates with tasks via a message queue to receive specification of task utilities and CPU demands. It uses the utility-greedy or energy-greedy heuristic for global coordination (Section 3.2), and indicates the resulting application QoS level, CPU budget allocation, and CPU frequency to the respective adaptors.

The GRACE-1 scheduler does not replace the standard Linux scheduler. Instead, it is added into the Linux *timer task queue*, called `tq_timer`, whose elements are executed on the next timer interrupt (every 10ms for x86 machines). Hence, the GRACE-1 scheduler is invoked every 10ms, and changes task scheduling parameters according to the frequency-aware scheduling algorithm. Specifically, it sets the scheduling policy of a task

Table 2. QoS levels and corresponding utility of the MPEG player. (C^{avg} is specified at the highest frequency.)

		Dithering method		
		gray	**ordered**	**color2**
Frame rate	**20**	Utility 0.31 CPU(C^{avg}=5.8ms, P=50ms)	Utility 0.37 CPU(C^{avg}=8.9ms, P=50ms)	Utility 0.41 CPU(C^{avg}=10.6ms, P=50ms)
	25	Utility 0.34 CPU(C^{avg}=5.8ms, P=40ms)	Utility 0.42 CPU(C^{avg}=8.9ms, P=40ms)	Utility 0.46 CPU(C^{avg}= 10.6ms, P=40ms)
	33	Utility 0.39 CPU(C^{avg}=5.8ms, P=30ms)	Utility 0.49 CPU(C^{avg}=8.9ms, P=30ms)	Utility 0.55 CPU(C^{avg}= 10.6ms, P=30ms)

to real-time mode `SCHED_FIFO` if the task has a positive CPU budget, and to best-effort mode `SCHED_OTHER` otherwise. It also sets the `rt_priority` of tasks according to their deadlines — the earlier the deadline, the higher the `rt_priority`. Hence, the multimedia task with the earliest deadline and positive budget has the highest `goodness` value, and is dispatched first by the standard Linux scheduler.

The hardware platform for our implementation and experiments is the HP Pavilion N5470 laptop with a single Mobile AMD Athlon 4 processor[15] and 256MB RAM. This processor supports six different frequencies and voltages (Table 1), which can be adjusted dynamically under OS control. The GRACE-1 CPU adaptor kernel module implements CPU frequency/voltage adjustment by writing the frequency and corresponding voltage to a special register `FidVidCtl`, based on the approach by Pillai and Shin.[2]

4.2. Experimental evaluation

For applications in our experimental evaluation, we use (soft real-time) MPEG video players, integrated with task adaptors that adapt the frame rate and the dithering method for different QoS levels. When the frame rate changes, the task period changes. When the dithering method changes, the task exits the current decoding thread, and starts another thread to play the video from the current frame number using the new dithering method. The experimental video is the *4Dice* MPEG video with frame size 352×240 pixels and 1679 frames.

We combine three dithering methods (*gray*, *ordered*, and *color2*) and three frame rates (20, 25, and 33 fps) to give nine QoS levels of the MPEG player (Table 2). For each QoS level, to determine the average CPU demand, we profile the CPU processing time for each frame and use the average across all 1679 frames. The profiling is performed at the highest CPU frequency of 1 GHz.

Defining a utility function for a QoS level is difficult, since it must represent subjective user satisfaction. In this paper, we define the utility function of an MPEG player as $\alpha + \frac{C^{avg}(q,f_{max})}{P(q)}$, where α represents the utility for its running in a soft real-time mode (α is 0.2 in our experiments), and $\frac{C^{avg}(q,f_{max})}{P(q)}$ represents the utility for its output quality at the QoS level q. Intuitively, the system utility is high if there are many soft real-time tasks and/or each task runs at a high QoS level. We leave the derivation of more sophisticated utility functions that better capture user satisfaction (e.g., by directly capturing the perceptual quality of video, accounting for task duration, etc.) to future work.

4.2.1. Overheads

Our first experiment measures overheads of global coordination and scheduling. To do so, we assume 1 to 30 tasks running concurrently, each with nine QoS levels of period 50ms and average CPU time demand randomly distributed between 1 and 10ms. We use this synthetic rather than real workload to stress the coordinator and scheduler. In the coordinator process, we measure its runtime (at the highest frequency) to solve the constrained optimization problem with the utility-greedy or energy-greedy heuristic. In the kernel, we measure the elapsed cycles for each execution of the GRACE-1 scheduler for 10000 times and calculate the average as the overhead.

Figure 6-(a) and (b) show the overheads of global coordination and scheduling respectively. The coordination time for utility-greedy and energy-greedy heuristics is similar. The coordination overhead varies linearly with the number of concurrent tasks, but is quite small relative to the computation time for an MPEG frame. The scheduling overhead also varies linearly with the number of tasks (because the scheduling algorithm needs to check the status of each task; e.g., if it begins a new period or underruns/overruns), but again is quite small.

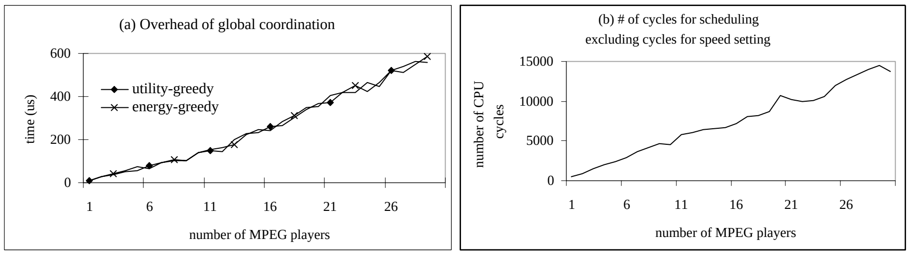


Figure 6. Overheads of global coordination and scheduling.

Other overheads incurred in our system include those for task QoS adaptation and CPU frequency/voltage scaling. The former is application-specific, and is quite large for our application; e.g., changing the dithering method costs about 100ms. We therefore currently do not perform local application adaptation, but could do so with other applications. The CPU voltage/frequency setting overhead has been reported by others,[2] and is small relative to the computation for an MPEG frame.

4.2.2. Comparison with other adaptation policies

We next compare GRACE-1 (both utility-greedy and energy-greedy) with a baseline policy with no adaptation, and with five other policies that adapt one, two, or three layers (possibly in response to changes in other layers and possibly with some coordination by the OS):

- *No-adapt.* The CPU runs at the highest frequency and the applications run at the highest QoS level.
- *App-only.* The CPU runs at the highest frequency. When an application joins the system, it configures its QoS level as high as possible, given the currently available CPU resource.
- *App-OS.* The CPU runs at the highest frequency. When an application joins or leaves the system, the OS coordinates the adaptation of all current applications to maximize the system utility (at the highest frequency). Effectively, this maximizes the accumulated system utility expression in Equation (1).
- *CPU-only.* The applications run at the highest QoS level. When an application joins or leaves the system, the CPU adapts according to the total CPU demand; i.e., it adjusts its frequency to $\min\{f_j : f_j \in F_{cpu}$ and $\sum_{i=0}^{n} \frac{C_i^{avg}(q_i, f_j)}{P_i(q_i)} \leq 1\}$ (i.e., the CPU resource constraint Equation (6)).
- *App-CPU.* This is uncoordinated adaptation in the application and hardware layer – the application and CPU perform their adaptation as in the *app-only* and *CPU-only* cases respectively.
- *App-OS-CPU.* This is an uncoordinated combination of app-OS and CPU-only. Application adaptation is coordinated by the OS as in App-OS. CPU frequency is set as in CPU-only, based on the CPU demand of the highest QoS level of the applications present.

It should be emphasized that, to our knowledge, no previous system adapts three layers. Our intent in designing the last app-OS-CPU policy and comparing with GRACE-1 was to show the benefits of fully coordinated adaptation. However, app-OS-CPU itself is already partially coordinated – the presence of the OS as one of the three layers inherently implies some coordination and made it difficult to develop a system with completely uncoordinated adaptation across all three layers. Note also that a modification of app-OS-CPU to set the CPU frequency based on the QoS levels of the applications after they adapt would make it a coordinated adaptation policy – it would be the same as the utility-greedy heuristic for GRACE-1.

For each of the above six policies, the scheduler allocates to a task the average CPU processing cycles at the appropriate QoS level, using the frequency-aware CPU scheduling algorithm (for a non-adaptive CPU, the frequency is fixed at 1 GHz). If there is not enough CPU resource for such an allocation (e.g., the frequency selected is higher than supported or the lowest application QoS considered cannot be supported), then the task

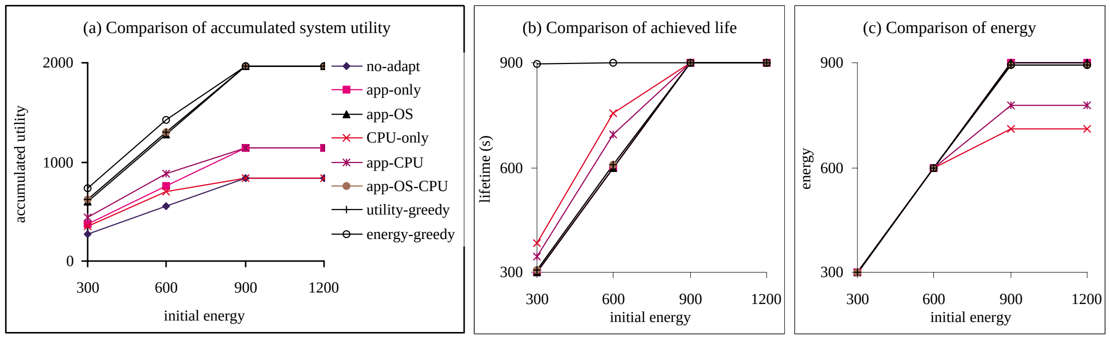


Figure 7. Comparing GRACE-1 with other policies with one task arrival every 12sec.

Table 3. Accumulated system utility of different policies with other arrival rates and with initial energy 300.

Arrival rate	no-adpt	cpu-only	app-only	app-cpu	app-os	app-os-cpu	utility-greedy	energy-greedy
every 20s	271	352	342	389	371	421	421	709
every 30s	266	350	266	350	266	350	350	546

runs in best effort mode with a utility of 0. Further, the scheduler does not handle overruns and underruns via local adaptation of CPU allocations and frequency for the above policies, since that assumes coordination between CPU and OS adaptation.¶

To evaluate the above policies, our default experiments start an MPEG player every few seconds on a new 4Dice video stream (a player exits after completing its stream). Each player has a weight of 1. We allocate a budget of 10ms every 100ms to all best-effort tasks, and run a best-effort, computation-intensive program in the background. As indicated above, an MPEG player may also run in best-effort mode if there is not enough CPU resource. We assume the desired battery lifetime to be 900 seconds. We perform experiments with different initial energy availability of 300, 600, 900, and 1200 units (normalized according to Table 1). We also perform experiments with different task arrival rates – an MPEG player arrives either every 12, 20, or 30 seconds.

We measure accumulated system utility, the achieved battery lifetime, and the total energy consumption. Accumulated system utility and energy consumption are defined as $\int_0^{T_{ach}} u(t)dt$ and $\int_0^{T_{ach}} p(t)dt$, respectively, where T_{ach} is the achieved lifetime, $u(t)$ and $p(t)$ are the system utility and CPU power at time $t, 0 \leq t \leq T_{ach}$, respectively. Figure 7 reports all of these metrics for the case where the MPEG players arrive every 12 seconds. Table 3 reports the accumulated system utility for the cases where the MPEG players arrive every 20 and 30 seconds, for initial energy of 300 units.

GRACE-1 vs. adaptation in 0 or 1 layer: Compared to no-adapt, app-only, and CPU-only, both GRACE-1 policies generally achieve a much higher accumulated system utility since the former policies are oblivious to utility.‖ CPU-only achieves a higher battery life than utility-greedy (but not higher than energy-greedy) only because it runs at a much lower utility and in fact does not exhaust all its energy (Figure 7-(c)).

GRACE-1 vs. app-OS: Both heuristics of GRACE-1 achieve higher or the same accumulated utility and battery lifetime than app-OS – the difference is larger with low initial energy availability (i.e., 300 and 600 units) and when the load on the system is not excessive (i.e., when the MPEG players arrive at 20sec and 30sec intervals, as in Table 3). In the case where initial battery energy is high, all of these policies perform well and in a similar way. In the case where the system workload is excessive (and initial energy is low), the accumulated utility and achieved lifetime of app-OS is very similar to that of utility-greedy (but less than that of energy-greedy). This is because the excessive system load makes utility-greedy run at the highest CPU frequency to maximize the current utility. Thus, it does not exploit CPU adaptation and appears similar to app-OS. This is not the case at lower system load or for energy-greedy (at low initial energy).

¶Previous work (e.g., DSRT[6] and SMART[25]) can handle overruns and underruns by adapting task CPU allocations, but this is for changes over global rather than local time scales.

‖In Figure 7-(a), the curve for utility-greedy is the second from the top (hidden behind app-OS and app-OS-CPU).

GRACE-1 vs. app-CPU: The uncoordinated combination of application and CPU adaptation gives better (or the same) accumulated utility than either adaptation alone. However, it is never better compared to GRACE-1, and gives significantly lower accumulated system utility than either heuristic of GRACE-1 for most cases.

GRACE-1 vs. app-OS-CPU: Energy-greedy GRACE-1 achieves higher (or the same) accumulated utility than app-OS-CPU. The difference is large with low initial energy and low to moderate task arrival rates.

Compared to utility-greedy GRACE-1, however, we find that app-OS-CPU gives the same accumulated system utility. The only potential difference between these policies is in the CPU frequency, which is set to be the same by both policies for these experiments. This is because both policies effectively use application QoS levels that would maximize the CPU utilization at the highest frequency. This results in running applications at either their highest QoS levels (in which case both policies pick the same CPU frequency) or running at virtually full CPU utilization (in which case, again both pick the same frequency).

However, there could be scenarios where utility-greedy GRACE-1 gives better accumulated system utility than app-OS-CPU. We performed another experiment with a similar MPEG player to show this distinction. This MPEG player decodes two 4Dice video streams simultaneously with two possible QoS levels. Both QoS levels have the same frame rate of 33 fps, but use different dithering methods, color2 or gray. The average processing time at the highest frequency for color2 and gray is 21.2ms and 11.6ms, respectively. We start two players together every 60 seconds. Both app-OS-CPU and utility-greedy policies configure the two players at the low QoS level. This, however, does not saturate the CPU, and so utility-greedy is able to set the CPU frequency at 800MHz. App-OS-CPU, however, sets the frequency at 1GHz because it over-estimates the total CPU demand. As a result, utility-greedy sees a longer battery lifetime and hence higher accumulated system utility.

Energy-greedy vs. utility-greedy GRACE-1: Our results show that the energy-greedy heuristic for GRACE-1 is often better (and never worse) than the utility-greedy heuristic, especially with low initial energy availability and moderate to low task arrival rate. This is because energy-greedy attempts to maximize battery lifetime, achieving the longest lifetime of all policies (Figure 7(b)). Since in our workload, new and equally important applications arrive at regular intervals, this long battery lifetime translates to high accumulated utility.

Utility-greedy, on the other hand, maximizes the current (i.e., instantaneous) utility, but sacrifices battery lifetime, achieving lower accumulated system utility in many cases. It is nevertheless possible to conceive of workloads where utility-greedy may give a higher accumulated system utility than energy-greedy; e.g., if applications that join and exit the system early have higher weight than those that join late. We confirmed this by running experiments with correspondingly different weights for the MPEG players. For such cases, GRACE-1 would do best by using a flexible policy, where energy-greedy is used if important applications are predicted to join late and utility-greedy is used if the current applications are more important. This flexibility, however, requires some knowledge of the workload, and we leave it to future work.

Summary: Overall, our results show that coordinated adaptation in multiple system layers provides significant benefits when there is limited available energy. In our experiments, the energy-greedy GRACE-1 heuristic provided the highest accumulated system utility and longest achieved battery lifetime overall, and significantly outperformed all the other policies in some cases. For some workloads (where the most important applications join and leave the system early), however, a combination of the energy-greedy and utility-greedy heuristics would work best, but requires predicting the future workload.

4.2.3. Effectiveness of local adaptation for overrun and underrun handling

In the final experiment, we assess the effectiveness of local adaptation in GRACE-1 to handle overruns and underruns. We run an MPEG player at the highest QoS level, and start a best-effort, computation-intensive program in the background. In GRACE-1, when the MPEG player overruns, the scheduling adaptor predicts the demand for the remaining part, allocates an additional budget to the player by adjusting the CPU frequency, and lets it run in soft real-time mode (as long as the CPU frequency can be adjusted to accommodate the overrun) (Section 3.3). We compare this to a version of GRACE-1 without such local adaptation – the overrun player is always made to run in best-effort mode and compete for CPU time with the background program. We found that the addition of local adaptation reduced the deadline miss ratio from 26% to less than 1% with both the utility-greedy and energy-greedy heuristics in GRACE-1. Thus, local adaptation in GRACE-1 is clearly effective.

5. RELATED WORK

Recently, there have been numerous research contributions on adaptation in the hardware, OS, and application layers. In the hardware, dynamic voltage scaling (DVS) is typically used to adjust CPU speed and power, based on the application workload. The workload is either predicted using heuristics[4] (which could violate timing constraints for soft real-time multimedia applications) or estimated from worst-case CPU time[2, 3] (which are generally too conservative for multimedia applications). GRACE-1 integrates DVS with CPU resource management, to achieve the energy savings of DVS while delivering soft real-time guarantees. GRACE-1 also integrates DVS with application adaptation. There is also work on hardware architecture adaptation[14] that can similarly be integrated into the GRACE-1 framework; we leave this work for the future.

Much work has been done in the operating system and middleware to provide predictable CPU allocation[5, 7, 25, 26] and adaptation services.[9, 11, 12, 18] Like GRACE-1, these CPU resource managers, such as SMART[25] and RT-Mach,[26] deliver soft real-time performance guarantees. The DSRT,[6] BERT,[5] and BEST[7] schedulers further adapt the scheduling policy to handle variations in application runtime CPU usage. Unlike GRACE-1, these approaches assume that the processor always runs at a constant speed. Odyssey[9] adds system support for mobile application adaptation, focusing on data fidelity and adaptation agility. Puppeteer[12] and DQM[27] are middlewares to facilitate applications to adapt their QoS. In contrast to the above work, GRACE-1 coordinates adaptations in the hardware, OS, and application layers. Like GRACE-1, TIMELY[28] integrates and coordinates cross-layer adaptations within the network protocol stack, but focuses on the network bandwidth resource.

Several projects advocate energy saving in the application layer. For instance, the Milly Watt project[10] proposes applications to be the driving force for higher-level power management. Flinn et al.[23] explore how to adapt application behavior to save energy. Mesarina et al.[8] discuss how to reduce energy in MPEG decoding. This application-layer energy saving work is orthogonal and complementary to GRACE-1. Furthermore, GRACE-1 provides a mechanism to coordinate adaptations of various applications and the CPU allocation and frequency.

Other closely related work includes dynamic QoS-aware resource allocation.[19, 29] Q-RAM[19] proposes a constraint optimization model to maximize the overall system utility while guaranteeing the minimum resource to each application. The IRS system[29] coordinates allocation and scheduling of multiple heterogeneous resources, maximizing the number of admitted applications in the long run. These approaches, however, assume that the hardware layer is static. In contrast, GRACE-1 targets a mobile system with adaptive CPU hardware and software applications, and optimizes the system utility under resource and energy constraints.

6. CONCLUSIONS

This paper presents a cross-layer adaptation framework and prototype implementation, *GRACE-1*, to coordinate adaptations of CPU frequency, CPU allocation, and application quality in mobile systems. The goal is to deliver high system utility in the presence of CPU time and energy constraints. GRACE-1 achieves this goal with low overhead through a combination of global coordination and local adaptation of the hardware, OS, and application layers. Our experiments show that compared with systems that exploit adaptation in only some of the layers or in an uncoordinated way, GRACE-1 can significantly increase system utility and battery lifetime in many cases.

Our work with GRACE-1 has provided valuable experience with cross-layer adaptation, but much remains to be done. We are currently working on integrating network and hardware architecture adaptations as well as local application adaptations. We are also developing more sophisticated utility functions, incorporating new forms of adaptation, and developing better global and local policies within our framework.

ACKNOWLEDGMENTS

We would like to thank the members of the GRACE project, especially Daniel Sachs, for many discussions, and the anonymous reviewers for their constructive comments. This work was supported in part by the National Science Foundation under Grant No. CCR-0205638, CCR-0209198, CCR-0096126, CCR-99-88199, and EIA-99-72884, and by a NASA grant NASA NAG 2-1250. Sarita V. Adve is also supported by an Alfred P. Sloan Research Fellowship. 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 the funding agencies.

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 (SHAMAN02), New York City, NY*, June 2002.
2. 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, Banff, Canada*, Oct. 2001.
3. R. Melhem, N. AbouGhazaleh, H. Aydin, and D. Mosse, "Power management points in power-aware real-time systems," in *Power Aware Computing*, R. Graybill and R. Melhem, eds., Plenum/Kluwer Publisher, 2002.
4. M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proc. of USENIX Symposium on Operating Systems Design and Implementation, 13-23*, Nov. 1994.
5. A. Bavier and L. Peterson, "The power of virtual time for multimedia scheduling," in *Proc. of 10th International Workshop for Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 2000.
6. H. H. Chu and K. Nahrstedt, "CPU service classes for multimedia applications," in *Proc. of IEEE Int. Conf. on Multimedia Computing and Systems (ICMCS'99), Florence, Italy*, pp. 296–301, June 1999.
7. S. Banachowski and S. Brandt, "The BEST scheduler for integrated processing of best-effort and soft real-time processes," in *Proc. of SPIE Multimedia Computing and Networking Conference, San Jose, CA*, Jan. 2002.
8. M. Mesarina and Y. Turner, "Reduced energy decoding of MPEG streams," in *Proc. of SPIE Multimedia Computing and Networking Conference, San Jose, CA*, Jan. 2002.
9. B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker, "Agile application-aware adaptation for mobility," in *Proc. of the 16th Symposium on Operating Systems Principles, Saint Malo, France*, Dec. 1997.
10. A. Vahdat, A. Lebeck, and C. Ellis, "Every joule is precious: A case for revisiting operating system design for energy efficiency," in *Proc. of 9th ACM SIGOPS European Workshop, Kolding, Denmark*, Sept. 2000.
11. B. Li and K. Nahrstedt, "A control-based middleware framework for quality of service adaptations," *IEEE J. Select. Areas Commun., 17(9)* , pp. 1632–1650, Sept. 1999.
12. J. Flinn, E. de Lara, M. Satyanarayanan, D. Wallach, and W. Zwaenepoel, "Reducing the energy usage of office applications," in *Proc. of Middleware 2001, Heidelberg, Germany*, Nov. 2001.
13. A. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits, Vol. 27* , pp. 473–484, Apr. 1992.
14. C. Hughes, J. Srinivasan, and S. Adve, "Saving energy with architectural and frequency adaptations for multimedia applications," in *Proc. of 34th International Symposium on Microarchitecture, Austin, TX*, Dec. 2001.
15. AMD, "Mobile AMD Athlon 4 processor model 6 CPGA data sheet." http://www.amd.com/products/cpg/athlon/techdocs/pdf/24319.pdf, Nov. 2001.
16. H. Tokuda and T. Kitayama, "Dynamic QoS control based on real-time threads," in *Proc. of the 3rd International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Nov. 1993.
17. S. Brandt, "Performance analysis of dynamic soft real-time systems," in *Proc. of the 20th IEEE International Performance, Computing, and Communications Conference (IPCCC 2001)*, Apr. 2001.
18. M. Corner, B. Noble, and K. Wasserman, "Fugue: time scales of adaptation in mobile video," in *Proc. of SPIE Multimedia Computing and Networking Conference, San Jose, CA*, Jan. 2001.
19. R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for QoS management," in *Proc. of 18th IEEE Real-Time Systems Symposium, San Francisco, CA*, Dec. 1997.
20. T. Kunz, M. Shentenawy, A. Gaddah, and R. Hafez, "Image transcoding for wireless WWW access: The user perspective," in *Proc. of SPIE Multimedia Computing and Networking, San Jose, CA*, Jan. 2002.
21. 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 (NOSSDAV '02), Miami Beach, FL*, May 2002.
22. C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan, "Variability in the execution of multimedia applications and implications for architecture," in *Proc. of International Symposium on Computer Architecture*, 2001.
23. 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.
24. D. Pisinger, "A minimal algorithm for the multiple-choice knapsack problem," *European Journal of Operational Research, 83* , pp. 394–410, 1995.
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, St-Malo, France*, Oct. 1997.
26. C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. of IEEE Int. Conf. on Multimedia Computing and Systems (ICMCS'94)*, May 1994.
27. S. Brandt and G. J. Nutt, "Flexible soft real-time processing in middleware," *Real-Time Systems* **22**(1-2), 2002.
28. V. Bharghavan, K. Lee, S. Lu, S. Ha, J. Li, and D. Dwyer, "The TIMELY adaptive resource management architecture," *IEEE Personal Communications Magazine, 5(4)* , Aug. 1998.
29. K. Gopalan and T. Chiueh, "Multi-resource allocation and scheduling for periodic soft real-time applications," in *Proc. of SPIE Multimedia Computing and Networking Conference, San Jose, CA*, Jan. 2002.