ReCalendar: Calendaring and Scheduling Applications with CPU and Energy Resource Guarantees for Mobile Devices

Wanghong Yuan, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
{wyuna1, klara}@cs.uiuc.edu

Abstract

In this paper, we present an advance reservation scheme for CPU and energy resources, called ReCalendar. *The goal is to enable soft real-time applications on mobile devices to achieve desired performance and lifetime. ReCalendar achieves this goal via two distinctive phases, calendaring and scheduling. In the calendaring phase, a calendar is used to arrange applications and to request CPU and energy reservations for calendared applications. In the scheduling phase, the resource manager enforces admitted reservations via CPU frequency/voltage adaptation and frequency-aware soft real-time scheduling. We have implemented the ReCalendar prototype and experimentally shown that, compared with previous approaches that support only immediate reservation or CPU advance reservation, ReCalendar achieves longer battery lifetime and higher overall system utility.*

1 Introduction

Battery-powered mobile devices are becoming increasingly an integral part of pervasive computing environment. They serve both to schedule our daily activities in advance and to run corresponding applications at the scheduled time. A laptop user, for example, may use a calendar to schedule activities such as reminder for a group meeting at 9:00 AM and DVD movie playback during a flight. These applications impose quality of service (QoS) requirements. In mobile systems, application QoS includes not only performance, such as throughput and deadline miss ratio, but also application lifetime (duration) such as DVD playback session. Therefore, in order to deliver desirable QoS in terms of performance and lifetime, pre-scheduled applications need predictable management of system resources such as CPU and energy.

Resource reservation provides a predictable resource management to support application performance. In general, reservations can be classified into two categories: *immediate* [10, 7] and *advance* [16, 6, 14, 8] reservations. The former requests reservations at the resource demand time, while the latter requests reservations in advance. The previous reservation approaches focus on CPU, I/O, and network bandwidth only, but does not consider energy.

Energy saving can be achieved by adapting hardware components [15, 12] and/or software applications [5, 19]. None of these approaches support *energy reservation* that allows individual applications to reserve energy for their lifetime. Unlike CPU reservation, energy reservation is also useful even in lightly loaded environments. For example, the meeting reminder needs to reserve energy for its future operation. Otherwise, it may not give the scheduled reminder, since previously-executed applications have used up the battery energy.

To support energy reservation, we need to solve two key problems. First, *what is the admission control for energy reservation*? An application cannot specify its energy demand in the reservation request, because, unlike its CPU demand, its energy demand cannot be predicted via profiling [2]. The reason is that application energy consumption depends not only on its workload but also on the device operating modes (see Section 2.3). Second, *how to enforce energy reservation*? Unlike CPU time, energy is not allocated to individual applications directly. Instead, it is consumed by resources such as CPU and wireless bandwidth that an application needs.

The OS can allocate and account for energy to individual applications by treating energy as a first class resource [11, 20]. This approach can solve the second problem only, and needs to modify the OS structure to add energy management for individual applications. In our work, we take an alternative approach that relies on the predictable CPU resource management. The reason behind our approach is that a predictable CPU management is needed anyway for supporting soft real-time performance.

Specifically, this paper presents a framework, called *ReCalendar*, that provisions QoS via an integrated advance

reservation of CPU and energy in mobile end-systems. ReCalendar consists of two distinctive phases: *calendaring* and *scheduling*. In the calendaring phase, the device user arranges activities and corresponding applications with a calendar. The calendar reserves CPU and energy resources in advance for the calendared applications, by specifying their CPU demands and desired lifetime *only*. Based on such specification, the OS resource manager makes admission control. In the scheduling phase, the resource manager enforces energy reservations via CPU frequency/voltage adaptation and enforces CPU reservations via frequency-aware soft real-time scheduling. In doing so, it provides soft guarantees of CPU and energy to admitted applications.

In this paper, we make two major contributions. First, we propose an energy reservation scheme and integrate it with CPU management. Unlike previous work on advance reservation for high-performance desktop systems [16, 6], ReCalendar aims to assist mobile systems, which impose both new challenges and new opportunities. New challenges lie in the energy constraint. New opportunities arise because a calendar application is typically used to schedule user activities and hence can conveniently provide future knowledge for advance reservations. Second, we have implemented a prototype of ReCalendar and experimentally evaluated it on an HP N5470 laptop with a variable speed processor and with soft real-time applications. Based on the performance evaluation, we show that, compared with previous reservation schemes, ReCalendar provides a better support for soft real-time performance and lifetime of applications.

The rest of the paper is organized as follows. Section 2 introduces models. Section 3 presents the ReCalendar framework. Sections 4 and 5 show implementation and evaluation, respectively. Section 6 compares ReCalendar with related work. Finally, Section 7 summarizes the paper.

2 Models and Assumptions

Table 1 lists symbols defined in this section, though we explain them when we first introduce them.

2.1 CPU and energy model

We consider mobile devices with a single processor that has a set of frequencies $F_{cpu} = \{f_1, ..., f_{max} | f_1 < \cdots < f_{max}\}$. The average CPU power at frequency f is $p(f) \propto C_L f V_f^2$, where C_L is the effective load capacitance and V_f is the voltage at frequency f [3]. Table 2 shows the supported frequencies and the corresponding voltages and relative power of an AMD Athlon CPU as an example [1].

2.2 User activity model

The user of a mobile device often knows how long the battery needs to last (e.g., the duration of a flight or a lecture) before it can be recharged [5]. We refer to this time length as *desired battery lifetime*, T^{life}. We assume that the user schedules her activities and corresponding applications during the desired battery lifetime. For example, a laptop user may use a calendar to schedule tomorrow activities, as illustrated in Table 3. The priority of an activity determines its importance (the higher the priority, the more important the activity); the most important activity should be guaranteed with resource availability first.

Table 1. List of symbols used in this paper.

f	A CPU frequency
$\{f_1, ..., f_{max}\}$	Discrete CPU frequency set
$p(f)$	CPU power at frequency f
T^{life}	Desired battery lifetime
E^{res}	Residual battery energy
T_i^s	Starting time of task i
T_i^e	Ending time of task i
P_i	Period of task i
C_i	CPU cycle demand per period of task i
$B_i = \frac{C_i}{P_i}$	CPU bandwidth (cycles per second) of task i
$B_i^*(t)$	CPU Bandwidth demand of task i at time t
$p_i^*(f(t))$	Energy consumption rate of task i at time t

Table 2. Speed and power of an Athlon 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	36	47	60	74	100

Table 3. An example of calendared activities.

Application	*Starting time*	*Ending time*	*Priority*
Lecture reminder	9:20 AM	9:25 AM	5
Lecture recording	9:30 AM	10:30 AM	4
CD Music playing	1:30 PM	3:00 PM	1
Video encoding	2:00 PM	2:45 PM	3
Video playback	2:00 PM	2:45 PM	2

2.3 Application model

Applications running on mobile devices consist of best-effort applications such as web browser, multimedia applications such as DVD player, and event applications such as reminder. We consider each multimedia application as a *periodic task*. A task releases a job, e.g., a frame decoding, every period during its lifetime. The job has a soft deadline, typically defined as the end of the period. Event applications are a special kind of periodic tasks that release only one job. We aggregate all best-effort applications into one *logical* periodic task (e.g., using a periodic server [18]). Hence, we use a unified periodic task model for all kinds of applications on mobile devices.

The CPU demand of a task i can be characterized with (T_i^s, T_i^e, P_i, C_i), where T_i^s, T_i^e, P_i, and C_i are its starting

time, ending time, period, and CPU cycles per period, respectively. The ratio $B_i = \frac{C_i}{P_i}$ denotes the required CPU bandwidth (cycles per second) of the task i during its lifetime. We use $B_i(t)$ to denote the task i's CPU bandwidth at time t. Clearly, $B_i(t)$ is B_i if $t \in [T_i^s, T_i^e]$ and 0 otherwise, since the task requires no CPU beyond its lifetime.

The starting and ending times are provided by the calendar. The period can be calculated from application performance requirements. The number of requested cycles is between the average and worst-case, both of which can be predicted via measurement-based (profiling) methodologies [2]. We assume that the number of required cycles is roughly constant at different CPU frequencies. This assumption holds for several kinds of multimedia applications since cycles spent on memory stalls are negligible [9].

If a task i occupies the CPU during the time interval $[t_1, t_2]$, its energy consumption during this interval is $\int_{t_1}^{t_2} p(f(t))dt$, where $f(t)$ is the frequency at time t. Therefore, its total energy consumption during its lifetime is $\int_{T_i^s}^{T_i^e} p_i^*(f(t))dt$, where $p_i^*(f(t))$ is $p(f(t))$, if the task occupies the CPU at time t, and 0 otherwise. It means that the energy demand of a task depends on the runtime scheduling (i.e., when, how long, and how fast the task runs) and thus cannot be predicted via profiling.

3 Design and Algorithms

The ReCalendar framework (Figure 1) consists of two major components, a calendar and a resource manager. The resource manager includes a battery monitor, a broker, and a scheduler. The *calendar* acts as a user agent to allow the user to arrange activities. It also requests CPU and energy reservations in advance, on behalf of the arranged applications. Applications themselves may request immediate reservations during the runtime as well[1]. The *broker* makes admission control on each reservation request, based on its CPU demands and requested duration, as well as energy availability provided by the *battery monitor*. The *scheduler* enforces admitted reservations via CPU frequency/voltage adaptation and soft real-time scheduling.

Operationally, ReCalendar supports QoS via two distinctive phases, *calendaring* and *scheduling*. In the calendaring phase (Section 3.2), the calendar requests reservation in advance; the broker makes admission control and returns a handle for an admitted reservation. In the scheduling phase (Section 3.3), applications bind corresponding reservations by using handles passed from the calendar; the scheduler provides them with guarantees of soft real-time performance and lifetime.

[1]Unless specified otherwise, we treat immediate reservations as a special kinds of advance reservations that use the current time and desired battery lifetime as their starting and ending times, respectively.

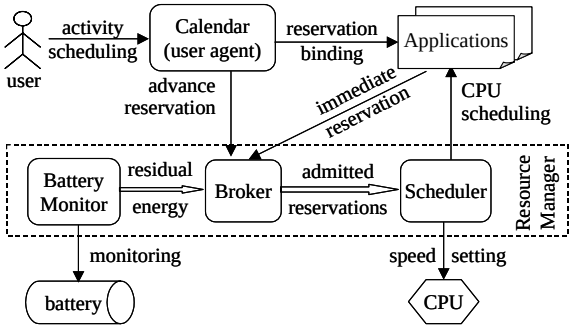


Figure 1. ReCalendar architecture.

3.1 Calendaring with advance reservation

The calendar provides an interface for the user to arrange what applications to run, when to start and stop an application, and how important to run an application. It also reserves CPU and energy in advance for the calendared applications. Specifically, it submits reservation requests in the order of application importance (priority). As the admission control is first-come-first-service in nature, reservations of important applications are admitted first. This is in sharp contrast to immediate reservations, where application requests are submitted in the order of resource demand times, typically when applications start.

3.1.1 Admission control criteria

Each reservation request specifies the CPU demand of the corresponding application, i.e., its starting and ending times, period, and required cycles. For such a request, the broker makes admission test to check if the available CPU and energy resources in the requested duration can meet the requirement. Specifically, assume that there are n admitted reservations; each reservation i (say, for the task i) has parameters $(T_i^s, T_i^e, P_i, C_i), 1 \leq i \leq n$. The admission criteria for a new reservation $(T_{n+1}^s, T_{n+1}^e, P_{n+1}, C_{n+1})$ is:

$$\max_{t \in [T_{n+1}^s, T_{n+1}^e]} \left\{ \sum_{i=1}^{n+1} B_i(t) \right\} \leq f_{max} \quad (1)$$

$$\sum_{i=1}^{n+1} \int_{T_i^s}^{T_i^e} p_i^*(f(t))dt \leq E^{res} \quad (2)$$

where E^{res} is the residual battery energy. Equation (1) and (2) check CPU and energy availability, respectively.

At the time of admission control, Equation (1) can be evaluated based on the requested CPU demand and duration; Equation (2), however, cannot be evaluated since the knowledge of task energy consumption, $\int_{T_i^s}^{T_i^e} p_i^*(f(t))dt$, is unavailable. To evaluate the energy constraint during ad-

mission test, we rewrite it as follows [2]:

$$\int_{T^{min}}^{T^{max}} p(f(t))dt \leq E^{res} \quad (3)$$

where T^{min} is the minimum of all task starting times, and T^{max} is the maximum of all task ending times.

Equation (3) is another view of energy availability — whether the battery can last until all admitted tasks complete their durations. It can be evaluated at the admission control time without future runtime information. The reason is that the CPU frequency, $f(t)$, at any time t can be set to the minimum one that meets the aggregate CPU demand, as shown in Equation (4), and $p(f(t))$ is available from the CPU power-frequency function (e.g., Table 2).

$$f(t) = \min\left\{f : f \in F_{cpu} \text{ and } f \geq \sum_{i=1}^{n+1} B_i(t)\right\} \quad (4)$$

3.1.2 Data structure and operations

ReCalendar uses a right-threaded binary search tree to store admitted reservations. Each node in the tree contains a time, at which a reservation will start or end. Each reservation is represented by two nodes: one for its starting time and the other for its ending time. If several reservations start or end at the same time, one node is used for the corresponding time. Each node also stores the aggregate CPU bandwidth and minimum required frequency at the node time t. The aggregate bandwidth is $\sum_{i=1}^{n} B_i(t)$. The minimum required frequency, $f(t)$, is the lowest frequency that meets the aggregate bandwidth demand (Equation (4)).

Each node α and its in-order successor β form a time interval $[t_\alpha, t_\beta)$, where t_α and t_β are the time of the node α and β, respectively. The energy consumption during the interval is $p(f(t_\alpha)) \times (t_\beta - t_\alpha)$, where $f(t_\alpha)$ is the minimum required frequency of node α. Hence, if we linearly traverse the tree from the earliest node to the latest node (via right threads), we can calculate the total energy consumption of all intervals. Figure 2-(a) shows an example of such a tree.

The admission control for a new reservation involves three steps. First, we use a binary search to insert its starting time node into the tree. We set the aggregate bandwidth of this starting time node to that of its predecessor (0 if it has no predecessor). Second, we use a linear search to insert the ending time node of the new reservation. For each node α in the linear search path, we increase its aggregate bandwidth by the new requested bandwidth, and update its minimum required frequency correspondingly. We also increase the total energy consumption by $(p(f(t_\alpha)) - p(f_{old}(t_\alpha))) \times (t_\beta - t_\alpha)$, where $f(t_\alpha)$

[2]Because $\sum_{i=1}^{n+1} \int_{T_i^s}^{T_i^e} p_i^*(f(t))dt = \sum_{i=1}^{n+1} \int_{T^{min}}^{T^{max}} p_i^*(f(t))dt = \int_{T^{min}}^{T^{max}} \sum_{i=1}^{n+1} p_i^*(f(t))dt = \int_{T^{min}}^{T^{max}} p(f(t))dt$.

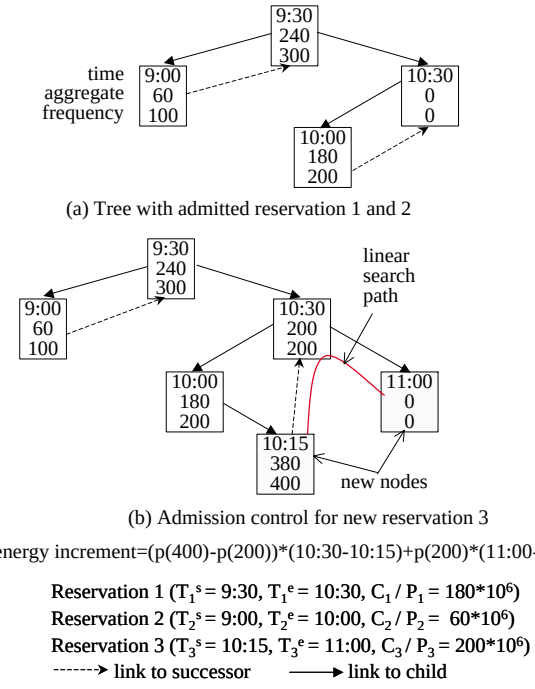


Figure 2. Data structure and operations.

and $f_{old}(t_\alpha)$ are α's updated and old minimum required frequencies, respectively, and t_β is the time of its successor. Finally, if (1) the aggregate bandwidth of all updated nodes is below the highest frequency and (2) the total energy consumption is less than the residual energy, we admit the reservation. Otherwise, we reject it and recover the tree. Figure 2-(b) illustrates the admission control process.

If there are n admitted reservations, the number of nodes in the tree is at most $2n$. Each node contains two pointers to its children or successor, one flag to tell linking to its right child or successor, and three integers (time, aggregate bandwidth, and minimum required frequency). Let us assume four bytes per pointer, four bytes per integer, and one byte for the flag. Each node requires 21 bytes. Hence, the total memory requirement of the tree is $21 \times 2n$ bytes.

The time for inserting the starting time node of a new reservation is $O(\log(2n))$. The time for inserting its ending time node is $O(m)$, where m is the number of nodes in the requested duration. If the admission test fails, the time for tree recovery is $O(m)$. Hence, the time complexity of admission control is $O(m + \log(2n))$.

We also considered other data structures (e.g., lists or segment trees [14, 8]), but found that they either incur large overheads or require a minimum length of requested durations. In contrast, the right-threaded binary search tree has small overheads, and is also flexible to support fine granularity of durations. This flexibility is needed since mobile application durations may be either long (e.g., DVD movie) or very short (e.g., the event reminder).

3.2 Scheduling

During the runtime, the scheduler enforces admitted reservations to provide soft CPU and energy guarantees. To enforce energy reservations, the scheduler linearly traverses the reservation tree established in the calendaring phase. To do so, the scheduler maintains a variable, *current node*, that points to the latest node whose time is not greater than the current time. It sets the CPU frequency (and corresponding voltage) to the minimum required frequency stored in the current node. When the current time is equal to the time of the successor of the current node, the scheduler advances the current node to its successor and adjusts the CPU frequency/voltage correspondingly.

To enforce CPU reservations, the scheduler adopts the EDF-based frequency-aware CPU scheduling algorithm [18]. The algorithm allocates a *budget* of C cycles every period P during the interval $[T^s, T^e]$ to a task, that binds a reservation (T^s, T^e, P, C). It dispatches tasks based on their deadlines and budgets — selecting the task with the earliest deadline and a positive budget. As the task executes, its budget is decreased by the number of cycles it consumes. That is, if a task executes for Δt time units at frequency f, its budget is decreased by $\Delta t \times f$. In doing so, the algorithm ensures that tasks always consume cycles based on their reservations regardless of frequency changes.

4 Implementation

We have implemented a prototype of the ReCalendar framework, using the Red Hat Linux 7.2. The calendar is implemented as a user-level daemon process. The broker and scheduler are implemented as loadable kernel modules in Linux kernel 2.4.7. We have not implemented the battery monitor, because we currently do not have power meters like PowerScope [5]. Instead, we calculate normalized energy consumption based on the relative power in Table 2. We add several new system calls (Table 4) for the calendar and applications to request and demand reservations.

The ReCalendar scheduler does not replace the standard Linux scheduler. Instead, it is added into the Linux *timer task queue*, called `tq_timer`, and hence is invoked every 10 ms. Upon an invocation, it checks the reservation tree and adjusts the CPU frequency/voltage if a reservation starts or stops. It also changes task scheduling parameters according to the frequency-aware scheduling algorithm. Specifically, it sets the scheduling policy of a task to real-time mode `SCHED_FIFO` if the task has a positive CPU budget, and to best-effort mode `SCHED_OTHER` otherwise. It sets the `rt_priority` of tasks according to their deadlines — the earlier the deadline, the higher the `rt_priority`. Hence, the soft real-time task with the earliest deadline and positive budget has the highest `goodness` value, and is dispatched first by the standard Linux scheduler.

Table 4. New APIs for advance reservation.

System call	Description
request_adv_resv	The calendar requests an advance reservation and gets its handle.
demand_adv_resv	A task demands reserved resources via reservation handle from the calendar.
create_srt_task	A task enters soft real-time mode when its advance reservation starts.
exit_srt_task	A task enters best effort mode when its advance reservation expires.
finish_job	A task tells the scheduler that it has finished its current job.
set_life_energy	Set the desired lifetime and initial energy (normalized according to Table 2).

The hardware platform for our implementation and experiments is the HP Pavilion N5470 laptop with a single AMD Athlon CPU [1] and 256MB RAM. This CPU supports six different speeds (Table 2), which can be adjusted by writing the frequency and corresponding voltage to a special register `FidVidCtl` [12, 19].

5 Experimental Evaluation

We use soft real-time, including multimedia and event, applications in our experimental evaluation. Table 5 summarizes these applications and their inputs. For each application, we profile its maximum and average number of cycles using the CPU cycle counter. Unless specified otherwise, we use reservation parameters in Table 5, where requested cycles is between the average and maximum.

The primary metrics for our evaluation include overheads, achieved battery lifetime, and utility. Utility is a quantitative measure on application quality from the user's point of view [13, 19]. We define the utility function of an application as $\frac{J^{com}}{J^{tot}} \times (1-\delta)$, where J^{tot} and J^{com} are its total and completed number of jobs, respectively, and δ is its deadline miss ratio. The factor $\frac{J^{com}}{J^{tot}}$ represents application lifetime, and $(1-\delta)$ represents its performance. We define system utility as the weighted sum of utilities of all applications, i.e., $\sum_{i=1}^{n} \rho_i (\frac{J_i^{com}}{J_i^{tot}} \times (1-\delta_i))$, where we use application priority ρ as its weight. Intuitively, the overall system utility is high if important applications achieve a high performance and long duration.

5.1 Overheads

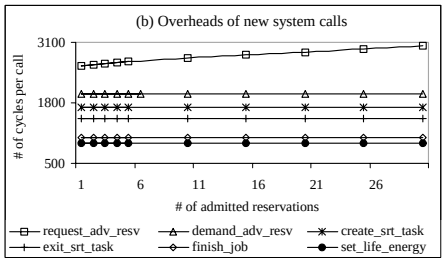
To measure the overhead of a CPU frequency change, we adjust the CPU from one frequency to another one for 1000 times, measure the number of cycles for each adjustment, and calculate the average as the overhead. The results (Figure 3-(a)) indicate that the frequency change overhead

Table 5. Experimental soft real-time applications

Application	Type	Frames	Profiled Cycles		Advance Reservation Parameters				
			Maximum	Average	T^s	T^e	P	C	Priority
mpgplay	MPEG video decoder	7691	30×10^6	9×10^6	290s	600s	40ms	18×10^6	6
reminder	event task	1	320	300	400s	410s	10s	300	5
tmn	H263 video encoder	6000	54×10^6	21×10^6	180s	600s	70ms	35×10^6	4
toast	GSM speech encoder	8132	7.2×10^6	3×10^6	0	250s	30ms	4.5×10^6	3
reminder	event task	1	320	300	195s	200s	5s	300	2
madplay	MP3 audio decoder	16324	4.1×10^6	2.6×10^6	0	500s	30ms	3×10^6	1

(a) CPU cycles (x10 ^{3}) for a frequency change

		to frequency (MHz)					
		300	500	600	700	800	1000
from freq. (MHz)	300		255	305	355	405	505
	500	155		305	355	405	504
	600	155	255		355	405	506
	700	155	256	305		406	506
	800	167	256	306	356		16
	1000	156	256	306	377	14	



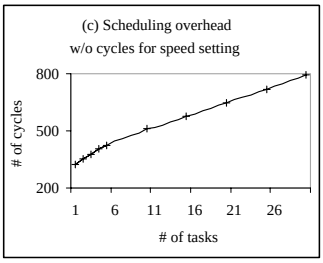


Figure 3. Overheads of CPU frequency adjustment, new system calls, and scheduling.

is small relative to multimedia application execution. We have not measured the overhead for a voltage change, but the AMD documentation [1] shows that it is below $100\mu s$. As described in Section 3.2, the CPU speed is adjusted only when a reservation starts or expires (hence not frequently). Therefore, the overheads of CPU speed adjustment are negligible for multimedia application execution, which is the major workload in our target systems.

To measure overheads of the new system calls (Table 4), we run 1 to 30 `toast` tasks concurrently. Each task has random reservation parameters as follows: T^s and T^e are uniformly distributed within interval $[0, 240s]$, P is uniformly distributed within $[20ms, 100ms]$, and C is uniformly distributed within $[0.1\times10^6, 10\times10^6]$. We measure the number of elapsed cycles (which is independent of the CPU frequency) for each new system call, and repeat the experiment for 40 times to calculate the average. Figure 3-(b) shows that the costs of these system calls are relatively small. Clearly, only the overhead of *request_adv_resv* is dependent on the number of reservations, since it incurs the admission test. Except *finish_job* that is called per job, other five system calls are called only once per task. Hence, the overheads of the new system calls are small relative to the application execution.

Finally, in the kernel, we measure the elapsed cycles for each execution of the ReCalendar scheduler for 10000 times and calculate the average as the scheduling overhead. Figure 3-(c) shows that this overhead is quite small. The scheduling overhead depends on the number of tasks, because the scheduler needs to check the status of each task (e.g., whether it begins a new period).

5.2 Comparison with other schemes

We next compare ReCalendar with two baseline no-reservation schemes and three schemes that support only immediate reservation or CPU advance reservation:

- *Best effort at the minimum speed (BE-Min)*. All applications run in best effort mode. The CPU always runs at the lowest frequency (300MHz in our case).
- *Best effort at the maximum speed (BE-Max)*. All applications run in best effort mode. The CPU always runs at the highest frequency (1GHz in our case).
- *Immediate CPU reservation (IM-C)*. All applications request immediate CPU reservations at their starting time. The resource manager makes admission control on CPU availability only, and adapts the CPU speed based on the total CPU reservation.
- *Immediate CPU and energy reservation (IM-CE)* [19, 17]. All applications request immediate CPU and energy reservations at their starting time. The resource manager checks (i) if the currently available CPU meets the CPU demand, and (ii) if the battery can last for the desired lifetime after increasing the CPU speed for the new request. It also adapts the CPU speed based on the total CPU reservation.
- *Advance CPU reservation (AD-C)*. It is the same as ReCalendar except that the admission control checks the CPU constraint in Equation (1) only.

To evaluate the above schemes, we run the application scenarios in Table 5. A soft real-time task runs in best-effort mode, if its reservation is rejected. We also start a best-effort, computation-intensive program in the background to

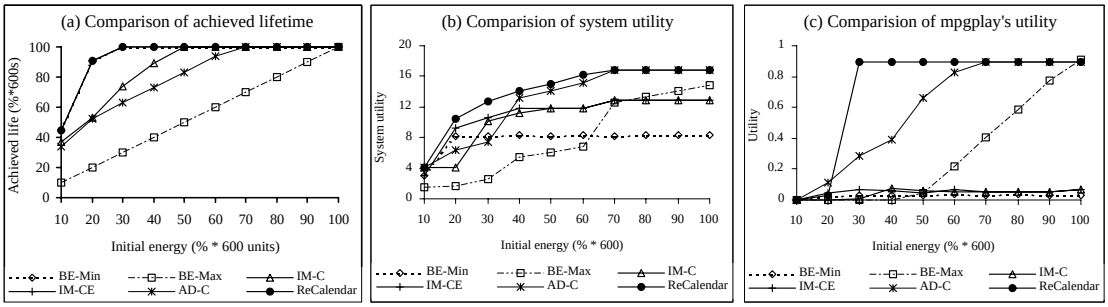


Figure 4. Comparing ReCalendar with other schemes.

compete for CPU with soft real-time tasks. We assume the desired battery lifetime to be 600 seconds. Clearly, if the normalized initial energy is greater than $600 \times 100\%$, then the CPU can always run at the highest speed for the desired battery lifetime. Hence, we perform experiments with different percentage (from 10% to 100%) of the normalized 600 energy units. We measure the achieved battery lifetime, system utility, and utility of the `mpgplay` (the most important application in our scenario). Figure 4 plots all of these metrics. We next use these results to evaluate ReCalendar.

ReCalendar vs. best-effort schemes: Compared with BE-Min and BE-Max, ReCalendar clearly achieves much higher system and `mpgplay` utility, since the former two best-effort schemes do not provide soft real-time performance guarantees. When the initial energy is high, BE-Max achieves high utility (but lower than ReCalendar), because it provides a high CPU performance for all applications to compete for, thus decreasing their miss ratios.

ReCalendar vs. immediate reservation schemes: ReCalendar achieves higher system and `mpgplay` utility than IM-C and IM-CE, because the latter two immediate reservation schemes ignore application importance. In particular, they reject `mpgplay`'s reservation request, thus resulting a very low utility for `mpgplay`. Another interesting result is that immediate reservations may actually hurt important applications that start later (e.g., when the initial energy is high, IM-C and IM-CE achieve a much lower `mpgplay` utility). This implies that advance reservation is desirable to save resources for future important applications.

ReCalendar vs. CPU reservation schemes: Compared with IM-C and AD-C, ReCalendar almost always achieves higher utility, since the former two schemes do not consider the energy constraint. Note that when the initial energy is 20%, AD-C achieves a higher `mpgplay` utility than ReCalendar. The reason is that it decreases the deadline miss ratio of `mpgplay` by providing soft CPU guarantee. This implies that when the battery energy is low, the user needs to trade off application performance and lifetime.

Summary: Overall, our results show that the integrated advance reservation of CPU and energy resources provides better support for application performance and lifetime. In our experiments, ReCalendar provides the highest utility and longest battery lifetime, and significantly outperforms all the other schemes in most cases.

6 Related Work

Resource reservation is a common mechanism to provide temporal and performance isolation for soft real-time applications in a general-purpose open environment [10, 7]. Advance reservations [16, 6, 14, 8] have been proposed to guarantee resources to important applications that demand resources late. Most of the above approaches focus on CPU, I/O, and network bandwidth only, and do not consider energy. In contrast, ReCalendar integrates CPU and energy advance reservation to guarantee CPU and energy availability for soft real-time applications on a mobile device.

Several groups have proposed efficient data structures for advance reservations. For example, Schelen et al. [14] presented a segment tree for advance reservations in Internet. Based on the segment tree, Kim and Nahrstedt [8] proposed a timely adaptive state tree for multi-user environments. These data structures assume a minimum granularity of requested durations. In contrast, ReCalendar uses a right-threaded binary search tree to support any length of durations. This flexibility is needed since application durations on mobile devices are either long (e.g., DVD movie) or every short (e.g., the event reminder).

Dynamic voltage scaling (DVS) is often used to reduce CPU speed and power based on application workload. The workload is either predicted using heuristics [15] (which could violate application timing constraints) or estimated from worst-case CPU time [12, 9] (which are generally too conservative for multimedia applications). ReCalendar integrates the DVS mechanism into the reservation-based resource management, to achieve energy saving of DVS while delivering soft real-time performance guarantees.

Odyssey [4], Nemesis [11], and ECOSystem [20] are closely related to ReCalendar. The goal of Odyssey is to

meet the desired battery lifetime by triggering application adaptation. Nemesis and ECOSystem treat energy as a first class resource. Specifically, ECOSystem explicitly allocates energy to tasks based on residual battery energy. None of them provide soft real-time performance requirements and consider future important applications.

7 Conclusion

This paper presents an integrated advance reservation scheme and prototype implementation, called *ReCalendar*, to provide soft guarantees of CPU and energy resources. The goal is to enable soft real-time applications on mobile devices to achieve a desired performance and lifetime. We have implemented the ReCalendar prototype using the Linux OS. Our experiments have demonstrated that ReCalendar (i) supports soft real-time performance and lifetime of applications under CPU and energy constraints, and (ii) imposes small overheads. In particular, compared with systems that support only immediate reservation or CPU advance reservation, ReCalendar significantly increases application performance and durations as well as battery lifetime.

8 Acknowledgement

We would like to thank the members of the GRACE project, especially Professor Sarita Adve, for informative discussions on energy saving and utility function, and the anonymous reviewers for their constructive comments. This work was supported by NSF grant under CCR 02-05638 and CISE EIA 99-72884, and the NASA grant under NASA NAG 2-1250. 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] AMD. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. http://www.amd.com/products/cpg/athlon/ techdocs/pdf/24319.pdf, Nov. 2001.

[2] 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.

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

[4] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, Dec. 1999.

[5] 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.

[6] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proc. of 7th International Workshop on Quality of Service (IWQoS'99)*, June 1999.

[7] 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.

[8] K. Kim and K. Nahrstedt. A resource broker model with integrated reservation scheme. In *Proc. of IEEE International Conference on Multimedia and Expo 2000*, July 2000.

[9] R. Melhem, N. AbouGhazaleh, H. Aydin, and D. Mosse. Power management points in power-aware real-time systems. In R. Graybill and R. Melhem, editors, *Power Aware Computing*. Plenum/Kluwer Publisher, 2002.

[10] 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.

[11] R. Neugebauer and D. McAuley. Energy is just another resource:energy accounting and energy pricing in the nemesis OS. In *Proc. of 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

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

[13] 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*, Dec. 1997.

[14] O. Schelen, A. Nilsson, J. Norrgard, and S. Pink. Performance of QoS agents for provisioning network resources. In *Proc. of the 7th international IWQoS conference*, June 1999.

[15] 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.

[16] L. C. Wolf and R. Steinmetz. Concepts for resource reservation in advance. *Multimedia Tools and Applications*, 4(3):255–278, 1997.

[17] W. Yuan and K. Nahrstedt. A middleware framework coordinating processor/power resource management for multimedia applications. In *Proc. of IEEE Globecom 2001*, Nov. 2001.

[18] 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)*, May 2002.

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

[20] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proc. of 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.