A Battery Lifetime Guarantee Scheme for Selective Applications in Smart Mobile Devices

Jungwook Cho, Youngjoo Woo, Suntae Kim, and Euiseong Seo

Abstract — *Unpredictable battery lifetime arising from multitasking can have a significant negative effect on availability for mobile devices. In practice, some applications are prioritized and required to remain in operation for certain duration. This paper suggests a battery lifetime guaranteeing scheme for prioritized applications in multitasking mobile systems. The proposed approach profiles and analyzes the battery usage pattern of each task dynamically, and preserves the energy budget for operation of prioritized tasks for a guaranteed time. In addition, this paper proposes an energy-constrained scheduler that limits the energy consumption of tasks while preserving scheduling patterns, which translates to the QoS. The suggested scheme is implemented in a commercial smartphone and evaluated. The evaluation showed that the proposed scheme successfully provides guaranteed operation time for prioritized tasks*[1].

Index Terms — energy-aware scheduling, battery management, mobile operating systems, scheduling algorithms, smartphones.

I. Introduction

Despite the explosive growth in smartphone hardware and software, battery lifetime remains the primary source of user dissatisfaction [1]. The battery capacity of mobile devices, which primarily determines the battery lifetime, is limited by the form-factor and external design of the devices. Therefore, user dissatisfaction with battery lifetime is expected to remain at least for the next few years.

Moreover, continual addition of diverse peripheral components such as global positioning system (GPS), near-field communication (NFC) and ambient sensors to smartphones is broadening the fields for third-party applications. This has resulted in a high degree of multitasking that leads to shorter battery lifetime. The unpredictability of battery lifetime owing to high degrees of multitasking is another source of user dissatisfaction.

Most smart phone systems provide tools for monitoring battery charge and estimating the lifetime remaining for batteries. However, most of the estimation methods currently used are based on a rule-of-thumb approach that divides the remaining battery charge by the energy-draining rate observed for the entire system. This approach cannot correctly predict battery lifetime when applications start and finish spontaneously because these sudden changes affect the battery draining rate and consequently the actual battery lifetime. Despite this unpredictability in battery lifetime, users have virtually no means to control the system operation time except terminating inessential applications, hopefully, to save energy and extend the operation time for other important applications.

Many researchers have attempted to control the energy consumption of each application according to its characteristics and priority. However, existing approaches require redesign of applications so that applications provide the operating systems with information about their energy requirements, and the operating systems change behavior of applications in response to the remaining battery charge. In addition, operating systems have to be modified because they have to deal with the complexity of different energy demands from various applications. Such major restructuring usually requires complicated programming interfaces for precise control of energy usage, and all applications, even insignificant background applications, have to explicitly notify the kernel about their energy requirements through these interfaces.

This paper proposes a scheme that guarantees the battery lifetime for each application in a mobile system. Instead of significant changes to both operating systems and applications, the proposed scheme does not require any changes in existing applications and can be applied with only minor modification of operating systems. The approach provides lifetime-guaranteed scheduling for selected applications and best-effort scheduling for other applications.

A prototype of the suggested scheme was implemented in a commercially successful open-source mobile operating system and was ported to a commercial smart phone for evaluation.

The remainder of this paper is organized as follows. The background and related works are introduced in Section II. Section III proposes the battery lifetime scheme for selected applications in a mobile device. Section IV describes

[1] This research was supported by Basic Science Research Program (2012R1A1A2A10038823) through the National Research Foundation of Korea (NRF), and also by the Core Technology Development of SW-SoC Convergence Platform for Hyper-Connection Services among Smart Devices based on Heterogeneous Multi-core Clusters through KEIT funded by the Ministry of Science, ICT and Future Planning.

J. Cho was with the College of ICE, Sungkyunkwan University, Suwon, Rep. of Korea. Currently, he is a Ph.D. student at the University of Utah, Salt Lake City, USA (e-mail: jmanbal@csl.skku.edu).

Y. Woo is with the College of ICE, Sungkyunkwan University, Suwon, Rep. of Korea (e-mail: mongmio@csl.skku.edu).

S. Kim is with the School of ECE, Ulsan National Institute of Science and Technology, Ulsan, Rep. of Korea (E-mail: makeyoudance@unist.ac.kr).

E. Seo is with the College of ICE, Sungkyunkwan University, Suwon, Rep. of Korea (e-mail: euiseong@skku.edu).

Contributed Paper
Manuscript received 12/22/13
Current version published 03/20/14
Electronic version published 03/20/14.

implementation of the prototype and presents evaluation results, and Section V concludes the research and discusses future research directions.

II. Background and Motivation

Most mobile electronic devices, including laptops and smart phones, predict remaining battery lifetime based on system-wide power consumption history and remaining battery charge. Although this type of simple estimation scheme provides users with information about expected battery lifetime, virtually no commercial systems provide effective means for controlling lifetime [2].

Currently, a representative mobile operating system (OS) monitors use of the following resources to estimate energy consumption:

1) Processors: The OS calculates the processor energy consumption by counting both the amount of processor utilization time and clock frequency in an interval.
2) Networks: Four types of network resources are monitored; phone, radio, Wi-Fi and Bluetooth. Phone and radio are for phone calls and stand-by through 2G or 3G networks, respectively.
3) Screen: The power consumption of the screen varies according to its brightness.

Using this power consumption model, the OS estimates how much energy usage each resource contributes to the overall system energy consumption [3]. However, the model only considers the contribution ratio. Absolute system energy consumption values are obtained from the battery sensor. By aggregating information from these two sources, the OS can calculate the amount of energy each resource and each application consumes. The kernel battery driver notifies the user-level energy management service of every 1% decrease in battery charge, and actual calculations of energy consumption by each resource and each application are conducted in response to these notifications.

Although this typical approach is used in many mobile electronic devices and is useful for predicting remaining battery lifetime, as mentioned before, a mobile embedded OS has no means to control the battery lifetime of the entire system or to guarantee a predefined lifetime for an application.

Applications that run on a mobile system at the same time usually have different lifetime requirements. For example, a movie player is only required for the duration of the movie, whereas the operation time for a mail fetcher is much longer. Some applications may have no lifetime requirements. These applications should be scheduled when the system energy status allows this. However, it is difficult for the system to tell the lifetime requirement of each application and to decide when to kill what applications for securing required battery lifetime [4].

New applications are continually added to a mobile system, and many applications are frequently updated. The same application may even consume different amounts of energy, depending on the underlying hardware. In addition to the hardware, other coexisting applications may affect the energy consumption of an application. Therefore, the amount of energy required for an application is difficult to predict in its design or implementation stage, and a battery lifetime guarantee scheme should perform reactively to dynamic changes in application energy consumption.

Users may run applications spontaneously at any time and thus they should be able to request a battery lifetime guarantee for newly executed applications whenever necessary. A suitable system should allow lifetime guarantee requests as long as the system conditions allow this.

Finally, a battery lifetime guarantee scheme should be easily applicable to existing operating systems as well as applications, and should provide simple and intuitive user interfaces. To meet these goals, the scheme cannot depend on energy usage information provided directly by applications.

Many models for predicting and controlling the battery lifetime of mobile systems have been proposed.

For example, the Odyssey operating system [5] predicts future energy use and remaining lifetime for a mobile system based on the energy consumption history of the system. When the remaining system energy becomes insufficient for the user-requested lifetime, Odyssey requests that running applications reduce their energy consumption by entering a low-power mode. To support this scheme, developers have to design and implement low-power modes for their applications.

Ravi et al. [6] proposed a system for context-aware battery management that warns the user when it detects that the phone battery can run out before the next charging opportunity is encountered. Their approach collects diverse information about user behavior and system status such as battery charging interval, call-time log, current set of active applications and battery discharging characteristics.

The Nemesis operating system [7] is similar to Odyssey in terms of adaptive feedback to application fidelity according to the energy status of the system. Nemesis uses an energy consumption estimation model based on the activities of devices after measuring the energy consumption of activities for each device.

Bellosa et al. [8] proposed an energy-aware scheduler that schedules threads in such a way that they will not exceed their average energy consumption rate. Similar to Nemesis, the energy-aware scheduler estimates the energy consumption of threads based on performance counters in processors. If a thread consumes more energy than is allowed, the scheduler will schedule halt cycles instead of that thread. This approach is partially employed by the proposed scheme in this paper to control the energy consumption rate of each application.

The energy-centric operating system (ECOSystem) [9], [10] considers energy as a first class resource. Each application is given a certain amount of currentcy, a unit of energy provision, periodically. The currentcy of an application is calculated as the amount of energy the application consumes using system resources such as processors, disks or network devices. When an application has no available currentcy, it will not be scheduled.

Banga et al. [11] introduced a resource container that sees a process as a resource principal and accurately accounts for energy consumption for a specific activity carried out by an application. Roy et al. [2], [12] proposed an energy-controllable operating system for mobile devices named Cinder by extending the resource container model.

Previous approaches to guarantee battery lifetime [5], [7] required modification of applications to provide low-power modes in which applications save energy at the expense of service fidelity. In many studies [5], [8], [9], [10], the unit for battery lifetime guarantee was the whole system, and not an individual application. To guarantee battery lifetime for individual applications, existing applications must be modified to explicitly control their energy use via dedicated programming interfaces [2], [12], [13]. Different from the previous approaches, this paper proposes a scheme that guarantees battery lifetime for individual applications without any modification of existing applications. In addition, the proposed scheme in this paper does not require users to be aware of the energy consumption rate or resource usage of the target applications. Instead, users only need to know how long the target applications should last.

III. Selective Battery Lifetime Guarantee Scheme

The proposed lifetime guarantee system monitors the energy consumption of each application and predicts the remaining lifetime by aggregating information about monitored energy consumption rates and current battery charge, as shown in Fig. 1. If the remaining lifetime is shorter than the longest application lifetime requirement, the scheme notifies the kernel so that the kernel scheduler limits the energy consumption rates of applications by scheduling applications restrictively or suspending execution of applications without lifetime requirements.

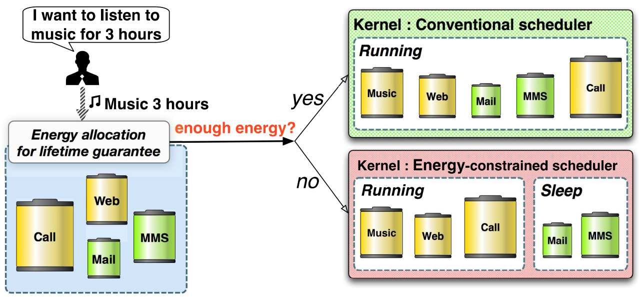

I want to listen to music for 3 hours
Music 3 hours
Energy allocation for lifetime guarantee
Call
Web
Mail
MMS
enough energy?
yes
no
Kernel : Conventional scheduler
Running
Music
Web
Mail
MMS
Call
Kernel : Energy-constrained scheduler
Running
Music
Web
Call
Sleep
Mail
MMS

Fig. 1. Conceptual diagram of the proposed approach.

A. System Architecture

The proposed approach consists of seven components, as shown in Fig. 2. Among these, the battery manager, activity manager, battery device driver and Linux kernel scheduler already exist in many mobile operating systems, and these components were extended to fit the needs of the proposed scheme. The other components were newly added.

The managers belonging to the application framework, including the activity and battery managers, are similar to daemon processes and provide system services. The battery manager monitors the battery status of the system during operation. It periodically retrieves remaining battery charge data given by the battery device driver in the kernel, and analyzes the energy consumption that each application contributes to the overall system consumption based on the scheduling time for each application. The activity manager tracks the scheduling time for each application and provides information to the battery manager. The battery driver is an in-kernel device driver that enables the kernel to control the battery management circuits. It provides information about battery status to the kernel.

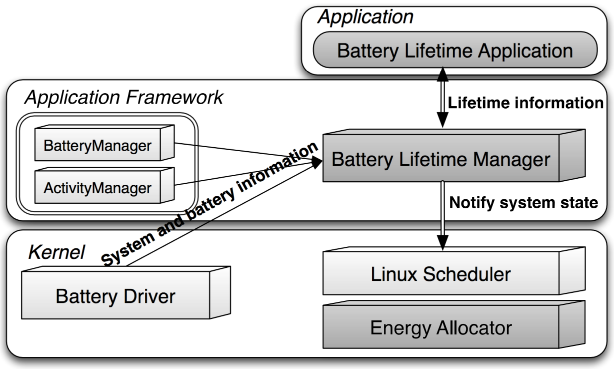


Fig. 2. Components in the proposed scheme.

The battery lifetime manager is the center of the scheme. It monitors the system status and controls energy expenditure and task scheduling according to the system status observed.

There are five system states, as defined in Section III.B, and the system is always in one of these states; Normal, Green, Yellow, Red and Black. The system state determines the task scheduling policy of the kernel. The battery lifetime manager periodically observes the battery consumption rate, and analyzes how much energy each application is consuming through the battery manager and activity manager. The battery lifetime manager also monitors the remaining battery charge. Based on the information collected, it determines the desired system state and changes the current system state accordingly.

When the remaining energy becomes insufficient to last for the farthest application lifetime required, the battery lifetime manager changes the system state to Red and orders the kernel scheduler to enter the *energy-constrained scheduling* mode, which is explained in detail in Section III.D. In the energy-constrained scheduling mode, applications without lifetime requirements are put into the sleep state and are therefore not scheduled, so that they no longer consume energy. Applications with lifetime requirements are allowed to run under energy-constrained scheduling; however, the energy consumption rate of each application is limited to its average energy consumption rate observed so far.

To impose this energy consumption restriction, the battery lifetime manager continually reports the energy consumption rate of each application to the kernel. This information is delivered to the energy allocator in the kernel. The energy allocator determines the energy consumption rate allowable for each application and notifies the scheduler of this when the kernel enters the energy-constrained scheduling mode.

Finally, the battery manager application is a GUI-based application like many other smart phone applications. Users make requests to ensure the lifetimes of applications, or to change or cancel previous lifetime guarantee requests via the battery manager application. The requests are simply given in terms of time such as '1 hour 30 minutes for the music player'. The requests are delivered to the battery lifetime manager to take effect.

B. System Modes

According to the lifetime requirements, remaining battery charge and charging status, the system status is categorized as one of the following five states:

1) Normal: No application requires a lifetime guarantee, or the system is connected to an external power supply such as an adapter. Consequently, no scheduling restriction is enforced in this state.
2) Green: Although some applications require lifetime guarantees, the remaining energy and system energy consumption rate mean that there are no concerns regarding battery lifetime.
3) Yellow: Some applications have lifetime requirements and the system cannot survive to the longest lifetime owing to the current energy consumption rate and remaining battery charge. However, the scheduler may defer the energy-constrained scheduling for a while.
4) Red: The scheduler must enter into the energy-constrained scheduling mode immediately to meet the lifetime requirements.
5) Black: Guaranteeing of the lifetime requirements is not feasible. The system falls into this state when the user newly requests a lifetime guarantee that is not feasible or there are unexpected changes in the remaining charge.

From the viewpoint of the operating system, the Yellow state is not different from the Green state. The operating system kernel does nothing for the lifetime guarantee in either state. However, the Yellow state exists to warn the user that the system will enter the energy-constrained scheduling mode soon because applications without lifetime requirements are sent to sleep under energy-constrained scheduling and consequently the user cannot manipulate these applications until the system returns to the Normal state.

The energy consumption rate of the entire system significantly changes when an application starts to execute or terminates. In addition, some applications may involve enormous changes in energy consumption rates because of their activity characteristics. According to changes in energy circumstances, the system state makes transitions as shown in Fig. 3.

When the smart phone gets connected to an external power supply, the system returns to the Normal state, regardless of what the current state is. The system makes a transition to the Green state from the Normal state when an application requests the lifetime guarantee while the system is not connected to the external power supply.

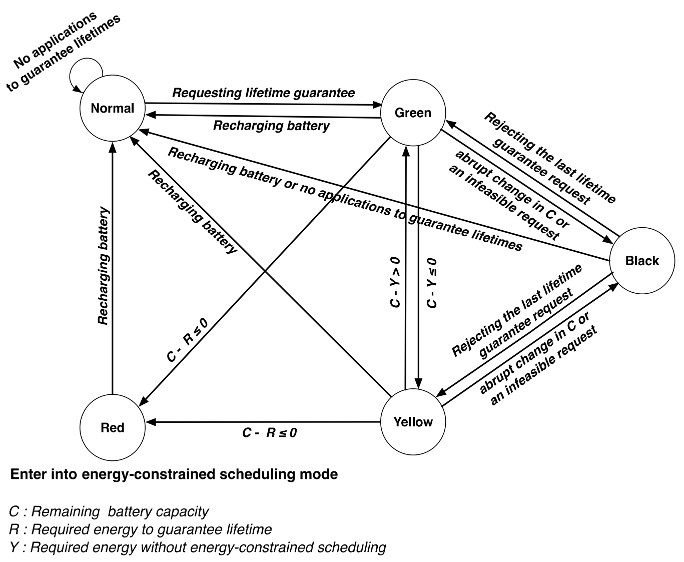

No applications to guarantee lifetimes
Normal
Requesting lifetime guarantee
Recharging battery
Green
Rejecting the last lifetime guarantee request
abrupt change in C or an infeasible request
Recharging battery or no applications to guarantee lifetimes
Recharging battery
Recharging battery
Black
C - Y > 0
C - Y ≤ 0
C - R ≤ 0
Rejecting the last lifetime guarantee request
abrupt change in C or an infeasible request
Red
C - R ≤ 0
Yellow
Enter into energy-constrained scheduling mode
C : Remaining battery capacity
R : Required energy to guarantee lifetime
Y : Required energy without energy-constrained scheduling

Fig. 3. Transition diagram among the five energy management states.

The transition from Green to Yellow occurs when the remaining battery charge is not sufficient for the longest lifetime requested owing to the current energy consumption rate. In the Yellow state, C gradually decreases. Therefore, before the longest lifetime requested is due, the system eventually enters the Red state. When this occurs, the battery lifetime manager advises the kernel via a system call to conduct energy-constrained scheduling.

The lifetime guarantee is surrendered under the Black state. The transition to the Black state is made when a catastrophic battery hardware failure or malfunction occurs, or a user requests infeasible lifetime guarantee in the Green or Yellow state. In the latter case, the transition to the Black state can be thought as an advance notice that the recently added requests are infeasible and should be withdrawn.

When an application with a lifetime requirement temporarily decreases its energy consumption for a short time in the Red state, the energy margin may increase to a positive number. In this case, if the system returns to the Green state, the system lifetime and the quality of service (QoS) for the application that temporarily decreased in energy consumption will be threatened by insufficient energy after the application recovers its energy consumption rate. To prevent this problem, once the system enters the Red state, it remains in this state until either an application with a lifetime requirement finishes or the device is connected to an external power supply.

There are multiple background system services in a mobile device and these usually exist as daemon processes for system management and support of diverse applications such as video play and sound multiplexing. Even when there is sufficient processor time for scheduling applications, they may not work properly if the system services are not satisfactorily scheduled.

Thus, system services are scheduled under all states. By this approach, all system services are guaranteed to be scheduled as they are in the Normal state until the end of the longest application lifetime. As a result, the applications can obtain the same QoS from the system services.

C. Lifetime Estimation

The battery manager tracks the energy consumption of each application. The purpose of this monitoring is not to accurately estimate the remaining battery lifetime, but to approximate the current status of the system. Therefore, the proposed scheme extends the energy consumption rate estimation model of PowerVisor [14].

The activity manager periodically reads the scheduled time duration for each application in the last time interval from the *proc* file system. This information is sent to the battery manager. From this information and the device usage history for each application, the battery manager estimates the energy consumption for each application in a time interval. An application may consist of multiple tasks or threads. In the OS used in this research, all tasks and threads belonging to the same application have the same user ID (UID). Therefore, the system can easily identify the overall energy consumption of an application by adding the energy consumption of all threads and tasks with an UID corresponding to that application. The proposed scheme sees an application, which is a collection of threads and tasks, as the unit of lifetime management.

For more accurate and rapid estimation, the battery lifetime manager initiates the data collection process of the activity manager whenever necessary. The prototype implementation assesses the energy consumption and scheduling information for each application at 1 min intervals. The smart phone, which was used for the evaluation, updates the battery charge register every 3.5 s, with an accuracy of 1.6 *mAh*. Therefore, use of sampling intervals that are too short will induce a significant overhead without improving the accuracy.

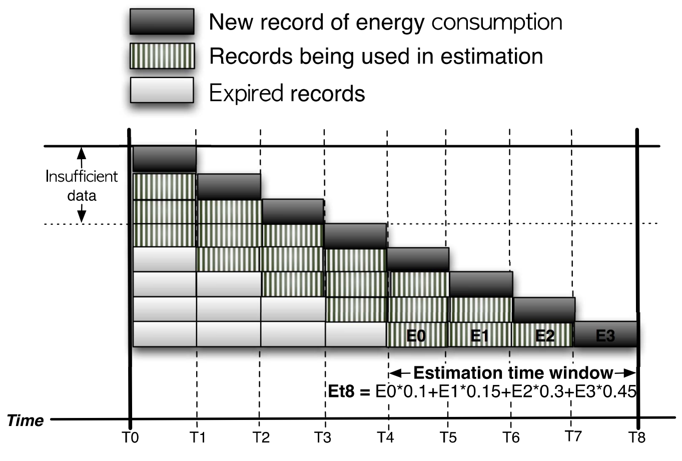

New record of energy consumption
Records being used in estimation
Expired records
Insufficient data
E0
E1
E2
E3
Estimation time window
Et8 = E0*0.1+E1*0.15+E2*0.3+E3*0.45
Time
T0
T1
T2
T3
T4
T5
T6
T7
T8

Fig. 4. Determination of application energy consumption.

An application may show dramatic changes in energy consumption depending on its activities. If the system determines the future energy consumption for an application based on a transiently high consumption rate, the system will enter the Red state prematurely and will interrupt the execution of other applications without any gains. Therefore, the system should not make a transition to the Red state based on a transient increase in energy consumption.

Empirically, it was revealed that sampling and use of the last 4 min of data are sufficient to provide both sufficient accuracy and stability if a greater weight is applied to more recent data, as shown in Fig. 4. This approach is similar to that for conventional DVFS algorithms [15]. However, both the sampling interval and the estimation time window length, which is 4 min in the prototype, may differ depending on the underlying hardware and the purpose of the system.

When an application starts, its energy consumption rate is not determined for 4 min, which is the estimation time window. If the user requests a lifetime guarantee for a newly started application before the first 4 min, the energy consumption of that application cannot affect the system state. Thus, it is possible for the system to enter the Black state immediately after the first 4 min of that application if the application uses excessive energy during this interval. If this situation occurs, the system notifies the user that the lifetime guarantee request is not feasible, rejects the request and returns the system to its previous state.

The display takes significant portion in the overall energy consumption of the system. To identify the display time for each application, the application framework was modified so that it can keep track of the display time of each application by recording the state transition time-stamps for all applications. The modified Battery Manager obtains the information about which applications are on the display at a time point, and delegates the energy consumed by the display to those applications.

Since the battery capacity, or its lifetime, is significantly affected by its discharge rate [8], the battery lifetime is not exactly in inverse proportion to the discharge rate. The remaining charge may decrease faster than the energy consumption rate of the system. However, this nature does not affect the accuracy of the proposed scheme since it measures the energy consumption rate of an application not by the system power consumption, but by the remaining charge value changes of the battery, of which the results come out of that characteristic. In addition, because the model used by the proposed scheme estimates based on the result of the energy consumption, not on the source of energy consumption, it is easily expected that the proposed estimation model performs more accurately than the existing algorithmic resource control or rule-based scheduling techniques [16], which ignore contextual factors [4].

D. Energy-Constrained Scheduler

In the Green and Yellow states, the system preserves energy for applications with lifetime requirements as illustrated in Fig. 5. Applications without lifetime requirements are scheduled using only surplus energy.

As explained previously, the system enters the Red state when the surplus energy is exhausted. Two significant changes in scheduling policy occur in the Red state. First, the scheduler suspends the execution of all applications without lifetime requirements. Next, the scheduler prohibits the applications with lifetime requirements from consuming more energy than they did before the state transition.

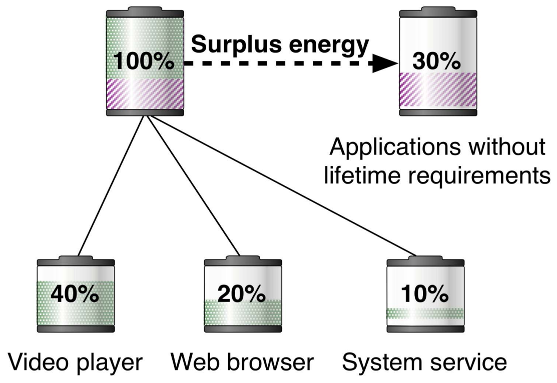


Fig. 5. Resource allocation proportional to applications' energy usage history.

The scheduler prepares a freezing queue for the energy-constrained scheduler. Applications without lifetime requirements are put into this queue in the Red state. The applications in this queue will be popped out of the queue and scheduled normally when the system exits from the Red state.

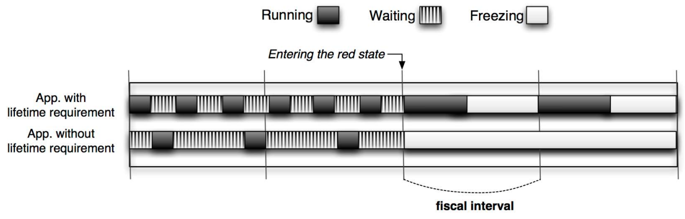
In the Red state, the scheduler limits the energy consumption of applications using a time interval, called a *fiscal interval*, which is usually a few seconds. The amount of energy that an application is allowed to consume in a fiscal interval, which is called the *energy budget*, is determined by the weighted average energy consumption rate obtained as explained in Section III.C.

When each fiscal interval begins, the scheduler distributes energy to applications according to their energy budget for the new interval. Scheduling of an application is allowed as long as its energy allocation remains. Each time the scheduler function in the kernel is invoked, the function calculates the amount of energy used by the current application after the last scheduling point. This is calculated according to the estimation model used to obtain the energy budgets of applications. The scheduler deducts the consumed energy from the energy allocation of the current application. When an application uses up its energy allocation, it is put into the sleep queue and remains there until the next fiscal interval begins. The energy remaining for an application after the previous interval is carried over to the next fiscal interval.

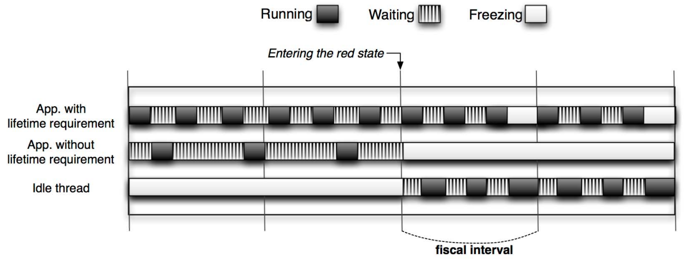
The kernel returns to the normal scheduling policy when the system exits from the Red state. Then applications that are frozen will be removed from the frozen queue and resume their execution.

Implementation of a prototype of the suggested energy-constrained scheduler revealed an unexpected phenomenon. Under the energy-constrained scheduling mode, a multimedia application is supposed to be scheduled with the same amount of energy that it has used under the normal scheduling mode. Consequently, no significant change in QoS is expected. However, the QoS for multimedia applications significantly decreased when they were scheduled by the energy-constrained scheduler. For example, a movie player in the Red state is scheduled with the same energy consumption rate and thus the same processor time as in the Green state. However, the frame-per-second (FPS) rate of the movie player showed a huge decrease after the system entered the Red state.

The reason for the QoS decrease was changes in application scheduling patterns due to the energy-constrained scheduler. As illustrated in Fig. 6(a), the applications without lifetime requirements are frozen and the applications with remaining lifetime requirements are continuously scheduled by the energy-constrained scheduler, whereas execution of all applications is interleaved under normal scheduling. If a multimedia or interactive application uses up its energy allocation in the early stage of a fiscal interval, it will not properly render movie frames or react to user inputs in the later part of the fiscal interval.


Running
Waiting
Freezing
Entering the red state
App. with lifetime requirement
App. without lifetime requirement
fiscal interval

(a) Without compulsory idle spreading.



(b) With compulsory idle spreading.

Fig. 6. Energy-constrained scheduling changes application scheduling patterns.

A movie player typically uses surplus processor cycles for buffering of future frames until the buffer is filled up. As a result, a movie player can use up its energy allocation for buffering only to find that it has no remaining energy to draw down the buffered frames on time. A few different movie players and games were tried, and all of them showed similar phenomena. Zeng et al. [10] also pointed out the possibility of this problem and suggested a conceptual solution named *self-pace*, which delays a task if its energy consumption is ahead of schedule during an interval. However, they did not show any real examples of this behavior, how to delay task execution to match the predetermined energy consumption rate, and the effectiveness of their solution.

To prevent this anomaly, the energy-constrained scheduler must preserve the scheduling patterns of applications as they are in the normal scheduling mode. To accomplish this goal, this paper also proposes a *compulsory idle spreading scheme*.

As shown in **Fig. 6**(b), the suggested scheme schedules an idle thread instead of frozen applications in the energy-constrained scheduling mode, so that the idle thread competes with live applications for processor time, as frozen applications do under normal scheduling. As a result, the scheduling patterns for live applications remain the same under energy-constrained scheduling as in the normal scheduling mode. In fact, an idle thread does nothing but idle.

Therefore, it consumes almost no energy except for a negligible amount for the increased scheduling overhead.

IV. Evaluation

A. Evaluation Environment

As mentioned, the suggested scheme was implemented in an open source mobile embedded OS, and ported to commercial smart phone hardware. Diverse applications were used for the evaluation, as listed in TABLE 1.

TABLE 1
Applications Used in the Evaluation

Name	Descriptions	Required Resource
Movie	Movie play with a media player app.	CPU, Screen and Audio
Game	Automated play of a 3D game	CPU, Screen and Audio
Web	Visiting 16 sites randomly with random thinking time from 500 ms to 13 s	CPU, Screen and Wi-Fi
Music	MP3 file play	CPU and Audio
Socket	Downloading small files from 420 bytes to 4200 bytes with random pause time from 1 s to 10 s	CPU and Wi-Fi
Loop	Repeating simple calculations continually	CPU
Random	Simple calculation for random time intervals from 1s to 10s with random pause times from 1 s to 10 s	CPU

Multiple applications were executed at the same time during the experiments. Some of them requested lifetime guarantees to measure the accuracy of the proposed scheme. Also, the FPS values for the movie player under energy-constrained scheduling were measured to evaluate the effectiveness of the compulsory idle spreading scheme.

B. Lifetime Guarantee Accuracy

Fig. 7 shows the time spent in each state when the system runs three applications concurrently and only one of them required a lifetime guarantee. At the beginning of each experiment run, the remaining battery charge was set to last for approximately two hours without using the proposed lifetime guarantee scheme. The requested lifetimes for the prioritized applications were 150 min. commonly in all experiments.

Unused means the leftover or surplus energy after passing the requested lifetimes. A negative Unused value means that the lifetime guarantee failed. A zero Unused value indicates that the proposed scheme used all energy other than that set aside for lifetime requests to run other applications. Finally, a positive Unused value means that the system tended to enter the Red state prematurely because of conservative energy management. Naturally, the zero Unused value is the ideal case.

A positive Unused value was obtained in every experiment. The amount of the unused energy differed in each experiment and ranged from 0.8% to 16.8%. Investigation revealed that this conservative behavior was caused by energy efficiency improvement due to decrease in the degree of multiprocessing in the Red state. The context switching overhead and the frequency of timer clock interrupts are inversely proportional to the degree of multiprocessing and processor utilization because many modern operating systems employ the tickless kernel architecture [17], which does not generate any timer interrupts during idling. Therefore, energy consumption by the kernel for scheduling and timer interrupt handling significantly decreased in the Red state and thus the overall system energy consumption tended to remain below the expected value. This tendency was stronger without the compulsory idle spreading scheme, as shown in Fig. 7, because the added idle thread induces extra scheduling overhead.

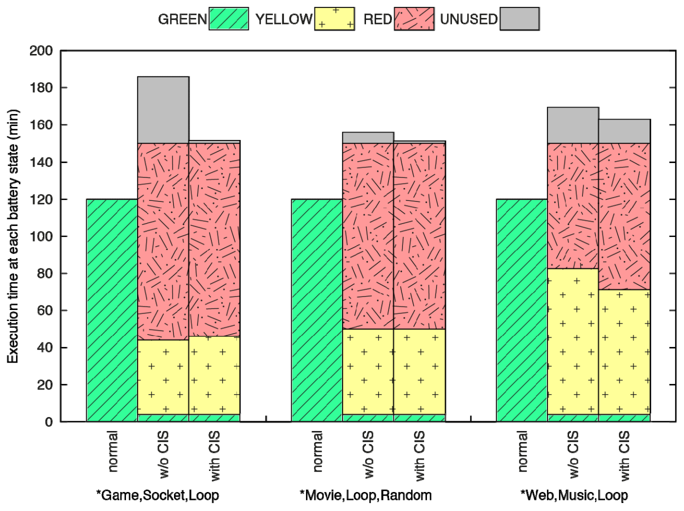


Fig. 7. Accuracy of the suggested schemes when only one application requests lifetime guarantee and the battery can last for approx. 120 min. without the suggested scheme. (Applications with * have lifetime requirements of 150 min.)

Although the system entered the Red state and therefore prohibited the execution of applications without lifetime requirements somewhat prematurely, the proposed scheme successfully guaranteed the requested lifetimes in every trial.

This conservative characteristic was also found in experiments that multiple applications have different lifetime requirements. Fig. 8 shows the results for two out of three concurrently running applications requiring guarantees of different lifetimes according to TABLE 2. The results show that 0.6~1.3% of the total energy remained unused after the last lifetime expired when with the compulsory idle spreading scheme and 4.1%~16% when without the compulsory idle spreading scheme.

TABLE 2
Original and Requested Lifetimes
in the Two-Prioritized-Application Experiments

Applications	Original	Requested
Game,Socket,Random	approx. 120 min.	Game: 132 min. Socket: 100 min.
Movie,Random,Loop	approx. 120 min.	Movie: 150 min. Random: 60 min.
Web,Music,Loop	approx. 120 min.	Web: 150 min. Music: 110 min.

Both Fig. 7 and Fig. 8 show that requesting lifetime guarantee for *Game* yielded significant Unused energy in experiments configured without compulsory idle spreading. Excluding the experiments with *Game*, the Unused energy was below 7.2% even without compulsory idle spreading. This significant Unused energy was caused by the same reason that produced the QoS drop for movie players in energy-constrained scheduling. When *Game* is not timely scheduled, it skips rendering missed frames. Due to the distorted scheduling pattern from energy-constrained scheduling without compulsory idle spreading, frame skipping occurred frequently in the Red state.

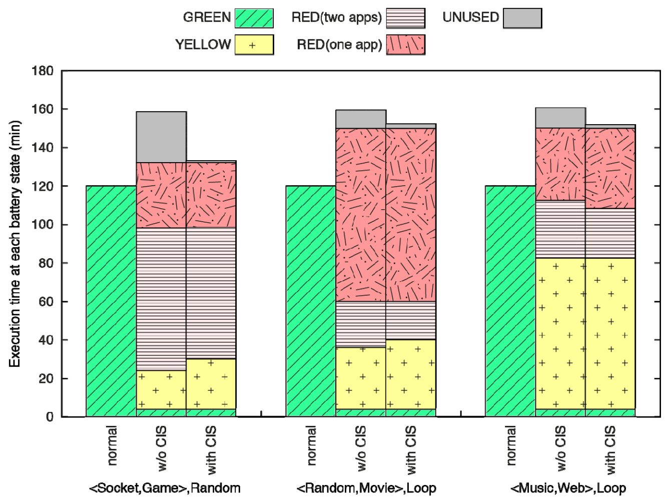


Fig. 8. Accuracy of the suggested scheme when two applications request lifetime guarantee. (Applications surrounded by <> have lifetime requirements.)

Consequently, the amount of computation load for *Game* significantly decreased in the Red state. Because the compulsory idle spreading scheme preserves scheduling patterns of applications, the experiments including *Game* did not yield distinctively large amount of unused energy in comparison to the other experiments when compulsory idle spreading was applied.

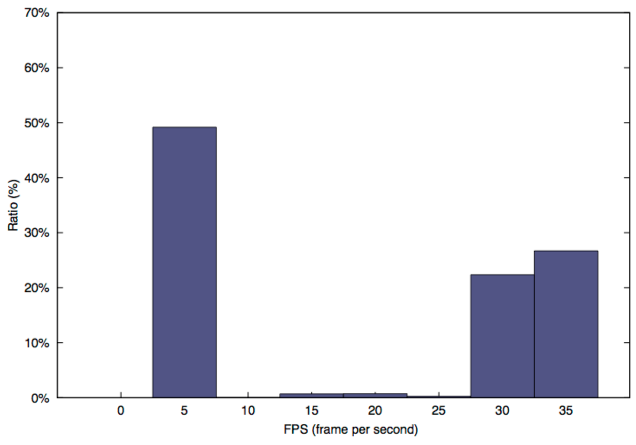
According to the evaluation results, the energy allocator should consider the energy efficiency improvement for frozen applications in the Red state to fully utilize the energy remaining. However, this approach may increase the possibility of lifetime guarantee failure.

C. *Quality of Multimedia Applications*

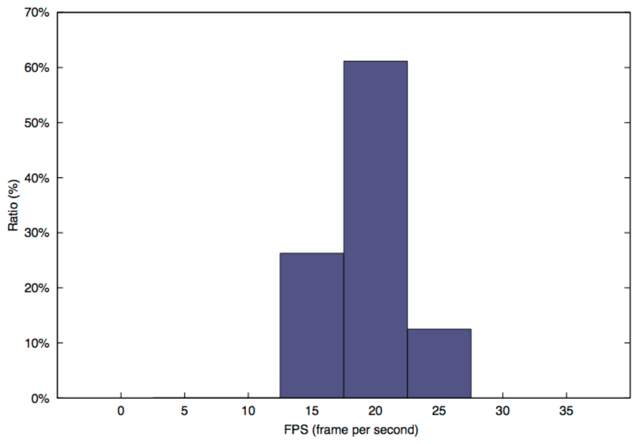
To assess the effectiveness of compulsory idle spreading, a simple tool was implemented that records FPS time series data for the movie player. The movie player requires approximately 50% of processor time to render video without frame drops in the Green state. In the experiments, the system was running a dummy background task consuming approximately 50%~60% of processor time to impose processing load. The system was set to enter the Red state forcibly immediately after the 4-min energy estimation window. The fiscal interval was extended to 2 s to clearly show the FPS changes.

Fig. 9 shows the FPS distribution for movie player operation for 1 h in the Red state. As expected, the FPS values are clustered around both extremes in the absence of compulsory idle spreading in Fig. 9(a). In the first half of a fiscal interval, the frame rate of the movie player was too fast, at approximately up to 50 FPS. On the contrary, in the second half, the frame rate dropped below 10 FPS because the energy allocation was used up early by too much buffering. As shown in Fig. 9(b), the FPS distribution remained stable around 20 FPS, which is similar to that in the Normal state.

Scheduling pattern changes in energy-constrained scheduling had a greater effect on sound quality than on video quality. Without compulsory idle spreading, movie sound was severely fragmented, whereas the sound played as smoothly under compulsory idle spreading as under normal scheduling.



(a) Scheduling without compulsory idle spreading.



(b) Scheduling with compulsory idle spreading.

Fig. 9. Distribution of frame-per-second values under energy-constrained scheduling.

V. Conclusion and Discussion

Currently, energy in a mobile smart device is provided to all applications in a best-effort manner and therefore users cannot predict and control the lifetime of each application accurately.

This paper proposes a scheme that guarantees the battery lifetime required for individual applications in a mobile device. Different from previous approaches, it does not require any modification of existing applications or operating systems. Applications do not need to be aware of remaining battery charge in the proposed scheme. In addition to the lifetime guarantee scheme, this paper proposed a QoS-preserving energy-constrained scheduler.

The evaluation results show that the proposed scheme successfully guaranteed required lifetimes when they are initially identified as feasible. In addition, the evaluation showed that the compulsory idle spreading scheme significantly reduces QoS decreases during energy-constrained scheduling.

The suggested battery lifetime scheme can improve the dependability and reliability of smart phone applications. Consequently, it is expected to improve the product value of smart phones by resolving the significant portion of user dissatisfaction.

The proposed approach, however, is only applicable to and effective for applications without severe fluctuation in energy consumption rates. For example, there may be a sudden phone call, which must be carried out by a prioritized application, after the system gets into the Red state. Future research will focus on resolving this issue by more accurate energy management and energy reservation based on behavior prediction of each application.

References

[1] CFI Group, “CFI Group smartphone satisfaction study 2009,” White Paper, 2009.

[2] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich, “Energy management in mobile devices with the Cinder operating system,” in Proceedings of the 6th European Conference on Computer Systems, 2011, pp. 139–152.

[3] M. Kim, J. Kong, and S. Chung, "Enhancing online power estimation accuracy for smartphones," IEEE Transactions on Consumer Electronics, vol.58, no.2, pp.333–339, 2012.

[4] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice, “Exhausting battery statistics,” Proc. ACM Mobiheld, 2010.

[5] J. Flinn and M. Satyanarayanan, “Energy-aware adaptation for mobile applications,” in Proceedings of the 17th ACM Symposium on Operating Systems Principles, 1999.

[6] N. Ravi, J. Scott, L. Han, and L. Iftode, “Context-aware battery management for mobile phones,” in Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications, 2008.

[7] R. Neugebauer and D. McAuley, “Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS,” in Proceedings of the 8th Workshop on Hot Topics in Operating Systems, 2002.

[8] F. Bellosa, “The benefits of event: driven energy accounting in power-sensitive systems,” in Proceedings of the 9th ACM SIGOPS European Workshop, 2000.

[9] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat, “ECOSystem: managing energy as a first class operating system resource,” ACM SIGPLAN Notices, vol. 37, no. 10, pp. 123–132, 2002.

[10] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat, “Currentcy: Unifying policies for resource management,” in Proceedings of the USENIX 2003 Annual Technical Conference, 2003.

[11] G. Banga, P. Druschel, and J. Mogul, “Resource containers: A new facility for resource management in server systems,” Operating Systems Review, vol. 33, pp. 45–58, 1998.

[12] S. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich, “Apprehending joule thieves with Cinder,” ACM SIGCOMM Computer Communication Review, vol. 40, no. 1, pp. 106–111, 2010.

[13] S.P. Kamat, "Energy management architecture for multimedia applications in battery powered devices," IEEE Transactions on Consumer Electronics, vol.55, no.2, pp.763–767, 2009.

[14] N. Zhang, P. Ramanathan, K. Kim, and S. Banerjee, “Powervisor: a battery virtualization scheme for smartphones,” in Proceedings of the third ACM workshop on Mobile Cloud Computing and Services, 2012.

[15] T. Pering, T. Burd, and R. Brodersen, “The simulation and evaluation of dynamic voltage scaling algorithms,” in Proceedings of the 1998 International Symposium on Low Power Electronics and Design, 1998.

[16] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday, “An architecture for the effective support of adaptive context-aware applications,” in Proceedings of the 2nd International Conference on Mobile Data Management, 2001.

[17] S. Siddha, V. Pallipadi, and A. V. D. Ven, “Getting maximum mileage out of tickless,” in Proceedings of the Linux Symposium, vol. Vol. 2, 2007, pp. 201–208.

Biographies

Jungwook Cho received his B.S. in Computer Engineering from Pukyong National University in 2009, and M.S. in Computer Engineering from Ulsan National Institute of Science and Technology (UNIST) in 2011. He was in the research staff at Sungkyunkwan University. Currently, he is a Ph.D. student at the University of Utah, USA. His research interests are in mobile operating systems, energy management and real-time systems.

Youngjoo Woo received her B.S. degree from Inha University in 2009, and received M.S. degree in electrical and computer engineering at UNIST in 2012. Currently she is a Ph.D. student at Sungkyunkwan University, Korea. Her research interests are embedded systems, power-aware computing, virtualization and cloud computing.

Suntae Kim received his BS degree in electrical and computer engineering at UNIST in 2013. Currently, he is in the MS degree program at UNIST. His research interest covers system software, virtualization, and embedded systems.

Euiseong Seo received his BS, MS, and PhD degree in computer science from KAIST in 2000, 2002, and 2007, respectively. He is currently an associate professor in college of ICE at Sungkyunkwan University, Korea. Before joining Sungkyunkwan University in 2012, he had been an assistant professor at UNIST, Korea from 2009 to 2012, and a research associate at the Pennsylvania State University from 2007 to 2009. His research interests are in power-aware computing, real-time systems, embedded systems, and virtualization.