An Energy-Aware Framework for Dynamic Software Management in Mobile Computing Systems

YUNSI FEI
University of Connecticut
LIN ZHONG
Rice University
and
NIRAJ K. JHA
Princeton University

27

Energy efficiency has become a very important and challenging issue for resource-constrained mobile computers. In this article, we propose a novel dynamic software management (DSOM) framework to improve battery utilization. We have designed and implemented a DSOM module in user space, independent of the operating system (OS), which explores quality-of-service (QoS) adaptation to reduce system energy and employs a priority-based preemption policy for multiple applications to avoid competition for limited energy resources. Software energy macromodels for mobile applications are employed to predict energy demand at each QoS level, so that the DSOM module is able to select the best possible trade-off between energy conservation and application QoS; it also honors the priority desired by the user. Our experimental results for some mobile applications (video player, speech recognizer, voice-over-IP) show that this approach can meet user-specified task-oriented goals and significantly improve battery utilization.

Categories and Subject Descriptors: C.4.1 [**Performance of Systems**]: Modeling techniques; D.m [**Software**]: Miscellaneous; I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*

General Terms: Performance, Management, Measurement

Additional Key Words and Phrases: Software adaptation, energy macromodel, runtime coordination

This work was supported by DARPA under contract no. DAAB07-02-C-P302, and also partly by University of Connecticut Research Foundation large grant 448483.

Authors' addresses: Yunsi Fei, Department of Electrical and Computer Engineering, University of Connecticut, Storrs, Connecticut 06269; email: yfei@engr.uconn.edu; Lin Zhong, Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77005; email: lzhong@rice.edu; Niraj K. Jha, Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08544; email: jha@princeton.edu.

ACM Reference Format:

Fei, Y., Zhong, L., and Jha, N. K. 2008. An energy-aware framework for dynamic software management in mobile computing systems. ACM Trans. Embedd. Comput. Syst. 7, 3, Article 27 (April 2008), 31 pages. DOI = 10.1145/1347375.1347380 http://doi.acm.org/10.1145/1347375.1347380

1. INTRODUCTION

As we enter the era of pervasive computing, users are expecting more ubiquitous services and higher productivity from mobile computing systems. However, mobile computers are constrained by scarce resources, such as small memory, slow CPU, etc. They are specially constrained by limited battery capacity because of the weight/size limits for the battery. In view of the slow battery capacity growth, it is increasingly important to develop techniques to achieve high energy efficiency for such systems. Energy efficiency in this context refers to the amount of service work that the system can accomplish given a battery capacity constraint.

Computer systems provide services to their users through software programs, which demand different hardware resources. Therefore, software and hardware form a pair of `consumer` and `supplier` of resources. From this resource-centered point of view, existing energy efficiency research can be coarsely categorized as follows.

—*Energy-efficient hardware design* techniques optimize the supplier so that less energy is consumed for the same supply of resources. Most processor and circuit power optimization techniques [Rabaey and Pedram 1996; Chandrakasan et al. 2000] fall into this category. These techniques are independent of the upper software system. They reduce the overall power level of the systems, including both the *idle power* when the computer does not perform any useful work and *active power* when the software workload actually executes on the computer.

—*Software optimization* techniques optimize the consumer so that less energy is required for the same software service. Most compilation [Kandemir et al. 2002] and software transformation techniques [Peymandoust et al. 2002; Tan et al. 2003; Fei et al. 2004] fit into this category. They mainly target *active power* during the execution of the software program.

—Another category of techniques *scales the supply according to demand*. All dynamic power management (DPM) and dynamic voltage/frequency scaling (DVFS) techniques belong to this category [Pering et al. 1998; Ishihara and Yasuura 1998; Jha 2001; Pouwelse et al. 2001]. They try to put the hardware components into low-power modes as often as possible. This category also includes OS-directed power management and optimization [Lee et al. 1999; Nahrstedt et al. 2001; Zeng et al. 2002; Benini et al. 2003; Zeng et al. 2005]. The ECOSystem [Zeng et al. 2002; 2005] presents a currentcy model to unify energy accounting over diverse hardware components and embeds this model in the operating system to enable fair allocation of available energy among applications. Each application needs to provide its own currentcy demand model to the OS. The techniques presented in Lee

et al. [1999] and Nahrstedt et al. [2001] use quality-of-service (QoS)-based resource reservation, in which applications pass specific resource demands to the OS for it to reserve the minimum required resources (fraction of CPU cycles, network bandwidth, etc.) for them. Note these techniques may also conversely adapt application QoS based on the resource availability (i.e., supply).

DPM and DVFS techniques have been shown to be quite effective in systems where the available hardware resource is more than adequate for the tasks being run. However, in resource-constrained mobile computing systems, the slow processors, small memory, and limited battery capacity may not always be able to cope with the increasing demands of software. In such cases, there may not be much room for DPM and DVFS techniques to save energy. Therefore, a number of recent works have tried a different approach: *scaling the demand according to supply*.

1.1 Related Work

Scaling the demand usually means reducing the service the application provides to the user. For data-intensive applications, the demand can be reduced by scaling data fidelity, i.e., the input to the application. The Odyssey system [Noble et al. 1997] adopts this approach and provides for a collaboration between the OS and applications to improve performance. Similarly, the Puppeteer system [De Lara et al. 2001] filters the input content through component-based adaptation. The Odyssey system was also extended by Flinn et al. to enable data fidelity adaptation for energy reduction [Flinn and Satyanarayanan 1999]. Shenoy et al. used the same philosophy to transform the requested network data stream to reduce receiving and decoding energy [Shenoy and Radkov 2003]. All these approaches are input data-centric and OS-based.

Another way to scale the demand is to change the QoS, which may imply altering the software computation fidelity instead of data fidelity. In mobile systems, the QoS refers to the service the software programs provide to the user by exploiting various hardware resources. For instance, the human visual perception of a video clip and aural perception of an audio segment define the QoS. An application adaptation framework is discussed in Bharghavan and Gupta [1997] which builds a general framework to communicate with the applications and manipulate the applications, but it does not target realistic applications and specific optimization objectives. The EQoS system develops a framework that can adapt the execution of applications to maximize energy-aware utility [Pillai et al. 2003]. They present several optimal algorithms and heuristics, but the system only targets real-time applications. Recent works [Chang and Karamcheti 2001; Narayanan and Satyanarayanan 2003] provide examples of how the computation fidelity of mobile applications can be altered. However, they do not present a methodology or framework for accomplishing this task automatically and target system delay reduction instead of energy saving. An integrated power-management approach is presented in Mohapatra et al. [2003] that unifies adaptive middleware techniques with

low-level architectural and OS-optimization mechanisms to reduce energy consumption. However, it only targets video streaming application and does not provide a general adaptation framework.

Another issue related to scaling the demand is how to coordinate multiple scalable applications. Competition among multiple applications for the constrained resources needs to be addressed. Efstratiou et al. [2002] demonstrated a platform to enable coordination among applications to avoid conflict. It is a general platform without specific objectives, such as improving performance or conserving energy. It is only targeted at Windows NT-based systems. In Rajkumar et al. [1997], a Q-RAM model is presented to allocate resources to the concurrently competing applications, such that the overall system performance utility is maximized and each application can meet its minimum needs. The Q-RAM approach is extended to maximize energy-aware system utility for battery-powered embedded systems in Park et al. [2003]. These approaches are still OS-based and it is cumbersome to derive accurate utility curve for each competing application and resource consumption curve for each kind of resource. The GRACE project proposes a hierarchical framework to adapt different applications at different system layers (including the hardware, network, operating system, and applications) in a coordinated fashion [Yuan et al. 2003; Sachs et al. 2004]. However, it is mainly restricted to multimedia applications running on wireless systems.

Except for a few works of Flinn et al. [Flinn and Satyanarayanan 1999], Shenoy et al. [Shenoy and Radkov 2003], EQoS project [Pillai et al. 2003], and GRACE group [Sachs et al. 2004], all the previous works target nonenergy related resources. Most investigate software adaptation for communication (network bandwidth) and computation resources (CPU cycles). The work of Flinn et al. in Flinn and Satyanarayanan [1999] actually demonstrates the difficulty of energy-aware application adaptation, since it requires extra equipment for real-time energy measurements and a data-processing computer, which is impractical for mobile systems. Previou work on application adaptation to reduce network bandwidth [Noble et al. 1997] was quite successful, since network bandwidth is relatively easy to estimate. Moreover, most of these approaches need OS support, which is difficult to implement for systems that use a closed-source OS. Therefore, there is a strong need for a practical and general dynamic software management (DSOM) framework for runtime energy optimization.

1.2 DSOM: Opportunities and Challenges

Traditional DPM techniques for computers have mainly exploited various low-power modes of hardware components, such as *active, idle, sleep*, and *off* states for the memory, hard disk, display, and network card [Li et al. 1994; Gauthier et al. 1996; Feeney and Nilsson 2001; Delaluz et al. 2001; Choi et al. 2002]. Analogous to hardware power modes, many software applications also have multiple working states, which correspond to multiple energy/power modes. Therefore, we envision automatic adjustment of adaptable applications to lower energy modes when the battery level is low.

The multiple software working states can be exploited by setting certain runtime parameters, or knobs. These tunable knobs represent alternative algorithms or implementation paths during application execution. The net effect perceived by the user is different output QoS for the same input. Each working state may use a different amount of hardware resources, including CPU cycles, memory, bandwidth, network interface card activation, etc., which leads to different energy/power consumption. The DSOM module should determine on the fly *when* to scale the application and to *what* working state so that an energy saving is achieved at minimal cost in quality.

We believe that DSOM will become an important and complementary approach to the conventional DPM and DVFS techniques in improving energy efficiency, especially for resource-constrained mobile systems. However, it raises several challenges in design and implementation:

1. First, we need to expose the adaptation points and the associated parameters such that manipulations can be performed effectively on the software programs and system. We focus on the adaptation points in software computation instead of data input. Some efforts may be necessary to modify the applications to expose software knobs [Narayanan and Satyanarayanan 2003]. However, many mobile software programs have built-in adaptable points.
2. The communication through the software layers becomes a critical issue. It occurs at the interfaces between the upper application level and the OS layer, in both directions. The question is what information can be generated and passed at each level? Appropriate application-programming interface (API) and efficient communication mechanisms need to be built.
3. The management of the software programs should be directed by certain policies, i.e., when to trigger application adaptations and what low-power mode to select? It is nontrivial to derive an effective policy, which is neither too aggressive in frequently adapting the applications and sacrificing QoS, nor too conservative in maintaining a high QoS level and draining the battery quickly.
4. In order to manage the software programs dynamically, energy estimation for the adaptable applications at each adaptation point should be performed, which requires an accurate and efficient on-line energy estimation module.

1.3 Article Contribution

In this work, we propose an application adaptation and multiple-application coordination framework, DSOM, based on software energy macro modeling techniques [Tan et al. 2002]. It is an *energy-aware* framework that dynamically adapts multiple mobile applications and differs from all the previous related work. The framework contains the following features:

— It utilizes software energy macro modeling and obviates the need for extra equipment for real-time energy measurements [Flinn and Satyanarayanan 1999], which is impractical for handheld computers.

— It is implemented as portable middleware using only POSIX-compliant system APIs. Thus, it requires no changes to the OS.

— It is task-oriented and goal-directed. The user can specify his/her goal in terms of expected task duration or number of tasks and different applications that need to be simultaneously run. The framework automatically finds the best QoS trade-off for the goal, in view of the available energy resource.
— The framework exploits multiple QoS knobs (in software computation) that can be tuned for embedded applications to meet the desired goals.

To summarize, our proposed framework is energy-aware, general, portable, and user-friendly. Our experiments establish its effectiveness for several real mobile applications, such as video player, speech recognizer and voice-over-IP.

The article is organized as follows. In Section 2, we provide background information for this work along with motivational examples. We present the overall DSOM framework in Section 3 and detail the design and implementation of the coordinator and dynamic software manager in Section 4. We present results of experiments conducted on a Linux-based iPAQ with our prototype implementation on several applications in Section 5. Finally, we discuss some limitations of the framework and conclude in Section 6.

2. MOTIVATION AND PRELIMINARIES

In this section, we first motivate, through an example the necessity and efficacy of application adaptation for energy saving. We then present the advantages of application coordination for battery-constrained systems. Finally, we provide the rationale for the management policy in an energy-aware framework for DSOM.

2.1 Motivational Example

We first need to characterize the resource usage (energy) for each working state of the application. Adaptation of applications will be futile if the energy saving is meager. We use a software video player application—*mpeg_play* [MPlayer]—as our motivational example. It is known that changing the data fidelity of an input video clip (by using different lossy compression methods when encoding video clips or using different display window sizes) will induce different energy consumption [Flinn and Satyanarayanan 1999]. However, changing the computation fidelity of applications for energy saving has not been adequately explored. We define a QoS space for the video player application, as shown in Figure 1. It has multiple parameters (dithering method, frame rate, frame display size), where each parameter represents a QoS dimension, which contains a finite set of discrete quality values. The combination of quality values in each dimension is referred to as a QoS point, which is associated with an energy cost. All possible QoS points together form the QoS space of the application. The QoS point in Figure 1 corresponds to the running state consisting of a 20 fps frame speed, 100×100 pixels display size, and monochrome dithering method.

The video player application consists of three functional steps for each frame: decode, dither, and display. Most time and energy are consumed in the first two steps. It is shown in Chakraborty and Yau [2002] that frame display size has little effect on energy consumption. A user may prefer to watch the video clip at a particular frame rate. Therefore, the frame rate is not frequently varied.

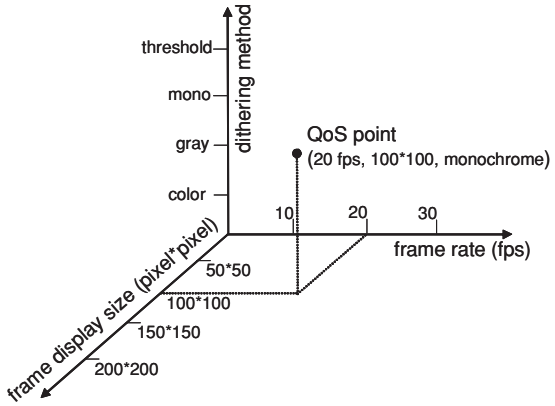


Fig. 1. QoS space for a video player.

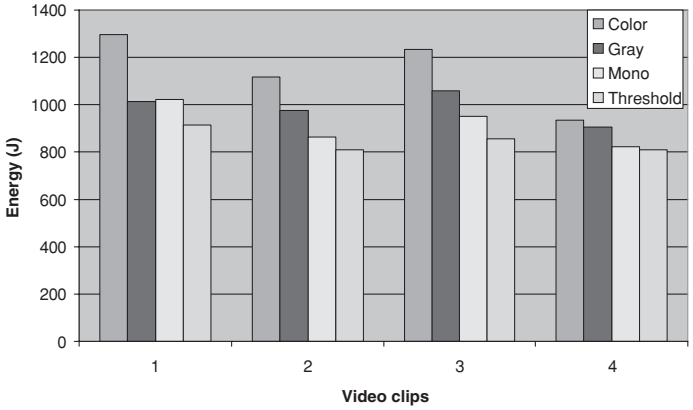


Fig. 2. Energy impact of the dithering method.

Dithering modes tune the color exposition of the frame and is present to us as one effective way to adapt the application. We analyzed various dithering methods and found four modes with human-perceptible differences: *color, gray, monochrome*, and *threshold*, out of a total of 19 modes.[1] Figure 2 shows the measured result of energy consumption of four video clips under different dithering methods on a Linux-based iPAQ 3870. We observed that the average energy saving for the lowest power mode (*threshold*) is 25.3% compared to the original full-color mode (*color*). This is a simple illustration of the impact of changing the computation fidelity on system energy consumption. Note that the QoS of application modes is not necessary to be in proportion to their energy usage. For each application, we need to characterize its specific relationship between energy consumption and QoS.

[1]The visual effect of using these different dithering methods is in the color of video images, for example, *color* refers to colorful exposition of frames, and all the other three modes refer to black and white frames where *gray* represents gray, *monochrome* is an intermediate black and white, and *threshold* represents sharp black and white.

2.2 Coordination among Multiple Applications

In mobile computers, it is common to have several applications running concurrently (e.g., a user may enjoy music using the software MP3 player, while downloading another MP3 file and playing Solitaire at the same time). They compete for the constrained resources of CPU cycles, memory, etc., and, ultimately, battery. It is the responsibility of the OS to assign and manage the resources among multiple applications in a fair way. However, since an OS is unaware of user intention (urgency level of each application), it may treat concurrently running applications equally, and cause all of them to simultaneously abort in the middle of execution when the battery goes down. To avoid this scenario, a user-defined application priority should be considered when a new application joins the system. We propose a coordinator as middleware, which can control the admission of a new application according to its priority and those of currently running ones. The coordinator also decides on adaptations for the newly admitted and other existing applications.

2.3 Policy for Dynamic Software Management

For the concept of DSOM to be useful, a policy is needed to determine *what* and *when* to adapt. A naive way to manage software adaptation is analogous to the time-out DPM technique frequently employed for hardware, where several time thresholds are used for transitioning hardware to lower power modes. One could similarly set a number of energy thresholds for DSOM. The battery energy status could be periodically checked and the appropriate execution mode for running applications selected accordingly. However, this policy is ad hoc, aggressive, and application-independent, and may have the unfortunate side effect of frequently annoying the user. Hence, we propose an application adaptation policy, which is directed by certain user-defined goals, and is oriented for each task, as discussed in detail in Section 3.

In mobile systems, a user may be able to estimate the needed duration for a task, for example, the length of a movie and the length of a conversation with peer mobile users. In such cases, we set the expected duration for the task as the goal. Only when the residual battery energy cannot sustain the goal is application adaptation triggered. Consider the following scenario: the user wants to watch a 60-minute long video clip with the energy consumption estimate of the video clip for the highest quality mode being 3600 J, whereas the residual battery energy is only 3200 J. Suppose the energy consumption under the four dithering modes are 3600, 3400, 3150, and 3000 J for *color*, *gray*, *monochrome*, and *threshold*, respectively. The application will adapt to the *monochrome* mode automatically to meet the goal of displaying the whole video clip, thus providing the highest possible QoS. However, in many cases, where the expected application duration could not be specified, the system will bypass the adaptation engine and always provide best-effort QoS.

3. FRAMEWORK DESIGN

To manage adaptable applications according to the battery status, in order to meet the task-oriented goal, we need an application-coordinating software

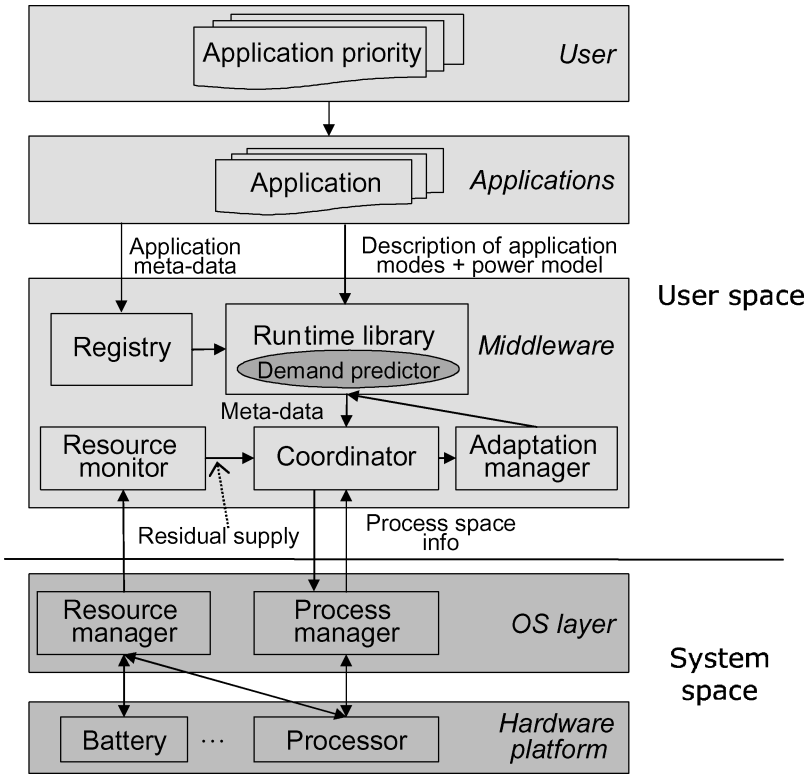


Fig. 3. A coordination framework for application adaptation.

manager in middleware, which fuses information from the application level above and the OS and platform below, and delivers adaptation and coordination commands to either the application or the underlying OS modules. In this section, we explain the design of a user-level DSOM framework geared toward energy savings.

3.1 Overview of the Framework

Figure 3 shows the overall framework for mobile systems. We envision the whole system to consist of several vertically communicating layers, which are categorized into user space and system space. The hardware platform contains a set of resources, such as a processor, memory, display, wireless card, and battery. The resource manager in the OS layer monitors the status of each resource and manages their usage. The process manager controls the creation, execution, and termination of processes, and provides information on running processes. The OS also provides a set of APIs to interact with the user-space modules. Above the OS layer is the middleware that we have developed, which consists of several modules that can interact with either the upper application layer or the OS layer.

In the framework, each scalable application should first register to an application registry with its metadata, e.g., the name of the service and the number

of low power modes. Meanwhile, the description of adaptation modes for each application is stored into a runtime library, along with the power macromodel to estimate the average power consumption for each low-power mode, in order to aid application adaptation.

On top of the middleware is the application and user layers. When multiple applications are running concurrently and competing for resources, the user can specify the priority level for each application if he/she so chooses. For example, when a soldier is using a PDA to talk with his peers in battlefield, and, at the same time, is downloading map information from an encrypted website, he might specify that the voice-over-IP application has higher priority over the network browser, or vice versa. For different instantiations of the same registered application, the user can specify a different priority level.

The resource monitor module in our framework polls the OS resource manager and calculates the residual battery energy value. Based on the application metadata, application-specific configurations and priority, residual battery energy value, and information on running processes obtained from the process manager, the coordinator performs admission control and adaptation arbitration. Thus, an appropriate application configuration is selected for the newly joining application, and coordination is carried out among the other running applications. We describe the coordinator in detail in Section 3.3.

3.2 Task-Oriented Goal-Directed Software Management

We consider three different cases for task-oriented software management and set up corresponding goals and policies to perform DSOM, as discussed next.

In the first case, altering application modes only changes average system power while the execution time remains the same. This case applies to a synchronized video player. The frame rate determines the sum of processing time (including decoding, dithering and displaying) and synchronization time for each frame. The length of a video clip can be calculated by the number of frames and frame rate beforehand. The task-oriented goal is set to duration of task (e.g., length of a movie). At the configuration and adaptation point, we detect the initial battery energy and calculate the expected average system power (supply). We also evaluate the power consumption for each mode (demand) by the demand predictor. We then select the adaptation level with average system power just under what is sustainable by the battery energy level. In this way, the goal is met with the least QoS degradation.

In the second case, the execution time is variable while the average system power is constant for different application modes. Our goal is now set as maximizing the number of tasks executed. The more tasks that complete execution with the same energy constraint, the higher the battery utilization. A speech recognizer falls into this category. When we make a slight sacrifice in speech recognition QoS, we gain in terms of execution speedup (thus also reducing system energy).

In the third case, both the power and execution time vary for different application modes. The goal now targets both task duration and number of tasks. The system should finish the execution of all the tasks before their

deadlines. If the system runs out of battery before the application finishes execution, or if the application execution does not complete before its deadline, the goal is not met, and the system degrades to provide a "best-effort" service.

Each registered application can be put into the above three categories, with its corresponding goal specified in the runtime library. The coordinator prompts the user for this information to make an adaptation and coordination decision.

The pseudocode for application QoS level configuration is shown in Algorithm 1. The initial residual energy ($ener$) and the user-specified time goal ($goal$) determine the expected average power consumption (P_{avg}) for the specified duration (line 3 in Algorithm 1). At each adaptation point, the coordinator evaluates the service for the adaptation block (line 7), estimates the average power ($\vec{P}$) and execution time ($\vec{T}$) for each QoS level, and selects the best QoS level to meet the user-specified goal (lines 8–16). We employ a history window-based power budget regulation method for the QoS level configuration. The average power of the selected QoS level should be just below the upper limit set by the minimum value between $\alpha * P_{avg}$ and $2 * P_{avg} - P_{last}$, where α is a constant value larger than 1, e.g., 1.1 is used in our experiments, which determines the laxity of the adaptation policy, and here we set the history window size at 2 and P_{last} is the average power for the last adaptation block. The algorithm tries to keep the average power value of two consecutive adaptation blocks below P_{avg} so that the user-specified duration goal can be met. The service ends either when the task goal is met, or the specified duration deadline is reached, or when the residual energy drops to zero (the conditions are described at line 6). The first case represents a successful completion of the application, while the others represent a failure. At the end of the experiment, the larger the residual energy, the more conservative software management may be, and the QoS level can be set higher.

Algorithm 1 ($* service\_command(name, goal, priority, \vec{param}) *$).

```
1.  initialize_service(name);
2.  detect_residual_energy(ener);
3.  P_avg ←ener/goal;(* average system power for the specified duration *)
4.  P_last ←0;(* average power for the last adaptation block *)
5.  T_elapsed ←0; i ←0;(* the elapsed time and task index *)
6.  while (i < n and T_elapsed < goal and ener > 0)
7.      do evaluate(Block_i, P⃗, T⃗, Num); (* estimate the power/time vector for number
           of Num different QoS levels *)
8.         P_upper ←min(α * P_avg, 2 * P_avg − P_last);
9.         if (P[num − 1] > P_upper)
10.           then level ←num − 1;
11.           else
12.                for j ←0 to Num
13.                    do if (P[j] < P_upper)
14.                          then level ←j;
15.                               break;
```

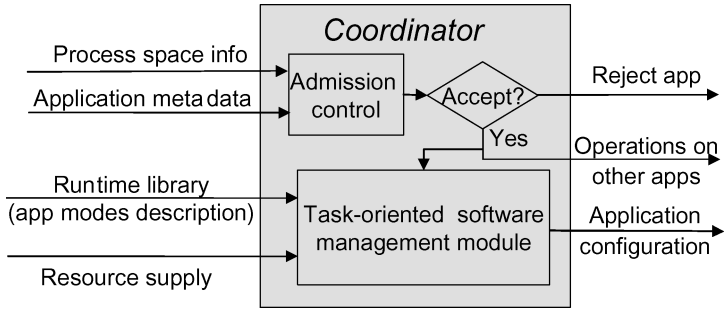


Fig. 4. The design and flow of the coordinator.

16. $P \leftarrow P[level]$;
17. execute($Block_i$, P, $level$);
18. $P_{last} \leftarrow P$;
19. i++;
20. detect_residual_energy($ener$);
21. report_time($T_{elapsed}$);

The demand predictor is also implemented in the runtime library and estimates the energy demand of each software mode at the adaptation point. This module is application-specific and is described in detail in Section 4.2.

3.3 Coordinator Design

Figure 4 depicts the design and flow of the coordinator. First, the metadata of the newly joined application and the process space information are given as input to the admission control unit. The application metadata should contain the priority level and number of power modes. The process space information obtained from the process manager covers the running processes and their association with applications. We employ a simple preemption and reservation policy for coordination. When there is any application running with a higher priority than the new application, the battery energy is reserved for the higher-priority applications, and the remaining energy is assigned to the new application for appropriate adaptation. Otherwise, it is admitted and other applications with lower priority yield the resources. Several fallback actions can be invoked for these yielding applications: suspend, abort, or rollback, which is explained in Section 3.4.

If the new application is admitted, it is evaluated under the constraint of current residual battery capacity and an appropriate configuration mode is obtained. Each admitted adaptable application has a supporting runtime library, which contains the description of various application modes and energy macromodels for both the demanded QoS and other QoS levels. By default, the demanded QoS of an application is assumed to be the highest quality level. If the demanded QoS energy is larger than the residual battery energy, the DSOM module triggers an adaptation for the application. If even at the lowest power mode, the residual battery energy is not sufficient, the application is forced to

launch at the lowest power mode. The application's configuration is delivered to the adaptation manager, which invokes the application with an appropriate mode from the runtime library.

Algorithm 2 describes this coordination policy in detail. Applications are classified into two categories, according to their user-specified priority. When a service with a high priority is called (line 2), we check the service list that contains the concurrently running applications (lines 4–8). The applications with a low priority yield the resources to the new service (line 6) and other high-priority applications keep running and competing for the battery with the new one. When the new service has a low priority (line 9), we still check the service list (line 10). If there are high-priority applications running (line 11), we reserve battery energy for them and the remaining energy is assigned to the new service for appropriate adaptation (line 12). Otherwise, each low-priority application views itself as the only one using the battery and all of them compete for the battery energy on a fair basis (line 14).

Algorithm 2 (∗ *Coordinator*(*servicelist*, *newservice*) ∗).

```
1.  prior←(newservice→priority);
2.  if prior=high
3.    then service←servicelist;
4.         while service ≠NULL
5.           do if (service→priority)=low
6.                then send_pause_signal(service→children);
7.              service←(service → next);
8.              evaluate(newservice, current_battery);(∗ evaluate the appropriate
                working state, competing with other high-priority applications ∗)
9.    else
10.        flag← search(servicelist);
11.        if flag=high
12.          then evaluate(newservice, remain_energy);(∗ energy reserved for high-
                  priority applications ∗)
13.          else
14.              evaluate(newservice, current_battery);(∗ compete with low-priority
                  applications ∗)
```

3.4 Scalable Adaptation Block

In our framework, the QoS level is negotiated between the application and coordinator before the application is launched. We define an *adaptation block* to be the block of application code that consists of a set of alternative sequences of execution, each associated with a different QoS level. The adaptation block can be the whole application when all the software knobs are global for the whole program. In other cases, sequential application execution can be divided into more than one adaptation block, each with its local software knobs. When the application has to yield to others, it may take one of three fallback actions. It may be suspended, aborted, or rolled back. A *suspended* application resumes at

```
struct component_t {
  char                  *servicename;
  void                  *component_handle;
  int                   levels;   //number of QoS levels
  int                   ac_level; //to indicate if AC-powered
  struct unit_t         *startunit;
  int                   numunits;
  register_t            register_handle;
  expand_t              expand_handle;
  evaluate_t            eval_handle;
  execute_t             exec_handle;
  struct component_t_t  *next;
};
```

Fig. 5. The data structure of *component*.

the original QoS level at the suspension point. An *aborted* application is dropped from the system. Under *rollback*, an application detects which adaptation block it is in, and when it resumes, it rolls back to the beginning of the adaptation block, renegotiates with the coordinator, determines the new quality level for the adaptation block, and executes at the new QoS level. Such scalability in adaptation block size is a characteristic of adaptable applications, which is not the focus of our work. Our main contribution is building a general energy-aware DSOM framework. Therefore, we currently assume the whole program to be an adaptation block and tune the global software knobs, although we recognize that this approach can be scaled down to finer-grained adaptation blocks with local knobs.

4. FRAMEWORK IMPLEMENTATION

We have built a prototype user space DSOM framework in middleware. The key components are the runtime library, which provides the adaptation configuration calls to the applications, and the coordinator, which negotiates/renegotiates with applications and assigns configuration modes or fallback operations to each application. The framework is implemented as a daemon server in user space, the communication between the applications and the coordinator is implemented in a client-server fashion [Pouwelse 2003] and the communication between the coordinator and processes is implemented with signals. More details of implementation can be found in Fei [2004]. We illustrate next each salient feature of the framework.

4.1 Registry

Each adaptable application needs to register its metadata in a registry, and is allocated a data structure called *component*, as shown in Figure 5. The name for the service provided by the application is assigned to the *servicename* entry of *component*. For example, we give the service name "video" to the MPEG-1 video player, *mpeg_play* [MPlayer], "voip" to a voice-over-IP application, *RAT* [RAT], and "speech" to a dynamic neuron-network based speech recognizer, DNN [Zhong et al. 1999]. The application also specifies the number of low-power modes that it can support to the *levels* entry of *component*.

```
struct service_t {
  char                  *name;
  int                   priority;//user-specified priority
  int                   verbose;
  int                   goal;  //user-specified goal
  double                energy;//energy estimation for this service
  double                left_energy; /*remaining battery energy
                                from high-priority reservation*/
  double                lastpower;  //average power for last unit
  struct component_t    *component;
  struct parameter_t    *parameters;
  struct child_t        *children; //the process space
  struct service_t      *next;     //for the link list
  struct service_t      *prev;     //for the link list
};
```

Fig. 6. The data structure of *service*.

For each service instantiation, we allocate a data structure called *service* with the specified name, which is illustrated in Figure 6. It inherits all the metadata from the registered component with the same service name, and appends some more parameters. For example, when the user launches a service and the associated application programs, he/she can specify the urgency level by filling the *priority* entry in the service object. Also, an entry called *goal* is reserved for the user to specify either the expected task duration or the workload expectation.

4.2 Runtime Library

The runtime library provides the functionalities required by the application to interact with the coordinator. Interaction between the application and the coordinator is structured as follows. The coordinator (on the server side) listens to the requests from clients at a default port. Upon startup (a client issuing text commands), the server receives the request, parses it, and executes corresponding commands. When the user launches an application, he/she issues a command specifying the service name, service priority, and execution goal, etc. The coordinator loads the corresponding runtime library for the application, which evaluates energy/power/execution time for all possible QoS levels, along with other application-specific information and handlers. The coordinator negotiates with the new application and other running applications and makes admission control, adaptation configuration, and fallback operation decisions. The runtime library can be viewed as a wrapper for each application. When an adaptation configuration is decided upon, it is passed on to the wrapper through the adaptation manager, and the application executed at the negotiated QoS level.

An important module in the runtime library is the energy estimator for an application at different QoS levels. The Odyssey system [Flinn and Satyanarayanan 1999] takes an on-line power/energy profiling approach, which requires extra measurement hardware and a computer. This is not appropriate for portable systems. Instead, we use software energy macro modeling to predict required system energy. Previous work [Krintz et al. 2002] has tried to predict program power using some application-level software tools. Currently,

we embed an application-specific energy estimation module in the runtime library. Based on the prebuilt energy macromodel, the application inputs, and other specific information, energy prediction is performed for all the QoS levels and used by the coordinator to make a decision. We briefly describe the energy macromodel for each application—video player *mpeg_player*, voice-over-IP *RAT*, and speech recognizer *DNN*.

4.2.1 *Energy Macromodel for the Video Player.* An MPEG-1 video stream consists of three frame types: *I frame* (intracoded), *P frame* (predictive-coded), and *B frame* (bidirectional-coded). The video stream is played at a fixed frame rate. In each frame period (inverse of the frame rate), a frame is decoded and displayed on the screen (note that the decoded frame and the displayed one may not be the same). It takes several steps to process each frame: *parsing*, *inverse discrete cosine transformation* (IDCT), *reconstruction*, and *dithering* [Mitchell et al. 1996]. Among these steps, the first three are CPU intensive and frame-type dependent (each frame type requires a different type of processing) and can be grouped as a *decode* step, while the *dithering* step is memory intensive (requires data movement between the processed video stream and the display frame buffer) and frame independent (i.e., it is independent of the frame type). Thus, we can divide the frame period into several functional processes: decode, dither, display, and idle. Hence, the energy consumption of the video player can be obtained from Equation (1).

$$E = \sum_{i=1}^{n} (P_{decode} \cdot T_{decode,i} + P_{dither} \cdot T_{dither,i} + P_{display} \cdot T_{display,i} + P_{idle} \cdot T_{idle,i}) \tag{1}$$

where n denotes the total number of the frames in the stream, P_{decode}, P_{dither}, $P_{display}$, and P_{idle} represent the average power consumption for each step, and $T_{decode,i}$, $T_{dither,i}$, $T_{display,i}$, and $T_{idle,i}$ represent the time spent in each step in frame i, respectively. The sum of execution times for each step in a frame period, T_{period}, satisfies the relationship in Equation (2).

$$T_{period} = T_{decode,i} + T_{dither,i} + T_{display,i} + T_{idle,i} \tag{2}$$

We first need to characterize the energy macromodel for the video player. The standard regression analysis method is adopted for this purpose. We ran a set of test programs, measured the total energy consumption and execution time for each step, and calculated the average power consumption coefficients, such as P_{decode}, P_{dither}, $P_{display}$, and P_{idle}. Note that for this application, we tune the software knob corresponding to dithering. Therefore, we obtain a vector for P_{dither} with four different values for the *color, gray, monochrome*, and *threshold* modes. Similarly, we also obtained a vector with twelve values for frame type-dependent P_{decode} and $P_{display}$, respectively.

When using the energy macromodel for each application mode, we need to estimate the execution times T_{decode}, T_{dither}, $T_{display}$, and T_{idle}, in each frame. The only available input is the video clip. A software package that comes with this application, *mpeg_play*, contains a module called *mpeg_stat*, which can extract some statistical information for the video, such as the frame rate, encoding

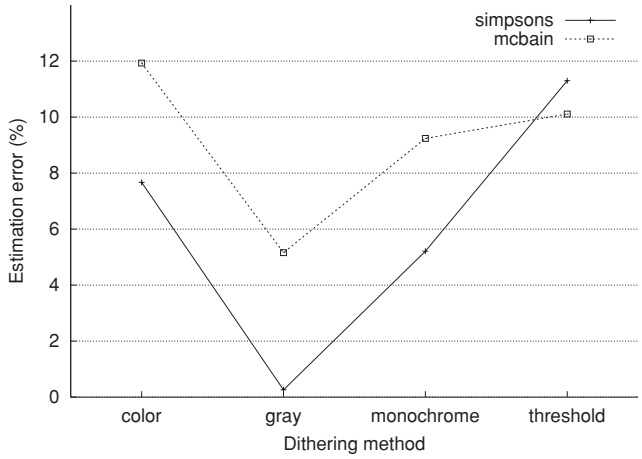


Fig. 7. Accuracy evaluation of energy macromodels for different QoS levels.

resolution (in pixels), frame structure, type, and size (in bytes) of each frame, etc. We derived relationships between the processing time and video characteristics for each processing step. It has been shown in a prediction model [Bavier et al. 1998] that the decoding time of each frame is linearly dependent on the frame size:

$$T_{decode,i} = \alpha_{j_i,k} + \beta_{j_i,k} \times X_i \tag{3}$$

where X_i is the frame size in bytes, j_i denotes the type of frame (I/P/B) of frame i, k represents the type of dithering method taken, and $\alpha_{j_i,k}$ and $\beta_{j_i,k}$ are constants. The dithering time and display time are linearly dependent on the frame resolution and independent of frame size and type. Equations (5) and (5) show this relationship.

$$T_{dither,i} = \zeta_k + \eta_k \times W \times H \tag{4}$$

$$T_{display,i} = \mu_k + \nu_k \times W \times H \tag{5}$$

where W is the horizontal size of a frame in pixels, H is the vertical size, and ζ_k, η_k, μ_k, and ν_k are constants dependent on the dithering method.

For each frame, the idle synchronization time can be calculated from Equation (2). For some mobile computer systems, the calculated idle time may be less than zero, which shows that the computer does not have enough computing capability to run the video at the given frame rate.

The impact of our energy macromodel and execution-time estimation techniques is shown for two video clips, each with four dithering methods (Figure 7). In this figure, the absolute energy estimation error is plotted for different dithering modes. The error is calculated with respect to measured results on an iPAQ. The overall error is under 12% with the average error being 7.6%, which is acceptable for on-line application adaptation.

4.2.2 *Energy Estimation for* RAT. We use a similar procedure to estimate energy consumption for a voice-over-IP application, *RAT* [RAT]. The application can execute under different audio quality settings, which contains a set of

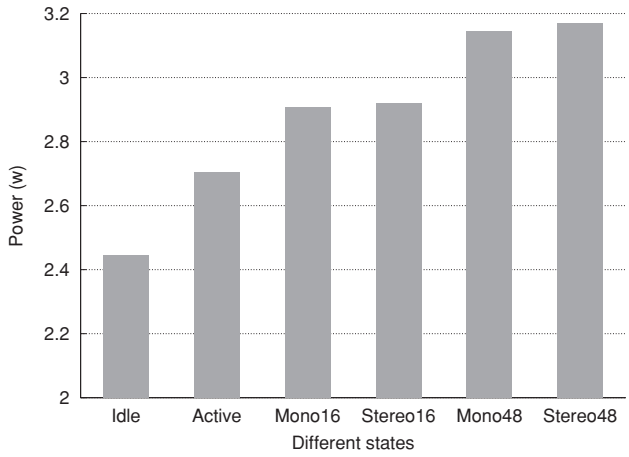


Fig. 8. Variation of power for different states.

parameters, such as mono or stereo channels, audio sample rate, received sample conversion quality, audio encoding algorithms, channel transmission encoding algorithms, etc. These parameters and their corresponding values compose the QoS space for *RAT*. For different QoS points, there are different active sending and receiving power consumptions.

Note that *RAT* is a real-time interactive system and there also exists idle time between sending and receiving audio. Thus, we need to consider the idle power as well. If a user specifies the duration goal for the conversation, two time values are considered: total duration and an approximate conversation time (assuming the sending and receiving times are equal). The energy macromodel is as shown in Equation (6).

$$E = P_{send} \cdot T_{send} + P_{receive} \cdot T_{receive} + P_{idle} \cdot T_{idle} \tag{6}$$

Figure 8 shows the average power measurement on the iPAQ for sending audio, when running *RAT*, under different settings and QoS levels. *Idle* represents the wireless card-on state, with *RAT* not running; *Active* is the state after launching *RAT*, but without audio communication. The other four states indicate different sampling rates and channel options. For example, *Mono16* uses a mono channel at a sampling rate of 16 KHz, while *Stereo48* uses a stereo channel at a sampling rate of 48 KHz. We observe that power consumption is more sensitive to the sampling rate than the channel option. The power differences among the states can be exploited for energy saving by choosing a lower sampling rate.

4.2.3 *Energy Estimation for the Speech Recognizer.* The dynamic neural network based speech recognizer, DNN [Zhong et al. 1999], is an application with rich configurations (adaptation opportunities) for the neural network. It contains two integrated parts: training and recognition. At the front end, a set of speech utterances is used to extract speech features for training proposes, and the neural network parameters are stored in a file. At the back end, when

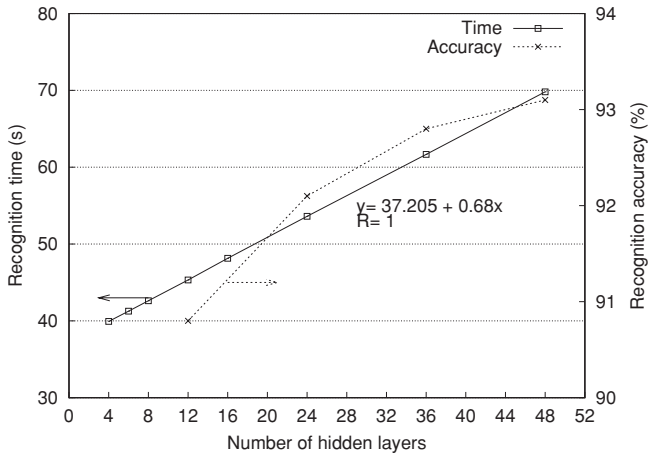


Fig. 9. Variation of recognition time/quality for different parameter values.

a new utterance is provided as input, the recognizer loads the neural network parameters and outputs the recognized text for this utterance.

There are a number of tunable parameters for the neural network structure, such as the time window size for the hidden layer, size of the input feature window, number of hidden layers, etc. Figure 9 shows the impact of one tunable parameter, number of hidden layers, on the recognition time. It shows a linear relationship: with an increase in the number of hidden layers, the recognition time, and corresponding energy (power remains roughly constant) increase. Meanwhile, increasing the number of hidden layers also improves the speech recognition accuracy, i.e., QoS level.

5. EVALUATION OF THE COORDINATION FRAMEWORK AND TASK-ORIENTED SOFTWARE MANAGEMENT

In this section, we establish the efficacy of our framework in meeting required goals and increasing system energy efficiency. We first discuss the experimental setup and then present the experimental results.

5.1 Experimental Setup

To validate our goal-directed application adaptation framework, we used the three applications described in Section 4. However, for brevity, we describe the experimental setup using the video player application. We evaluated our framework for 24 video clips and used each complete video clip as an adaptation block. QoS adaptation is achieved through a change in the dithering mode. We used an iPAQ HP H3870 (with Intel StrongARM microprocessor SA-1110 at 206 MHz, 64-MB DRAM and 32-MB flash), running under Familiar Linux, as our evaluation platform. The iPAQ H3870 can use AC or DC power. The battery that supplies the latter is the Danionics lithium-ion polymer battery (#DLP 345794). Its capacity is 1400 mAH, and its voltage range is specified as 3.7 to 4.3 V. For our experiment, we used an initial energy value of 2876 J. As an illustrative example, for a particular video clip (*simpsons*), the battery lasts 22

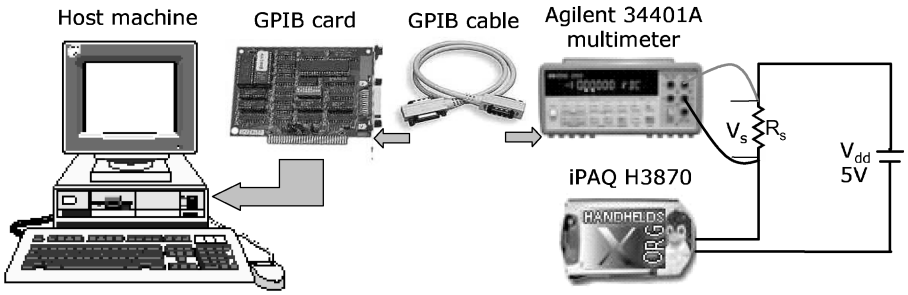


Fig. 10. Power measurement setup.

iterations, when the video player is at the highest QoS level, and 32 iterations at the lowest QoS level. This represents a 45.5% extension in battery lifetime. We selected a small initial energy value for experimental convenience. When extrapolated to the full capacity of the battery, we can run 179 iterations at the highest QoS level and 261 iterations at the lowest QoS level.

Since the current advanced power management interface in Familiar Linux can report residual battery capacity only at a very coarse-grained level (the basic unit is 1%), we used an external power supply with the battery removed to obtain the system energy dissipation (note that this measurement is done only for validation purposes, the framework actually uses the software energy macromodels described earlier for demand prediction).

For our experiments, we built a power measurement setup, as illustrated in Figure 10. The measurement setup consists of a host computer running certain data acquisition software such as Excel with Agilent IntuiLink tool bar add-in for multimeters [Agilent], an IEEE488 GPIB PCI card for high-speed data acquisition installed into the host machine, a GPIB cable connecting the measurement instrument, an Agilent 34401A digital multimeter to the card, iPAQ H3870, a small series resistor ($R_s < 100$ mΩ), and the DC power supply (5 V).

By this approach [Farkas et al. 2000], what we measure directly is the voltage across the sense resistor R_s, which is connected in series with the DC power supply. The current consumption of the iPAQ system can then be computed by dividing the voltage drop across the series resistor by the resistance value. The resistance value is sufficiently small that it can be considered nonintrusive, i.e., the voltage drop on the iPAQ is approximately equal to the supply voltage ($V_{dd} = 5\ V$). Thus, the instantaneous power consumption of the iPAQ is:

$$P_s(t) = V_{dd} * \frac{V_s(t)}{R_s} \tag{7}$$

The data-acquisition software, IntuiLink, adds a tool bar to Excel. After configurations on the GPIB card and multimeter, each voltage value measured by the multimeter is automatically logged into a spreadsheet. The GPIB card enables the sampling rate of measurement to be as high as 1000 samples per second. The add-in of Excel offers a sample rate around 12/s. After measurement data are collected for a software program, we get the system power profile

of the program running on the iPAQ. We identify the starting point (T_1) and finishing point (T_2) of each program, and integrate the power consumption over the execution time. Thus, the energy consumption value of the software program is obtained, as shown in Equation (8). We can also sample the power value points and the energy dissipation rate profile is drawn.

$$\begin{aligned} E_s(t) &= \int_{T_1}^{T_2} P_s(t)dt \\ &= \sum_{i=0}^{n-1} P_s(t_i) * (t_{i+1} - t_i) \end{aligned} \tag{8}$$

where t_0=T_1 and t_n=T_2 represent the starting and the finishing point of the software program, respectively.

To evaluate our coordination framework, we designed several scenarios with a new application joining the system that contains other concurrently running applications. The priority of the new application and other existing applications determines their coordination policy. The results are described next.

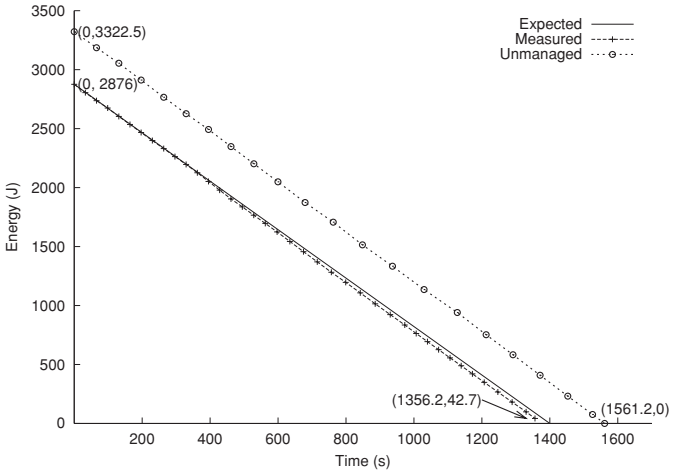
5.2 Experimental Results

In this section, we first present the experimental results on adaptation of single applications, then we evaluate the coordination framework.

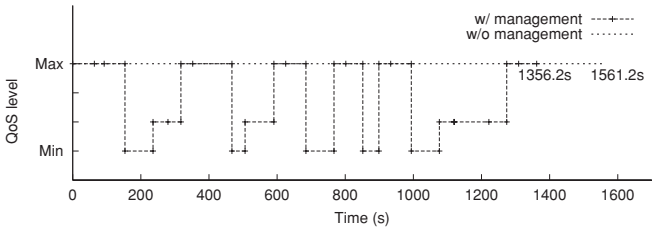
5.2.1 *Adaptation of Single Applications.* In our framework, when issuing a service command, the user should provide the execution goal together with the service name, and other service-specific parameters. For example, the typical video player service command looks like "`service name=video goal=1400 priority=0 filelist=/home/user/playlist`." Here `service` refers to command type, `name` indicates which service to call (the eligible service must be in the registry), `goal` specifies the expected duration in seconds, and `priority`, the urgency level. The *mpeg_player* also needs the list of video files, which contains the number of tasks n. Each user represents a client. The command is issued by user in a text message and passed to the *server* side through the client–server interface. When the *server* receives a command, it first parses the command. If it is a service request, the coordinator on the server side determines the application configuration (QoS level) and delivers it to the application.

Figure 11 shows the detailed result of an experiment with the goal set to 1400 s and for executing 24 video clips.[2] The *measured* line in Figure 11a shows how the battery energy supply changes over time, i.e., the energy-dissipation rate of *mpeg_player*. The *Expected* line connects the user-specified duration goal point (1400,0) on the x axis and the initial energy point (0, 2876) on the y axis. It represents the expected battery energy-dissipation rate to meet the duration goal. We can see that the measured line tracks the expected line very well and the goal is met with a battery residual energy of 42.7 J, which is only 1.5% of the initial energy. It shows that our DSOM framework is not conservative at all

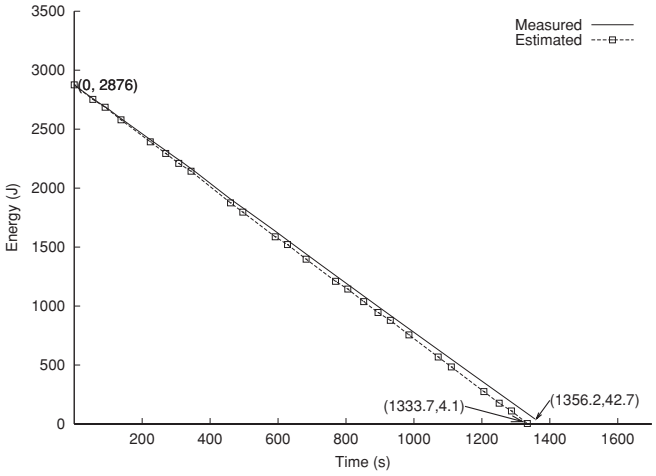
[2]Note in our experiments, the battery is removed from the PDA, the initial battery energy is set to a fixed value in the DSOM framework and we measure the energy-dissipation rate.



(a) Energy-dissipation rate of video player with and without goal-directed DSOM



(b) Variation of QoS in the two cases



(c) Comparison of estimated and measured energy-dissipation rates

Fig. 11. Application adaptation of the video player.

Table I. Knobs and Effects of Application Adaptation

Applications	Tunable parameter	Resource	Range of variation	Possible energy saving (%)
Unsynchronized video player	Dithering mode	CPU time for dithering Power	1.34–24.93 ms 1.5–3.6 W	30.2
Speech recognizer	Number of hidden layers	CPU time for recognition	39.92–69.81 s	54.1
Voice-over-IP	Sampling rate of audio device	Power	2.90–3.21 W	9.1

for this case. The third line, *Unmanaged*, represents the energy-consumption rate without DSOM. It starts from another initial energy value (3322.5 J) and drains the battery when finished. It can be seen that, in this case, the system needs an extra 446.5 J (15.5% of original battery residual energy 2876 J) of energy to finish the task in 1561.2 s. Thus, without DSOM, the goal cannot be satisfied.

Figure 11b shows how the software application adapts with and without software management during execution. We consider four different QoS levels, based on the dithering mode, as described earlier. Different video clips, even at the same QoS level, may have different average power. Hence, we see very frequent adaptation of the video player for clips from line w/ management. In general, adaptations of separate adaptation blocks are independent of each other. Thus, the user is not likely to be annoyed by the changes. The other line w/o management shows that the application keeps running at the highest QoS level, however, it takes longer to finish the list of tasks. We can conclude that the video player that we are using is unsynchronized.

To evaluate the accuracy of our energy macromodel, we compared the estimated and measured energy dissipation rates in Figure 11c. Our energy macromodel predicts that the total energy consumption is 2871.9 J and the video player runs for 1333.7 s. The estimation is very accurate, with the error in total energy consumption being only 1.4% and the error in total execution time only 1.7%.

We performed similar experiments on the other two applications—speech recognizer and voice-over-IP. Table I shows the tunable parameter we selected for each application, the resource that changed under different modes, the range of resource variation, and the possible energy saving for the same task from the highest to the lowest QoS level. Note that the adaptation policy and goal for each application differ slightly according to the associated variable resource. For the voice-over-IP application, *RAT*, the goal is set to the expected conversation time. Under the constraint of battery residual energy, we configure the appropriate running environment for *RAT*, and launch it with the corresponding audio device sampling rate. We have trained the speech recognizer with different numbers of network layers and stored the network parameters into different dictionary files. For the same set of utterances, the recognition time is shortened greatly by using a smaller dictionary corresponding to fewer network layers. Thus, the energy consumption is reduced greatly and the recognition rate is increased. Our experimental results show that the applications can meet their

user-specified goal only with goal-directed DSOM, and the battery utilization can be improved greatly.

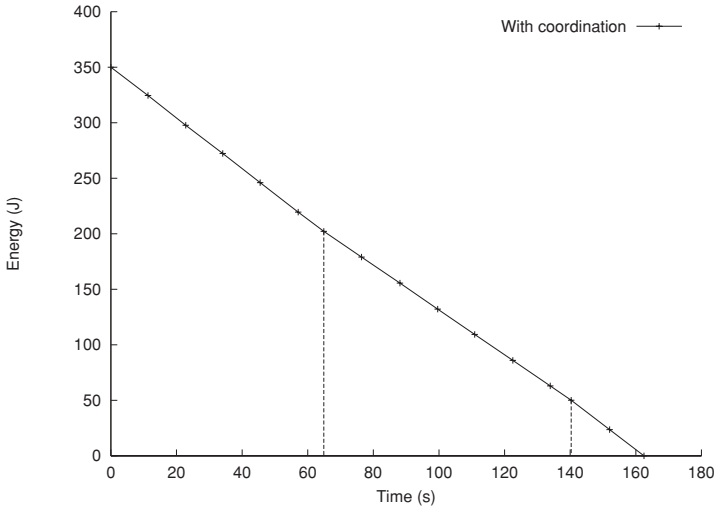
5.2.2 *Coordination among Multiple Applications.* We performed another set of experiments to evaluate the efficacy of coordination among multiple applications. In our experiments, we turn off the iPAQ backlight, thus, the base idle energy consumption of the system is a very small portion of the overall application energy consumption. Therefore, we assume that the energy expended to run two applications concurrently is equivalent to the sum of the energy expended to sequentially run the applications.

The coordinator acts as a user-level coarse-grained scheduler. It is energy-aware and enables multiple applications to run efficiently. It favors urgent applications and prevents other low-priority applications from competing. We perform two case studies to evaluate the efficacy of the coordinator.

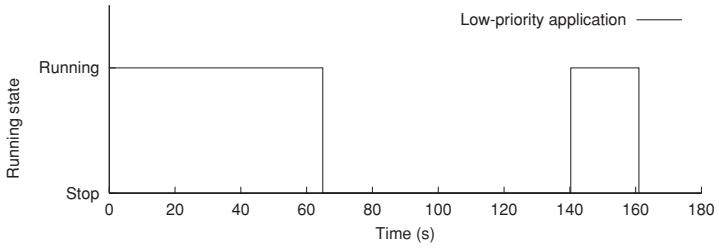
I. *Case study of a high-priority application joining the system with a low-priority application.* We first consider the following scenario. The user is watching a video clip using the video player, which is a low-priority service. In the middle, an urgent request comes for recognizing a set of speech utterances. Figure 12 shows the experimental result under this scenario with system coordination. Figure 12a shows the energy-dissipation rate. The starting and finishing time points for the speech recognizer are also marked on the figure. We can see from Figure 12b that the existing low-priority video player yields to the new high-priority speech recognizer when the latter arrives at time 64.9 s. Figure 12c shows that with coordination, the speech recognizer can finish under the battery energy constraint. The video clip is not able to finish, even though the video player resumes execution when the speech recognizer finishes. We guarantee the execution of the more urgent application first.

If there is no coordination, multiple applications simultaneously compete for the battery. Figure 13 represents the experimental results in this case. Figure 13a shows the energy-dissipation rate and the starting point for the speech recognizer. Because of competition from the existing video player application, the speech recognizer does not finish, as shown in Figure 13c, neither does the video player. We also notice that without coordination, the battery drains faster, even though it starts with the same residual battery energy.

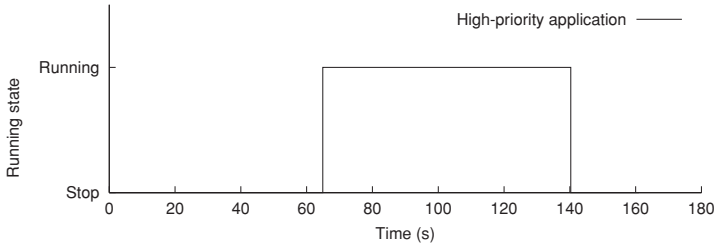
II. *Case study of a low-priority application joining the system with a high-priority application.* We performed another similar experiment in which the existing video player application is of high priority while the newly joining speech recognizer is of low priority. When the speech recognizer joins the system, the system knows the currently running QoS level of the video player and its estimated energy from the energy macromodel. With coordination, the system reserves the energy for the video player, thus executing the speech recognizer at the fourth QoS level instead of first (the highest). Both the video player and speech recognizer complete their execution. Figure 14 shows the experimental results under this scenario. Figure 14a presents the energy-dissipation rate. It shows that both the applications finish executing within the battery energy constraint. Figure 14b indicates that execution of the



(a) Energy-dissipation rate of the two applications



(b) Running state of the existing low-priority application

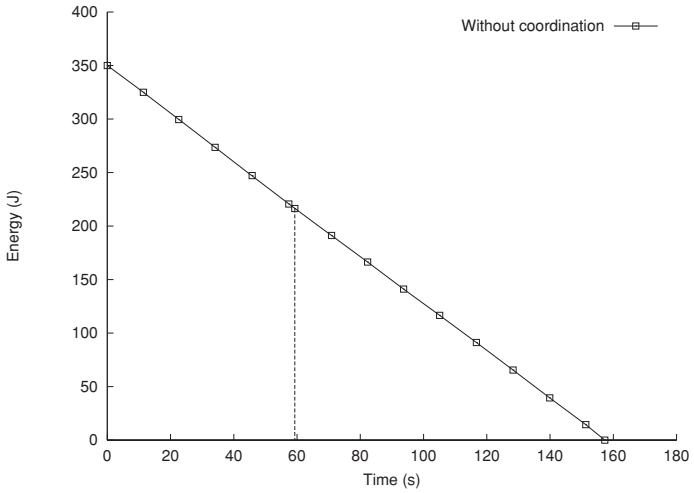


(c) Running state of the newly joining high-priority application

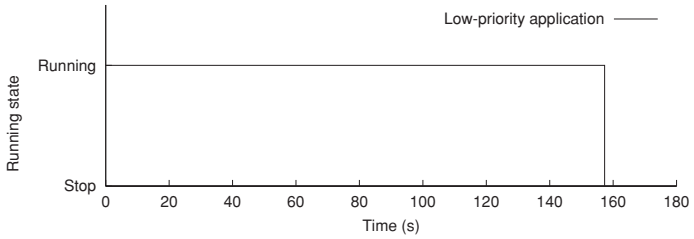
Fig. 12. System coordination when a high-priority application joins the system.

existing high-priority application is not affected by the joining of the low-priority application. Figure 14c shows that the newly joining application will run at the minimum QoS level, since it needs to yield resources for the already existing high-priority applications.

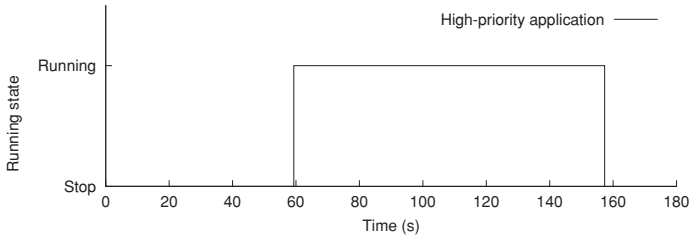
Without coordination, the speech recognizer checks the residual battery energy and views itself as the only one that is using the battery. Thus, it is executed



(a) Energy-dissipation rate of the two applications



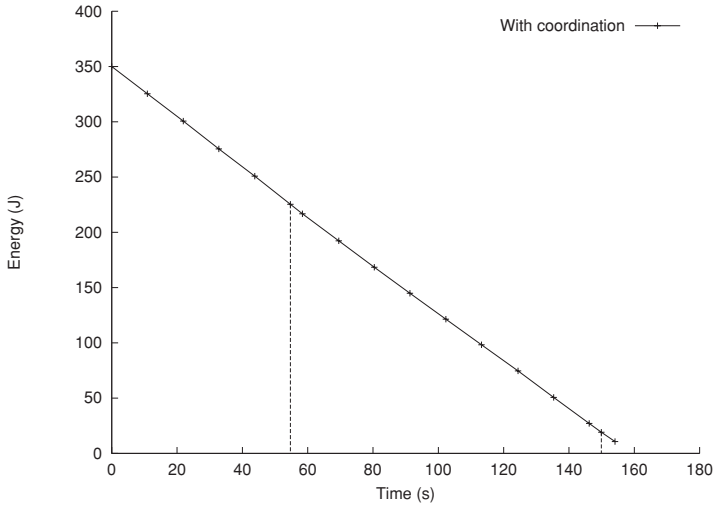
(b) Running state of the existing low-priority application



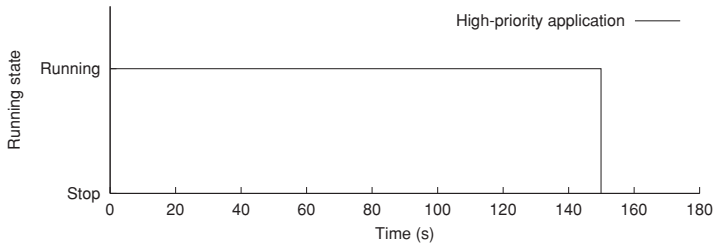
(c) Running state of the newly joining high-priority application

Fig. 13. The case of a high-priority application joining the system without coordination.

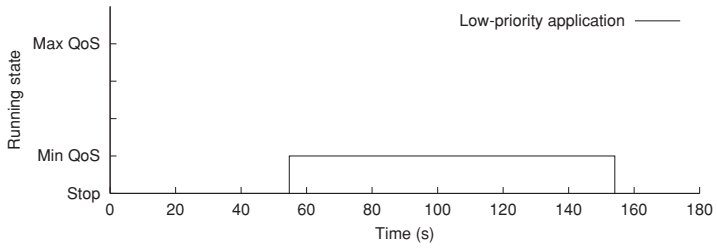
at the highest QoS level, and competes more avariciously with the high-priority application for the battery. Consequently, the battery drains faster and neither of the two applications finishes execution. Figure 15 represents the experimental results in this case. Figure 15a shows the energy-dissipation rate and the starting point for the speech recognizer. Figure 15b shows the execution state of the already existing high-priority application video player. Figure 15c shows



(a) Energy-dissipation rate of the two applications



(b) Running state of the existing high-priority application

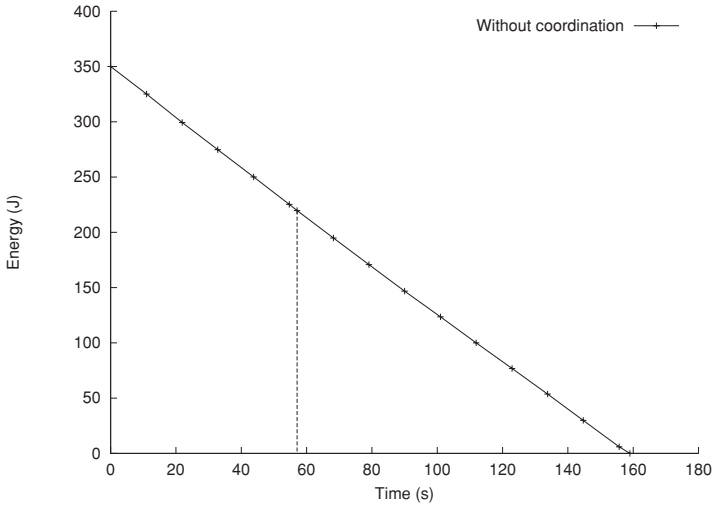


(c) Running state of the newly joining low-priority application

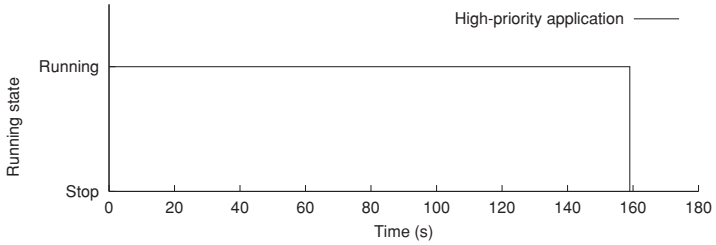
Fig. 14. System coordination when a low-priority application joins the system.

that the newly joining application of speech recognizer executes at the highest QoS level.

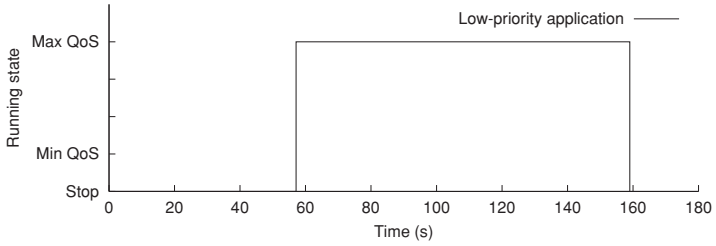
These experiments show the necessity and efficacy of both coordination among multiple applications and adaptation for each single application for mobile computers.



(a) Energy-dissipation rate of the two applications



(b) Running state of the existing high-priority application



(c) Running state of the newly joining low-priority application

Fig. 15. The case of a low-priority application joining the system without coordination.

6. CONCLUSIONS

We have proposed and implemented a DSOM framework, which not only meets user-specified goals under battery energy constraints, but also abides by the user's intention through the use of a user-specified priority. It is implemented in the user space, with no changes needed in the underlying OS. It is also

portable to any OS and mobile platform that is POSIX-compliant. It increases the energy efficiency of the mobile computer system at the expense of acceptable QoS degradation. It is complementary to other low-level energy efficiency techniques (such as those at the OS and compiler levels, and DVFS, etc.), and exploits the new concept of software low-power modes.

Our framework does have several limitations that inspire future research work. It relies greatly on the application being adaptable. In practice, however, we found many mobile applications have adaptable features, including the mobile applications used in the article and other nonmultimedia applications, e.g., a web browser. In the future, adaptability could be built into the application to further exploit our framework. Currently, for each adaptable application, we need to derive the energy macromodel offline and embed the model in its associated runtime library. Developing a unified low-level device-based or architecture parameter-based energy macromodel will be a near future research work. Future work will also focus on cross-layer adaptation, which will exploit DVFS for each low-power mode as well, that is, both *the resource demand and supply can be scaled in an interdependent manner*. For example, under different dithering methods, the dithering time may vary and create different slacks in a frame period, so that different frequencies/voltages can be used for the same video clip under different modes. In such a case, the framework may be able to direct application-specified information to the underlying OS and platform for further energy savings.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Johan Pouwelse from Delft University of Technology, The Netherlands, for his kind offer of the source code for the client–server communication mechanism.

REFERENCES

Agilent. Agilent IntuiLink software. `http://www.testequipmentdepot.com/hp/IntuiLinkSoft-ware.htm`.

Bavier, A., Montz, A., and Peterson, L. 1998. Predicting MPEG execution times. In *Proceedings of the International Conference on Measurement & Modeling of Computer Systems*. 131–140.

Benini, L., Kandemir, M., and Ramanujam, J., Eds. 2003. *Compilers and Operating Systems for Low Power*. Kluwer Academic Publ., Norwell, MA.

Bharghavan, V. and Gupta, V. 1997. A framework for application adaptation in mobile computing environments. In *Proceedings of the Computer Software & Application Conference*. 573–579.

Chakraborty, S. and Yau, D. K. Y. 2002. Predicting energy consumption of MPEG video playback on handhelds. In *Proceedings of the International Conference Multimedia & Expo*. 317–320.

Chandrakasan, A. P., Bowhill, W. J., and Fox, F. 2000. *Design of High-Performance Microprocessor Circuits*. Wiley-IEEE Press, New York.

Chang, F. and Karamcheti, V. 2001. A framework for automatic adaptation of tunable distributed applications. *Cluster Comput. 4,* 1 (May), 49–62.

Choi, I., Shim, H., and Chang, N. 2002. Low-power color TFT LCD display for handheld embedded systems. In *Proceedings of the International Symposium on Low Power Electronics & Design*. 112–117.

De Lara, E., Wallach, D. S., and Zwaenepol, W. 2001. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the USENIX Symposium on Internet Technologies & Systems*. 159–170.

DELALUZ, V., KANDEMIR, M., VIJAYKRISHNAN, N., SIVASUBRAMANIAM, A., AND IRWIN, M. J. 2001. Memory energy management using software and hardware directed power mode control. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 159–169.

EFSTRATIOU, C., FRIDAY, A., DAVIES, N., AND CHEVERST, K. 2002. A platform supporting coordinated adaptation in mobile systems. In *Proceedings of the Workshop on Mobile Computing Systems & Applications*. 128–137.

FARKAS, K. I., FLINN, J., BACK, G., GRUNWALD, D., AND ANDERSON, J. M. 2000. Quantifying the energy consumption of a pocket computer and a Java Virtual Machine. In *Proceedings of the International Conference on Measurement & Modeling Computer Systems*. 252–263.

FEENEY, L. M. AND NILSSON, M. 2001. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *Proceedings of the IEEE INFOCOM*. 1548–1557.

FEI, Y. 2004. System-level Energy Analysis and Optimization of Embedded Systems. Ph.D. thesis, Department of Electrical Engineering, Princeton University.

FEI, Y., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. 2004. Energy-optimizing source code transformations for OS-driven embedded software. In *Proceedings of the International Conference VLSI Design*. 261–266.

FLINN, J. AND SATYANARAYANAN, M. 1999. Energy-aware adaptation for mobile applications. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 48–63.

GAUTHIER, P., HARADA, D., AND STEMM, M. 1996. Reducing power consumption for the next generation of PDAs: It's in the network interface. In *Proceedings of the International Workshop on Mobile Multimedia Communication*.

ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics & Design*. 197–202.

RABAEY, J. AND PEDRAM, M. (EDS.). 1996. *Low Power Design Methodologies*. Kluwer Academic Publ., Norwell, MA.

JHA, N. K. 2001. Low power system scheduling and synthesis. In *Proceedings of the International Conference Computer-Aided Design*. 259–263.

KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2002. Compiler optimizations for low power systems. In *Power-Aware Computing*, R. Melhem and R. Graybill, Eds. Kluwer Academic Publ, Norwell, M.A.

KRINTZ, C., WEN, Y., AND WOLSKI, R. 2002. Predicting program power consumption. Tech. Rep. 2002–20 July., Department of Electrical and Computer Engineering, University of California at Santa Barbara.

LEE, C., LEHOCZKY, J., SIEWIOREK, D., RAJKUMAR, R., AND HANSEN, J. 1999. A scalable solution to the multi-resource QoS problem. In *Proceedings of the Real-Time Systems Symposium*. 315–326.

LI, K., KUMPF, R., HORTON, P., AND ANDERSON, T. 1994. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the Winter Usenix*. 279–291.

MITCHELL, J., PENNEBAKER, W., FOGG, C., AND LEGALL, D. 1996. *MPEG Video Compression Standard*. Chapman & Hall, London.

MOHAPATRA, S., CORNEA, R., DUTT, N., NICOLAU, A., AND VENKATASUBRAMANIAN, N. 2003. Integrated power management for video straming to mobile handheld devices. In *Proceedings of the ACM International Conference on Multimedia*. 582–591.

MPlayer. MPEG Player. `http://bmrc.berkeley.edu/frame/research/mpeg/`

NAHRSTEDT, K., XU, D., WICHADUKUL, D., AND LI, B. 2001. QoS-aware middleware for ubiquitous and heterogeneous environments. *IEEE Commun. 39,* 11 (Nov.), 140–148.

NARAYANAN, D. AND SATYANARAYANAN, M. 2003. Predictive resource management for wearable computing. In *Proceedings of the International Conference on Mobile Systems, Applications, & Services*. 113–128.

NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. 1997. Agile application-aware adaptations for mobility. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 276–287.

PARK, S., RAGHUNATHAN, V., AND SRIVASTAVA, M. B. 2003. Energy efficiency and fairness tradeoffs in multi-resource multi-tasking embedded systems. In *Proceedings of the International Symposium on Low Power Electronics & Design*. 469–474.

PERING, T., BURD, T., AND BRODERSEN, R. 1998. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low Power Electronics & Design*. 76–81.

PEYMANDOUST, A., SIMUNIC, T., AND DE MICHELI, G. 2002. Low power embedded software optimization using symbolic algebra. In *Proceedings of the Design Automation & Test Europe Conference* 1052–1059.

PILLAI, P., HUANG, H., AND SHIN, K. G. 2003. Energy-aware quality-of-service adaptation. Tech. Rept. CSE-TR-479-03, University of Michigan.

POUWELSE, J. 2003. Power Management for Portable Devices. Ph.D. thesis, Faculty of Information Technology and Systems, Delft University of Technology, The Netherlands.

POUWELSE, J., LANGENDOEN, K., AND SIPS, H. 2001. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the International Conference on Mobile Computing & Networking*. 251–259.

RAJKUMAR, R., LEE, C., LEHOCZKY, J. P., AND SIEWIOREK, D. P. 1997. A resource allocation model for QoS management. In *Proceedings of the IEEE Real-Time Systems Symposium*. 298–307.

RAT. Robust Audio Tool. `http://internet2.motlabs.com/ipaq/rat.htm`.

SACHS, D. G., YUAN, W., HUGHES, C. J., HARRIS, A., ADVE, S. V., JONES, D. L., HRAVETS, R. H., AND NAHRSTEDT, K. 2004. GRACE: A hierarchical adaptation framework for saving energy. Tech. Rep. UIUCDCS-R-2004-2400, Computer Science, University of Illinois. Feb. http://rsim.cs.uiuc.edu/grace/papers/grace-tr.pdf.

SHENOY, P. AND RADKOV, P. 2003. Proxy-assisted power-friendly streaming to mobile devices. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing & Networking*. 177–191.

TAN, T. K., RAGHUNATHAN, A., LAKSHMINARAYANA, G., AND JHA, N. K. 2002. High-level energy macro modeling of embedded software. *IEEE Trans. Comput.-Aided Design 21,* 9 (Sept.), 1037–1050.

TAN, T. K., RAGHUNATHAN, A., AND JHA, N. K. 2003. Software architecture transformations: A new approach to low energy embedded software. In *Proceedings of the Design Automation & Test Europe Conference* 1046–1051.

YUAN, W., NAHRSTEDT, K., ADVE, S., JONES, D., AND KRAVETS, R. 2003. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Proceedings of the SPIE/ACM Multimedia Computing & Networking Conference*. 1–13.

ZENG, H., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. 2002. ECOSystem: Managing energy as a first class operating system resource. In *Proceedings of the International Conference on Architectural Support for Programming Languages & Operating Systems*. 123–132.

ZENG, H., ELLIS, C. S., AND LEBECK, A. R. 2005. Experiences in managing energy with ECOSystem. *IEEE Pervasive Comput. 4,* 1 (Jan.–Mar.), 62–68.

ZHONG, L., SHI, Y., AND LIU, R. 1999. A dynamic neural network for syllable recognition. In *Proceedings of the International Joint Conference on Neural Networks*. 2997–3001.

Received August 2005; revised October 2006; accepted December 2006