

An Energy-aware Framework for Coordinated Dynamic Software Management in Mobile Computers

Yunsi Fei, Lin Zhong, and Niraj K. Jha

Dept. of Electrical Engineering, Princeton University, NJ 08544

Email: {yfei, lzhong, jha}@princeton.edu*

Abstract

Energy efficiency is a very important and challenging issue for resource-constrained mobile computers. In this paper, we propose a dynamic software management (DSM) framework to improve battery utilization, and avoid competition for limited energy resources from multiple applications. We have designed and implemented a DSM 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. It also employs energy macro-models for mobile applications to aid in this endeavor. By monitoring the energy supply and predicting energy demand at each QoS level, the DSM module is able to select the best possible trade-off between energy conservation and application QoS. To the best of our knowledge, this is the first energy-aware coordinated framework utilizing adaptation of mobile applications. It honors the priority desired by the user and is portable to POSIX-compliant OSs. 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 improve battery utilization significantly. They also show that prediction of application energy demand based on energy macro-models is a key component of this framework.

1 Introduction

Mobile computing systems are constrained by scarce resources, such as small memory, slow CPU, *etc.* They are specially constrained by limited battery capacity owing to the weight/size limits for the battery. On the other hand, as we enter the era of pervasive computing, users are expecting more flexible and ubiquitous services and higher productivity from mobile computing systems. 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 refers to the amount of

work that the system can accomplish given a battery capacity constraint.

Mobile computing 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. Existing energy efficiency research can be 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 [25, 7] fall into this category. *Software optimization* techniques optimize the consumer so that less energy is consumed for the same software service. Most compilation [14] and software transformation techniques [22, 27, 10] fit into this category. Another category of techniques *scales the supply according to demand*. All dynamic power management (DPM) and dynamic voltage/frequency scaling (DVFS) techniques [13, 21, 12, 24] belong to this category. It also includes OS-directed power management and optimization [4]. The techniques presented in [16, 18] use QoS-based resource reservation, in which applications pass specific resource demands to the OS, and the OS reserves the minimum required resources (fraction of CPU cycles, network bandwidth, *etc.*) for them and prevents other applications from competing.

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*. Our work belongs to this category.

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 [20] adopts this approach and provides for a collabo-

* Acknowledgments: This work was supported by DARPA under contract no. DAAB07-02-C-P302.

ration between the OS and applications to improve performance. Similarly, the Puppeteer system [8] filters the input content through component-based adaptation. The Odyssey system was also extended to enable data fidelity adaptation for energy reduction [11]. The same philosophy is used in [26] to transform the requested network data stream to reduce receiving and decoding energy. All these approaches are OS-based and input data-centric.

Another way to scale the demand is to change the QoS, which may or may not depend on altering data fidelity. An application adaptation framework is discussed in [5], but it does not target realistic applications. A recent work [19] provides examples of how the computation fidelity of mobile applications can be altered. However, it does not present a methodology or framework for accomplishing this task automatically, and it targets system delay reduction instead of energy saving.

Another issue related to scaling the demand is how to coordinate multiple scalable applications. The competition among multiple applications for the constrained resources needs to be addressed. A platform is proposed in [9] 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.

Except for the work in [11, 26], all the above works target non-energy related resources. Most investigate software adaptation for communication (network bandwidth) and computation resources (CPU cycles). The work in [11] 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. Its predecessor work on application adaptation to reduce network bandwidth [20] 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 DSM framework for run-time energy optimization.

1.2 Paper contributions

In this work, we propose an application adaptation framework based on software energy macro-modeling techniques [28], DSM, and multiple-application coordination. The framework makes the following contributions:

- It utilizes software energy macro-modeling and obviates the need for extra equipment for real-time energy measurement [11], which is impractical for handheld computers.
- It is implemented as portable middleware using only POSIX compliant system application-programming interfaces (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.

- Unlike most previous work, the framework exploits multiple QoS knobs 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 real applications, such as video player, speech recognizer and voice-over-IP.

The paper is organized as follows. In Section 2, we provide background information for this work along with motivational examples. We present the overall DSM 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 future work 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 and coordination for battery-constrained systems, and then provide the rationale for an energy-aware framework for DSM.

2.1 Dynamic software management for low energy

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 hard disk, display, and network card. 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 run-time parameters, or knobs. These tunable knobs represent alternative algorithm or implementation paths during application execution. The knobs may be local to some blocks, or global throughout the whole program. 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 DSM module determines on the fly *when* to scale the application and to *what* working state. Thus, energy saving is achieved at minimal cost in quality.

2.2 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* [1] 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 size) will induce different energy consumption [11]. However, changing the computation fidelity of applications for energy saving has not been adequately explored before. 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.

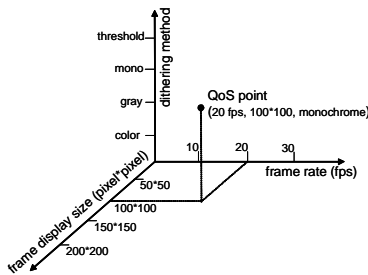


Figure 1. QoS space for a video player

The video player application consists of three steps for each frame: decode, dither, and display. Most time and energy are consumed in the first two steps. It is shown in [6] that display size has little effect on energy consumption. Also, a user may prefer to watch the video clip at a particular frame rate. Dithering modes tune the color exposition of the frame and present us with one way to adapt the application. We analyzed various dithering methods, and found four modes with human-perceptible difference: *color*, *gray*, *monochrome*, and *threshold*, out of a total of 19 modes. Figure 2 shows the energy consumption of four video clips under different dithering methods on a Linux-based iPAQ 3870 (this is based on current measurements on the iPAQ). 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.

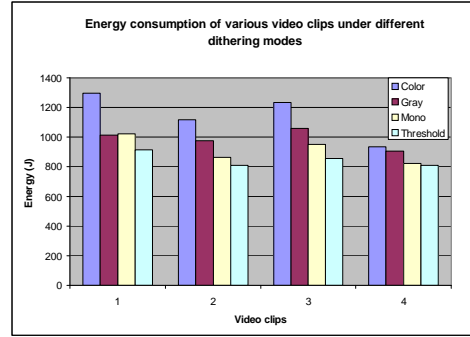


Figure 2. Energy impact of the dithering method

2.3 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 new adaptations for the newly admitted and other applications.

2.4 Policy for dynamic software management

For the concept of DSM 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 DSM. The battery energy status could be checked periodically and the appropriate execution mode for running applications selected accordingly. However, this policy is ad-hoc and aggressive, 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 Section 3.

In mobile systems, a user may be able to estimate the needed duration for a task, *e.g.*, the length of a movie, the length of a conversation with peer mobile users, *etc.* In

such cases, we set the expected duration for the task as the goal. Only when the residual battery energy cannot sustain the goal, application adaptation is triggered. Consider the following scenario: the user wants to watch a 30-minute long video clip with the energy consumption estimate of the video clip for the highest quality mode being 3600 Joules, whereas the residual battery energy is only 3200 Joules. Suppose the energy consumption under the four dithering modes are 3600, 3400, 3150, 3000 Joules 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. In the absence of user input on expected application duration, the system will 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 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 DSM 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, battery, etc. 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 to the upper modules as well. The OS also provides a set of APIs to interact with the user-space modules. Above the OS layer sits the middleware that we have developed, which consists of several modules that can interact either with the upper application layer, or the OS layer below.

On top of the middleware lie 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. In our framework, on joining the system, the application first registers with an application registry and provides some meta-data, e.g., the name of the service, the number of low power modes, etc. Meanwhile, the description of adaptation modes for each application is stored into a run-time library, along with some application-specific information, such as the power macro-model to estimate the average power consumption for each low power mode, the inputs, etc., in order to aid application adaptation. For different instantiations of

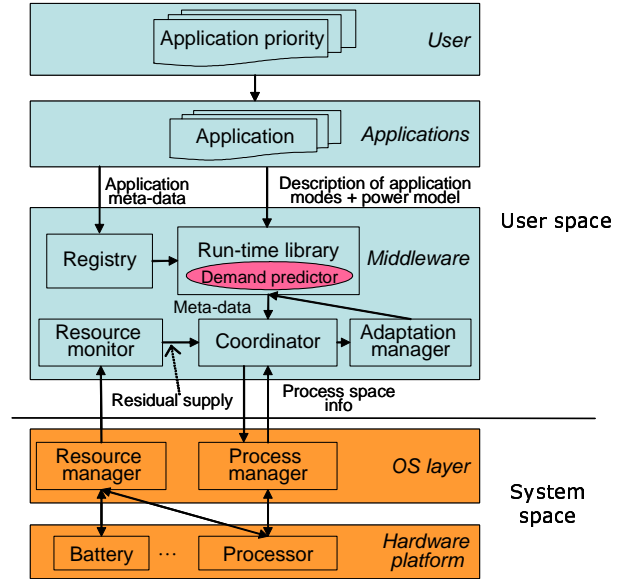


Figure 3. A coordination framework for application adaptation

the same registered application, the user can specify a different priority level.

The resource monitor module in our framework polls the OS resource manager, acquires the battery status information, and calculates the residual battery energy value. Based on the application meta-data, 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, and sends process operation commands to the OS process manager and adaptation decisions to the adaptation manager. 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 DSM, 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. Then we 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, and also before the available energy drops to zero. 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 provides a “best-effort” service.

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

The demand predictor is also implemented in the run-time 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 meta-data of the newly registered application and the process space information are given as input to the admission control unit. The application meta-data at least 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 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 run-time li-

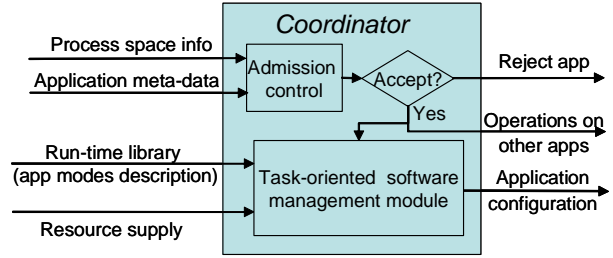


Figure 4. The design and flow of the coordinator

brary, which contains the description of various application modes, and energy macro-models 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 DSM 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 run-time library.

3.4 Scalable adaptation block

In our framework, the QoS level is negotiated between the application and coordinator before the application is launched. Adaptation is invoked at the beginning of the application. 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 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 paper. Our main contribution is building a general energy-aware DSM 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.

4 Framework implementation

We have built a prototype user space DSM framework in middleware. The key components are the run-time library, which provides the adaptation configuration calls to the applications, and the coordinator, which negotiates/re-negotiates 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 [23], and the communication between the coordinator and processes is implemented with signals. We illustrate next each salient feature of the framework.

4.1 Registry

Each adaptable application needs to register its meta-data in a registry, and is allocated a data structure called *component*. 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* [1], “voip” to a voice-over-IP application, *RAT* [2], and “speech” to a dynamic neuron-network based speech recognizer, *DNN* [29]. The application also specifies the number of low-power modes that it can support to the *levels* entry of *component*.

For each service instantiation, we allocate a data structure called *service* with the specified name. It inherits all the meta-data 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 Run-time library

The run-time 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 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 run-time 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 based on information from the run-time library. The run-time 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 run-time library is the energy estimator for an application at different QoS levels. The Odyssey system [11] 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 [15] has tried to predict program power using some application-level software tools. Currently, we embed an application-specific energy estimation module in the run-time library. Based on the pre-built energy macro-model for each low power mode, 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 macro-model for each application - video player *mpeg_player*, voice-over-IP *RAT*, and speech recognizer *DNN*.

4.2.1 Energy macro-model for the video player

An MPEG-1 video stream consists of three frame types: *I-frame* (intra-coded), *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 a frame is displayed on the screen (note that the decoded frame and the displayed one may not be the same). It takes several steps to decode each frame: *parsing*, *inverse discrete cosine transformation* (IDCT), *reconstruction*, and *dithering* [17]. Among these steps, the first three are CPU-intensive and frame-type-dependent (each frame type requires a different type of processing), 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 processes: decode, dither, display and idle. Hence, the energy consumption of the video player can be obtained from Equation (1).

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

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).

$$T_{period} = T_{decode,i} + T_{dither,i} + T_{display,i} + T_{idle,i} \quad (2)$$

We first need to characterize the energy macro-model for the video player. We adopt the standard regression analysis method 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 four values for P_{decode} and $P_{display}$, respectively.

When using the energy macro-model 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 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 [3] that the decoding time of each frame is linearly dependent on the frame size:

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

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. Similarly, the dithering time and display time are linearly dependent on the frame resolution, and independent of frame size and type. Equations (4) and (5) show this relationship.

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

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

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 macro-model and execution time estimation techniques is shown for two video clips, each with four dithering methods, in Figure 5. 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 can use a similar procedure for energy estimation for a voice-over-IP application, *RAT* [2]. The application can execute under different audio quality settings, which contains a set of 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

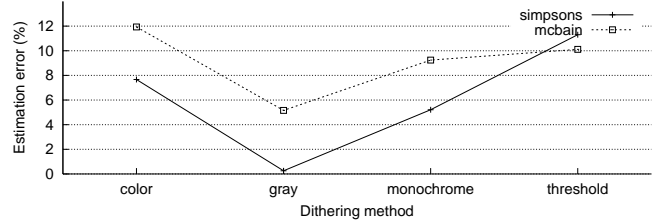


Figure 5. Accuracy evaluation of energy macro-models for different QoS levels

values compose the QoS space for *RAT*. For different QoS points, there are different active sending power 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 macro-model is as shown in Equation (6).

$$E = P_{send} \cdot T_{send} + P_{receive} \cdot T_{receive} + P_{idle} \cdot T_{idle} \quad (6)$$

Figure 6 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 16KHz, while *Stereo48* uses a stereo channel at a sampling rate of 48KHz. 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.

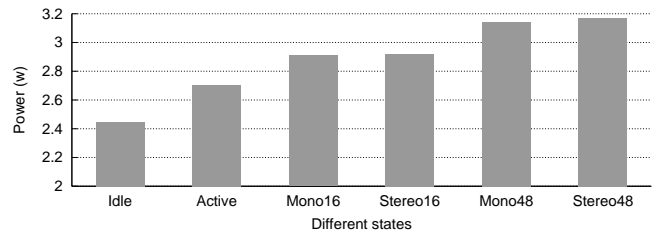


Figure 6. Variation of power for different states

4.2.3 Energy estimation for the speech recognizer

The dynamic neural network based speech recognizer, DNN [29], 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 purposes, and the neural network parameters are stored in a file. At the back end, when 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 7 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. However, increasing the number of hidden layers also improves the speech recognition accuracy, *i.e.*, QoS level.

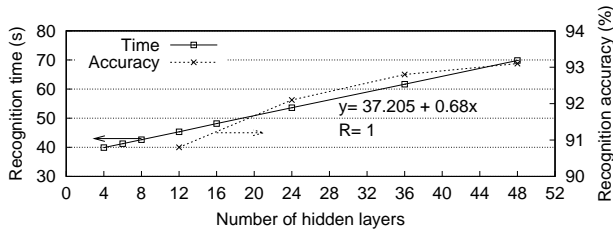


Figure 7. Variation of recognition time/quality for different parameter values

5 Evaluation of the DSM framework

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 206MHz, 64MB DRAM and 32MB flash), running under Familiar Linux, as our evaluation platform.

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 fully charged to obtain the system energy dissipation (note that this measurement is done only for validation purposes, the framework actually uses the macro-models described earlier).

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 (*filelist*).

When the *client* receives a command, it passes it on to the *server*. The *server* first parses the command. If it is a service request, it determines the application configuration (QoS level) and delivers it to the application. The pseudocode for QoS level configuration is shown in Algorithm 1. At each adaptation point, the coordinator on the server side evaluates the service, estimates the average power and execution time for each QoS level, and selects the best QoS level to meet the user-specified goal. The service ends either when the duration goal is met, or when the residual energy drops to zero, or the specified deadline is reached. 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.

The iPAQ can use AC or DC power. The battery that supplies the latter is the Danionics Lithium-Ion Polymer battery (#DLP 345794). Its capacity is 1400mAh, 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 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.

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 in Section 5.2.2.

5.2 Experimental results

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

Algorithm 1 *service_command(name, goal, priority, param)*

```

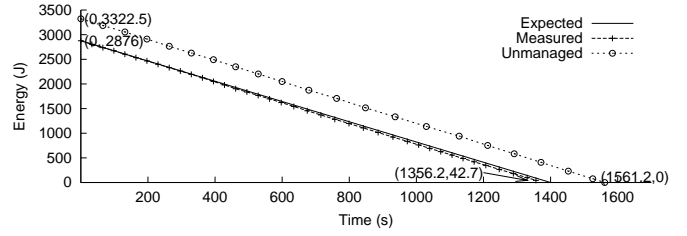
1: initialize_service(name);
2: detect_residual_energy(ener);
3:  $P_{avg} = ener/goal$ ; //average system power for the specified duration
4:  $P_{last} = 0$ ; //average power for the last adaptation block
5:  $T_{elapsed} = 0; i=0$ ; //the elapsed time and task index
6: while ( $i < n \ \& \ T_{elapsed} < goal \ \& \ ener > 0$ ) do
7:   evaluate( $Block_i, \bar{P}, \bar{T}, Num$ ); /*estimate the power/time vector for different QoS levels, Num is the number of levels*/
8:    $P_{upper} = \min(\alpha * P_{avg}, 2 * P_{avg} - P_{last})$ ;
9:   if ( $P[num - 1] > P_{upper}$ ) then
10:    level = num - 1;
11:   else
12:    for  $j = 0$  to  $Num$  do
13:      if ( $P[j] < P_{upper}$ ) then
14:        level = j;
15:        break;
16:      end if
17:    end for
18:   end if
19:    $P = P[level]$ ;
20:   execute( $Block_i, P, level$ );
21:    $P_{last} = P$ ;
22:    $i++$ ;
23:   detect_residual_energy(ener);
24:   report_time( $T_{elapsed}$ );
25: end while

```

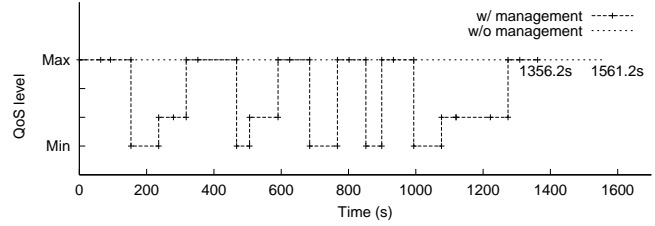
5.2.1 Adaptation of single applications

Figure 8 shows the detailed result of an experiment with the goal set to 1400 seconds and for executing 24 video clips. The **Measured** line in Figure 8(a) 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 DSM is not conservative at all. The third line, **Unmanaged**, represents the energy consumption rate without DSM. 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 seconds. Thus, without DSM, the goal cannot be satisfied.

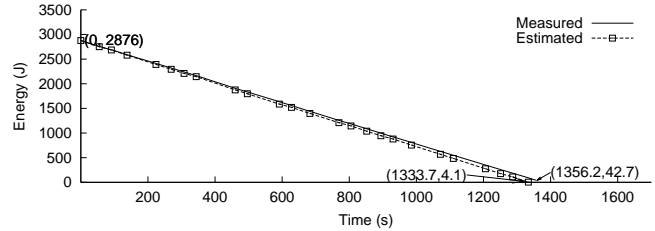
Figure 8(b) shows how the software application adapts with and without software management during execution. We consider four different QoS levels based on the dithering mode, as described earlier. Different video clips, even at the same QoS level, may have different average power. Hence, we see very frequent adaptation of the video player for clips



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



(b) Variation of QoS in the two cases



(c) Comparison of estimated and measured energy dissipation rates

Figure 8. Application adaptation of the video player

from line **w/ management**. In general, the adaptation of separate adaptation blocks is 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.

To evaluate the accuracy of our energy macro-model, we compared the estimated and measured energy dissipation rates in Figure 8(c). Our energy macro-model predicts that the total energy consumption is 2871.9 J and the video player runs for 1333.7 seconds. 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 1 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

Table 1. Knobs and effects of application adaptation

Applications	Tunable parameter	Resource	Range of variation	Possible energy saving
Video player	Dithering mode	CPU time for dithering	1.34-24.93 ms	30.2%
		Power	1.5-3.6 W	
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 the same task from the highest QoS level 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 DSM, and the battery utilization can be improved greatly.

5.2.2 Coordination among multiple applications

We performed another experiment to evaluate the efficacy of coordination among multiple applications. Applications are classified into two categories according to their user-specified priority. When a service with a high priority is called, we check the service list that contains the concurrently-running applications. The applications with a low priority yield the resources to the new service, and other high-priority applications keep running and competing for the battery with the new one. When the new service has a low priority, we still check the service list. If there are high-priority applications running, we reserve battery energy for them and the remaining energy is assigned to the new service for appropriate adaptation. 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. Algorithm 2 describes this coordination policy in detail.

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. 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 9 shows the experimental result under this scenario with system coordination. Figure 9(a) shows the energy dissipation rate. The starting and finishing time points for the speech recognizer are also

Algorithm 2 *coordinator(service_list, new_service)*

```

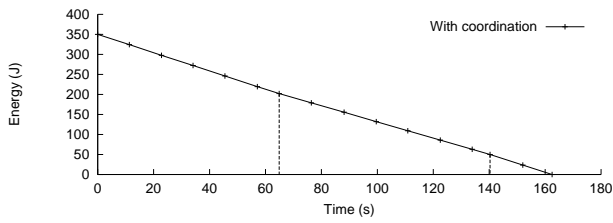
1: prior = new_service→priority;
2: if prior=high then
3:   service = service_list;
4:   while service!=NULL do
5:     if service→priority=low then
6:       send_pause_signal(service→children);
7:     end if
8:     service = service→next;
9:   end while
10: evaluate(new_service, current_battery); /*evaluate the
    appropriate working state, competing with other high-
    priority applications*/
11: else
12:   flag=search(service_list);
13:   if flag=high then
14:     evaluate(new_service, remain_energy); //energy re-
    served for high-priority applications
15:   else
16:     evaluate(new_service, current_battery); //compete
    with low-priority applications
17:   end if
18: end if

```

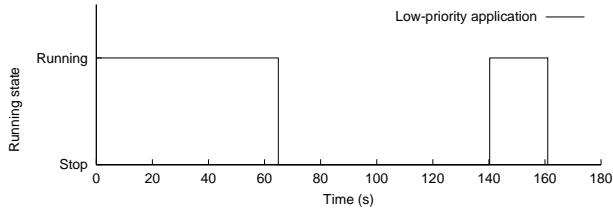
marked on the figure. We can see from Figure 9(b) that the existing low-priority video player yields to the new high-priority speech recognizer when the latter arrives at time 64.9 s. Figure 9(c) shows that with coordination, the speech recognizer can finish under the battery energy constraint. The video 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 compete for the battery simultaneously. Figure 10 represents the experimental results for this case. Figure 10(a) shows the energy dissipation rate and the starting point for the speech recognizer. Due to competition from the existing video player application, the speech recognizer does not finish, as shown in Figure 10(c). 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.

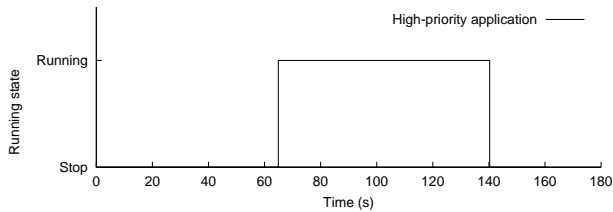
We performed another similar experiment in which the existing video player application is of high priority while the newly-joining speech recognizer is of low priority. When the speech recognizer joins the system, the system



(a) Energy dissipation rate of the two applications



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



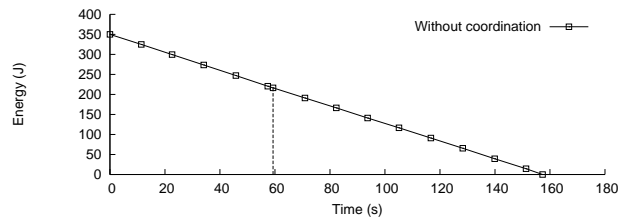
(c) Running state of the newly-joining high-priority application

Figure 9. System coordination when a high-priority application joins the system

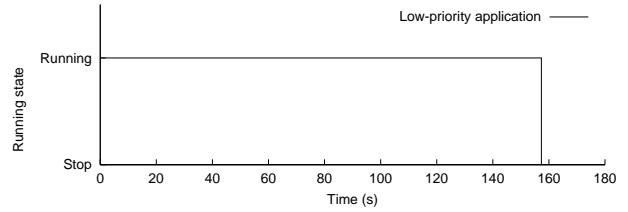
knows the currently running QoS level of the video player and its estimated energy from the energy macro-model. With coordination, the system reserves the energy for the video player, thus executing the speech recognizer at the fourth QoS level instead of first (the highest). Both the video player and speech recognizer complete their execution. 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 at the highest QoS level, and competes with the high-priority application for the battery more avariciously. Consequently, the battery drains faster and neither of the two applications finish execution. These experiments show the efficacy of both coordination among multiple applications and adaptation for each single application.

6 Conclusions and future work

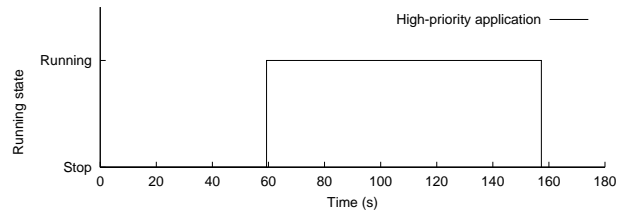
We have proposed and implemented a DSM framework, which not only meets user-specified goals under battery energy constraints, but also abides by the user's intention



(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

Figure 10. The case of a high-priority application joining the system without coordination

through 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 some limitations. It relies greatly on the application being adaptable. In practice, however, we found many mobile applications have adaptable features. In the future, adaptability could be built into the application to further exploit our framework. Future work will focus on cross-layer adaptation, which will exploit DVFS for each low power mode as well, *i.e.*, both the resource demand and supply can be scaled in an inter-dependent 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

- [1] MPEG player. <http://bmrc.berkeley.edu/frame/research/mpeg/>.
- [2] Robust Audio Tool. <http://internet2.motlabs.com/ipaq/rat.htm>.
- [3] A. Bavier, A. Montz, and L. Peterson. Predicting MPEG execution times. In *Proc. Int. Conf. Measurement & Modeling of Computer Systems*, pages 131–140, June 1998.
- [4] L. Benini, M. Kandemir, and J. Ramanujam, editors. *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, Boston, MA, Oct. 2003.
- [5] V. Bharghavan and V. Gupta. A framework for application adaptation in mobile computing environments. In *Proc. Computer Software & Application Conf.*, pages 573–579, Nov. 1997.
- [6] S. Chakraborty and D. K. Y. Yau. Predicting energy consumption of MPEG video playback on handhelds. In *Proc. Int. Conf. Multimedia & Expo*, pages 317–320, Aug. 2002.
- [7] A. P. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. Wiley-IEEE Press, 2000.
- [8] E. De Lara, D. S. Wallach, and W. Zwaenepol. Puppeteer: Component-based adaptation for mobile computing. In *Proc. USENIX Symp. Internet Technologies & Systems*, pages 159–170, Mar. 2001.
- [9] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. A platform supporting coordinated adaptation in mobile systems. In *Proc. Wkshp. Mobile Computing Systems & Applications*, pages 128–137, June 2002.
- [10] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha. Energy-optimizing source code transformations for OS-driven embedded software. In *Proc. Int. Conf. VLSI Design*, pages 261–266, Jan. 2004.
- [11] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. ACM Symp. Operating Systems Principles*, pages 48–63, Dec. 1999.
- [12] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 197–202, Aug. 1998.
- [13] N. K. Jha. Low power system scheduling and synthesis. In *Proc. Int. Conf. Computer-Aided Design*, pages 259–263, Nov. 2001.
- [14] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler optimizations for low power systems. In R. Melhem and R. Graybill, editors, *Power Aware Computing*. Kluwer Academic Publishers, Boston, MA, 2002.
- [15] C. Krintz, Y. Wen, and R. Wolski. Predicting program power consumption. Technical Report 2002-20, Department of Electrical and Computer Engineering, University of California at Santa Barbara, July 2002.
- [16] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource QoS problem. In *Proc. Real-Time Systems Symp.*, pages 315–326, Dec. 1999.
- [17] J. Mitchell, W. Pennebaker, C. Fogg, and D. LeGall. *MPEG Video Compression Standard*. Chapman & Hall, London, 1996.
- [18] K. Nahrstedt, D. Xu, D. Wichadukul, and B. Li. QoS-aware middleware for ubiquitous and heterogeneous environments. *IEEE Communications*, 39(11):140–148, Nov. 2001.
- [19] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *Proc. Int. Conf. Mobile Systems, Applications, & Services*, pages 113–128, May 2003.
- [20] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptations for mobility. In *Proc. ACM Symp. Operating Systems Principles*, pages 276–287, 1997.
- [21] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 76–81, Aug. 1998.
- [22] A. Peymandoust, T. Simunic, and G. De Micheli. Low power embedded software optimization using symbolic algebra. In *Proc. Design Automation & Test Europe Conf.*, pages 1052–1059, Mar. 2002.
- [23] J. Pouwelse. *Power Management for Portable Devices*. PhD thesis, Faculty of Information Technology and Systems, Delft University of Technology, The Netherlands, Oct. 2003.
- [24] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proc. Int. Conf. Mobile Computing & Networking*, pages 251–259, July 2001.
- [25] J. M. Rabaey and M. Pedram, editors. *Low Power Design Methodologies*. Kluwer Academic Publishers, Boston, MA, 1995.
- [26] P. Shenoy and P. Radkov. Proxy-assisted power-friendly streaming to mobile devices. In *Proc. SPIE/ACM Conf. Multimedia Computing & Networking*, pages 177–191, Jan. 2003.
- [27] T. K. Tan, A. Raghunathan, and N. K. Jha. Software architectural transformations: A new approach to low energy embedded software. In *Proc. Design Automation & Test Europe Conf.*, pages 1046–1051, Mar. 2003.
- [28] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. High-level energy macro-modeling of embedded software. *IEEE Trans. Computer-Aided Design*, 21(9):1037–1050, Sept. 2002.
- [29] L. Zhong, Y. Shi, and R. Liu. A dynamic neural network for syllable recognition. In *Proc. Int. Joint Conf. Neural Networks*, pages 2997–3001, July 1999.