Power-Aware Resource Management Techniques for Low-Power Embedded Systems

Jihong Kim

School of Computer Science & Engineering

Seoul National University

Seoul, Korea 151-742

E-mail: jihong@davinci.snu.ac.kr

Tajana Simunic Rosing

Department of Computer Science & Engineering

University of California, San Diego

La Jolla, CA 92093, USA

E-mail: tajana@ucsd.edu

July 2, 2006

1 Introduction

Energy consumption has become one of the most important design constraints for modern embedded systems, especially for mobile embedded systems that operate with a limited energy source such as batteries. For these systems, the design process is characterized by a tradeoff between a need for high performance and low power consumption, emphasizing high performance to meeting the performance constraints while minimizing the power consumption. Decreasing the power consumption not only helps with extending the battery lifetime in portable devices, but is also a critical factor in lowering the packaging and the cooling costs of embedded systems. While better low-power circuit design techniques have helped to lower the power consumption [9, 41, 34], managing power dissipation at higher abstraction levels can considerably decrease energy requirements [14, 3].

Since we will be focusing on dynamic power P_{dynamic} in this chapter, the total power dissipation P_{CMOS} can be approximated as $P_{\text{CMOS}} \approx P_{\text{dynamic}}$. The dynamic power of CMOS circuits is dissipated when the output capacitance is charged or discharged, and is given by $P_{\text{dynamic}} = \alpha \cdot C_{\text{L}} \cdot V_{\text{dd}}^2 \cdot f_{\text{clk}}$ where α is the switching activity (the average number of high-to-low transitions per cycle), C_{L} is the load capacitance, V_{dd} is the supply voltage, and f_{clk} is the clock frequency. The energy consumption during the time interval $[0, T]$ is given by $E = \int_0^T P(t)dt \propto V_{\text{dd}}^2 \cdot f_{\text{clk}} \cdot T = V_{\text{dd}}^2 \cdot N_{\text{cycle}}$ where $P(t)$ is the power dissipation at t and N_{cycle} is the number of clock cycles during the interval $[0, T]$. These equations indicate that a significant energy saving can be achieved by reducing the supply voltage V_{dd}; a decrease in the supply voltage by a factor of two yields a decrease in the energy consumption by a factor of four.

In this chapter, we focus on the *system-level* power-aware resource management techniques. System-level power management technqiues can be roughly classified into two categories, dynamic voltage scaling (DVS) and dynamic power management (DPM). Dynamic voltage scaling (DVS) [7], which can be applied in both hardware and software desgin abstractions, is one of most effective design techniques in minimizing the energy consumption of VLSI systems. Since the energy consumption E of CMOS circuits has a quadratic dependency on the supply voltage, lowering the supply voltage reduces the energy consumption significantly. When a given application does not require the peak performance of a VLSI system, the clock speed (and its corresponding supply voltage) can be dynamically adjusted to the lowest level that still satisfies the performance requirement, saving the energy consumed without perceivable performance degradations.

For example, consider a task with a deadline of 25 msec, running on a processor with the 50 MHz clock speed and 5.0 V supply voltage. If the task requires $5 \cdot 10^5$ cycles for its execution, the processor executes the task in 10 msec and becomes idle for the remaining 15 msec. (We call this type of an idle interval the *slack* time.) However, if the clock speed and the supply voltage are lowered to 20 MHz and 2.0 V, it finishes the task at its deadline (= 25 msec), resulting in 84% energy reduction.

Since lowering the supply voltage also decreases the maximum achievable clock speed [43], various DVS algorithms for real-time systems have the goal of reducing supply voltage dynamically to the lowest possible level while satisfying the tasks' timing constraints. For real-time systems where timing constraints must be strictly satisfied, a fundamental energy-delay tradeoff makes it more challenging to dynamically adjust the supply voltage so that the energy consumption is minimized while not violating the timing requirements. In this chapter, we focus on DVS algorithms for hard real-time systems.

On the other hand, dynamic power management decreases the energy consumption by selectively placing idle components into lower power states. At the minimum, the device needs to stay in the low-power state for long enough (defined as the break even time) to recuperate the cost of transitioning in and out of the state. The break even time T_{BE}, as defined in Equation 1.1, is a function of the power consumption in the active state, P_{on}, the amount of power consumed in the low power state, P_{sleep}, and the cost of transition in terms of both time, T_{tr}, and power, P_{pr}. If it was possible to predict ahead of time the exact length of each idle period, then the ideal power management policy would place a device in the sleep state only when idle period will be longer than the break even time. Unfortunately, in most real systems such perfect prediction of idle period is not possible. As a result, one of the primary tasks DPM algorithms have is to predict when the idle period will be long enough to amortize the cost of transition to a low power state, and to select the state to transition to. Three classes of policies can be defined - timeout based, predictive and stochastic. The policies in each class differ in the way prediction of the length of the idle period is made, and the timing of the actual transition into the low power state (e.g., transitioning immediately at the start of an idle period vs. after some amount of idle time has passed).

$$T_{BE} = T_{tr} + T_{tr}\frac{P_{tr} - P_{on}}{P_{on} - P_{sleep}} \tag{1.1}$$

This chapter provides an overview of state-of-the-art dynamic power management and dynamic voltage scaling algorithms. The remainder of the chapter is organized as follows. An overview of dynamic voltage scaling techniques is presented in Section 2. Section 3 provides an overview of dynamic power management algorithms and the models used in deriving the policies. We conclue with a summary in Section 4.

2 Dynamic Voltage Scaling

For hard real-time systems, there are two kinds of voltage scheduling approaches depending on the voltage scaling granularity: intra-task DVS (IntraDVS) and inter-task DVS (InterDVS). The intra-task DVS algorithms adjust the voltage within an individual task boundary, while the inter-task DVS algorithms determine the voltage on a task-by-task basis at each scheduling point. The main difference between them is whether the slack times are used for the current task or for the tasks that follow. InterDVS algorithms distribute the slack times from the current task for the following tasks, while IntraDVS algorithms use the slack times from the current task for the current task itself.

2.1 Intra-Task Voltage Scaling

The main feature of IntraDVS algorithms is how to select the program locations where the voltage and clock will be scaled. Depending on the selection mechanism, we can classify IntraDVS algorithms into five categories: segment-based IntraDVS, path-based IntraDVS, memory-aware IntraDVS, stochastic IntraDVS, and hybrid IntraDVS.

2.1.1 Segment-based IntraDVS

Segment-based IntraDVS techniques partition a task into several segments [29, 33]. After executing a segment, they adjust the clock speed and supply voltage exploiting the slack times from the executed segments of a program. In determining the target clock frequency using the slack times available, different slack distribution policies are often used. For example, all the identified slack may be given to the immediately following segment. Or, it is possible to distribute the slack time evenly to all remaining segments. For a better result, the identified slack may be combined with the estimated future slacks (which may come from the following segments' early completions) for a slower execution.

A key problem of the segment-based IntraDVS is how to divide an application into segments. Automatically partitioning an application code is not trivial. For example, consider the problem of determining the granularity of speed changes. Ideally, the more frequently the voltage is changed, the more efficiently the application can exploit dynamic slacks, saving more energy. However, there is energy and time overhead associated with each speed adjustment. Therefore, we should determine how far apart any two voltage scaling points should be. Since the distance between two consecutive voltage scaling points varies depending on the execution path taken at run time, it is difficult to determine the length of voltage scaling intervals statically.

One solution is to use both the compiler and the operating system to adapt performance and reduce energy consumption of the processor. The collaborative IntraDVS [1] uses such an approach and provides a systematic methodology to partition a program into segments considering branch, loop and procedure call. The compiler does not insert voltage scaling codes between segments but annotates the application program with so-called power management hints (PMH) based on program structure and estimated worst-case performance. A PMH conveys path-specific run-time information about a program's progress to the operating system. It is very low cost instrumentation that collects path-specific information for the operating system about how the program is behaving relative to the worst-case performance. The operating system periodically invokes a power management point (PMP) to change the processor's performance based on the timing information from the power management hints. This collaborative approach has the advantage that the lightweight hints can collect accurate timing information for the operating system without actually changing performance. Further, the periodicity of performance/energy adaptation can be controlled independently of power management hints to better balance the high overhead of adaptation.

We can also partition a program based on the workload type. For example, the required decoding time for each frame in an MPEG decoder can be separated into two parts [10]: a frame-dependent (FD) part and a frame-independent (FI) part. The FD part varies greatly according to the type of the incoming frame whereas the FI part remains constant regardless of the frame type. The computational workload for an incoming frame's FD part (W_{FD}^{P}) can be predicted by using a frame-based history, i.e., maintaining a moving-average of the frame-dependent time for each frame type. The frame-independent time is not predicted since it is constant for a given video sequence (W_{FI}). Because the total predicted workload is ($W_{FD}^{P} + W_{FI}$), given a deadline D, the program starts with the cock speed f_{FD} as follows:

$$f_{FD} = \frac{W_{FD}^{P} + W_{FI}}{D} \tag{1.2}$$

For the MPEG decoder, since the FD part (such as the IDCT and motion compensation steps) is executed before the FI part (such as the frame dithering and frame display steps), the FD time prediction error is recovered inside that frame itself, i.e., during the FI part, so that the decoding time of each frame can be maintained satisfying the given frame rate. This is possible because the workload of the FI part is constant for a given video stream and easily obtained after decoding the first frame. When a misprediction occurs (which can be detected by comparing the predicted FD time (W_{FD}^{P}) with the actual FD time (W_{FD}^{A})), an appropriate action must be taken during the FI part to compensate for the misprediction.

If the actual FD time was smaller than the predicted value, there will be an idle interval before the deadline. Hence, we can scale down the voltage level during the FI part's processing. On the other hand, if the actual FD time was larger than the predicted value, a corrective action must be taken to meet the deadline. This is accomplished by scaling up the voltage and frequency during the FI part so as to make up for the lost time. Since the elapsed time consumed during the FD part is W_{FD}^{A}/f_{FD}, the FI part should start with the following cock speed f_{FI}:

$$f_{FI} = \frac{W_{FI}}{D - \frac{W_{FD}^{A}}{f_{FD}}} \tag{1.3}$$

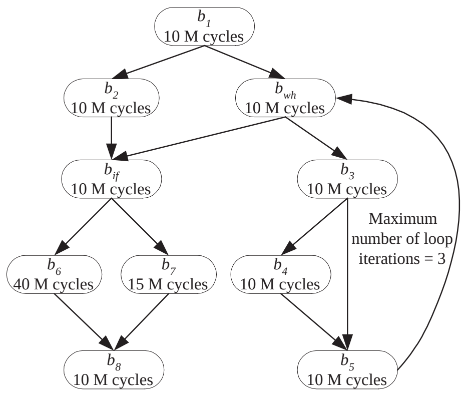
2.1.2 Path-based IntraDVS

At a specific program point, two kinds of slack times can be identified: backward slack and forward slack. While the backward slack is generated from the early completion of the executed program segments, the forward slack is generated when the change of remaining workload is estimated. Though the segment-based IntraDVS utilizes the backward slack times, the path-based IntraDVS exploits the forward slack times based on the program's control flow.

```
main() {
  S1;
  if (cond1) S2;
  else
     while (cond2) {
        S3;
        if (cond3) S4;
        S5;
     }

  if (cond4) S6;
  else S7;
  S8;
}
```

(a)



(b)

Figure 1.1: An example program P: (a) an example real-time program with the 2 seconds deadline and (b) its CFG representation G_P.

Consider a hard real-time program P with the deadline of 2 seconds shown in Figure 1.1(a). The control flow graph (CFG) G_P of the program P is shown in Figure 1.1(b). In G_P, each node represents a basic block of P and each edge indicates the control dependency between basic blocks. The number within each node indicates the number of execution cycles of the corresponding basic block. The back edge from b_5 to b_{wh} models the **while** loop of the program P. The worst-case execution cycles of this program is 2×10^8 cycles.

Path-based IntraDVS [45] adjusts the clock speed within the task depending on the control flow. For example, when the program control flow follows the execution path $\pi_1 = (b_1, b_2, b_{if}, b_6, b_8)$ of Figure 1.1(b), the clock speed is initially determined to complete the worst-case execution path (WCEP) before the deadline, i.e., 100 MHz. However, we can reduce the clock speed at the edge (b_1, b_2) because we know this control flow does not follow the WCEP. In the path-based IntraDVS algorithm, we identify appropriate program locations where the clock speed should be adjusted, and inserts clock and voltage scaling codes to the selected program locations at compile time. The branching edges of the CFG, i.e., branch or loop statements, are the candidate locations for inserting voltage scaling codes because the remaining execution cycles are changed at those locations.

The path-based IntraDVS consists of two key steps: (1) one to predict the execution path of application program at compile time and (2) the other to adjust the clock speed depending on the real execution path taken at run time. In the first step, using the predicted execution path, we calculate the remaining predicted execution cycles (RPEC) $\delta(b_i)$ at a basic block b_i which is a branching node in G_P as follows:

$$\delta(b_i) = c(b_i) + \mathcal{P}(\delta(b_j), \delta(b_k)) \tag{1.4}$$

where $c(b_i)$ is the execution cycles for the basic block b_i and $\mathcal{P}$ is the prediction function. The basic blocks b_j and b_k are the immediate successor nodes of b_i in the control flow graph. Depending on the prediction method for execution path, the function $\mathcal{P}$ is determined. For example, if we take the WCEP as an execution path to be taken at run time, $\mathcal{P}(\alpha, \beta)$ will be equal to $max(\alpha, \beta)$. With the predicted value of $\delta(b_i)$, we set the initial clock frequency and its corresponding voltage assuming that the task execution will follow the predicted execution path. We call the predicted execution path as the *reference path* because the clock speed is determined based on the execution path.

For a loop, we use the following equation to predict the remaining execution cycles for the loop L:

$$\delta(L) = c(H_L) + (c(H_L) + c(B_L)) \cdot N_{pred}(L) + \delta(post_L) \tag{1.5}$$

where $c(H_L)$ and $c(B_L)$ mean the execution cycles of the header and the body of the loop L, respectively. $N_{pred}(L)$ is the predicted number of loop iterations and $post_L$ denotes the successor node of the loop, which is executed just after the loop termination.

At run time, if the actual execution deviates from the (predicted) reference path (say, by a branch instruction), the clock speed can be adjusted depending on the difference between the remaining execution cycles of the reference path and that of the newly deviated execution path. If the new execution path takes significantly longer to complete its execution than the reference execution path, the clock speed should be *raised* to meet the deadline constraint. On the other hand, if the new execution path can finish its execution earlier than the reference execution path, the clock speed can be *lowered* to save energy.

For run-time clock speed adjustment, voltage scaling codes are inserted into the selected program locations at compile time. The branching edges of the CFG, i.e., branch or loop statements, are the candidate locations for inserting voltage scaling codes. They are called *Voltage Scaling Points* (VSPs) because the clock speed and voltage are adjusted at these points. There are two types of VSPs, the B-type VSP and L-type VSP. The B-type VSP corresponds to a branch statement while the L-type VSP maps into a loop statement. VSPs can be also categorized into Up-VSPs or Down-VSPs, where the clock speed is raised or lowered, respectively. At each VSP (b_i, b_j), the clock speed is determined using $\delta(b_i)$ and $\delta(b_j)$ as follows:

$$f(b_j) = \frac{\delta(b_j)}{T_j} = f(b_i) \cdot \frac{\delta(b_j)}{\delta(b_i) - c(b_i)} = f(b_i) \cdot r(b_i, b_j) \tag{1.6}$$

where $f(b_i)$ and $f(b_j)$ are the clock speeds at the basic blocks b_i and b_j, respectively. T_j is the remaining time until the deadline from the basic block b_j. $r(b_i, b_j)$ is the *speed update ratio* of the edge (b_i, b_j).

For a loop, if the actual number of loop iterations is N_{actual}, the clock speed is changed after the loop as follows:

$$f(post_L) = f(pre_L) \cdot \frac{max((c(H_L) + c(B_L)) \cdot (N_{actual}(L) - N_{pred}(L)), 0) + \delta(post_L)}{max((c(H_L) + c(B_L)) \cdot (N_{pred}(L) - N_{actual}(L)), 0) + \delta(post_L)} \tag{1.7}$$

where $f(pre_L)$ is the clock speed before executing the loop L. If N_{actual} is larger (smaller) than N_{pred}, the clock speed is increased (decreased).

Although the WCEP-based IntraDVS reduces the energy consumption significantly while guaranteeing the deadline, this is a pessimistic approach because it always predicts that the longest path will be executed. If we use the average-case execution path (ACEP) as a reference path, a more efficient voltage schedule can be generated. (The ACEP is an execution path that will be most likely to be executed.) To find the average-case execution path, we should utilize the profile information on the program execution.

2.1.3 Other IntraDVS Techniques

Memory-aware IntraDVS utilizes the CPU idle times due to external memory stalls. While the *compiler-driven* IntraDVS [19] identifies the program regions where the CPU is mostly idle due to memory stalls at compile time, the *event-driven* IntraDVS [51, 11] uses several performance monitoring events to capture the CPU idle time at run time. The memory-aware IntraDVS differs from the path-based IntraDVS in the type of CPU slacks being exploited. While the path-based IntraDVS takes advantage of the difference between the predicted execution path and the real execution path of applications, the memory-aware IntraDVS exploits slacks from the memory stalls. The idea is to identify the program regions in which the CPU is mostly idle due to memory stalls, and slow them down for energy reduction. If the system architecture supports the overlapped execution of the CPU and memory operations, such a CPU slow-down will not result in a serious system performance degradation, hiding the slow CPU speed behind the memory hierarchy accesses which are on the critical path. There are two kinds of approaches to identify the memory-bound regions: analyzing a program at compile time and monitoring run-time hardware events.

The compiler-directed IntraDVS [19] partitions a program into multiple program regions. It assigns different slow-down factors to different selected regions so as to maximize the overall energy savings without violating the global performance penalty constraint. The application program is partitioned not to introduce too much overhead due to switches between different voltages/frequencies. That is, the granularity of the region needs to be large enough to compensate for the overhead of voltage and frequency adjustments.

Event-driven IntraDVS [51, 11] makes use of run-time information about the external memory access statistics in order to perform CPU voltage and frequency scaling with the goal of minimizing the energy consumption while controlling the performance penalty. The technique relies on dynamically-constructed regression models that allow the CPU to calculate the expected workload and slack time for the next time slot. This is achieved by estimating and exploiting the ratio of the total off-chip access time to the total on-chip computation time. To capture the CPU idle time, several performance monitoring events (such as ones collected by the performance monitoring unit (PMU) of the XScale processor) can be used. Using the performance monitoring events, we can count the number of instructions being executed and the number of external memory accesses at run time.

Stochastic IntraDVS uses the stochastic information on the program's execution time [17, 31]. This technique is motivated by the idea that, from the energy consumption perspective, it is usually better to "start at low speed and accelerate execution later when needed" than to "start at high speed and reduce the speed later when the slack time is found" in the program execution. It finds a speed schedule that minimizes the expected energy consumption while still meeting the deadline. A task starts executing at a low speed and then gradually accelerates its speed to meet the deadline. Since an execution of a task might not follow the WCEP, it can happen that high speed regions are avoided.

Hybrid IntraDVS overcomes the main limitation of IntraDVS techniques described before, which is that they have no global view of the task set in multi-task environments. Based on an observation that a cooperation between IntraDVS and InterDVS could result in more energy efficient systems, the *hybrid IntraDVS* technique selects either the *intra mode* or the *inter mode* when slack times are available during the execution of a task [44]. At the inter mode, the slack time identified during the execution of a task is transferred to the following other tasks. Therefore, the speed of the current task is not changed by the slack time produced by itself. At the intra mode, the slack time is used for the current task, reducing its own execution speed.

2.2 Inter-Task Voltage Scaling

InterDVS algorithms exploit the "run-calculate-assignrun" strategy to determine the supply voltage, which can be summarized as follows: (1) run a current task, (2) when the task is completed, calculate the maximum allowable execution time for the next task, (3) assign the supply voltage for the next task, and (4) run the next task. Most InterDVS algorithms differ during step (2) in computing the maximum allowed time for the next task τ which is the sum of the worst-case execution time (WCET) of τ and the slack time available for τ.

A generic InterDVS algorithm consists of two parts: slack estimation and slack distribution. The goal of the slack estimation part is to identify as much slack times as possible while the goal of the slack distribution part is to distribute the resulting slack times so that the resulting speed schedule is as uniform as possible. Slack times generally come from two sources; static slack times are the extra times available for the next task that can be identified statically, while dynamic slack times are caused from run-time variations of the task executions.

2.2.1 Slack Estimation and Distribution Methods

One of the most commonly used *static* slack estimation methods is to compute the maximum constant speed, which is defined as the lowest possible clock speed that guarantees the feasible schedule of a task set [46]. For example, in EDF scheduling, if the worst case processor utilization (WCPU) U of a given task set is lower than 1.0 under the maximum speed f_{max}, the task set can be scheduled with a new maximum speed $f'_{max} = U \cdot f_{max}$. Although more complicated, the maximum constant speed can be statically calculated as well for RM scheduling [46, 17].

Three widely-used techniques of estimating *dynamic* slack times are briefly described below. *Stretching-*

to-NTA is based on a slack between the deadline of the current task and the arrival time of the next task. Even though a given task set is scheduled with the maximum constant speed, since the actual execution times of tasks are usually much less than their WCETs, the tasks usually have dynamic slack times. One simple method to estimate the dynamic slack time is to use the arrival time of the next task [46]. (The arrival time of the next task is denoted by NTA.) Assume that the current task τ is scheduled at time t. If NTA of τ is later than $(t + WCET(\tau))$, task τ can be executed at a lower speed so that its execution completes exactly at the NTA. When a single task τ is activated, the execution of τ can be stretched to NTA. When multiple tasks are activated, there can be several alternatives in stretching options. For example, the dynamic slack time may be given to a single task or distributed equally to all activated tasks.

Priority-based slack stealing exploits the basic properties of priority-driven scheduling such as RM and EDF. The basic idea is that when a higher-priority task completes its execution earlier than its WCET, the following lower-priority tasks can use the slack time from the completed higher-priority task. It is also possible for a higher-priority task to utilize the slack times from completed lower-priority tasks. However, the latter type of slack stealing is computationally expensive to implement precisely. Therefore, the existing algorithms are based on heuristics [2, 27].

Utilization updating is based on a observation that the actual processor utilization during run time is usually lower than the worst case processor utilization. The utilization updating technique estimates the required processor performance at the current scheduling point by recalculating the expected worst case processor utilization using the actual execution times of completed task instances [38]. When the processor utilization is updated, the clock speed can be adjusted accordingly. The main merit of this method is its simple implementation, since only the processor utilization of completed task instances have to be updated at each scheduling point.

In distributing slack times, most InterDVS algorithms have adopted a greedy approach, where all the slack times are given to the next activated task. This approach is not an optimal solution, but the greedy approach is widely used because of its simplicity.

2.2.2 Example InterDVS Algorithms

In this section, we briefly summarize some of the representative DVS algorithms proposed for hard real-time systems. Here, eight InterDVS algorithms are chosen, two [38, 46] of which are based on the RM scheduling policy, while the other six algorithms [38, 46, 2, 27] are based on the EDF scheduling policy.

In these selected DVS algorithms, one or sometimes more than one slack estimation methods explained in the previous section were used. In lppsEDF and lppsRM which were proposed by Shin et al. in [46], slack time of a task is estimated using the maximum constant speed and Stretching-to-NTA methods.

The ccRM algorithm proposed by Pillai et al. [38] is similar to lppsRM in the sense that it uses both the maximum constant speed and the Stretching-to-NTA methods. However, while lppsRM can adjust the voltage and clock speed only when a single task is active, ccRM extends the stretching to NTA method to the case where multiple tasks are active.

Pillai et al. also proposed two other DVS algorithms [38], ccEDF and laEDF, for EDF scheduling policy. These algorithms estimate slack time of a task using the utilization updating method. While ccEDF adjusts the voltage and clock speed based on run-time variation in processor utilization alone, laEDF takes a more aggressive approach by estimating the amount of work required to be completed before NTA.

DRA and AGR, which were proposed by Aydin et al. in [2], are two representative DVS algorithms that are based on the priority-based slack stealing method. The DRA algorithm estimates the slack time of a task using the prioritybased slack stealing method along with the maximum constant speed and the Stretching-to-NTA methods. Aydin et al. also extended the DRA algorithm and proposed another DVS algorithm called AGR for more aggressive slack estimation and voltage/clock scaling. In AGR, in addition to the priority-based slack stealing, more slack times are identified by computing the amount of work required to be completed before NTA.

lpSHE is another DVS algorithm which is based on the priority-based slack stealing method [27]. Unlike DRA and AGR, lpSHE extends the priority-based slack stealing method by adding a procedure that estimates the slack time from lower-priority tasks that were completed earlier than expected. DRA, AGR, and lpSHE algorithms are somewhat similar to one another in the sense that all of them use the maximum constant speed in the off-line phase and the Stretching-to-NTA method in the on-line phase in addition to the priority-based slack stealing method.

3 Dynamic Power Management

Dynamic power management (DPM) techniques selectively place system components into low-power states when they are idle. A power managed system can be modeled as a *power state machine*, where each state is characterized by the power consumption and the performance. In addition, state transitions have power and delay cost. Usually, lower power consumption also implies lower performance and longer transition delay. When a component is placed into a low-power state, such as a sleep state, it is unavailable for the time period spent there, in addition to the transition time between the states. The transitions between states are controlled by commands issued by a *power manager* (PM) that observes the workload of the system and decides when and how to force power state transitions. The power manager makes state transition decisions according to the *power management policy.* The choice of the policy that minimizes power under performance constraints (or maximizes performance under power constraint) is a constrained optimization problem.

The system model for power management therefore consists of three components: the user, the device and the queue as shown in Figure 3. The user, or the application that accesses each device by sending requests, can be modeled with a request interarrival time distribution. When one or more requests arrive, the user is said to be in the active state, otherwise it is in the idle state. Figure 3 shows three different power states for the device: active, idle and sleep. Often the device will have multiple active states, which can be differentiated by the frequency and voltage of operation. Similarly, there can be multiple inactive states - both idle, each potentially corresponding to a specific frequency of operation in the active state (e.g., XScale processor [23]) , and sleep states. Service time distribution describes the behavior of the device in the active state. When the device is in either the idle or the sleep state, it does not service any requests. Typically, the transition to the active state is shorter from the idle state, but the sleep state has lower power consumption. The transition distribution models the time taken by the device to transition between its power states. The queue models a buffer associated with each device. The combination of interarrival time distribution (incoming requests to the buffer) and service time distribution (requests leaving the buffer) can

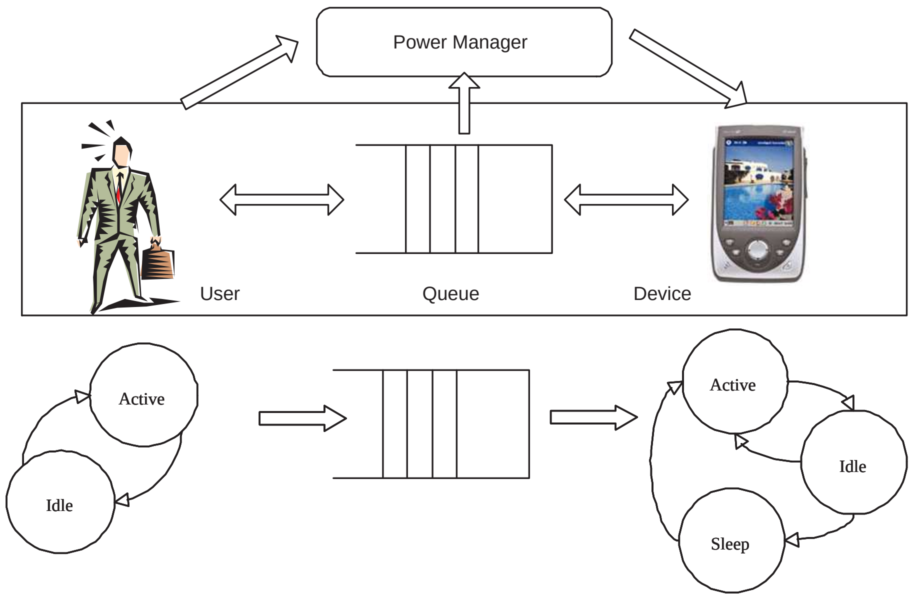


Figure 1.2: System model.

be used to fully characterize the behavior of the queue.

Power manager's job can consist of a number of tasks: (1) tracking and modeling of incoming service requests with a goal of predicting when longer idle periods occur; (2) traffic shaping - buffering the traffic so larger idle periods are created that enable the device to stay asleep for a longer time period; (3) deciding when and to what sleep state a device should transition to; (4) making a decision as to when a device should wake up to process requests. In this section we provide an overview of DPM policies that address various combinations of these tasks.

3.1 Heuristic policies

Most commercial power management implementations focus only on deciding when a device should go to sleep. The cost of transition between the active and the sleep state typically determines how aggressive a policy can be. When the cost is reasonably low, or is at least not perceivable by the users, then policies transition device to sleep as soon as it becomes idle. For example, μSleep has been introduced in [6] as a way of reducing the idle power consumption of HP's IPAQ platform. IBM's wristwatch enters standby state as soon as idle [25]. Intel's QuickStart technology [22] puts a processor to sleep between the keystrokes. In all three cases the cost of waking up is not perceivable by the user. When the cost of transition to a low power state is significant, then commercial implementations typically implement policies based on timeout. For example, HP's IPAQ uses a timeout value to decide when display should be dimmed. The timeout value is usually either fixed at design time, or can be set by a user.

Timeout policies assume that if the incoming workload has an idle period that is longer than some timeout value T_{to}, then there is a very high likelihood that the idle period will be long enough to justify going to a sleep state. Therefore, the total length of the idle time needs to be longer than $T_{to} + T_{be}$. The timeout policies waste energy during the timeout period, as they keep a device in the active state until the timeout expires. Therefore, shorter timeout values are better for saving energy while waiting to transition to sleep. On the other hand, if the timeout value is too long, then there is a good chance that the rest of the idle period is not long enough to amortize the cost of transition to sleep, so the overall cost is actually then higher than it would have been if the device had not gone to sleep at all. A good example of this situation is setting too short of a timeout on a hard drive. Typical hard drive can take up to a few seconds to spin up from sleep. The spinning up process can cost more than twice the power consumption of the active state. Thus, if the timeout value is only a few hundred milliseconds, chances are that a number of times just as the hard drive spins down, it'll have to immediately spin back up, thus causing large performance and energy overhead. Therefore, the selection of the timeout value has to be done with a good understanding of both device characteristics and the typical workloads. A study of hard drive traces is presented in [30]. One of the major conclusions is that timeout values on the order of multiple seconds are appropriate for many hard drives. This timeout value for hard disks can be shown to be within a factor of two of the optimal policy using competitive ratio analysis [26].

Although most commercial implementations have a single fixed timeout value, a number of studies have designed methods for adapting the timeout value to changes in the workloads (e.g., [13, 18, 28, 24]). Machine learning techniques [18], along with models based on economic analysis [28, 24] have been employed as ways to adapt the timeout value. In [42] adaptive policy learns the distribution of idle periods and based on that it selects which sleep state is most appropriate. Competitive analysis is then used to show how close the adaptive policy is to optimum. Adaptive timeout policies also suffer from wasting energy while waiting for the timeout to expire, but hopefully the amount of energy wasted is lower as the timeout value is more closely fine tuned to the changes in the workload.

Predictive policies attempt to not only predict the length of the next idle period by studying the distribution of request arrivals , but also try to do so with enough accuracy to be able to transition a device into a sleep state with no idleness. In addition, some predictive policies also wake up a device from the sleep state in anticipation of service request arrival. When prediction of timing and the length of the idle period is correct, then predictive policies provide a solution with no overhead. On the other hand, if the prediction is wrong, then potential cost can be quite large. If a power manager transitions a device to sleep expecting a long idle period, but the idle period is actually short, the overall cost in terms of both energy and performance can be quite large. On the other hand, if the manager predicts an idle period that is short, then it'll wake up a device before it is needed and thus waste energy the same way standard timeout policies do. The quality of idle period prediction is the keystone of these policies. A study on prediction of idleness in hard drives is presented in [16]. Policies based on study of idle period prediction are presented in both [16] and [50]. Two policies are introduced in [50]: the first one is based on a regression model of idle periods, while the second one is based on the analysis of a length of a previous busy period. Exponential averaging is used to predict the idle period length in [20]. Analysis of user interface interactions is used to guide a decision on when to wake up a given component [54]. In multi core designs a signal has been proposed that can be used to notify the system components of impeding request arrival and of instances when no more requests are expected [49]. These signals enable both predictive wakeup and predictive sleep. Both timeout and predictive policies have one thing in common - they are heuristic. None of these policies can guarantee optimality. In contrast, policies that use stochastic models to formulate and solve the power management problem can provide optimal solutions within restrictive assumption made when developing the system model.

3.2 Stochastic Policies

Stochastic policies can loosely be categorized into time and event-driven, and based on assumption that all distributions modeling the system are memoryless (e.g., geometric and exponential distribution) or that some

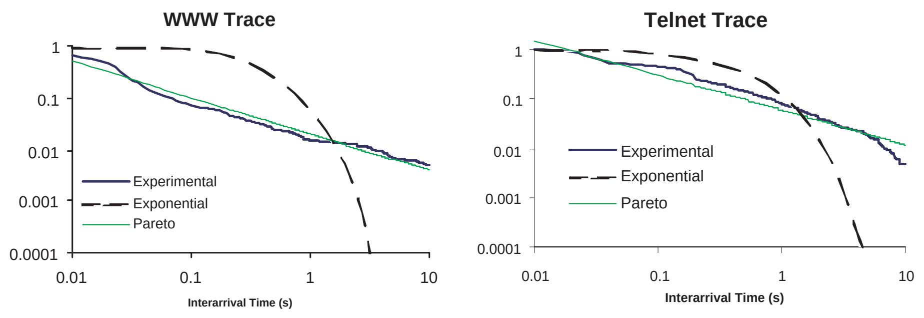


Figure 1.3: WLAN idle state arrival tail distribution.

distributions are history dependent. Power management policies can be classified into two categories by the manner in which decisions are made: *discrete time* (or clock based) and *event driven.* In addition, policies can be *stationary* (the same policy applies at any point in time) or *non-stationary* (the policy changes over time). In both discrete and event-driven approaches optimality of the algorithm can be guaranteed since the underlying theoretical model is based on Markov chains. An overview of stochastic DPM policies follows.

Benini et al. [4] formulated a probabilistic system model using stationary discrete-time Markov decision processes (DTMDP). They rigorously formulate the policy optimization problem and showed that it can be solved exactly and in polynomial time in the size of the system model. The DTMDP approach requires that all state transitions follow stationary geometric distributions, which is not true in many practical cases. Non-stationary user request rates can be treated using an adaptive policy interpolation procedure presented in [12]. A limitation of both stationary and adaptive DTMDP policies is that decision evaluation is repeated periodically, even when the system is idle, thus wasting power. For example, for a 10 W processor, the DTMDP policy with evaluation period of 1s would waste as much as 1800 J of energy from the battery during a 30min break. The advantage of the discrete time approach is that decisions are re-evaluated periodically so the decision can be reconsidered thus adapting better to arrivals that are not truly geometrically distributed.

An alternative to the DTMDP model is a continuous-time Markov decision process (CTMDP) model [40]. In a CTMDP, the power manager (PM) issues commands upon event occurrences instead of at discrete time settings. As a result, more energy can be saved since there is no need to continually re-evaluate the policy in the low power state. Results are guaranteed optimal assuming that the exponential distribution describes well the system behavior. Unfortunately, in many practical cases the transition times may be distributed according to a more general distribution. Figure 1.3 shows a tail distribution of wireless LAN (WLAN) service request interarrival times. The plot highlights that for longer idle times of interest to power management, the exponential distribution shows a very poor fit, while a heavy tailed distribution, such as Pareto, is a much better fit to the data. As a result, in real implementation the results can be far from optimal [48]. Work presented in [39] uses series and parallel combinations of exponential distributions to approximate general distribution of transition times. Unfortunately, this approach is very complex and does not give a good approximation for the bursty behavior observed in real systems [48, 37].

Time-Indexed Semi-Markov Decision Process (TISMDP) model has been introduced to solve the DPM optimization problem without the restrictive assumption of memoryless distribution for system transitions [48]. The power management policy optimization is solved *exactly* and in polynomial time with guaranteed optimal results. Large savings are measured with TISMDP model as it handles general user request interarrival distributions and make decisions in the event-driven manner. The resulting DPM policy is event driven and can be implemented as a randomized timeout. The value of randomization depends on the parameters of the policy derived from TISMDP optimization. The policy itself is in to form of a distribution that provides the probability of transitioning into a sleep state for every timeout value. Renewal theory has been used in [49] for joint optimization of both DPM and DVS on multi core systems-on-a-chip. As in TISMDP, the renewal model allows for modeling a general transition distribution. Probabilistic model checking has been employed to verify the correctness of stochastic policies [47]. Although both TISMDP and renewal models limit the restriction of only memoryless distributions, they do require that all transitions be statistically independent and stationary. An approach presented in [24] removes the stationary assumption by learning the interarrival distribution online. Although this approach does not guarantee optimality, the quality of estimation has been evaluated using competitive analysis.

3.3 OS and cross-layer DPM

Most power management policies are implemented as a part of an operating system. Intel, Microsoft and Toshiba created an Advanced Configuration and Power Interface specification (ACPI) that can be used to implement power management policies in Windows OS [21]. ACPI gives only a structure for implementing

DPM, but does not actually provide any policies beyond simple timeouts for specific devices. In contrast, NemesisOS and ECOSystem both use pricing mechanisms to efficiently allocate system's hardware resources based on energy consumption. Both of these approaches use high level approximations for energy cost of access to any given hardware resource, thus incurring significant inaccuracies. Their implementation for a different system requires that both kernel and the hardware specific functions be changed. Another OS level approach is to compute an equivalent utilization for each hardware component and then to transition a device to sleep when its utilization falls below a predefined threshold [32]. A few energy-aware software implementations integrate information available at multiple system layers to be able to manage energy more efficiently [15, 52].

All DPM policies discussed thus far do no perform any traffic reshaping of the incoming workload. Thus their primary goal is to evaluate for a given idle period if a device should transition to sleep. Various buffering mechanisms have been suggested by a number of researchers as a way to enhance the availability of longer idle periods suitable for power management. Buffering for streaming multimedia applications has be proposed as a methodology to lower the energy consumption in wireless network interface cards [5]. Similar techniques have been used to help increase the idle times available for spinning down the hard disk power [36]. A general buffer management methodology based on the idea of inventory control has been proposed in [8]. An advantage of this approach is that it can be applied to multiple devices in a single system. When combined with OS level scheduling, the adaptive workload buffering can be used for both power management and voltage scaling. Shorter idle times created by adaptive buffering can be used to slow down a device, while the longer ones are more appropriate for transitions to sleep.

4 Conclusions

We have described two representative system-level power-aware resource management approaches for low-power embedded systems. DVS techniques take advantage of workload variations within a single task execution as well as workload fluctuations from running multiple tasks, and adjust the supply voltage, reducing the energy consumption of embedded systems. DPM techniques identify idle system components using various heuristics and place the identified components into low-power states. We reviewed the key steps of these techniques.

In this chapter, we reviewed DVS and DPM as an independent approach. Since both approaches are based on the system idleness, DVS and DPM can be combined into a single power management framework. For example, shorter idle periods are more amiable to DVS, while longer ones are more appropriate for DPM. Thus, a combination of the two approaches is necessary for more power-efficient embedded systems.

References

[1] N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem and M. Craven, Energy management for real-time embedded applications with compiler support, in *Proc. of Conference on Language, Compiler, and Tool Support for Embedded Systems*, pp. 238-246, 2002.

[2] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez, Dynamic and aggressive scheduling techniques for power-aware real-time systems, in *Proc. of IEEE Real-Time Systems Symposium*, December 2001.

[3] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*, Kluwer, 1997.

[4] L. Benini, G. Paleologo, A. Bogliolo and G. DeMicheli, Policy optimization for dynamic power management, *IEEE Transactions on Computer-Aided Design*, 18(6), pp. 813-833, June 1999.

[5] D. Bertozzi, L. Benini, and B. Ricco, Power aware network interface management for streaming multimedia, in *Proc. of IEEE Wireless Communnication Network Conf.*, pp. 926-930, March 2002.

[6] L. S. Brakmo, D. A. Wallach, and M. A. Viredaz, Sleep: A technique for reducing energy consumption in handheld devices, in *Proc. of USENIX/ACM Int. Conf. Mobile Systems, Applications, & Services*, pp. 12-22, June 2004.

[7] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, A dynamic voltage scaled microprocessor system, *IEEE Journal of Solid State Circuits*, 35(11), 2000.

[8] L. Cai and Y.-H. Lu, Energy management using buffer memory for streaming data, *IEEE Trans. Computer-Aided Design of IC & Systems*, 24(2), pp. 141-152, February 2005.

[9] A. Chandrakasan and R. Brodersen, *Low Power Digital CMOS Design*, Kluwer, 1995.

[10] K. Choi, W-C. Cheng and M. Pedram, Frame-based dynamic voltage and frequency scaling for an MPEG player, *Journal of Low Power Electronics*, 1(1), pp. 27-43, 2005.

[11] K. Choi, R. Soma and M. Pedram, Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times, *IEEE Transactions on Computer Aided Design*, 24(1), pp. 18-28, 2005.

[12] E. Chung, L. Benini and G. De Micheli, Dynamic power management for non-stationary service requests, in *Proc. of Design, Automation and Test in Europe*, pp. 77-81, 1999.

[13] F. Douglis, P. Krishnan, and B. N. Bershad, Adaptive disk spin-down policies for mobile computers, in *Proc. of 2nd USENIX Symp. Mobile & Location-Independent Computing*, pp. 121-137, April 1995.

[14] C. Ellis, The case for higher-level power management, in *Proc. of the 7th IEEE Workshop on Hot Topics in Operating Systems*, pp. 162-167, 1999.

[15] J. Flinn and M. Satyanarayanan, Managing battery lifetime with energy-aware adaptation, *ACM Trans. Computer Systems*, 22(2), pp. 137-179, May 2004.

[16] R. Golding, P. Bosch, and J. Wilkes, Idleness is not sloth, in *Proc. of USENIX Winter Tech. Conf.*, pp. 201-212, January 1995.

[17] F. Gruian, Hard real-time scheduling using stochastic data and DVS processors, in *Proc. of International Symposium on Low Power Electronics and Design*, pp. 46-51, 2001.

[18] D. P. Helmbold, D. D. E. Long, and B. Sherrod, A dynamic disk spin-down technique for mobile computing, in *Proc. of ACM Ann. Int. Conf. Mobile Computing & Networking*, pp. 130-142, November 1996.

[19] C-H. Hsu and U. Kremer, The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction, in *Proc. of ACM SIGPLAN Conference on Programming Languages, Design, and Implementation*, pp. 38-48, 2003.

[20] C. -H. Hwang and A. Wu, A predictive system shutdown method for energy saving of event-driven computation, in *Proc. of International Conference on Computer Aided Design*, pp. 28-32, 1997.

[21] Intel, Microsoft and Toshiba, *Advanced Configuration and Power Interface specification*, Intel, Microsoft, Toshiba, 1996.

[22] Intel QuickStart Technology, www.intel.com.

[23] Intel XScale Technology, http://developer.intel.com/design/intelxscale, 2001.

[24] S. Irani, S. Shukla, and R. Gupta, Online strategies for dynamic power management in systems with multiple power-saving states, *ACM Trans. Embedded Computing Systems*, 2(3), pp. 325-346, July 2003.

[25] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami, Energy tradeoffs in the IBM wristwatch computer, in *Proc. of IEEE Int. Symp. Wearable Computers*, pp. 133-140, 2001.

[26] A. Karlin, M. Manesse, L. McGeoch and S. Owicki, Competitive randomized algorithms for nonuniform problems, *Algorithmica*, pp. 542-571, 1994.

[27] W. Kim, J. Kim, and S. L. Min, A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis, in *Proc. of Design, Automation and Test in Europe (DATE'02)*, pp. 788-794, March 2002.

[28] P. . Krishnan, P. Long, and J. Vitter, Adaptive disk spindown via optimal rent-to-buy in probabilistic environments, *Algorithmica*, 23(1), pp. 31-56, January 1999.

[29] S. Lee and T. Sakurai, Run-time voltage hopping for low-power real-time systems, in *Proc. of Design Automation Conference*, pp. 806-809, 2000.

[30] K. Li, R. Kumpf, P. Horton, and T. E. Anderson, A quantitative analysis of disk drive power management in portable computers, in *Proc. of USENIX Winter Tech. Conf.*, pp. 279-291, January 1994.

[31] J. R. Lorch and A. J. Smith, Improving dynamic voltage scaling algorithms with PACE, in *Proc. of ACM SIGMETRICS Conference*, pp. 50-61, 2001.

[32] Y.-H. Lu, L. Benini, and G. De Micheli, Power-aware operating systems for interactive systems, *IEEE Trans. VLSI Systems*, 10(2), pp. 119-134, April 2002.

[33] D. Mosse, H. Aydin, B. Childers and R. Melhem, Compiler-assisted dynamic power-aware scheduling for real-time applications, in *Proc. of Workshop on Compiler and OS for Low Power*, 2000.

[34] W. Nabel, J. Mermet (Editors), *Lower Power Design in Deep Submicron Electronics*, Kluwer, 1997.

[35] R. Neugebauer and D. McAuley, Energy is just another resource: energy accounting and energy pricing in the Nemesis OS, in *Proc. of Workshop on Hot Topics in Operating Systems*, May 2001.

[36] A. E. Papathanasiou and M. L. Scott, Energy efficient prefetching and caching, in *Proc. of USENIX Ann. Tech. Conf.*, pp. 255-268, 2004.

[37] V. Paxson, S. Floyd, Wide area traffic: the failure of poisson modeling, *IEEE Transactions on Networking*, 3(3), pp. 226-244, June 1995.

[38] P. Pillai and K. G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pp. 89-102, October 2001.

[39] Q. Qiu and M. Pedram, Dynamic power management of complex Systems using generalized stochastic Petri nets, in *Proc. of Design Automation Conference*, pp. 352-356, 2000.

[40] Q. Qiu and M. Pedram, Dynamic power management based on continuous-time Markov decision processes, in *Proc. of Design Automation Conference*, pp. 555-561, 1999.

[41] J. Rabaey, M. Pedram (Editors), *Low Power Design Methodologies*, Kluwer, 1996.

[42] D. Ramanathan, S. Irani, and R. Gupta, Latency effects of system level power management algorithms, in *Proc. of IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 350-356, November 2000.

[43] T. Sakurai and A. Newton, Alpha-power law MOSFET model and its application to CMOS inverter delay and other formulas, *IEEE Journal of Solid State Circuits*, 25(2), pp. 584-594, 1990.

[44] D. Shin, W. Kim, J. Jeon and J. Kim, SimDVS: an integrated simulation environment for performance evaluation of dynamic voltage scaling algorithms, *Lecture Notes in Computer Science*, 2325, pp. 141-156, 2003.

[45] D. Shin, J. Kim and S. Lee, Intra-task voltage scheduling for low-energy hard real-time applications, *IEEE Design and Test of Computers*, 18(2), pp. 20-30, 2001.

[46] Y. Shin, K. Choi, and T. Sakurai, Power optimization of real-time embedded systems on variable speed processors, in *Proc. of the International Conference on Computer-Aided Design*, pp. 365-368, November 2000.

[47] S. Shukla and R. Gupta, A model checking approach to evaluating system level power management for embedded systems, in *Proc. of HLDVT*, 2001.

[48] T. Simunic, L. Benini, P. Glynn, G. De Micheli, Event-driven power management, *IEEE Transactions on CAD*, pp.840-857, July 2001.

[49] T. Simunic, S. Boyd, and P. Glynn, Managing power consumption in networks on chips, *IEEE Transactions on VLSI*, pp. 96- 107, January 2004.

[50] M. B. Srivastava, A. P. Chandrakasan, and R.W. Brodersen, Predictive system shutdown and other architectural techniques for energy efficient programmable computation, *IEEE Trans. VLSI Systems*, 4(1), pp. 42-55, January 1996.

[51] A. Weissel and F. Bellosa, Process cruise control: event-driven clock scaling for dynamic power management, in *Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 238-246, 2002.

[52] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets, GRACE: cross-layer adaptation for multimedia quality and battery energy, *IEEE Trans. Mobile Computing*, 2005.

[53] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, ECOSystem: managing energy as a first class operating system resource, in *Proc. of Int. Conf. Architectural Support for Programming Languages & Operating Systems*, pp. 123-132, October 2002.

[54] L. Zhong and N. K. Jha, Dynamic power optimization for interactive systems, in *Proc. of Int. Conf. VLSI Design*, pp. 1041-1047, January 2004.

Index