# Power-Aware Operating Systems for Interactive Systems

Yung-Hsiang Lu, Luca Benini, *Member, IEEE*, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—Many portable systems deploy operating systems (OS) to support versatile functionality and to manage resources, including power. This paper presents a new approach for using OS to reduce the power consumption of IO devices in interactive systems. Low-power OS observes the relationship between hardware devices and processes. The OS kernel estimates the utilization of a device from each process. If a device is not used by any running process, the OS puts it into a low-power state. This paper also explains how scheduling can facilitate power management. When processes are properly scheduled, power reduction can be achieved without degrading performance. We implemented a prototype on Linux to control two devices; experimental results showed nearly 70% power saving on a network card and a hard disk drive.**

***Index Terms*—Interactive systems, operating systems, power management, scheduling.**

## I. Introduction

DUE TO rapid advance in hardware, electronic systems support wide ranges of applications and often deploy operating systems, such as Windows, Palm OS and Linux in personal computers, personal digital assistants (PDAs) and thin clients. Operating systems (OS) have two major roles: providing an abstraction and managing resources. A file system is an abstraction of storage; programs create files without knowing the number of cylinders on hard disks. Similarly, programs transfer files through networks without considering the bandwidth of the network cards. An OS also manages resources, such as CPU time, memory allocation, and disk quota. Power is a precious resource; hence, it should be properly managed. Reducing power consumption has become one major goal in designing electronic systems. Lower power consumption prolongs operation hours of battery-powered systems. High power raises temperatures and deteriorates reliability. Rising concern about the environmental impact of electronic systems further highlights the importance of power reduction [1].

Power-reduction techniques can be classified into static and dynamic [2]. Static techniques are applied at design time, such as compilation and synthesis for low power. Dynamic techniques are applied at run time based on the variations in workloads. These techniques are called *dynamic power management* (DPM) [3]; DPM changes *power states* at run time. When high performance is required, DPM allows hardware to consume more power; otherwise, the hardware enters a lower-power state. DPM techniques include dynamic voltage/frequency scaling (DVS/DFS) and clock gating. DVS and/or DFS have been implemented in commercial products such as Transmeta's Crusoe processors [4] and the StrongARM processors [5]. In addition to processors, DPM can reduce power of input–output (IO) devices [6], such as hard disks, network cards, and displays. IO devices are different from processors in two major ways. First, they often have fewer power states; many devices have only two power states. Second, they take much longer time to change power states, up to several seconds.When an IO device is not used (also called *idle* [7]), it can enter a low-power *sleeping state*; this is called "shut down." When a device is being used (*busy*), it has to stay in a high-power *working state*. Most existing power management schemes consider whether a device is idle or busy, regardless of the reason it is idle or busy. Hardware devices are busy to serve requests from software; software provides valuable information for power reduction. Such information is available during compilation or at run time. Recently, power reduction through compilers and OS has attracted great interest in research community [8]. This paper concentrates on saving the power of IO devices using shutdown techniques through operating systems.

In this paper, we present a new method in OS to reduce the power of IO devices. We target single-processor interactive systems such as laptop computers. Our approach is divided into two parts. First, the OS kernel observes the relationship between devices and processes to estimate the utilization of each device. When the utilization is low, the OS puts this device into a sleeping state. Second, the OS provides a system call that allows application programs to specify their future hardware requirements. Then, the OS schedules processes to reduce power without degrading performance. We implemented a prototype in Linux on a notebook computer; it saved nearly 70% power on a Linksys Ethernet card and on a Hitachi 2.5″ hard disk drive. This paper has three contributions. First, we point out the importance to distinguish individual processes for power management. Second, we propose power-aware scheduling for IO requests. Finally, we implement our approach in Linux and obtain significant power saving of two IO devices.

This paper is organized as follows. Section II provides the background of OS resource management and power management. Section III describes related work. In Section IV, we present a new method to estimate device utilization in OS kernel

Manuscript received January 11, 2001; revised September 9, 2001. This work was supported in part by Gigascale Silicon Research Center and in part by NSF under Contract CCR-9901190.

Y.-H. Lu and G. De Micheli are with the Department of Electrical Engineering at Stanford University, Stanford, CA 94305 USA (e-mail: luyung@stanford.edu; nanni@stanford.edu).

L. Benini is with the Department of Electronics and Computer Science at University of Bologna, Bologna 40136 Italy (e-mail: lbenini@deis.unibo.it).

Publisher Item Identifier S 1063-8210(02)00480-8.

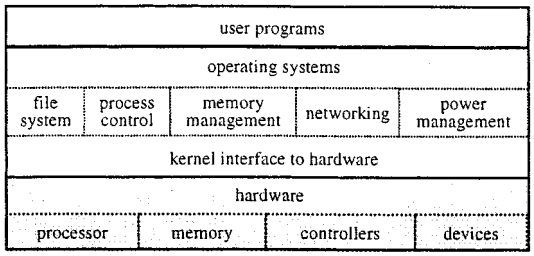


Fig. 1. Layers of computers.

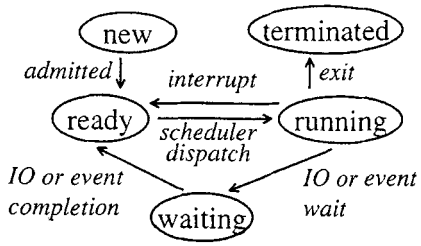


Fig. 2. Process states.

and the condition to shut down a device. Section V explains how to improve power management by scheduling. We propose a low-power scheduling algorithm for interactive systems in Section VI. Section VII describes the implementation on Linux. Our experimental results are shown in Section VIII. Finally, Section IX concludes the study and points out directions for future research.

## II. Background

### A. Resource Management by Operating Systems

Conceptually, computers are structured in layers as shown in Fig. 1 [9]. At the bottom, there are hardware components, such as processors, memory, and IO devices. OS communicates with the hardware through privileged instructions and basic IO system (BIOS) calls. OS provides services such as file systems, process control, and memory management. User programs access hardware by issuing *system calls* through OS. An OS manages resources, including CPU time, memory allocation, and disk space.

When a user starts a program, a *process* is created. This process occupies memory and takes CPU time; it may also read or write files to a hard disk. A process is an instantiation of a program. Fig. 2 shows the life of a process [9]. It is created, runs, and finally terminates. Most operating systems support *multiprogramming*: many processes can execute concurrently and share resources. Two processes are *concurrent* if one starts before the other terminates; namely, their execution times overlap. When a process is *alive* (between its creation to termination), operating systems manage when it occupies a processor, how much memory it possesses, which files it opens, and which IO devices it uses. Through the services from OS, each process has the illusion that the whole machine is dedicated to it. When multiple processes require the same resource, such as CPU or hard disks, the operating systems determine their access order; this is called *scheduling*. Commonly adopted scheduling schemes include round-robin, priority, and first-in–first-out (FIFO) [9].

### B. Dynamic Power Management

Most computers do not operate at their peak performance continuously. Some devices are idle even when other devices are busy. Examples of IO devices on personal computers include hard disk drives, network interface cards, and displays. Occasional idleness provides opportunities for power reduction [7]. DPM puts idle devices into sleeping states (sometimes called standby) to reduce their power consumption [3]. DPM provides the illusion that devices are always ready to serve requests (abstraction) even though they occasionally sleep and save power (management). DPM can be controlled by hardware, software, or the collaboration of both.

*1) Concept of Power Management:* DPM changes the power states of a device based on the variations of workloads. A workload consists of the requests generated from all processes. Workloads on a disk are read and write commands; these commands may come from a text editor such as `emacs` or a compiler such as `gcc`. Workloads on a network card are packets; these packets may come from `netscape` or `telnet`.

Fig. 3 illustrates the concept of power management. When there are requests to serve, the device is busy; otherwise, the device is idle. In this figure, the device is idle between $t_1$ and $t_3$. When the device is idle, it can enter a low-power sleeping state. Changing power states takes time; $t_{sd}$ and $t_{wu}$ are the shutdown and wakeup delays. These delays can be substantial: waking up a disk or a display takes several seconds, or hundreds of millions of processor cycles. Furthermore, waking up a sleeping device may take extra energy. In other words, power management has overhead. If there were no overhead, power management would be trivial—shutting down a device whenever it is idle. Unfortunately, there is overhead; a device should sleep only if the saved energy can justify the overhead. The rules to decide when to shut down a device are called power-management *policies*.

*2) Break-Even Time:* The *break-even time* ($t_{be}$) is the minimum length of an idle period to save power [6]; it depends on the device and is independent of requests or policies (Table I summarizes the symbols and their meanings in this paper). Consider a device whose state-transition delay is $t_o$ and the transition energy is $e_o$. Suppose its power in the working and sleeping states is $p_w$ and $p_s$ respectively. Fig. 4 shows two cases: keeping the device in the working state or shutting it down. The break-even time makes energy in both cases equal; it can be found by $p_w \cdot t_{be} = e_o + p_s \cdot (t_{be} - t_o)$. Also, the break-even time has to be larger than the transition delay; therefore

$$t_{be} = \max\left(\frac{e_o - p_s \cdot t_o}{p_w - p_s}, t_o\right). \tag{1}$$

For simplicity of explanation, this article assumes that a device has one working and one sleeping states. It serves requests only in the working state.

*Example 1:* Fig. 5 shows the power consumed by a 2.5″ Hitachi hard disk. It wakes up from the sleeping state, serves requests, and then becomes idle. ♢

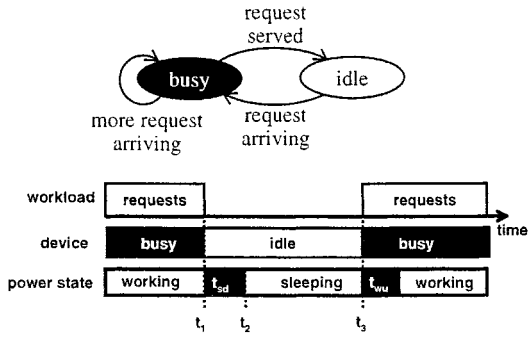


Fig. 3. Sleep during an idle period.

### C. Scheduling and Power Management

Running processes generate requests for a device; their execution orders directly affect the arrival times of requests, hence, the length of idle periods.

*1) Concept of Jobs:* In this paper, a job is defined as a unit to finish a specific task and it can be scheduled to start at a specific time. Consider a user running a text editor; this editor formats text, checks spelling, saves contents, and so on. Formatting, spell checking, and saving are three distinct jobs. Another example is an email reader that downloads email from a server. Downloading is a job because it can be scheduled to occur periodically.

*2) Precedence and Timing Constraints:* Consider three independent processes $pc_1$, $pc_2$, and $pc_3$. Suppose each process has three jobs: $pc_i$ has jobs $j_{i,1}$, $j_{i,2}$, and $j_{i,3}$ here $i \in [1, 3]$. There are nine jobs to schedule. Because $j_{1,1}$ and $j_{1,2}$ belong to the same process, $j_{1,1}$ must execute before $j_{1,2}$. Similarly, $j_{2,1}$ must execute before $j_{2,3}$. These orders are called *precedence constraints* [10]. Precedence constraints are expressed as directed acyclic graphs (DAG): $\mathcal{G} = (\mathcal{J}, \mathcal{E})$ where $\mathcal{J}$ is a subset of jobs and $\mathcal{E}$ are directed edges connecting jobs. If two jobs, $j_x$ and $j_y$, are connected by an edge $(j_x, j_y) \in \mathcal{E}$, then $j_x$ (*predecessor*) must execute before $j_y$ (*successor*). The precedence graph of the nine jobs is shown in Fig. 6.

Another type of constraints is *timing constraints*: a job has to finish before its *deadline*. Deadlines can be classified into three categories: firm, soft, and on-time [10]. Fig. 7 illustrates the differences between them. Suppose there is a "value" if a job finishes before the deadline. For a firm deadline, the value drops sharply if the job finishes after the deadline. Examples of firm deadlines are flight control systems; finishing a job after the deadline can lead to severe damages or even loss of lives. For a soft deadline, the value decreases more smoothly after the deadline. If a job has an on-time constraint, it should finish near the deadline, neither too early nor too late.

*3) Scheduling Jobs for Power Management:* Suppose three jobs, $j_{1,1}$, $j_{1,2}$, and $j_{2,3}$ need a specific device. For simplicity, we assume that each job takes $t$ to execute. Fig. 8 shows two schedules. A black rectangle indicates that this job needs the device. One difference between the two schedules is the lengths of idle periods. In the first schedule, the device is idle three times, each of length $2t$; in the second schedule, the device is idle for $6t$. The idle period in the second schedule is "continuous and long." If the break-even time of this device is between $2t$ and $6t$, power management saves power only in the second schedule.

TABLE I
SYMBOLS AND THEIR MEANINGS

| | |
|---|---|
| lower cases for scalars | |
| $t_{sd}$ ($t_{wu}$) | shutdown (wakeup) delay |
| $e_{sd}$ ($e_{wu}$) | shutdown (wakeup) energy |
| $p_w$ ($p_s$) | power in working (sleeping) state |
| $t_o$ | transition delay ($t_{sd} + t_{wu}$) |
| $t_{be}$ | break-even time |
| $e_o$ | transition energy ($e_{sd} + e_{wu}$) |
| $p_{w,k}$ | power in working state of device $d_k$ |
| $pc_i$ | a process |
| $j_{i,k}$ | a job of $pc_i$ |
| $d_i$ | a device |
| $u_{i,j}$ | utilization of $d_i$ by processes $pc_j$ |
| $u_i$ | utilization of $d_i$ by all processes |
| $c_i$ | CPU utilization by process $pc_i$ |
| $tbr$ | time between requests |
| upper cases for sets or sequences | |
| $\mathcal{J}$ | a set of jobs |
| $\mathcal{S}$ | a schedule of jobs |
| $\mathcal{RDS}$ | required device set |

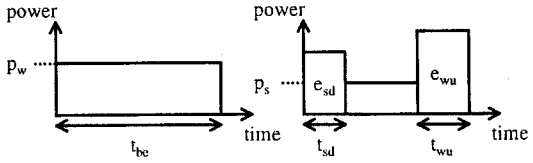


Fig. 4. Keeping the device in the working state (left) and shutting down the device (right). The energy is equal if the idle time is $t_{be}$.

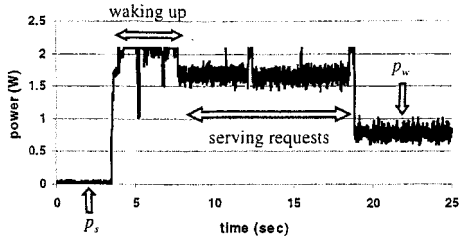


Fig. 5. State transitions of a Hitachi hard disk.

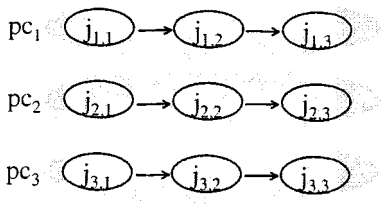


Fig. 6. Precedence of three independent processes.

*4) Scheduling in Inactive Systems:* In personal computers, some IO requests are scheduled to occur in the future. For example, text editors often have "autosavers" that save the contents periodically. An email reader retrieves emails from a mail server and stores them on a local hard disk. Both the editor and the reader generate periodic requests for a local hard disk. If their requests are not arranged properly, the disk has more and shorter idle periods. If the requests are arranged so they arrive at approximately the same time, the disk can remain idle and sleep for longer durations.

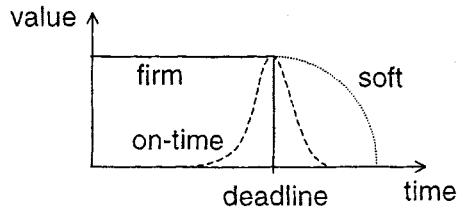


Fig. 7. Three types of deadlines.

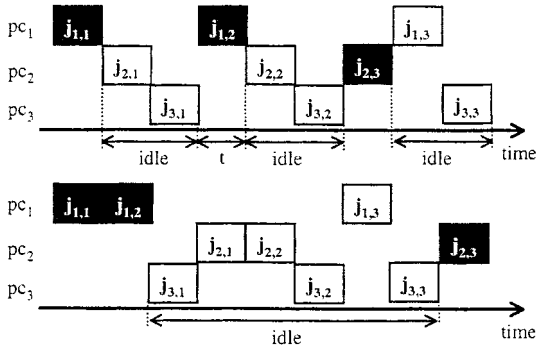


Fig. 8. Two schedules of three independent processes.

## III. Related Work

Run-time power reduction using software techniques can be achieved by operating systems or application programs. Operating systems can reduce the power of IO devices, processors, or memory systems. On the other hand, application programs can adjust their quality of service for available power budgets.

### A. OS-Based Power Reduction

*1) Power Management on IO Devices:* These policies can be divided into three categories: timeout, predictive, and stochastic. A detailed survey is available in [11]; another study compares both power saving and performance impact of some policies [12], [13].

Timeout is widely used in commercial products. A timeout policy shuts down a device after it is idle longer than a timeout value, $\tau$. For the example in Fig. 3, a timeout policy shuts down the device at $t_1 + \tau$. These policies assume that if a device is idle longer than $\tau$ then it will remain idle for at least $t_{be}$ [7]. When $\tau$ equals to the break-even time, the device consumes at most twice power compared to a perfect policy; this is called a 2-competitive policy [14]. An obvious drawback of timeout policies is the wasted energy during the timeout period if $\tau$ is large.

Predictive policies explicitly predict the length of an idle period before it starts. If the predicted length is larger than $t_{be}$, the device is shut down immediately after it becomes idle. These policies compute the length of an idle period according to previous idle and busy periods [1], [15], [16]. The major problem is that some policies have low prediction accuracy.

Stochastic policies use stochastic models for request generation and device state changes. Under such formulation, power management is a stochastic optimization problem. These policies can explicitly trade off between power saving and performance while timeout and predictive policies cannot. Stochastic policies include discrete-time and continuous-time stationary models [17], [18], time-indexed semi-Markov models [19], Petri Nets [20], and nonstationary models [21]. A common problem is that most policies require the characteristics of the workloads for off-line optimization. There is no universally adopted method to adjust at run time if the workload behavior changes.

*2) Low-Power Scheduling for Processors:* Instead of passively predicting the lengths of idle periods, scheduling for low power actively "controls" their lengths by rearranging job execution. Scheduling for behavioral synthesis is investigated in [22] and [23]. The former uses scheduling and guarded evaluation to reduce useless computation; the latter schedules computation to increase operand reuse. In [24], the authors propose scheduling for pipelined systems; they derive the conditions when scheduling with buffer insertion reduces power. Several studies investigate the relationship between scheduling and dynamic voltage scaling [25]–[27]. In [28], the authors discuss the effect of voltage scaling and quality of service.

*3) Reducing Memory Power:* In addition to IO devices and processors, memory also consumes significant amount of power. Reducing memory power can be achieved by selecting different power-performance modes at compile time or run time [29]. In [30], the authors study how page allocation affects energy and performance.

### B. Adaptive Programs for Low Power

A recent approach brings the awareness of power consumption to application programs, for example, by providing programming interfaces between application programs and operating systems [31], [32]. Application programs are modified to trade off between quality of service and available power budgets [33]. While this approach is promising, it requires further study to develop a set of application programming interfaces (API) that can be widely accepted.

## IV. Process-Based Power Management

OS kernel has the information about process execution and request generation; thus, power management should be controlled by OS kernel. This section presents a new approach to use kernel information for estimating device utilization and shutting down idle devices.

### A. Request Generation Models

Existing DPM policies (see Section III-A-1) do not distinguish request sources: requests are generated by an abstract entity called a *requester* [17]. These policies implicitly assume that the arrival of requests, regardless of their sources, is sufficient for predicting the length of future idle periods.

In reality, requests are generated by running processes. Studies show that different processes consume different amounts of power [34], [35]. Process-based power management is first proposed in [36]. Processes provide valuable information for predicting the lengths of idle periods. Processes have states; Fig. 2 shows five process states, including *ready*, *running*, and *waiting* states. A process generates new requests in the running state. Operating systems know the current state of each process. Several factors affect the state of a process. A process enters the running state after being selected by the

process scheduler; a process can wait for synchronization or other events; a process can also stop running and terminate.

When a process is created, a new requester is born. We found that parent processes do not provide enough information about request generation of their child processes. This is because most child processes execute different programs (in UNIX by calling `execl`) right after they are born (by `fork`). On the other hand, when a process terminates, it cannot generate new requests; process termination is important for predicting future request generations. We discovered that most processes have short lifetimes. Fig. 9 is the cumulative distribution of process lifetimes. While the percentages change in different workloads, the shape of the curve is likely to remain the same. Consequently, it is important to detect process termination for power saving.

We also observed that a program which generates many IO requests within short time periods often have short lifetimes; examples are `ftp` and `gcc`. Some other programs, such as `emacs`, generate requests less frequently and have longer lifetimes. Daemons like `syslogd` have long lifetimes, possibly as long as the computer is on; they rarely generate IO requests. Fig. 10 illustrates our discovery.

Based on these observations, we adopt a new model for request generations. It differs from existing models in three ways: 1) It separates request sources by processes; 2) it detects the termination of a process; and 3) it considers how often a process executes. This model incorporates additional information to predict the idleness of a device.

### *B. Device Utilization*

Because a process can generate requests when it is running, the relationship between a process and a device can be estimated by two factors 1) how often the process generates requests when it is running and 2) how often the process runs. They are represented by the device utilization and processor utilization. We use $u_{i,j}$ to indicate how often process $pc_j$ uses device $d_i$. We use $c_j$ to estimate how often this process runs. The range of $i$ is the number of devices; this is determined by the system configuration. The range of $j$ is the number of processes currently under consideration. All quantities are computed at run time. The following paragraphs explain our heuristics to estimate device utilization based on per-process information.

*1) Device Utilization:* Some processes are "CPU-burst"—using CPU mostly; some processes are "IO-burst"—using IO devices mostly [9]. Some other processes change between CPU-burst and IO-burst. We explain how to estimate device utilization for either CPU-burst or IO-burst processes; then we explain how to handle processes that change between two kinds of bursts. The device utilization by a process, $u_{i,j}$, is computed as the reciprocal of *time between requests*, $tbr$. It is the duration when a device is idle while a process is running. It is the time between the completion of the previous request and the arrival of the next request. Fig. 11 shows an example of two processes. The first process is IO-burst and generates many requests while it is running; its $tbr$ is shorter and its device utilization is higher. In contrast, the second process rarely generates requests and its $tbr$ is larger.

There are various ways to use $tbr$ for estimating $u_{i,j}$, for example, using the latest $tbr$ or using the running average.

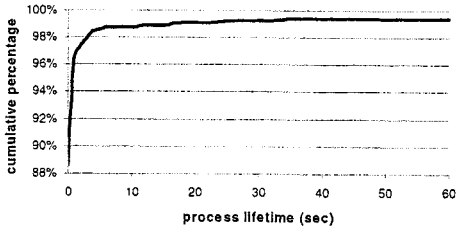


Fig. 9. Most processes have short lifetimes.

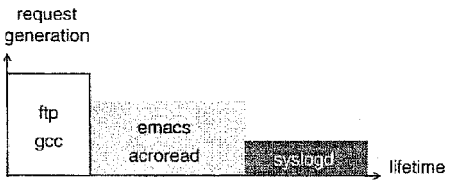


Fig. 10. Programs that generate intensive requests usually have short lifetimes.

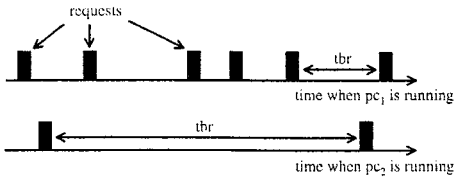


Fig. 11. Time between requests of two processes.

The former considers only one $tbr$ while the later consider all $tbr$s; neither is appropriate. Using the latest $tbr$ may make $u_{i,j}$ change quickly and possibly unstable; using the running average causes $u_{i,j}$ to update too slowly when the run-time behavior of a process changes. We use *discounted average* (also called exponential average [16]) as a balance between these two methods. Discounted average puts more weight on the latest $tbr$ but also considers previous $tbr$s. Suppose $n$ requests have been generated by this process and $tbr$ is latest time between requests [between the $(n-1)$th and the $n$th requests]. Let $tbr_n$ be the estimated time between requests after these $n$ requests We compute $u_{i,j}$ by this formula

$$tbr_n = a \cdot tbr + (1-a) \cdot tbr_{n-1}$$
$$u_{i,j} = \frac{1}{tbr_n}. \tag{2}$$

Here $a$ is a constant between zero and one. The value of $a$ determines how much "weight" is put on the latest $tbr$. When $a$ is large, we consider the latest $tbr$ as more important in estimating the overall utilization. For example, when $a$ is one, we consider only the latest $tbr$ and completely ignore earlier $tbr$s. When $a$ is small, more emphasis in placed on earlier $tbr$s.

Next, we explain how to estimate the device utilization if a process changes from IO-burst to CPU-burst. When a process changes from IO-burst to CPU-burst, its device utilization is overestimated during the CPU-burst period. This can be illustrated in the following example.

*Example 2:* Consider running a spreadsheet program with four stages illustrated in Fig. 12. It reads data from a hard disk (IO-burst during $t_0$ to $t_1$), gets user inputs, computes the results (CPU-burst during $t_2$ to $t_3$), and writes the results back to the disk (IO-burst during $t_3$ to $t_4$). Since updating $tbr$s is triggered

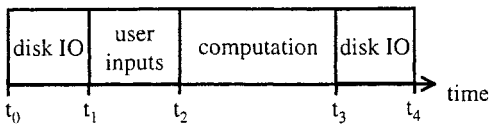


Fig. 12. A process with four phases.

by requests, the device utilization is overestimated during $t_1$ to $t_3$ because $tbr$ is not updated. ♢

The above example suggests the need to adjust the estimation of device utilization when the process changes from IO-burst to CPU-burst. We define $l_{i,j}$ as the time since process $pc_j$ generated the last request for device $d_i$. The adjusted estimation should be the same as $u_{i,j}$ when $l_{i,j}$ is small; the estimation should be zero when $l_{i,j}$ is large. This "small" and "large" are relative to the parameters of this device. We choose the break-even time of the device as the reference and use an adjustment function as

$$f_{i,j} = e^{-(l_{i,j}/t_{be,i})}. \tag{3}$$

When a process changes from IO-burst to CPU-burst, $l_{i,j}$ is large so $f_{i,j}$ becomes small. In contrast, when a process changes from CPU-burst to IO-burst, $l_{i,j}$ is small to make $f_{i,j}$ almost one. This is desirable because IO-burst requests are accurately estimated. After the adjustment, (2) is replaced by the new utilization estimation

$$w_{i,j} = u_{i,j} \times f_{i,j}. \tag{4}$$

*2) Processor Utilization:* While $w_{i,j}$ considers the interaction between a device and a process, it ignores other processes. A process may generate many requests while it is running. However, this process may rarely execute because, for example, it has a low priority or it is triggered by infrequent events. From the device's point of view, this process rarely generates requests. This effect is considered by including the processor utilization of the process.

Processor utilization of process $pc_j$ is represented by $c_j$. It is the percentage of CPU time occupied by this process in a sliding window because discounted average does not reflect processor utilization. Discounted average underestimates processor utilization for an IO-bounded process. When a process is IO-bounded, it uses CPU only momentarily each time it is selected by the process scheduler. While $w_{i,j}$ correctly indicates that this process has short $tbr$ and high utilization on this device, the same method does not indicate how often and how long this process executes. Consequently, we use the percentage of CPU time on this process to compute $c_j$

$$c_j = \frac{\text{CPUTime}(pc_j)}{\sum_{\text{all process } pc_i} \text{CPUTime}(pc_i)}. \tag{5}$$

This formula uses a sliding window; only processes running in this window are considered. The window size should be large enough to include most processes; on the other hand, it should be sufficiently small to quickly reflect changes in process behavior.

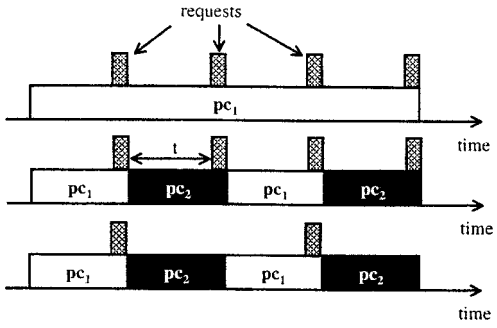


Fig. 13. Three examples of device utilization.

*3) Aggregate Device Utilization:* The aggregate utilization for device $d_i$ is $u_i$; it can be computed as the summation of device utilization and processor utilization from all processes

$$u_i = \sum_{\text{all process } pc_j} w_{i,j} \times c_j. \tag{6}$$

*Example 3:* Fig. 13 shows three examples to compute the device utilization. In the first example, only process $pc_1$ is running; it generates requests every $t$. In the second example, two processes are running; each generates requests every $t$. In the third example, only $pc_1$ generates request. The time between requests for each process in the three examples is $t$. In the first example, $c_1$ is one; in the second and third example, $c_1 = c_2 = 0.5$. The aggregate utilization for each example is $(1/t) \cdot 1 = (1/t)$, $(1/t) \cdot 0.5 + (1/t) \cdot 0.5 = 1/t$, and $(1/t) \cdot 0.5 = 1/2t$ respectively. This reflects accurately how often the device receives a request. ♢

### C. Shutdown Condition

A device is shut down when its aggregate utilization is small. Since $t_{be,i}$ is the minimum length of an idle period to save power of device $d_i$, the shutdown condition is determined based on $t_{be,i}$. The shutdown condition is

$$u_i < \frac{k}{t_{be,i}} \tag{7}$$

where $k$ is the "aggressiveness factor." If $k$ is one, a device is shutdown when the utilization is smaller than $1/t_{be,i}$; namely, the time between requests from all processes is longer than $t_{be,i}$. When $k$ is smaller than one, the power manager is "conservative" because it shuts down the device when the utilization is lower; this may lose opportunities to save power. In contrast, when $k$ is larger than one, the power manager is "aggressive" because it "takes chances" to save power by shutting down the device even when the utilization is still high. When $k$ is too large, however, the power manager shuts down the device too often. State-transition delays can significantly degrade performance; furthermore, state-transition energy may make actually increase power. Hence, we suggest a $k$ value equal to or slightly larger than one.

Emphasis should be stressed that our approach is fundamentally different from existing policies described in Section III-A1. They observe requests directly at the hardware or at device drivers but do not consider how requests are generated. Our method uses high-level (software) information

by distinguishing individual processes. Our method requires no modification in application programs. The following sections explain how schedulers can improve power management by collaborating with programs.

## V. Off-Line Scheduling

We developed an on-line scheduling scheme to improve power management. Before explaining our method, we start with off-line scheduling as the background. *Off-line* scheduling is performed before the execution of any job; it is possible if the complete knowledge of all jobs is available in advance. In contrast, *on-line* scheduling is performed at run time. When the behavior of a process changes or new jobs are created according to run-time conditions, scheduling must be performed on-line. In particular, interactive systems must use on-line scheduling because it is impossible to perfectly predict user behavior. We formulate off-line scheduling as the basis for understanding on-line scheduling.

### A. Scheduling for Multiple Devices

When there are multiple devices, even off-line scheduling is a complex problem. Consider the three processes in Fig. 6 again. Each job may use device $d_1$, device $d_2$, both, or neither. The job-device relationship is expressed by *required device set* ($\mathcal{RDS}$). Suppose the $\mathcal{RDS}$ of each job is expressed in Table II and each job takes $t$ to execute. Fig. 14 shows two schedules of these jobs. In the first schedule, $d_2$ is idle for $5t$ first, busy for $2t$, and idle again for $2t$. In the second schedule, $d_2$ is idle for $7t$ continuously. In contrast, $d_1$ is idle continuously for $4t$ in the first schedule. In the second schedule, this idle period is divided into two periods, each of $2t$. It is unclear which schedule saves more power. In fact, it depends on the hardware parameters. For example, the first schedule is better if $(t_{be,1}, t_{be,2}) = (3t, 8t)$ because $d_1$ can sleep and save power. On the other hand, the second schedule is better if $(t_{be,1}, t_{be,2}) = (5t, 6t)$ because $d_2$ can sleep and save power.

### B. Problem Formulation

Consider $n$ jobs: $\mathcal{J} = \{j_1, j_2, \ldots, j_n\}$ on a single-processor system with $m$ devices: $\mathcal{D} = \{d_1, d_2, \ldots, d_m\}$. Jobs share devices but no two jobs can use the same device simultaneously. Each job may use some of these devices. We use $r_{a,b}$ for such relationship: if job $j_a$ uses device $d_b$, $r_{a,b}$ is one; otherwise, it is zero. A schedule, $\mathcal{S} = (j_{s_1}, j_{s_2}, \ldots, j_{s_n})$, is a linear order of these jobs; $j_{s_{i+1}}$ executes immediately after $j_{s_i}$ for $i \in [1, n-1]$. A schedule has to satisfy all timing and precedence constraints. Low-energy scheduling is the problem of finding a schedule to minimize energy through power management. We define $t_i$ as the time when $j_{s_i}$ starts execution; $j_{s_i}$ executes during $[t_i, t_{i+1})$. The total energy of one schedule is the sum of the energy of all devices. The energy of device $d_k$ is divided into three parts

1) $e_{\text{busy},k}$ when $d_k$ is busy;
2) $e_{\text{sleep},k}$ when $d_k$ is idle and sleeping;
3) $e_{\text{idle},k}$ when $d_k$ is idle but remain in the working state.

We use $p_{w,k}$ as the power of device $d_k$ when it is in the working state; $p_{s,k}$ is the power when $d_k$ is sleeping. To compute $e_{\text{busy},k}$, we have to find the time when $d_k$ is busy. It is busy if $j_{s_i}$ executes and $r_{s_i,k} = 1$. Since $j_{s_i}$ executes during $[t_i, t_{i+1})$, $d_k$ is busy during $[t_i, t_{i+1})$

$$e_{\text{busy},k} = \sum_{\substack{i \in [1,n] \\ \text{such that } r_{s_i,k}=1}} p_{w,k} \cdot (t_{i+1} - t_i). \tag{8}$$

Then, we find the time when $d_k$ is idle. An idle period of $d_k$ is a period when it is not used but it is used before and after this period. In other words, an idle period of $d_k$ is defined by three conditions

1) a sequence of jobs, $j_{s_w}, j_{s_{w+1}}, \ldots, j_{s_x}$, that do not use $d_k$; namely, $r_{s_w,k} = r_{s_{w+1},k} =, \ldots, = r_{s_x,k} = 0$;
2) $d_k$ is used before this sequence: $r_{s_{w-1},k} = 1$;
3) $d_k$ is used after this sequence: $r_{s_{x+1},k} = 1$

where $w-1, w, \ldots, x+1$, are between 1 and $n$.

Let's now compute $e_{\text{sleep},k}$. Suppose $\text{idle}_k$ is an idle period of $d_k$; the length of this period is $|\text{idle}_k|$ and $|\text{idle}_k| = t_{x+1} - t_w$. When $|\text{idle}_k|$ is larger than $t_{\text{be},k}$, $d_k$ sleeps to save power. In order to compute $e_{\text{sleep},k}$, we find all idle periods that are longer than $t_{\text{be},k}$. Let $\mathcal{IS}_k$ be the set of idle periods that are longer than $t_{\text{be},k}$: $\mathcal{IS}_k = \{\text{idle}_k\text{: } |\text{idle}_k| > t_{\text{be},k}\}$. The total energy during these long idle periods is $e_{\text{sleep},k}$. Because $d_k$ changes power states, $e_{\text{sleep},k}$ includes the state-transition energy, $e_{o,k}$

$$e_{\text{sleep},k} = \sum_{is \in \mathcal{IS}_k} (p_{s,k} \cdot |is| + e_{o,k}) \tag{9}$$

where $|\text{is}|$ is the length of the corresponding idle period.

Finally, we consider $e_{\text{idle},k}$ for idle periods shorter than the break-even time. The device stays in the working state even though it is idle. Let $\mathcal{IW}_k$ be the set of these idle periods: $\mathcal{IW}_k = \{\text{idle}_k\text{: } |\text{idle}_k| \le t_{be,k}\}$

$$e_{\text{idle},k} = \sum_{iw \in \mathcal{IW}_k} p_{w,k} \cdot |iw| \tag{10}$$

where $|iw|$ is the length of the corresponding idle period.

TABLE II
Devices Required by Each Job

| $j_{1,1}$ | $j_{1,2}$ | $j_{1,3}$ | $j_{2,1}$ | $j_{2,2}$ | $j_{2,3}$ | $j_{3,1}$ | $j_{3,2}$ | $j_{3,3}$ |
|---|---|---|---|---|---|---|---|---|
| $d_1$ | $d_1$ | $d_1$ | $d_1$ | $d_1$ | $\phi$ | $d_2$ | $d_2$ | $\phi$ |

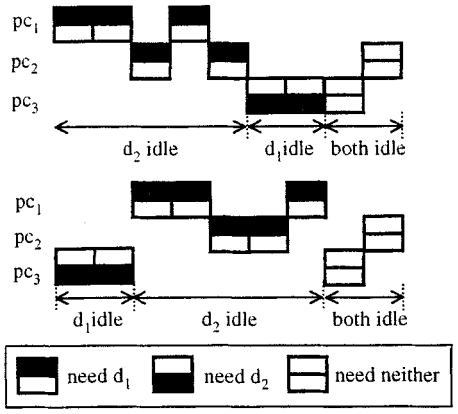


Fig. 14. Three processes using two devices.

The energy of device $d_k$ is $e_{\text{busy},k} + e_{\text{sleep},k} + e_{\text{idle},k}$ and the energy of all devices is

$$e = \sum_{k=1}^{m} e_{\text{busy},k} + e_{\text{sleep},k} + e_{\text{idle},k}. \tag{11}$$

Finding a schedule with the minimum power is an NP-complete problem even without timing or precedence constraints; its proof can be found in the Appendix.

## VI. On-Line Scheduling

The appendix shows that even simplified off-line scheduling is NP-complete; hence, we do not intend to find optimal solutions. Furthermore, we target interactive systems on which on-line scheduling is necessary. This section presents heuristic rules for low-power scheduling on interactive systems.

### A. Requests Created by Timers

The arrival times of requests are not completely unpredictable; in particular, some requests are created by "timers" so that they arrive at specific time in the future. The periodic jobs in Section II-C4 are such examples. In UNIX, the jobs are created in four steps: 1) calling `setitimer` to create a timer; 2) registering a callback function for the `SIGALRM` signal; 3) OS issues the `SIGALRM` signal when the timer expires; and 4) executing the callback function when this signal is issued. The job performed by the callback function is scheduled by the timer. If a power manager knows when a job executes and which devices are used by this job, the additional information helps the manager save power more effectively. Existing timer mechanism in UNIX does not allow programs to specify which devices will be used. The following paragraphs explain how to add such information and improve power management.

### B. Scheduling in Linux

On-line scheduling algorithms are often "priority-based"; at any moment, the scheduler selects a ready job with the highest priority. A *ready* job can start execution immediately; a job is ready after all its predecessors have completed. Priorities can be determined in different ways. In Linux, priorities are classified into different levels. Fig. 15 shows the flow of a Linux scheduler [37]. When the scheduler is invoked, it first checks whether there is a job in a task queue. A *task queue* is a method to inform OS kernel that a job is ready to execute. For instance, a device driver can put a job into a task queue for data retrieval after the device is ready to transfer data. Additionally, interrupts are used to inform OS of new events. Because interrupts can "interrupt" a running job, they are used for urgent events. Timers have the third-level priority; a timer is used to execute a job at a specific time. After checking task queues, interrupts, and timers, the scheduler chooses a user process with the highest priority.

We extend the Linux scheduler for power management. In order to maintain interactivity and reduce the impact on existing programs, power reduction is considered after the steps in Fig. 15. The extension is divided into two parts, 1) it wakes up a sleeping device before scheduling a job that requires this device and 2) it arranges execution orders to facilitate power management.

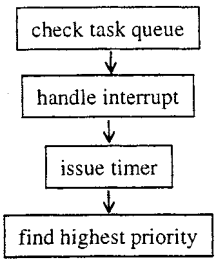


Fig. 15. Steps of a Linux scheduler.

### C. Predictive Wakeup

If a requests requires a sleeping device, this request has to wait for the wakeup delay. Ideally, the device should wake up before the request arrives to eliminate waiting; this is called "predictive wakeup" [38]. A device should wake up just before requests arrive. Waking up too early wastes energy; waking up too late does not eliminate waiting. Our experiments show that the method in [16] actually increases energy because it mixes requests from all processes and has low prediction accuracy.

To perform predictive wakeup accurately, the scheduler has to know which devices are used by a job. This can be achieved in two ways. The first is to predict using the history of a process. The advantage is that no user program needs to be modified; the disadvantage is that the accuracy can be low, especially when a process changes from CPU-burst to IO-burst or vice versa. An alternative is to provide an interface for processes to specify their device requirements. The advantage is that only specified devices wake up; no device is woken up if there is no request. The disadvantage is that programs need to be modified to specify their device requirements.

We take the second approach because it provides precise information about device requirements. A system call is added so that programmers can provide explicit information for predictive wakeup [32]. When a program creates a timer, it can also specify which device will be used when the timer expires

```
RequireDevice (device, time, callback)
device: hardware device
time: when to start
callback: a callback function.
```

*Example 4:* In Section II-C4, an editor saves contents onto a hard disk every five minutes. This can be specified by `RequireDevice (HardDisk, five minutes, save file)`. A mail reader needs both the hard disk and the network card; therefore, `Netscape` issues two system calls: `RequireDevice ({HardDisk, NetworkCard}, five minutes, download)`. ◇

If the timer expires at $tm$ and this job uses device $d_k$, then our scheduler informs the power manager at $tm - t_{wu,k}$ to check the power state of $d_k$. If $d_k$ is sleeping, it is woken up so that this job does not have to wait for the wakeup delay. If there are multiple jobs, the scheduler finds the job with the earliest timer and wakes up devices needed by this job. Fig. 16 shows a pseudocode of predictive wakeup.

If a program does not specify which device will be used in the future, the device will wake up "on demand:" only when a request actually arrives. This increases the response time of the

```
PredictiveWakeup (J := jobs to schedule)
begin
    sort J by their timer values;
    j := a job in J with the earliest timer;
    for each d_k in RDS(j)
        if (d_k is sleeping) and
        (timer(j) - now ≤ t_wu,k)
            wake up d_k;
end
```

Fig. 16. Predictive wakeup.

request. Predictive wakeup improves performance but does not save power. The real benefit will be clearer after we explain how to schedule jobs later.

### D. Flexible Timers

Some jobs have the flexibility to start execution before or after their timer expires. For example, editors do not have to save the contents precisely every five min; it is usually acceptable if the period is close enough to five min. The concept of flexible timers is illustrated in Fig. 17; this figure modifies Fig. 7. In Fig. 17, the original (i.e., inflexible) timer is shown by the solid line; its value drops quickly before and after the specified start time. In contrast, the flexible timer uses the dashed line. It is acceptable to execute the timer's callback function earlier or later, as long as the difference is less than the tolerance. We enhance the previous system call so that a program can specify its flexibility

```
RequireDevice (device, tolerance, time,
               callback)
device: hardware device
tolerance: acceptable variation
time: when to start
callback: a callback function.
```

### E. Scheduling Jobs for Power Reduction

`RequireDevice` creates jobs and specifies devices to be used. It is further enhance to include an estimated execution time of a job. Based on this information, we can schedule the jobs for power management.

```
RequireDevice (device, execution, tolerance,
               time, callback)
device: hardware device
execution: execution time
tolerance: acceptable variation
time: when to start
callback: a callback function.
```

We use an example to convey the basic idea before explaining our method.

*Example 5:* Fig. 18 is an example of scheduling for power management on two devices. The meaning of each rectangle is explained earlier in Fig. 14. At the top of Fig. 18, the jobs are arranged by the order of their timers. The idle periods are short and scattered. At the bottom of the figure, the execution order is rearranged to make idle periods continuous and long. The scheduler can rearrange the order because these jobs are created by flexible timers. ♢

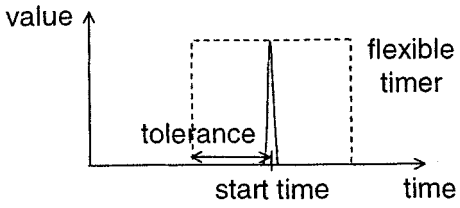


Fig. 17. Flexible timer.

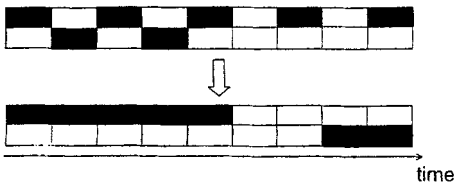


Fig. 18. Group jobs according to their device requirements.

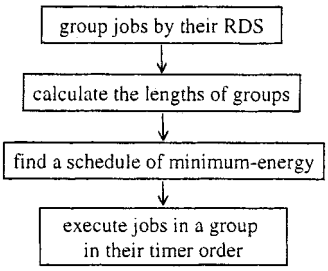


Fig. 19. Low-power scheduling.

Fig. 19 outlines our heuristics of our low-power scheduler. First, it groups jobs according to their device requirements and calculates the length of each group; jobs in the same group execute together. Suppose there are $q$ groups: $\mathcal{J}_1, \mathcal{J}_2, \ldots, \mathcal{J}_q$. The devices used by jobs in $\mathcal{J}_i (i \in [1, q])$ is represented as $\mathcal{RDS}(\mathcal{J}_i)$. Let $ex_x$ be the execution time of job $j_x$. The length of a group is the sum of execution time of all jobs in this group

$$|\mathcal{J}_i| = \sum_{\forall j_x \in \mathcal{J}_i} ex_x. \tag{12}$$

Let $\mathcal{LPGS}$ (low-power group schedule) be a schedule of these groups: $\mathcal{LPGS} = (\mathcal{J}_{s_1}, \mathcal{J}_{s_2}, \ldots, \mathcal{J}_{s_q})$. Then, the scheduler computes the energy of this schedule by treating a group of jobs as a single job and applying formulae (8)–(11). For example, $|\mathcal{J}_i|$ corresponds to the execution time of one job $(t_{i+1} - t_i)$ in (8); $e_{\text{busy},\,k}$ is calculated by

$$e_{\text{busy},\,k} = \sum_{d_k \in \mathcal{RDS}(\mathcal{J}_i)} p_{w,\,k} \cdot |\mathcal{J}_i|. \tag{13}$$

Energy $e_{\text{sleep},\,k}$ and $e_{\text{idle},\,k}$ are computed in a similar way. After computing the energy of each schedule, the scheduler finds one schedule with the minimum energy. Fig. 20 shows a pseudocode of the scheduler.

*Example 6:* Consider nine jobs waiting for execution. The devices required by each job is shown in Table III. These nine jobs belong to three groups. First, $\mathcal{RDS}_1 = \{d_1\}$ has five jobs: $\{j_1, j_2, j_3, j_4, j_5\}$. Second, $\mathcal{RDS}_2 = \{d_2\}$ has two jobs: $\{j_7, j_8\}$. Third, $\mathcal{RDS}_3 = \phi$ has two jobs: $\{j_6, j_9\}$. Suppose the execution time of each job is $t$. The length of these groups are $5t$, $2t$, and $2t$, respectively. Let's assume the break-even times are $t_{be,\,1} = 5t$ and $t_{be,\,2} = 6t$.

```
/* initialization */
LPGS := empty; /* low-power group schedule */
/* end of initialization */

..... /* the steps in Figure 15; extension starts below */
if (new jobs are created)
    group jobs by their RDS's;
    calculate the length of each group;
    /* find one schedule first */
    S := one schedule of these groups;
    eng := energy of S;
    LPGS := S;
    /* check whether there are better schedule */
    for each schedule of these groups
        ethis := the energy of this schedule;
        if (ethis < eng) /* a better schedule */
            eng := ethis;
            LPGS := this schedule;
```

Fig. 20. Scheduling jobs by their $\mathcal{RDS}$ groups.

TABLE III
DEVICE REQUIREMENTS FOR EXAMPLE 6

| $j_1$ | $j_2$ | $j_3$ | $j_4$ | $j_5$ | $j_6$ | $j_7$ | $j_8$ | $j_9$ |
|---|---|---|---|---|---|---|---|---|
| $d_1$ | $d_1$ | $d_1$ | $d_1$ | $d_1$ | $\phi$ | $d_2$ | $d_2$ | $\phi$ |

There are six ways to schedule these three groups

1) $\{\mathcal{RDS}_1, \mathcal{RDS}_2, \mathcal{RDS}_3\}$;
2) $\{\mathcal{RDS}_1, \mathcal{RDS}_3, \mathcal{RDS}_2\}$;
3) $\{\mathcal{RDS}_2, \mathcal{RDS}_1, \mathcal{RDS}_3\}$;
4) $\{\mathcal{RDS}_2, \mathcal{RDS}_3, \mathcal{RDS}_1\}$;
5) $\{\mathcal{RDS}_3, \mathcal{RDS}_1, \mathcal{RDS}_2\}$;
6) $\{\mathcal{RDS}_3, \mathcal{RDS}_2, \mathcal{RDS}_1\}$.

Four schedules cause $d_2$ to be idle longer than its break-even time and save power. They are schedules 3 to 6.

All possible $\mathcal{RDS}$ groups form a power set of the power-managed devices; this is determined by the number of power-managed devices in the system, and is independent of the number of jobs. In this example, there are at most four groups. ♢

Although, it seems that Fig. 20 requires large amount of computation to compare all possible schedules, this does not happen in practice due to two reasons 1) the number of groups is small for a system with only a few power-managed IO devices and 2) The scheduler considers only jobs created by the flexible timers.

### F. Effects of Caching

Caching is widely used to improve the performance of accessing IO devices. It is difficult to predict whether an IO request can be served by the cache or it actually reaches the device. In some cases this can be predicted with certainty; for example, the request from an autosaver should be flushed directly to the hard disk. Also, the network card has to wake up in order to check whether any new email has arrived at the server. In these cases, caching does not avoid waking up the devices.

### G. Meeting Timing Constraints

Because jobs are not executed strictly by their timer values, it is possible that a job is executed after its timer expires. The scheduler has to guarantee the job executes within the interval specified by the flexible timer `RequireDevice`. Suppose a job is created by `RequireDevice` and $t_e$, $t_t$, and $t_s$ are the execution time, tolerance, and starting time. The scheduler will start the job no later than $t_s + t_t - t_e$.

### H. Handling Requests From Other Programs

The low-power scheduler does not change a request if it is not created by `RequireDevice`. This is necessary to run all legacy programs. Such requests come from two types of sources: from a program invoked by the user or from the original UNIX (inflexible) timer. Sometimes, these requests wake up a sleeping device. When the power state of a device changes, the scheduler reevaluates the energy of different schedules.

*Example 7:* Fig. 21 is an example how the schedule changes. Suppose four jobs are created by `RequireDevice`. At $t0$, both devices are sleeping and a preferred schedule is shown at the top of the figure. At time $t1$, a request (enclosed by dotted circle) arrives and it requires device $d_2$. If this request is not created by `RequireDevice`, $d_2$ has to wake up immediately. At $t2$, after serving this request, $d_2$ is still in the working state. Because the power state of $d_2$ has changed, the original schedule is now inferior. The scheduler changes the execution order as shown at the bottom of the figure. ♢

As this example demonstrates, our method can still improve power management even if there are requests from other programs.

## VII. IMPLEMENTATION IN LINUX

We implemented power management in Redhat 6.2 Linux on a Sony VAIO notebook. Recollect that systems are structured in layers as shown in Fig. 1. Our implementation is divided into three parts. At the bottom, device drivers set hardware power states. In the middle, the OS kernel estimates device utilization from processes and shuts down a device of low utilization. The OS also schedules jobs to cluster idle periods. At the top, application programs generate requests for devices; these programs may actively inform OS about their device requirements by using a new system call we provide. Fig. 22 shows the interaction between different components in our implementation.

Our OS controls the power states of IO devices using PCMCIA interfaces. PCMCIA has `suspend` and `resume` commands to control devices. These commands can shut down and wake up any PCMICA devices. When a device is suspended, its power consumption is virtually zero.

We call the method presented in Section IV "process-based" because it distinguishes requesters as processes. This method updates the estimation of device and processor utilization when a process changes states (Fig. 2) or when the device receives a request. A process changes states by the process scheduler; request generation is detected by device drivers. This method does not require any modification in user programs.

A notebook computer has two factors that limits low-power scheduling. First, it is an interactive system; fast response is expected by users. Second, wide varieties of programs are ported; most programs assume certain properties in scheduling. In particular, when they use timers, they expect these timers to expire

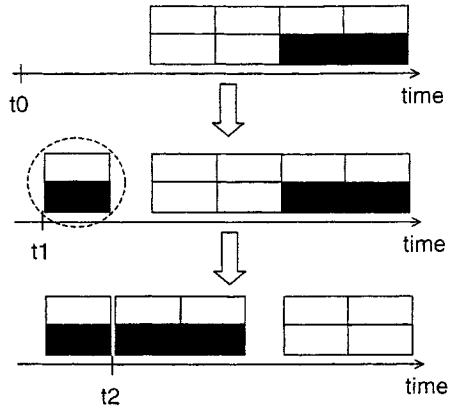


Fig. 21. Group jobs according to their device requirements.

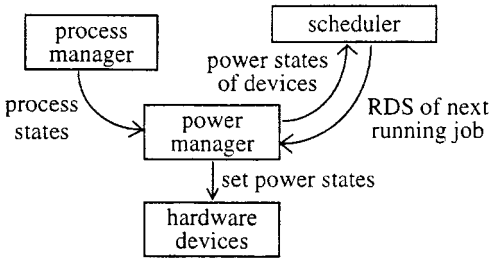


Fig. 22. Interaction between process manager, scheduler, and power manager.

at the specified time. If a request is not created by the flexible timer, it is served immediately, possibly after waiting for the wakeup delay of the device needed.

## VIII. Experiments

### A. Measurement Setup

Two IO devices are power managed: a Linksys Ethernet card and a Hitachi 2.5″ hard disk. Table IV shows the parameters of these devices. Both devices are connected through the PCMCIA interface. The hard disk is a standard 2.5″ IDE disk; it is connected to the computer through a PCMCIA-IDE converter as a second disk. An Accurite PCMCIA Extender Card is inserted between either device and the computer. We connect the extender to a National Instrument data acquisition card (DAQ). Fig. 23 illustrates the setup for measuring the power of the hard disk.

### B. Workloads

Three types of workloads are considered. The first is a trace of user activities by recording idle periods longer than two seconds. The trace is then replayed while a policy is running. The second workload uses probability models for transitions from idleness to busyness (Fig. 3). In [19], the authors discover that Pareto distributions closely approximate interactive workloads such as `telnet`. A Pareto distribution is expressed by its cumulative function: $1 - \alpha t^{-\beta}$, where $t$ is time and $\alpha$ and $\beta$ are constants. We use 0.7/s for $\alpha$ and 0.5 for $\beta$ because they reside in the range presented in [19]. In addition to Pareto distributions, we also consider uniform distributions for comparison.

TABLE IV
Device Parameters

| parameter | hard disk | network card |
|---|---|---|
| vendor | Hitachi | Linksys |
| model | DK23AA-60 | NP 100 |
| $p_w$ | 0.77 | 0.76 |
| $p_s$ | 0.0 | 0.0 |
| $t_o$ | 10.61 | 2.75 |
| $e_o$ | 18.90 | 2.88 |
| $t_{be}$ | 24.41 | 3.61 |

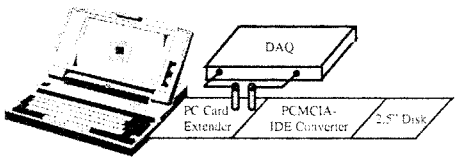


Fig. 23. Setup for measuring disk power.

The range of the uniform distribution is zero to ten minutes. The third workload generates periodic requests with timers. We consider both fixed timers (created by `setitimer`) and flexible timers (created by `RequireDevice`). The period of each timer is between one and five min; the tolerance is one min.

There are up to six requesters at any time. A requester may generate three types of requests: `ping` for the network card, `fput` for the hard disk, and `ftp` for both devices. After a requester generates a request, it has 10% probability to terminate. Once a requester terminates, another requester is created two min later. Each workload runs for two h.

### C. Policy Comparison

Seven policies are compared. The last two policies are used for the third workload in which requests are generated with timers:

1) no power management;
2) timeout of three minutes;
3) 2-competitive [14];
4) exponential average (16);
5) process-based;
6) predictive wakeup;
7) scheduling for flexible timers with predictive wakeup.

We compare these policies by five criteria, including power and performance. Power is determined by the time in the working state and the number of state transitions. Performance is affected by the time spent during state transitions.

1) $p_a$: average power (Watt). The measurements include the energy for serving requests;
2) $t_s$: average time in the sleeping state (s);
3) $t_t$: total time during state transitions (s);
4) $sd$: number of shutdowns;
5) $sd_w$: number of wrong shutdowns. They count the shutdowns during idle periods shorter than the break-even time.

It is desirable to have low power ($p_a$), low overhead ($t_t$ and $sd$), low error rate ($sd_w$), and long sleeping time ($t_s$).

TABLE V
POWER SAVING AND PERFORMANCE OF DIFFERENT POLICIES

| workload | policy | hard disk | | | | | Ethernet card | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $p_a$ | $t_s$ | $t_t$ | $sd$ | $sd_w$ | $p_a$ | $t_s$ | $t_t$ | $sd$ | $sd_w$ |
| 1 trace | 1 | 0.91 | 0 | - | - | - | 0.77 | 0 | - | - | - |
| | 2 | 0.89 | 44 | 53 | 5 | 0 | 0.58 | 352 | 14 | 5 | 0 |
| | 3 | 0.58 | 68 | 435 | 41 | 19 | 0.26 | 93 | 149 | 54 | 8 |
| | 4 | 0.74 | 19 | 774 | 73 | 42 | 0.57 | 17 | 262 | 95 | 21 |
| | 5 | 0.50 | 42 | 647 | 61 | 17 | 0.27 | 136 | 94 | 34 | 1 |
| 2 Pareto | 1 | 0.90 | 0 | - | - | - | 0.77 | 0 | - | - | - |
| | 2 | 0.88 | 16 | 74 | 7 | 4 | 0.77 | 24 | 19 | 7 | 0 |
| | 3 | 0.57 | 43 | 721 | 68 | 25 | 0.43 | 40 | 283 | 103 | 12 |
| | 4 | 0.51 | 28 | 1113 | 105 | 30 | 0.42 | 14 | 605 | 220 | 46 |
| | 5 | 0.45 | 41 | 954 | 90 | 11 | 0.41 | 66 | 157 | 57 | 7 |
| 2 uniform | 1 | 0.84 | 0 | - | - | - | 0.76 | 0 | - | - | - |
| | 2 | 0.81 | 46 | 106 | 10 | 2 | 0.73 | 89 | 11 | 4 | 0 |
| | 3 | 0.42 | 80 | 551 | 52 | 5 | 0.36 | 45 | 206 | 75 | 0 |
| | 4 | 0.44 | 45 | 837 | 79 | 17 | 0.43 | 16 | 497 | 181 | 29 |
| | 5 | 0.39 | 88 | 530 | 50 | 4 | 0.35 | 46 | 190 | 69 | 6 |
| 3 timer | 1 | 0.88 | 0 | - | - | - | 0.76 | 0 | - | - | - |
| | 2 | 0.87 | 0 | 0 | 0 | 0 | 0.76 | 0 | 0 | 0 | 0 |
| | 3 | 0.45 | 105 | 392 | 37 | 13 | 0.31 | 33 | 302 | 110 | 9 |
| | 4 | 0.64 | 14 | 1198 | 113 | 65 | 0.39 | 41 | 225 | 82 | 22 |
| | 5 | 0.37 | 72 | 583 | 55 | 9 | 0.30 | 104 | 96 | 35 | 3 |
| | 6 | 0.35 | 85 | 530 | 50 | 0 | 0.29 | 80 | 140 | 51 | 1 |
| | 7 | 0.25 | 220 | 265 | 25 | 0 | 0.19 | 228 | 73 | 25 | 0 |

### D. Experimental Results

*1) Parameter Setting:* Three parameters affect the experimental results: discount factor $a$ in (2), window size in (5), and aggressiveness $k$ in (7).

In general, if a power manager responds to requests and changes of requesters more quickly, it saves more power. Meanwhile, it causes more shutdowns and degrades performance more seriously. Consequently, these parameters trade off power with performance. A power manager responds more quickly under the following conditions.

1) Large $a$. When $a$ is large, more weight is put on the latest $tbr$ and $u_{i,j}$ changes more quickly.
2) Small window size. When the window is smaller, the denominator is smaller and $c_j$ changes more quickly.
3) Large $k$. When $k$ is large, the power manager shuts down a device quickly when its utilization drops.

Our experiments show that when $a$ increases from 0.1 to 0.9, average power reduces by nearly 17%; however, the number of shutdowns increases by 20%. When the window size decreases from sixty seconds to ten seconds, power reduces by 18% and the number of shutdowns increases by 29%. When $k$ increases from 0.5 to 1, power reduces by 17% and the number of shutdowns increases by 21%. When $k$ increases from 0.5 to 2, the percentage of wrong shutdowns increases by more than 25%. All measurements are conducted on the hard disk for the second workload with Pareto distributions. We chose 0.5 for $a$, one min for the window size, and 1 for $k$ in generating Table V.

*2) Power Saving and Performance Impact:* Table V compares power and performance of different policies; several important facts can be observed.

If a policy saves more power (small $p_a$), it usually has more shutdowns (large $sd$); the device spends more time on state transition (large $t_t$). Timeout with three minutes rarely has shutdown opportunities to save power. It makes power consumption much higher compared to other policies. The 2-competitive method (policy 3) has comparable power saving with process-based power management (policy 5) for the network card. However, 2-competitive generally has higher misprediction rates.

Some shutdowns actually waste energy because the device does not sleep long enough; these shutdowns are "wrong" and are expressed as $sd_w$. The ratio of $sd_w$ and $sd$ is the misprediction rate. Misprediction is reduced when requests are generated by timers (policies 6 and 7). Fig. 24 shows the misprediction rates of policies 2 to 5; lower misprediction rates are preferred. Because the break-even time of the hard disk is longer, it is more likely to have higher misprediction rates on the hard disk.

Even though power saving depends on workloads and devices, policy 5 consistently achieves nearly 50% power saving for both devices and all request generation methods. Other policies have large variations in their power saving. For example, policy 4 saves 20% to 48% power. Policy 6 does not change the time when requests are generated but it reduces misprediction ($sd_w$). In contrast, policy 7 changes the time when requests are generated and significantly reduces power, up to 72% on the network card for workload 3.

Figs. 25–27 show the power ($p_a$) and transition time ($t_t$) of different policies; shorter bars (less power and transition overhead) are preferred. Figs. 25 and 26 are normalized by policy 5; Fig. 27 is normalized by policy 7. As we can see from these figures, policy 1 has the least transition time (zero) but the highest power consumption. Policy 4 has the largest numbers of shutdowns; it also has high misprediction rates as shown in Fig. 24.

*3) Lengths of Idle Periods:* Fig. 28 shows the distribution of idle periods with and without low-power scheduling. Circles are the idle periods without low-power scheduling and squares are the idle periods with low-power scheduling. It is preferred to have long idle periods. Without low-power scheduling, the

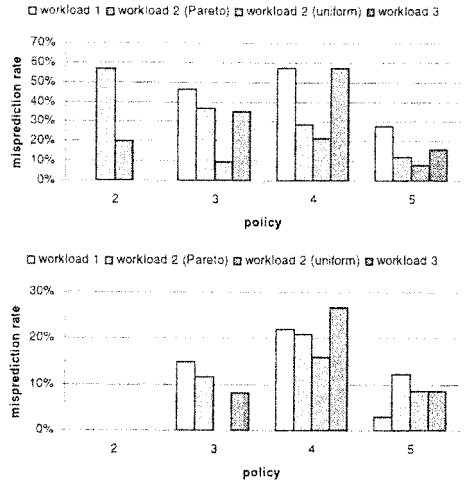


Fig. 24. Misprediction rates of policies 2 to 5, hard disk (top) and network card (bottom).

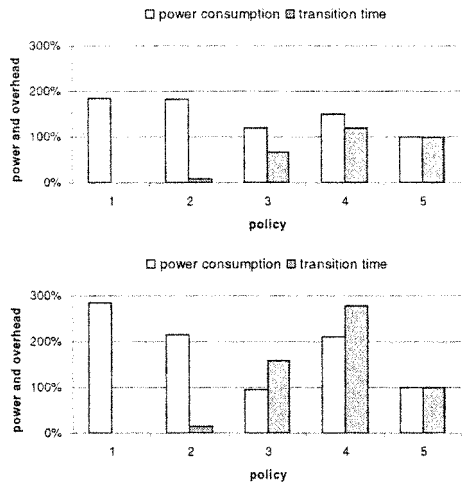


Fig. 25. Power and overhead for workload 1, hard disk (top) and network card (bottom).

lengths of idle periods is more widely spread. In particular, the dotted oval shows some "medium-length" idle periods between 60 seconds to 180 s. In contrast, low-power scheduling makes requests bursty and prevents these medium-length idle periods. Idle periods are either very short (the far left outside this figure) or very long (longer than 180 s); power managers use long idle periods to save power.

*4) Repetitive Wrong Shutdowns:* We discovered that the method presented in [16] has low prediction accuracy (large $sd_w$). This policy predicts the length of a future idle period based the previous idle period and the previous prediction. Let $len_{\text{actual}}$ and $len_{\text{predict}}$ be the actual and predicted lengths of the latest idle period. This policy predicts that the length of the next idle period is $a \cdot len_{\text{actual}} + (1-a) \cdot len_{\text{predict}}$ where $a$ is between zero and one. Consider a device whose break-even time is $t_{be}$. Let us call the latest idle period $idle_0$. The first idle period in the future is called $idle_1$ and the second idle period in the future is $idle_2$. Suppose the length of $idle_0$ is $nt_{be}$ where $n \gg 1$ and future idle periods are very short ($\approx 0$). This policy predicts $idle_1$ to be $a \cdot n \cdot t_{be}$. If $a \cdot n > 1$, the device is shut down. Since $idle_1 \approx 0$, the policy predicts $idle_2$ as $a^2 \cdot n \cdot t_{be}$.

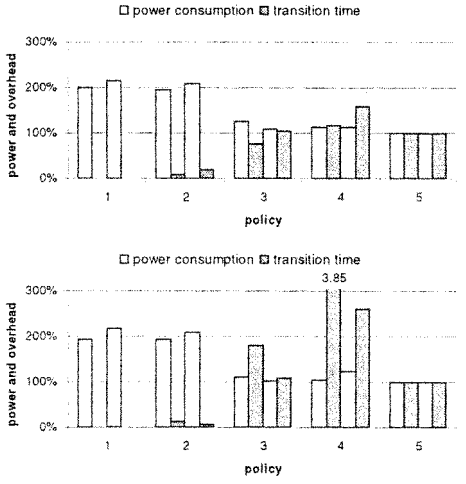


Fig. 26. Power and overhead for workload 2, hard disk (top) and network card (bottom).

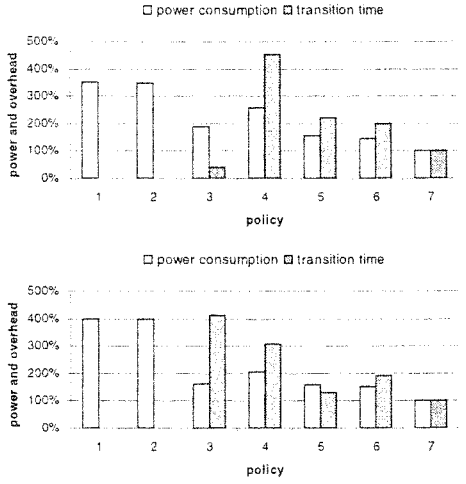


Fig. 27. Power and overhead for workload 3, hard disk (top) and network card (bottom).

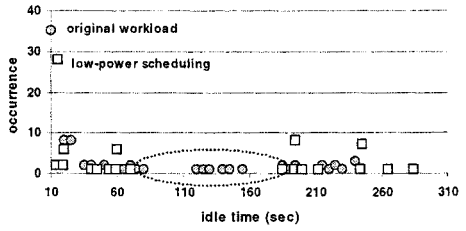


Fig. 28. Low-power scheduling reduces medium-length idle periods (60 to 180 s).

Applying the same rule, the policy predicts $idle_m$ as $a^m \cdot n \cdot t_{be}$. Even though these idle periods ($idle_1$, $idle_2$, ..., $idle_m$) are short, this policy still shuts down the device until $a^m \cdot n < 1$ or $m > -(\log n / \log a)$. In other words, this policy requires a "learning period" to correct its prediction; the length of this period is proportional to the logarithm of the length of a long idle period, $idle_0$ in this example. Our method avoids this problem by including both device and processor utilizations. When a program generates bursty requests after a long idle period, its both utilizations increase promptly so our method will not shut down the device.

*5) Computation Overhead of Our Methods:* The process-based policy presented in Section IV updates the estimation of device and processor utilization only when a process changes states. When a process changes states, the process scheduler is invoked to determine the next process to execute. The computation required for our policy is considerably less than the computation performed by the process scheduler (Fig. 15). The overhead of the low-power scheduling method is affected by two factors, 1) how many devices are power managed and 2) how many jobs are created by the flexible timer. Our experiments consider only two devices and at most six jobs. On a typical notebook, the number of power-managed IO devices is likely to be small enough so the overhead is acceptable.

## IX. Conclusion

This paper presents a new approach to reduce power consumption in interactive systems. Our method uses OS kernel to distinguish individual processes for estimating the utilization of IO devices. Unused devices are shut down to save power. This is more accurate than traditional methods that do not distinguish processes. We also propose a system call for programs to specify their hardware requirements. Such information allows process schedulers to make idle periods continuous and long; power reduction can be achieved without performance degradation. We implemented the utilization estimation and the new system call in Linux; experimental results achieved up to 72% power saving.

This study can be extended in several directions. First, some systems, such as servers, have multiple identical devices. It is possible to trade off performance and power with power-aware load balancing. Second, battery capacities reduce at high peak power; low-power scheduling may reduce peak power and increases battery capacities. Third, software techniques for power reduction, such as the interactions between compilers and OS, require further investigation. Fourth, more investigation is needed to understand how OS can support the tradeoff between power consumption and quality of service. Finally, it remains a research topic how to scale low-power scheduling to manage the power of many devices.

## Appendix
## Complexity of Low-Energy Scheduling

Suppose there are $n$ jobs: $\mathcal{J} = \{j_1, j_2, \ldots, j_n\}$ on a single-processor system with $m$ devices: $\mathcal{D} = \{d_1, d_2, \ldots, d_m\}$. Let $\mathcal{S}$ be a schedule: $\mathcal{S} = (j_{s_1}, j_{s_2}, \ldots, j_{s_n})$; $j_{s_{i+1}}$ executes immediately after $j_{s_i}$ for $i \in [1, n-1]$. The energy of a schedule is computed using formula (11). The low-energy scheduling problem can be transformed into a decision problem: *given a bound $k$, is there a schedule that makes energy lower than $k$*?

For an off-line scheduling problem with timing and precedence constraints, it is NP-complete to answer whether a schedule exists [39]. Since low-energy scheduling considers energy in addition to the constraints, its complexity is at least NP-hard. On the other hand, when a schedule is known, it takes polynomial time to find the energy consumption using formula (11) and to answer whether the energy is lower than $k$. The low-energy scheduling problem is in NP; therefore, it is an NP-complete problem.

### A. Simplification Assumptions

Even without timing and precedence constraints, low-energy scheduling is still an NP-complete problem. We simplify the problem so that there always exist schedules.

1) There is no timing or precedence constraints. Jobs can execute in any order and any schedule is a valid schedule.
2) All devices have the same parameters, such as the power, and their break-even time. Devices have equal importance in power reduction.
3) All jobs have the same execution time. Each job causes the same duration of busyness of a device that is used by this job.

For a device, its total idle time is the same, regardless of the schedule. However, the idle time may consist of many short idle periods or few long idle periods. Only idle periods longer than the break-even time can save power using power management. Long idle periods are preferred; short idle periods are "wasted." Under the three assumptions, low-energy scheduling is equivalent to reducing the number of idle periods and to enlarging the length of each idle period. In order words, the scheduler intends to reduce the "switches" between idleness and busyness of devices.

### B. State Switches

A device switches from idleness to busyness if a job does not need this device while the following job does. Specifically, device $d_k$ switches from idleness to busyness if $r_{s_i,k} = 0$ and $r_{s_{i+1},k} = 1$ for any $i \in [1, n-1]$. Similarly, $d_k$ switches from busyness to idleness if $r_{s_i,k} = 1$ and $r_{s_{i+1},k} = 0$. When $r_{s_i,k} = r_{s_{i+1},k}$, the device remains either idle or busy with no switch. Therefore, $d_k$ switches if and only if $r_{s_i,k} \neq r_{s_{i+1},k}$. We define $sw_k$ as the number of switches of $d_k$ in this schedule; it can be computed by

$$sw_k = \sum_{i=1}^{n-1} r_{s_i,k} \oplus r_{s_{i+1},k} \tag{14}$$

here $\oplus$ this is the *exclusive-or* function. The total number of switches of all devices, $sw$, is

$$sw = \sum_{k=1}^{m} sw_k. \tag{15}$$

### C. Problem Statement

With the simplification, the low-energy scheduling problem is equivalent to finding a schedule such that the total number of switches is the minimum.

$$\min \sum_{k=1}^{m} \sum_{i=1}^{n-1} r_{s_i,k} \oplus r_{s_{i+1},k}. \tag{16}$$

### D. Distance Between Jobs

We define the "distance" between two jobs as the number of switches when these jobs execute consecutively. For jobs $j_x$ and $j_y$, their distance is

$$dt_{x,y} = \sum_{k=1}^{m} r_{x,k} \oplus r_{y,k}. \tag{17}$$

*Example 8:* Consider an example of three jobs and two devices. The required device set ($\mathcal{RDS}$) of $j_1$ is $\{d_1\}$, $\mathcal{RDS}(j_2) = \{d_2\}$, $\mathcal{RDS}(j_3) = \{d_1, d_2\}$. These relationships are expressed in Table VI.

If $j_2$ executes after $j_1$, $d_1$ becomes idle while $d_2$ becomes busy; two devices change between idleness and busyness. The distance between $j_1$ and $j_2$ is two. If $j_3$ executes after $j_1$, $d_1$ remains busy while $d_2$ becomes busy. Only one device changes from idleness to busyness; the distance between $j_1$ and $j_3$ is one. We can construct a matrix $\mathcal{M}_{3\times 3}$ to encodes these distances; $m_{x,y}$ is the distance between $j_x$ and $j_y$. The distance matrix of these three jobs is

$$\begin{bmatrix} 0 & 2 & 1 \\ 2 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}. \tag{18}$$

$\mathcal{M}$ is a symmetric matrix. Since a job cannot execute after itself, the elements along the diagonal are not used. We assign zeros to the diagonal for simplicity. ♢

### E. Scheduling Jobs

Without loss of generality, we assume there is a *starting job* ($j_0$) that must execute first and there is a *terminating job* ($j_{n+1}$) that must execute last. It takes no time to execute these two jobs. A matrix $\mathcal{M}_{(n+2)\times(n+2)}$ represents the distances between jobs; $m_{x,y}$ is the distance between $j_x$ and $j_y$. Since $j_0$ and $j_{n+1}$ are not real jobs, the distance between any job and $j_0$ or $j_{n+1}$ is zero. Zeros are assigned to the diagonal, the first and the last rows, and the first and the last column: $\forall\, x \in [0, n+1]$, $m_{x,x} = m_{0,x} = m_{x,0} = m_{n+1,x} = m_{x,n+1} = 0$.

The matrix $\mathcal{M}$ can be treated as the *distance matrix* for a graph, $\mathcal{G} = (\mathcal{J}, \mathcal{E})$. In this graph, the vertices are jobs and they are connected by edges. Each edge has a weight; the weight for edge $(j_x, j_y)$ is $m_{x,y}$.

*Example 9:* Fig. 29 shows the distance graph of the jobs the previous example. In this figure, dashed lines have zero weights. ♢

Finding a schedule to execute all jobs is to find a "tour" that visits each vertex exactly once. The tour starts at $j_0$ and ends at $j_{n+1}$. Finding the minimum number of switches is to find a tour with the minimum total weight starting from $j_0$ and ending at $j_{n+1}$. We can merge $j_0$ and $j_{n+1}$ without changing the total weight. The problem is transformed to find a tour starting from $j_0$, visiting all jobs, and ending at $j_0$. This is equivalent to the *traveling salesperson problem* (TSP). It has been shown that TSP is an NP-complete problem [39, ND22, p. 211]. Consequently, the simplified scheduling problem is NP-complete.

TABLE VI
JOB–DEVICE RELATIONSHIP

| device | $j_1$ | $j_2$ | $j_3$ |
|---|---|---|---|
| $d_1$ | 1 | 0 | 1 |
| $d_2$ | 0 | 1 | 1 |

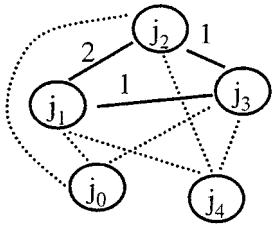


Fig. 29. Graph of jobs and their distances.

## REFERENCES


[1] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, "Predictive system shutdown and other architecture techniques for energy efficient programmable computation," *IEEE Trans. VLSI Syst.*, vol. 4, pp. 42–55, Mar. 1996.

[2] J. M. Rabaey and M. Pedram, Eds., *Low Power Design Methodologies*. Norwell, MA: Kluwer, 1996.

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

[4] Crusoe Processor. [Online]. Available: http://www.transmeta.com/crusoe/

[5] StrongARM Processor. [Online]. Available: http://developer.intel.com/design/strong/

[6] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. VLSI Syst.*, vol. 8, June 2000.

[7] R. Golding, P. Bosch, and J. Wilkes, "Idleness is not sloth," in *Proc. USENIX Winter Conf.*, New Orleans, LA, Jan., 1995, pp. 201–212.

[8] (2000, Oct.) Workshop on compilers and operating systems for low power. [Online]. Available: www.cse.psu.edu/~kandemir/colp.html

[9] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, 4th ed. Reading, MA: Addison-Wesley, 1994.

[10] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Norwell, MA: Kluwer, 1997.

[11] G. De Micheli and L. Benini, "System level power optimization: Techniques and tools," *ACM Trans. Design Automation Electron. Syst.*, vol. 5, no. 2, pp. 115–192, Apr. 2000.

[12] Y.-H. Lu, E.-Y. Chung, T. Šimunić, L. Benini, and G. De Micheli, "Quantitative comparison of power management algorithms," in *Design Automation Test Europe*, Paris, France, Mar. 2000, pp. 20–26.

[13] Y.-H. Lu and G. De Micheli, "Comparing system-level power management policies," *IEEE Design Test Comput.*, vol. 18, pp. 10–19, Mar.–Apr. 2001.

[14] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "Competitive randomized algorithms for nonuniform problems," *Algorithmica*, vol. 11, no. 6, pp. 542–571, June 1994.

[15] E.-Y. Chung, L. Benini, and G. De Micheli, "Dynamic power management using adaptive learning tree," in *Int. Conf. Comput.-Aided Design*, San Jose, CA, Nov., 1999, pp. 274–279.

[16] C.-H. Hwang and A. C. H. Wu, "A predictive system shutdown method for energy saving of event driven computation," *ACM Trans. Design Automation Electron. Syst.*, vol. 5, no. 2, pp. 226–241, 2000.

[17] L. Benini, A. Bogliolo, G. A. Paleologo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Trans. Comput.-Aided Design Integrated Circuits Syst.*, vol. 16, pp. 813–833, June 1999.

[18] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes," *Design Automation Conf.*, pp. 555–561, June, 1999.

[19] T. Šimunić, L. Benini, P. W. Glynn, and G. De Micheli, "Dynamic power management for portable systems," in *Proc. Int. Conf. Mobile Computing Networking*, Boston, MA, Aug. 2000, pp. 11–19.

[20] Q. Qiu, Q. Wu, and M. Pedram, "Dynamic power management of complex systems using generalized stochastic petri nets," in *Proc. Design Automation Conf.*, Los Angeles, CA, June 2000, pp. 352–356.

[21] E.-Y. Chung, L. Benini, A. Bogliolo, and G. De Micheli, "Dynamic power management for nonstationary service requests," in *Proc. Design Automation Test Europe*, Munich, Germany, Mar. 1999, pp. 77–81.

[22] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar, "Scheduling techniques to enable power management," in *Proc. Design Automation Conf.*, Las Vegas, NV, June 1996, pp. 349–352.
[23] E. Musoll and J. Cortadella, "Scheduling and resource binding for low power," in *Proc. Int. Symp. Syst. Synthesis*, 1995, pp. 104–109.
[24] J. J. Brown, D. Z. Chen, G. W. Greenwood, X. Hu, and R. W. Taylor, "Scheduling for power reduction in a real-time system," in *Proc. Int. Symp. Low Power Electron. Design*, Monterey, CA, Aug. 1997, pp. 84–87.
[25] J. R. Lorch and A. J. Smith, "Scheduling techniques for reducing processor energy use in MacOS," *Wireless Networks*, vol. 3, no. 5, pp. 311–324, 1997.
[26] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Proc. Design Automation Conf.*, New Orleans, LA, June 1999, pp. 134–139.
[27] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proc. Symp. Operating Syst. Design Implementation*, Monterey, CA, Nov. 1994, pp. 13–23.
[28] G. Qu and M. Ptokonjak, "Energy minimization with guaranteed quality of service," in *Proc. Int. Symp. Low Power Electron. Design*, Rapallo, Italy, July 2000, pp. 43–48.
[29] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin, "DRAM energy management using software and hardware directed power mode control," in *Proc. Int. Symp. High-Performance Comput. Architecture*, Monterrey, Mexico, Jan. 2001, pp. 159–169.
[30] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power aware page allocation," in *Proc. Int. Conf. Architectural Support Programming Languages Operating Syst.*, Cambridge, MA, Nov. 2000, pp. 105–116.
[31] C. S. Ellis, "The case for higher-level power management," in *Proc. Workshop Hot Topics Operating Syst.*, Rio Rico, AZ., Mar. 1999, pp. 162–167.
[32] Y.-H. Lu, L. Benini, and G. De Micheli, "Requester-aware power reduction," in *Proc. Int. Symp. Syst. Synthesis*, Madrid, Spain, Sept. 2000, pp. 18–24.
[33] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *Proc. ACM Symp. Operating Syst. Principles*, Kiawah Island, SC, Dec. 1999, pp. 48–63.
[34] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Power analysis of embedded operating systems," in *Proc. Design Automation Conf.*, Los Angeles, CA, June 2000, pp. 312–315.
[35] J. Flinn and M. Satyanarayanan, "PowerScope: A tool for profiling the energy usage of mobile applications," in *Proc. IEEE Workshop Mobile Comput. Syst. Applicat.*, New Orleans, LA, Feb. 1999, pp. 2–10.
[36] Y.-H. Lu, L. Benini, and G. De Micheli, "Operating-system directed power reduction," in *Proc. Int. Symp. Low Power Electron. Design*, Rapallo, Italy, July 2000, pp. 37–42.
[37] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, 2nd ed. Reading, MA: Addison-Wesley, 1997.
[38] J. Wilkes, "Predictive power conservation," Hewlett-Packard, Tech. Rep., HPL-CSP-92-5, 1992.
[39] M. R. Garey and D. S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Complteness*, New York: W. H. Freeman, 1979.

**Yung-Hsiang Lu** received the B.S.E.E. degree from National Taiwan University, Taipei, Taiwan, the M.S.E.E. degree, and the Ph.D degree in electrical engineering from Stanford University, Palo Alto, CA, in 1992, 1996, and 2002, respectively.

His research focus includes low-power system design.

**Luca Benini** (S'94–M'97) received the Ph.D. degree in electrical engineering from Stanford University, Palo Alto, CA, in 1997.

Since 1998, he has been with the University of Bologna, Bologna, Italy, Department of Electronics and Computer Science where he is currently an Associate Professor. He is a Visiting Researcher at Stanford University and the Hewlett-Packard Laboratories, Palo Alto, CA. His research interests include all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications, and in the design of portable systems. On these topics he has published more than 120 papers in international journals and conferences, a book, and several book chapters.

Dr. Benini is a member of the organizing committee of the International Symposium on Low-Power Design. He is a member of the technical program committee for several technical conferences, including the Design and Test in Europe Conference, International Symposium on Low Power Design, and the Symposium on Hardware–Software Codesign.

**Giovanni De Micheli** (S'79–M'79–SM'89–F'94) received the Nuclear Engineer degree from Politecnico di Milano, Milano, Italy, in 1979, and the M.S. degree and Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley in 1980 and 1983, respectively.

Currently, he is Professor in both the Department of Electrical Engineering and the Computer Science Department (by courtesy) at Stanford University. He has been with the IBM T.J. Watson Research Center, Yorktown Heights, NY, with the Department of Electronics, Politecnico di Milano, Italy and with Harris Semiconductor in Melbourne, Florida. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software co-design and low-power design.He is author of: *Synthesis and Optimization of Digital Circuits*(McGraw-Hill: New York, 1994), coauthor and/or coeditor of five books and over 250 technical articles.

Dr. De Micheli is a Fellow of ACM. He received the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000. He received the 1987 IEEE TRANSACTIONS ON CAD/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and in 1993. He is Editor in Chief of the IEEE TRANSACTIONS ON CAD/ICAS. He was Vice President (for publications) of the IEEE CAS Society in 1999–2000. He was the Program Chair and General Chair of the Design Automation Conference (DAC) in 1996–1997 and 2000, respectively. He was the Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively. He was also codirector of the NATO Advanced Study Institutes on Hardware/Software Codesign, held in Tremezzo, Italy, 1995 and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, 1986.