# POWER-AWARE OPERATING SYSTEMS FOR INTERACTIVE SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yung-Hsiang Lu
December 2001

# Abstract

Reducing power consumption is increasingly important due to three recent trends: (1) Battery-powered portable systems are more and more popular. Reducing power prolongs the operational time between recharging batteries. (2) Power is dissipated mainly as heat, and excessive heat has become a barrier for future performance improvement. Rising temperatures reduce the reliability of electronic components. Heat also increases the cost of packaging and cooling. (3) The concept of "green computers" requires better energy efficiency to decrease the demand for more power plants and to alleviate the impact on our environment.

Many systems use operating systems (OS), such as Linux and Windows, to manage resources; these resources include processor time, memory, and input-output (IO) devices. OS provides interfaces for other programs and makes it easier to port programs onto different systems. For instance, programs use network protocols without knowing the details of network interface cards. Programs perform computation, read files, send network packets, and so on; that is, they request services from hardware. Reading a file makes a hard disk consume power; sending a packet causes a network card to consume power. Even though power is consumed by hardware, hardware consumes power to serve software programs. OS is the interface between hardware and software; consequently, OS can play a pivotal role in power reduction.

This research focuses on using operating systems to control the power states of IO devices and processors dynamically. The operating systems estimate future utilization of these devices and processors. When the utilization is low, they are set to lower-power states. This approach is called dynamic power management. Power management may degrade performance. One example is to spin down the plates in hard disks to save power. It takes several seconds to spin up the plates before files can be read from or written to the disks; requests have to wait for the spin-up. Thus, it is essential to predict utilization accurately in order to save power and maintain satisfactory performance.

Operating systems can be structured as layers, including device drivers, process managers, and schedulers. Device drivers are the closest layer to hardware and can observe requests sent to hardware for services. Unfortunately, device drivers have only limited information about running programs for predicting future hardware utilization. Another approach is to combine power management with process management. Process managers are closer to application programs than device drivers; hence, process managers have more information regarding which programs generate requests. Process managers also maintain the current states of individual programs. With additional information, better accuracy for predicting utilization can be achieved. Moreover, process schedulers can collaborate with power managers. Schedulers determine the execution order of programs and directly control when requests are generated; in this way, utilization can be calculated accurately.

This thesis investigates how to use the information from different layers in operating systems to improve power management for interactive systems, such as a laptop computer. First, it describes the principles for designing power-management policies and a framework for implementing policies. Then, it presents methods that use process managers and process schedulers to perform power management on IO devices. The final part of this thesis introduces a software-based method for reducing the power consumption of processors which have multiple power states. Application programs are modified to insert data buffers. The buffers are filled when the processors run in the states with higher power and performance; then, the buffers are drained while the processors enter lower-power states. If buffers are inserted, it is possible to achieve nearly optimal power saving on processors that have only finite power states. Buffering can also improve the response time of sporadic jobs. A graph traversal technique is used for efficiently assigning power states to achieve minimum power consumption.

The methods presented in this thesis have been implemented in Microsoft Windows and Linux on desktop, laptop, and palmtop computers to demonstrate their effectiveness in power saving.

# Acknowledgments

This thesis would not be possible without the inspiration and patient guidance from my advisor, Professor De Micheli. I am very grateful to my reading committee, Professor Dawson Engler and Professor Teresa Meng, for their encouragement and suggestions during my Ph.D. study. I want to thank Professor Luca Benini of Università di Bologna, Italy, for the discussion in hundreds of email correspondences.

The support from Stanford CAD group is invaluable. I learned a lot from my officemates, Eui-Young Chung and Tajana Šimunić; they gave immediate feedback on my ideas. I also want to thank current and past members in our group: Armita Peymandoust, Terry Ye, Chaiyasit Manovit, Davide Bertozzi, Luc Semeria, Jim Smith, and Vincent Mooney. The administrative and technical staff made Computer Systems Laboratory an enjoyable working environment. Credits belong to Evelyn Ubhoff, Charles Origish, Kathleen DiTommaso, and Patrick Burke.

I am deeply in debt to my family, especially my sister Yung-Ling Lu in California. I want to thank my friends in Taiwan and USA; they have made my life full of fun.

This research project was sponsored by MARCO/DARPA Gigascale Silicon Research Center and NSF under contract CCR-9901190. Their financial support made this work possible.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Reducing power consumption is a major design goal for electronic systems, from portable computers to servers. Portable systems, such as notebook computers and personal digital assistants (PDA), are increasingly popular in recent years. These systems may obtain power from batteries. Batteries have only limited capacity; therefore, it is essential to reduce power consumption in order to prolong the operational time before recharging batteries. For systems that obtain power from power grids, reducing power consumption lowers electric bills, saves cooling costs, and lessens fan noise. Lately, the power consumption of high-performance processors has been rapidly increasing due to advanced architectural techniques, such as parallel and speculative execution. These processors consume more than thirty Watts of power; within five years, high-performance processors may consume hundreds of Watts. Today, a typical Internet server center has hundreds of servers and consumes hundreds of thousands of Watts to operate these servers and the air conditioners. Power-related expense is a major portion in the operating cost of a server center. Furthermore, power is dissipated mainly as heat; the heat raises temperatures and lessens reliability. Excessive heat has become a major barrier for future performance improvement. In a nutshell, reducing power consumption is critical for all types of electronic systems.

Power is consumed by hardware; however, hardware consumes power to serve software. For example, when a program sends network packets, it causes a network card to consume power. Similarly, when a program writes to a file, it causes a hard disk to consume power. Software generates requests; network packets and file writes are instances of requests, which make hardware consume power. The collaboration between software and hardware can reduce power.

An operating system (OS) is a special program as the interface between hardware and other software programs. Many interactive systems use operating systems; for example, iPAQ is a PDA supporting both Microsoft Windows and Linux. The OS makes it easier to support the same user interfaces like web browsers. If the same OS is supported by a platform, application programs can be ported more easily. As hardware advances, systems become increasingly complex; they use OS to manage resources such as CPU time and memory allocation. Power is a precious resource; it should be properly managed by the OS. Because OS is the interface between software and hardware, it can observe requests generated from software and predict future utilization of hardware. When a hardware component is predicted to have low utilization (or is even idle), this component can be set to a lower-power state to reduce power consumption.

Operating systems are structured into layers, including device drivers, process managers, and process schedulers. These layers provide distinct advantages and disadvantages for saving power. Device drivers are the layer closest to hardware and can observe requests sent to hardware for services; power management can be implemented in device drivers. Unfortunately, device drivers have limited information about running programs for predicting future utilization. Alternatively, we can combine power management with process management. Process managers are closer to application programs than device drivers; hence, process managers have more information regarding which programs generate requests. Process managers also maintain the current states of individual programs. With additional information, we can improve the prediction accuracy for power management. Moreover, we can integrate power management with schedulers. Process schedulers determine the execution order of programs and directly control when requests are generated; in this way, utilization can be calculated accurately.

In this thesis, we will discuss how to use the information from different parts of operating systems to facilitate power reduction. We will also explain the advantages and disadvantages in each part of operating systems for implementing power management.

## 1.2 Dynamic Power Management

A typical system does not have to operate at its peak performance continuously; instead, its workloads change from time to time. In particular, an interactive system is idle when there are no requests from a user, as illustrated in Figure 1.1. When the system is idle, it should enter a low-power *sleeping* state to save energy [31]. *Dynamic power management* puts an idle system (or subsystem) into its sleeping state to reduce power consumption [13] [58]. A commonly seen example is to spin down the plates of a hard disk drive (HDD) when it is not used.

A subsystem is in an *active* state if it can serve requests from a workload. For example, a hard disk is in an active state when its plates are spinning. In contrast, a subsystem is in an *inactive* state if it cannot serve requests. When the plates stop spinning, the hard disk is in an inactive state. This thesis uses inactive and sleeping states interchangeably.

Some processors have multiple active states; these processors consume different amounts of power and provide different performance in distinct active states. For CMOS-based processors, slowing down their clock speeds, namely lowering their frequencies, reduces their power consumption and their performance. This method is also called *frequency scaling*; we consider frequency scaling as a special case of power management. This thesis treats processors as devices with multiple active states, unless it is necessary to distinguish IO devices from processors.

### 1.2.1 Power Management Overhead

When a device is in a sleeping state, it consumes less power, but it cannot serve requests. Hence, when a new request arrives, the device has to wake up and enter a

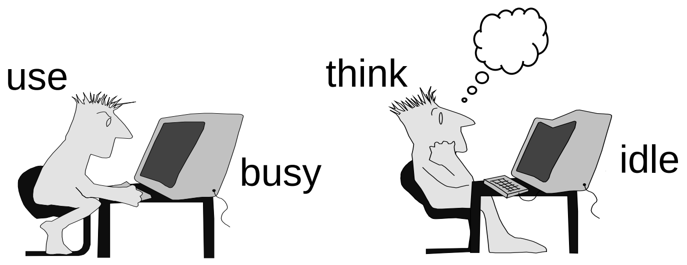


Figure 1.1: An interactive system is busy or idle depending on user requests.

higher-power *working* state to serve requests. The transition from working to sleeping states is called *shutdown*; the transition from sleeping to working states is called *wakeup*. State transitions have overhead, including delay and additional energy.

Figure 1.2 shows the measured power consumption of a 2.5” HDD (Hitachi DK23AA-60) during state transitions. The top of this figure shows the shutdown transition and the bottom shows the wakeup transition. Transitions have long delays and consume large amounts of energy. The details of this measurement are presented in Section A.1. For processors, the time to wake up is shorter: dozens of milliseconds [11]; this is equivalent to many thousands of clock cycles. Even if a processor remains active, it still takes thousands of cycles to change frequencies [36]. The overhead of frequency scaling cannot be ignored.

If there were no state-transition overhead, power management would be a trivial problem: shut down a device whenever it is idle. In reality, the overhead has to be amortized by keeping the device in a low-power state long enough. Because of the transition overhead, it is undesirable to shut down a device every time it becomes idle. Otherwise, it can actually increase energy consumption and substantially degrade performance. Power management is challenging because it has to predict the future utilization of a hardware device and to determine the power states.

When a device is idle, it may sleep to save power. If a device has multiple active states, it can enter a lower-power active state when the utilization is low. *Power management policies* (or more simply, *policies*) are the rules of determining when to change power states and which states to enter. In order to make such a decision, a policy has to predict idleness and utilization.

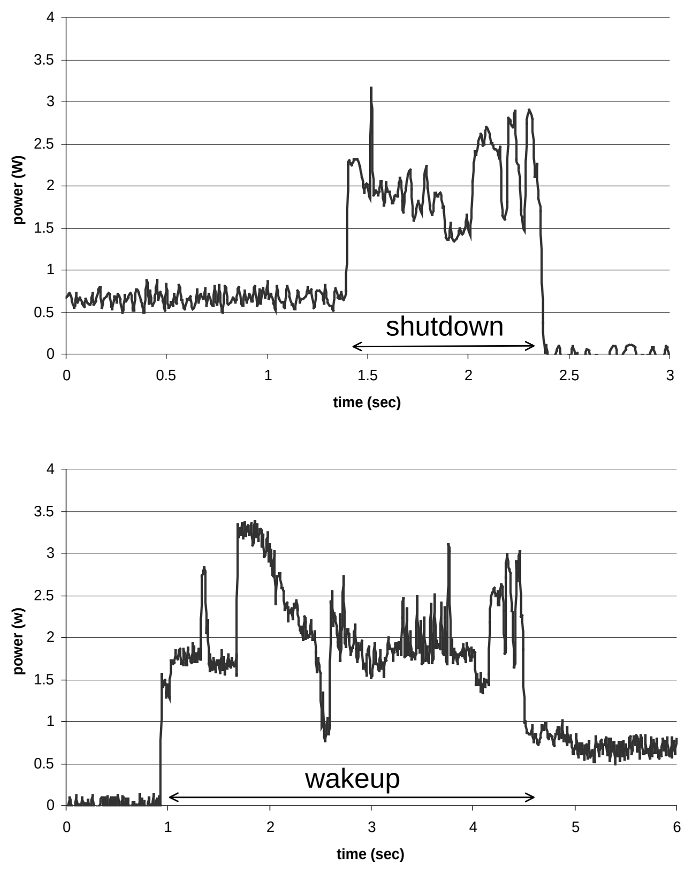


Figure 1.2: power consumption of 2.5 inch hard disk during state transitions

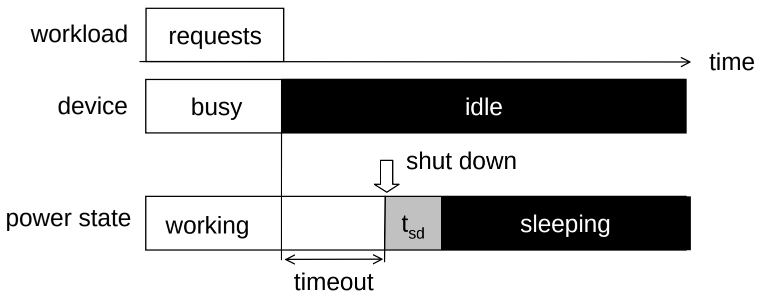


Figure 1.3: timeout policy

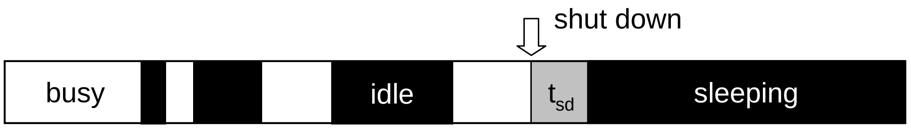


Figure 1.4: predictive policy

## 1.2.2 Policies Predicting Idleness

These policies predict when a device becomes idle so it can sleep. The policies are classified into three categories: timeout, predictive, and stochastic [20].

### Timeout Policies

A timeout policy shuts down a device after it has been idle longer than the “timeout” value. These policies assume that if a device is idle longer the timeout length, the device will remain idle. Figure 1.3 illustrates this concept; $t_{sd}$ is the time required to shut down the device. Timeout policies are widely used in today’s commercial products because of their simplicity. For example, Microsoft Windows allow users to set the values for both hard disks and monitors from several minutes to a few hours through the Control Panel.

### Predictive Policies

A predictive policy assumes that the lengths of idle periods follow a pattern; the length of a future idle period can be accurately predicted by the recent history of requests. This concept is illustrated in Figure 1.4. In this figure, each idle period is indicated by a black rectangle. The lengths of these idle periods become longer and longer. When an idle period is long enough to amortize the overhead, the power manager shuts down the device. Predictive policies capture such a pattern; its advantage is to save the energy during the timeout period.

#### Stochastic Policies

These policies model request generation as stochastic processes and solve stochastic optimization problems. Figure 1.5 is one simple stochastic model. When a device is busy, it has 90% probability to remain busy and 10% probability to become idle. When it is idle, the probability to remain idle is 95% and the probability to become busy is 5%. Stochastic policies formulate power management as constrained optimization problems; they provide the flexibility to trade off between power and performance explicitly. Some stochastic policies require solving complex optimization problems; hence, they need the characteristics of typical workloads in advance to do off-line analysis before a power manager can be applied at run time.

### 1.2.3 Policies Predicting Utilization

If a device has multiple active states, we can reduce its power consumption when the utilization is lower. Many policies have been proposed to manage processor power; these policies can be classified into two categories depending on their capability of obtaining priori knowledge or even affecting the workloads. If a policy has no knowledge about future workloads, it can use recent utilization to predict future utilization. If a policy can control workloads, it can *schedule* them to make the utilization at desirable levels.

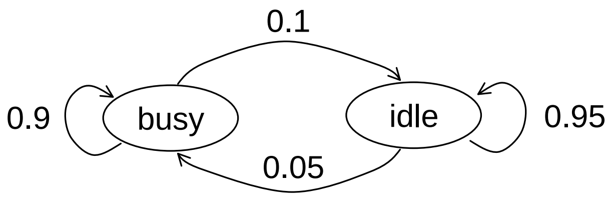


Figure 1.5: stochastic model

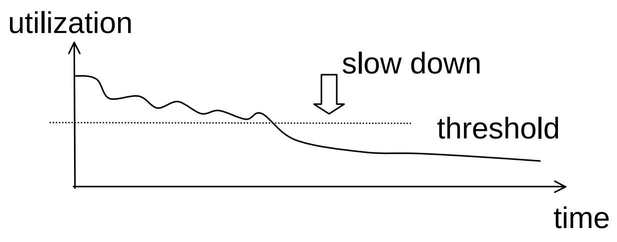


Figure 1.6: slow down a processor during low utilization

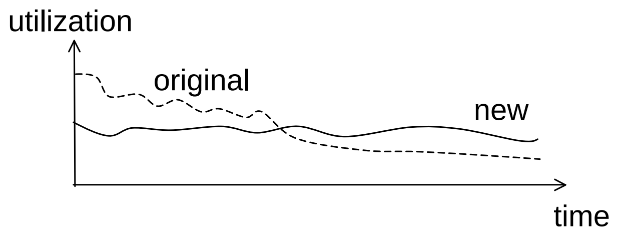


Figure 1.7: schedule workload to keep low utilization

### History-Based Policies

Figure 1.6 shows that the utilization of a processor is decreasing. When the utilization is below a threshold, the power manager slows down the processor. In [82], the authors use the utilization in the previous interval to predict the utilization of the next interval. In [64], the authors compare several variations of similar approaches. StrongARM is a commercial processor that support multiple power states; power management is implemented on StrongARM to investigate its effectiveness [36].

### Scheduling-Based Polices

These policies arrange the execution of the workloads so that the utilization remains low enough and the processor can slow down. This concept is illustrated in Figure 1.7. In order words, the policies do not simply *predict* future utilization; instead, they *control* the utilization by *scheduling* the execution of workloads. In [43], the authors prove that a processor consumes less power if it stays in a constant lower speed than varying between high and low speeds. Scheduling techniques have been applied to reduce power on various systems [15] [40] [47] [70].

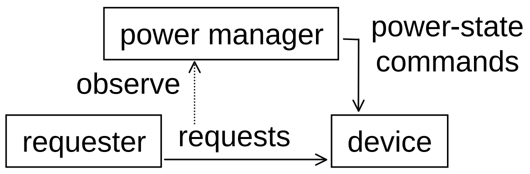


Figure 1.8: single-request model

## 1.3 Requester Models

Up until now, we have implicitly assume the existence of a special entity called "requester" that generates workloads, including IO requests and computation needs. Request modeling is one essential part of power management because policies predict future workloads based on their requester models. We consider four requester models for designing policies: single requester, multiple requesters, multiple requesters with creation and termination, and multiple requesters with timers. These models are increasingly complex and closer to the programs running on realistic interactive systems like a laptop computer. Additionally, periodic requests with timing constraints attract researchers' attention in recent years; such models consider multimedia workloads that require constant video and audio output rates.

### 1.3.1 Single-Requester Model

Figure 1.8 depicts the concept of the single-request model. The requester generates requests for the device; meanwhile, the power manager observes the requests. Based on this observation, the power manager issues commands to change the power states of the device. Some policies explicitly use this model in determining their rules to change power states [12] [19] [20] [66] [81]; some other policies implicitly assume a single requester [41] [48] [52] [73].

### 1.3.2 Multiple-Requester Model

In complex systems, there may be more than one entities that generate requests. For example, in a multiprogramming system, several processes may generate requests

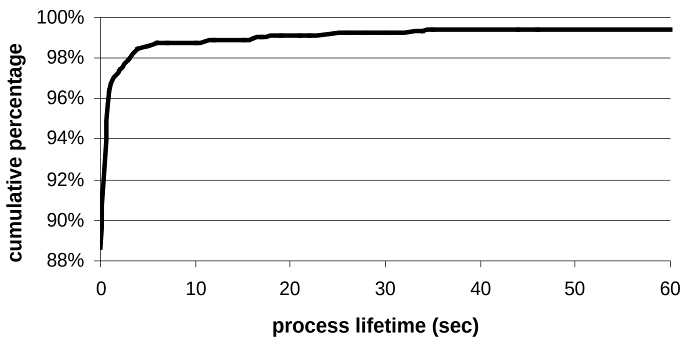


Figure 1.9: process lifetime

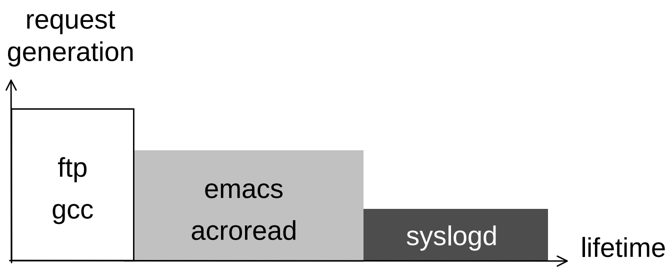


Figure 1.10: Programs that generate intensive requests usually have short lifetimes.

to the same device. In [28] [23], the authors demonstrate that different processes consume different energy; in particular, one study finds that an X server consumes large energy on both a network card and a hard disk.

### 1.3.3 Requester Creation and Termination

A more general model considers the creation and the termination of requesters. When a requester terminates, it can no longer generate any requests. One measurement shows that most processes have very short lifetime [50], as illustrated in Figure 1.9. While these numbers may change for different workloads, most processes are still likely to have short lifetimes because most activities in computers are bursty.

We also observe that a program which generates many IO requests within short time periods often have short lifetimes; examples are file writes from `ftp` and `gcc`. Some other programs, such as `emacs`, generate requests less frequently and have longer lifetimes. Daemons like `syslogd` have long lifetimes, possibly since the computer is powered on; they infrequently generate IO requests. This observation is illustrated in Figure 1.10.

Even though process termination is important in predicting future requests, it is hard to predict requests based on process creation. This is because interactive systems have few processes running the same code. When a process is created, it usually executes code different from its parents. In UNIX, new processes often call `execl` right after they are created by `fork`. Therefore, predicting future requests based on process creation is much more challenging.

### 1.3.4 Requesters with Timers

A process may create requesters "in the future" when a specific event occurs. For example, a text editor saves the content every five minutes. In UNIX, this process creates a timer that expires in five minutes. When the timer expires, a *signal* is sent to the process. This process captures the signal and saves the content into the hard disk. Existing UNIX timer mechanism does not consider power management; power managers cannot predict future requests based on timer information. Making application programs "power-aware" has been suggested lately [26] [51] [65]. One approach is to allow processes to specify which device will be used in the future. Such information helps power managers to make correct decisions for changing power states.

### 1.3.5 Periodic Requests with Timing Constraints

Multimedia workloads are now supported on palm-size battery-powered computers [38]. These workloads have distinct requirements: constant video and audio output to provide satisfactory user experience. A typical timing constraint is to display one video frame every 33 milliseconds to maintain 30 frames per second. Reducing power consumption for this type of workloads is an active research topic [5] [42] [59].

## 1.4 Thesis Contributions

This thesis focuses on dynamically adjusting the power states of IO devices and processors in interactive systems; examples of such systems include desktop, laptop, and PDA-like palm-size computers. The thesis has four major contributions: a framework to implement policies, a process-based power management policy, a low-power scheduling procedure for IO devices, and an efficient frequency scaling algorithm to reduce processor power.

### 1.4.1 Framework to Implement Policies

While many system-level power management policies have been proposed, few of them have been implemented in real systems. Most of the policies were evaluated by simulations. It was unclear how to implement these policies on real systems and the overhead of the implementation.

This thesis presents a framework in Windows 2000 for implementing policies. It is built using *filter drivers* and provides a template for experimenting new policies. This approach has two major advantages: (1) a human user can interact with the system while a policy is running (2) there is no need to modify OS kernel or application programs. This implementation allows users to judge the performance impact of a policy. Based on this framework, we implemented and compared nearly a dozen policies representing the three categories of policies (timeout, predictive, and stochastic) and pointed out their strength and drawbacks. This is the first time these policies were compared in the same environment for the same workloads while interacting with users. Two metrics are proposed to quantify the performance degradation of policies in an interactive system.

This study reveals a few important findings. First, most power-management policies do not need substantial computation. Second, it is important to maintain short response time and to avoid frequently shutting down a device. Third, some policies are inapplicable to certain devices whose state-transition delays are too long; hence, it is important to understand the limitation of policies.

### 1.4.2 Process-Based Power Management

The majority of existing policies were designed based on the single-requester model, explained in Section 1.3.1. While this model is simpler, it does not capture the essence of program execution in a computer: multiple programs are running concurrently. These programs may have different priorities. Some programs execute more frequently than the others; furthermore, sometimes a program terminates. A program can be computation-intensive, IO-intensive, or change between computation and IO intensive. The single-requester model ignores all the differences among programs.

We propose a policy using a more realistic requester model, explained in Section 1.3.3. This policy distinguishes requesters as individual processes and considers the execution time of these processes. It is called "process-based" power management. It also considers the creation and termination of processes as well as as how often these processes execute. Because it uses additional information from OS kernel, it predicts future utilization more accurately. This policy is implemented in Linux running on a laptop computer; experimental results show that our new policy can improve power reduction while maintaining satisfactory performance.

### 1.4.3 Low-Power Scheduling for IO Devices

Even though process-based policy enhances power saving, it still estimates future utilization based on history. In a computer, many requests can be accurately predicted through process schedulers. An example of such requests is to periodically download information from the Internet. Power managers can obtain information about future requests from process schedulers; power managers can also affect scheduling to rearrange the idle periods of devices.

The third part of this thesis explains how to combine process scheduling with power management. Schedulers determine when a process executes, so they directly control the generation of requests. Using the information from schedulers, the idle times of devices are predicted accurately. In order to allow programs to specify their device requirements and when requests are generated, we created an application programming interface (API) that allows programs to collaborate with power managers.

This method brings the awareness of power management to application programs. Because finding a minimum-power schedule is computationally expensive, heuristics are used. The basic idea is to classify jobs by the devices needed. If two jobs need the same devices, they execute together. Experimental results show better power saving and performance when power management is integrated into process schedulers.

### 1.4.4 Frequency Scaling to Reduce Processor Power

Many portable systems can execute multimedia programs, such as playing MPEG movies. Multimedia programs need to update images periodically, namely a constant frame rate, to provide satisfactory quality of service. It is desirable to maintain the output rates while reducing power consumption. Many modern processors have multiple power states; they provides lower performance in a lower-power state. Ideally, a processor should run at a low-power state while keeping the output rates. Meanwhile, it is important that the systems respond promptly to sporadic user inputs. Such a mixture of periodic and sporadic workloads creates new challenges in power saving.

The last part of this thesis addresses the issues to control finite power states of a processor for running mixed workloads. We modify a multimedia program and divide it into multiple stages; buffers are inserted between stages. When the processor runs at a higher-power state, it uses the slack time to fill buffers. Later, the processor enters a lower-power state while draining the data stored in the buffers. If a sporadic user input arrives, the buffered data are drained so that the processor can spend more time to handle this input without disrupting the output rate.

This problem can be formulated as integer linear programming (ILP). While ILP is flexible, it is often computationally expensive. We develop a graph-based method to find the assignments of power states efficiently. It builds a space-search graph whose vertices represent the current states of buffers, the power state, and how buffers are filled (or drained). The time complexity of this method depends on the number of vertices and is independent of the workloads; hence, it is able to handle multimedia workloads that have very large numbers of frames. Our experimental results suggest that buffering is effective without occupying large memory. We also show that only a few power states are sufficient to achieve nearly optimal power saving.

### 1.4.5 Relationship with Previous Work

This thesis differs from previous work in four ways. First, it is not "yet another policy". Most previous work surveyed in Section 1.2.2 adopts the single-requester model. In contrast, our work emphasizes the importance of using a better and realistic requester model. Second, this study stresses on the importance of implementation. It is essential to demonstrate the applicability of our schemes on real systems. Third, we consider both power saving and performance degradation due to power management. We propose metrics to quantify performance in interactive systems. Finally, we reduce the computation needed for power managers. Assigning power states to processors was previously formulated as integer linear programming (ILP) and suffered from the computation needed to solve ILP. We reformulate the problem and present a polynomial-time solution. In summary, this thesis emphasizes a practical requester model, the performance impact of power management, and the efficiency of power managers.

### 1.4.6 Limitations

This study takes a "macro view" of computers and ignores most details inside both software and hardware. One goal is to avoid rewriting application programs by confining most of the modification in operating systems. More power reduction may be achieved by optimizing application programs for power saving. Similarly, we use commercially available hardware components and treat them as "black-boxes" even though it is possible to improve power saving by modifying these components such as adding more power states.

Operating systems can affect power consumption in different ways. For example, page-replacement policies affect when page faults occur and the idle time of hard disks. In addition, some daemons periodically write memory contents to hard disks. While this maintains system consistency in case of crashes, it also reduces the idle time. This thesis does not discuss how to design power-efficient page-replacement policies or daemons.

This study assumes that performance degradation is tolerable on interactive systems. Some other types of systems may be timing-critical; missing their timing constraints can lead to catastrophic results. Reducing power on such systems is beyond the scope of this thesis.

## 1.5 Thesis Organization

Chapter 2 explains the criteria to design policies; these criteria are based on saving power and maintaining performance. This chapter proposes two performance metrics and discusses the relationship between frequencies and power consumption. Chapter 3 presents a framework for implementing policies to control the power states of IO devices. Using this framework, this chapter compares the power saving and performance impact of a dozen policies. Chapter 4 describes a policy that incorporates information from the process manager in OS kernel. With the additional information, the power manager can predict idle time more accurately and save more energy. Chapter 5 explains how to schedule processes to facilitate power management. By properly scheduling processes, power managers can control the lengths of idle periods to enhance power saving. Chapter 6 proposes a method that inserts data buffers in programs and adopts a graph-based algorithm to find optimal assignments of power states on a processor. This method greatly reduces the time complexity for workloads with very long time horizons. Finally, Chapter 7 concludes this thesis and discusses directions for future work.

# Chapter 2

# Policy Design Principles

This chapter discusses quantitative criteria for designing policies. An ideal policy achieves the maximum power saving with the minimum performance degradation.

## 2.1 Amortizing Energy Overhead

In order to amortize the energy overhead, a device has to stay in a low-power state long enough. The *break-even time* ($t_{be}$) is the criterion to determine whether a device should enter a low-power state [11]. The following explanation considers one active (working) and one inactive (sleeping) state. If a device has more power states, the break-even time between any two states can be calculated in the same way.

Figure 2.1 demonstrates how to compute the break-even time of a device. There

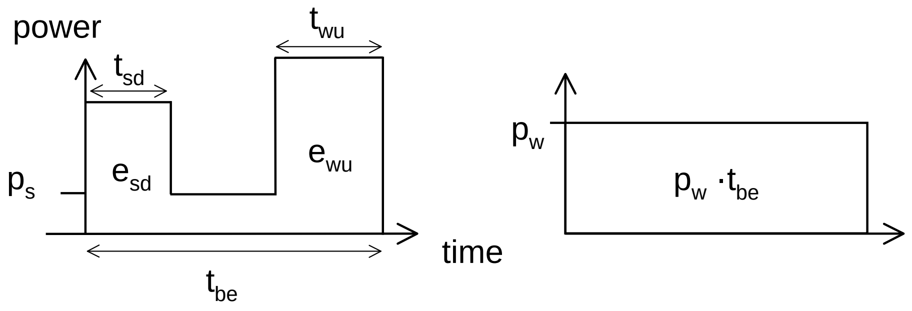


Figure 2.1: compute the break-even time of a device

| | |
|---|---|
| $t_{be}$ | break-even time |
| $t_{sd}$ | shutdown delay |
| $t_o$ | transition delay, $t_o = t_{sd} + t_{wu}$ |
| $t_{wu}$ | wakeup delay |
| $e_{sd}$ | shutdown energy |
| $e_{wu}$ | wakeup energy |
| $e_o$ | transition energy, $e_o = e_{sd} + e_{wu}$ |
| $t_{ms}$ | shortest sleeping time to save power |

Table 2.1: device parameters

are two cases: on the left, the device is shut down and consumes $e_{sd}$ and $e_{wu}$ during state transitions. On the right, the device stays in the working state and consumes constant power $p_w$. The length of $t_{be}$ makes the two cases "break-even". When the two cases consume the same energy, then

$$p_w \cdot t_{be} = e_{sd} + p_s \cdot (t_{be} - t_{sd} - t_{wu}) + e_{wu} \tag{2.1}$$

therefore, the break-even time of this device is

$$t_{be} = \frac{e_o - p_s \times t_o}{p_w - p_s} \tag{2.2}$$

Since $t_{be}$ includes the transition delays, it must be larger than $t_{sd} + t_{wu}$. Consequently,

$$t_{be} = \max(t_o, \frac{e_o - p_s \times t_o}{p_w - p_s}) \tag{2.3}$$

The break-even time is a parameter of each device; it is independent of requests. We can also define the minimum time in the sleeping state to save energy. Let $t_{ms}$ be this minimum sleeping time. It is computed by subtracting transition delays from the break-even time:

| | Hitachi (2.5") | IBM (3.5") |
|---|---|---|
| $t_{sd}$ | 0.97 sec | 0.52 sec |
| $t_{wu}$ | 3.71 sec | 6.97 sec |
| $e_{sd}$ | 1.98 J | 1.08 J |
| $e_{wu}$ | 7.46 J | 52.5 J |
| $p_w$ | 0.67 W | 3.48 W |
| $p_s$ | 0.03 W | 0.75 W |
| $t_{be}$ | 14.5 sec | 17.6 sec |

Table 2.2: parameters of a Hitachi disk and an IBM disk

$$t_{ms} = t_{be} - t_{sd} - t_{wu} = \frac{e_o - p_w \times t_o}{p_w - p_s} \tag{2.4}$$

The parameters of the disk shown in Figure 1.2 is summarized in Table 2.1. The table also lists the parameters of an IBM 3.5-inch disk (IBM DTTA 350640). The 3.5" disk consumes substantially more power than the 2.5-inch disk.

## 2.2 Power / Performance Tradeoff

In addition to saving power, power managers have to consider the impact on performance. Power management degrades performance because some requests have to wait for power-state transitions. This delay may be substantial: several seconds for a typical hard disk. Power management trades off performance for power reduction. On one hand, a device can remain in the working state and is always ready to serve requests; this wastes energy when the device is idle. On the other hand, the device can sleep to save power but this degrades performance. In addition to degrading performance, power management may also affect reliability. One study reports that a shutdown-wakeup cycle is equivalent to four hours of continuous usage of a hard disk [34].

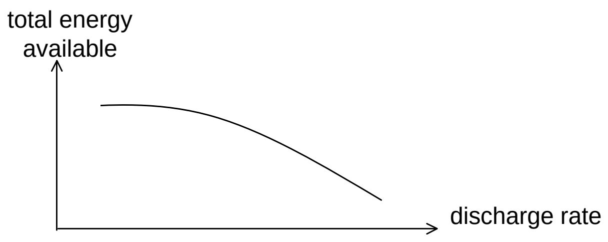


Figure 2.2: total energy and discharge rate

### 2.2.1 Power Metrics

The main purpose of power management is to reduce average power. Average power is defined as energy divided by time:

$$\text{average power} = \frac{\text{energy}}{\text{time}} \tag{2.5}$$

For portable systems, reducing power consumption increases operational time of batteries. In fact, the energy retrievable from a battery before the next recharging is not constant. When the discharge rate (i.e. current) is too large, the total energy (VAH, volt-amper-hour) available from a battery decreases [10] [63]; Figure 2.2 illustrates this situation. Lower power consumption has two effects: (1) the battery can provide more overall energy (2) the energy is consumed at a smaller rate. Consequently, the operational time is longer when the power consumption is lower.

In addition to average power, some other power metrics are considered, such as transient power. The transient power is the derivative of energy over time:

$$\text{transition power} = \frac{d \text{ energy}}{d \text{ time}} \tag{2.6}$$

The maximum power is the largest transient power. Restricting the maximum power is important because all power supplies have limits on the maximum transient power. Power managers should avoid waking up multiple devices simultaneously if the total wakeup power may exceed the limits of the power supplies.

| | |
|---|---|
| $ws_i$ | starting time of the $i^{th}$ waiting period |
| $we_i$ | ending time of the $i^{th}$ waiting period |
| $wd$ | size of the observation window |
| $wt$ | starting time of an observation window |
| $ml$ | length of user memory |

Table 2.3: performance parameters

### 2.2.2 Performance Metrics

Performance is commonly defined as the total time needed to perform specific tasks, such as running SPEC benchmarks [39]. These metrics measure the throughput of "batch" or transaction-based systems. They do not directly apply to interactive systems. For most users, response time is as important as throughput. When a device is power-managed, its performance degrades due to the delays when changing power states. It is essential to quantify the performance perceived by users.

Users have limited memory; they remember an event that just occurred more clearly than another event that had occurred long time ago. In other words, the negative effect of a delay decreases monotonically as time goes by. Also, users tend to "forgive" and forget delays if they do not happen often. Based on this observation, we propose two types of performance metrics: (1) how much waiting time is perceived (2) how often delays occur. Both metrics are measured in small observation windows (several minutes) that reflect limited user memory. In this section, "delay" and "waiting period" are used interchangeably.

#### Total Delay in a Small Window

Suppose $ws_i$ and $we_i$ are the starting and ending times of the $i^{th}$ waiting period due to power management. Consider an observation window of size $wd$. If this window starts at $wt$, the total waiting time is the sum of all waiting periods enclosed by this window. Namely, the starting time ($ws_i$) is after the beginning of the window ($ws_i \geq wt$) and the ending time ($we_i$) is before the end of the window ($we_i \leq wt + wd$). Figure 2.3 displays how to calculate the total waiting time in this window. The following is a

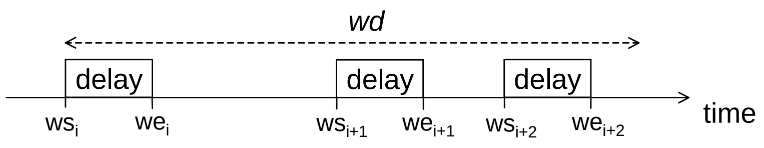


Figure 2.3: calculate total waiting time in an observation window of size *wd*

formula for calculating the total delay starting at *wt* with window size *wd*:

$$td(wt, wd) = \sum_{\substack{i \text{ such that} \\ ws_i \geq wt \\ we_i \leq wt + wd}} (we_i - ws_i) \tag{2.7}$$

We want to find the worst-case total waiting time by adjusting the beginning of this window:

$$d = \max_{wt} \; td(wt, wd) = \max_{wt} \sum_{\substack{i \text{ such that} \\ ws_i \geq wt \\ we_i \leq wt + wd}} (we_i - ws_i) \tag{2.8}$$

A desirable policy has a short worst-case waiting time, i.e. small value of $d$.

### Frequency of Delays

The second metric quantifies the number of waiting periods a user remembers. We assume the user has memory duration $ml$. If one delay occurs more than $ml$ before the next delay, the user forgets the first. However, if the time between two consecutive delays is shorter than $ml$, the user will remember both. In Figure 2.4, the user forgets the $(i-1)^{th}$ delay when the $i^{th}$ occurs. Since the following two delays are close, the user will perceive three consecutive waiting periods. The user remembers a sequence

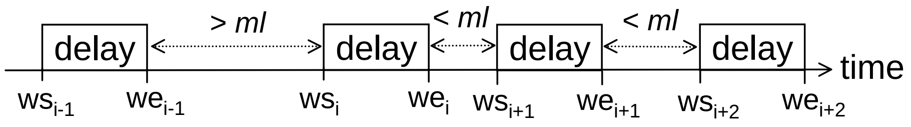


Figure 2.4: calculate the length of a delay sequence

of waiting periods if the sequence satisfies three conditions:

- The time between the first delay and its previous delay is longer than $ml$.
- The time between two consecutive delays in this sequence is shorter than $ml$.
- The time between the last delay and its next delay is longer than $ml$.

The user perceives a sequence of $nw + 1$ delays if there is a sequence formed by the $i^{th}$ to the $(i + nw)^{th}$ delays that meet the three conditions:

$$\begin{array}{lll} ws_i & - & we_{i-1} > ml \\ ws_k & - & we_{k-1} \leq ml, \qquad \forall k \in [i+1, i+nw] \\ ws_{i+nw+1} & - & we_{i+nw} > ml \end{array} \tag{2.9}$$

We are interested in finding the worst-case performance degradation; i.e. the longest delay sequence. It is represented by the largest value of $nw$:

$$s(ml) = \max_i \quad nw \tag{2.10}$$

Its value is affected by users' memory duration, $ml$. A good policy should have smaller $s(ml)$ for a reasonable range of $ml$. For human being, $ml$ can be up to about one hour. Any delay that occurs more than one hour ago is likely to be tolerated and "forgiven" by users.

## 2.3 Dynamic Voltage / Frequency Scaling

Many processors support multiple active states differentiated by their frequencies. For these processors, performance is proportional to power. CMOS-based processors consume power primarily during switching: from logic true to false or vice versa. The dynamic power of a CMOS gate can be approximated by

$$p = c \cdot {v_{dd}}^2 \cdot sw \cdot f \tag{2.11}$$

here $c$ is the load capacitance, $v_{dd}$ is the supply voltage, $sw$ is the switching activity, and $f$ is the clock frequency [83]. The energy consumption during $[0, T)$ is

$$e = \int_0^T p dt \tag{2.12}$$

If we replace the load capacitance and the switching activity by their averages, the energy is proportional to

$$e \propto \int_0^T {v_{dd}}^2 \cdot f dt \tag{2.13}$$

Power can be reduced by lowering $v_{dd}$ and/or $f$; this is called *voltage scaling* and *frequency scaling* [16] [29] [67]. The delay of a gate, as first-order approximation, is inversely proportional to $v_{dd}$. Clock frequencies are determined by gate delays; thus, frequencies are affected by the supply voltage. When a circuit operates at a lower frequency, the voltage can be reduced accordingly. Voltage is a monotonically increasing function of frequencies. Let $v(f)$ be the supply voltage at frequency $f$. If frequencies $f_1 > f_2$, then $v(f_1) \geq v(f_2)$. Frequencies and voltages are always positive. The following theorem proves that minimizing frequencies is equivalent to minimizing total energy.

**Theorem 1** *Suppose function $f(t)$ is the frequency at time $t$ over an interval $[0, T]$; $f(t)$ may vary within a finite range. If $f(\bullet)$ minimizes*

$$\int_0^T f(t)dt \tag{2.14}$$

*then, the pair $f(\bullet)$ and $v(f(\bullet))$ will minimize the total energy $e$, where*

$$e \propto \int_0^T v(f(t))^2 \cdot f(t)dt \tag{2.15}$$

*Proof*

We prove it by contradiction. Let $f^*(\bullet)$ be a frequency assignment that minimizes $\int_0^T f(t)dt$. Suppose $\hat{f}(\bullet)$ is another frequency assignment over the interval $[0, T]$. We assume the following is true:

$$\begin{aligned} &\textstyle\int_0^T f^*(t)dt < \int_0^T \hat{f}(t)dt \qquad \text{and} \\ &\textstyle\int_0^T v(f^*(t))^2 f^*(t)dt > \int_0^T v(\hat{f}(t))^2 \hat{f}(t)dt \end{aligned} \tag{2.16}$$

Namely, $f^*(t)$ can cause more energy consumption even though $\int_0^T f^*(t)dt$ is smaller. In order to satisfy both inequalities in (2.16), there must be $\tau \in [0, T]$ such that

$$f^*(\tau) < \hat{f}(\tau) \qquad \text{and} \qquad v(f^*(\tau))^2 f^*(\tau) > v(\hat{f}(\tau))^2 \hat{f}(\tau) \tag{2.17}$$

The second inequality can be written as

$$v(f^*(\tau))^2 f^*(\tau) - v(f^*(\tau))^2 \hat{f}(\tau) > v(\hat{f}(\tau))^2 \hat{f}(\tau) - v(f^*(\tau))^2 \hat{f}(\tau) \tag{2.18}$$

This is equivalent to

$$v(f^*(\tau))^2[f^*(\tau) - \hat{f}(\tau)] > \hat{f}(\tau)[v(\hat{f}(\tau))^2 - v(f^*(\tau))^2] \tag{2.19}$$

Since $f^*(\tau) < \hat{f}(\tau)$ and $v(f)$ is a monotonically increasing function, $v(f^*(\tau))^2 \leq v(\hat{f}(\tau))^2$. The left hand side (LHS) of the above inequality is negative while the right hand side (RHS) is positive or zero. It is impossible that LHS is larger than RHS. Therefore, the assumption is false; we cannot find $f^*(t)$ and $\hat{f}(t)$ such that both inequalities in (2.16) hold. In other words, it is impossible to minimize energy consumption without minimizing the integration of frequencies. $\diamondsuit$

If frequencies change only at time $t_1$, $t_2$, ..., $t_n$, the frequency during $[t_i, t_i + 1)$ is $f_i$, $i \in \{1, 2, ..., n-1\}$. We can rewrite (2.14) as

$$\int_0^T f(t)dt = \sum_{i=1}^{n-1} f_i(t_{i+1} - t_i) \tag{2.20}$$

Thus, selecting $f_i$ to minimize the following function minimizes the total energy.

$$\sum_{i=1}^{n-1} f_i(t_{i+1} - t_i) \tag{2.21}$$

Frequency scaling has two effects on performance. First, the processor has a slower clock rate, hence lower performance; second, it takes time to change frequencies. In [36], the authors report 200 microseconds for changing frequencies on a StrongARM processor. Even though this delay is too small to perceive by human users, it is more than ten thousand clock cycles and cannot be ignored in some cases. Some commercial processors provide software interfaces for dynamic frequency scaling. For example, StrongARM processors have special registers to indicate the current clock frequencies [2]. Modifying the values in these registers changes the frequencies. StrongARM SA1110 has eleven frequencies available, between 59 MHz and 206 MHz.

## 2.4 Chapter Summary

This chapter explains the criteria to design power-management policies. A desirable policy shuts down a device only when it can sleep long enough to justify the overhead in state-transition energy and delay. We also prove that, in CMOS circuits, choosing the lowest possible frequencies over time (according to the performance constraints) corresponds to minimizing total energy used over the same time interval.

# Chapter 3

# Implementing Policies

This chapter presents a framework to implement and compare policies. Most power-management policies were evaluated by simulations only; many policies have never been implemented on realistic systems. There are potential drawbacks in simulations. First, simulations often use simplified models of hardware devices and ignore details. In [25] [31] [49] [52] [68], simplified hard disk models are used to evaluate policies. Second, simulations often ignore the overhead of power managers even though some policies require substantial computation like traversing trees [20]. Third, simulations do not support direct interaction with users; hence, it is impossible to evaluate performance degradation perceived by users. This chapter explains how to implement policies as *filter drivers* in Microsoft Windows; experimental data by measurement will be presented to compare different policies. The experimental setup is described in Appendix A.

## 3.1 Power Management and System Layers

Before a policy is implemented, several issues have to be clarified: (1) where to implement power management (2) what information is available for power managers (3) how much overhead is added.

A complex system is usually structured in *layers* such as the one shown in Figure 3.1. Hardware is at the bottom of the layers and operating systems lie between

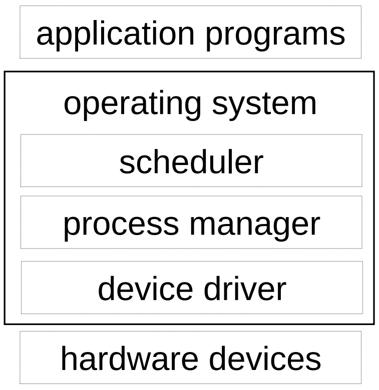


Figure 3.1: Power managers can be implemented at different layers in a system.

hardware and application programs. Each layer provides unique advantages and disadvantages for power management. A power manager that is implemented at a lower layer has less information about application programs and it provides less flexibility. On the other hand, a higher layer may incur unnecessary overhead; for example, if a power manager is implemented in the scheduler, then all processes are affected regardless whether they generate requests.

### 3.1.1 Power Management by Hardware

Power is consumed by hardware; thus, implementing policies directly in hardware is an intuitive approach. This has two advantages: independent of software and small overhead. On the other hand, hardware has limited information: it can detect only the arrival of requests. Due to the lack of higher level information, it is difficult for hardware-implemented power managers to predict the lengths of idle periods accurately. Furthermore, it is sometimes objectionable to implement "policies" in hardware. Instead, a lower layer should provide "mechanism" so that higher layers can determine the policies [72].

If a policy is directly implemented in hardware, there is no flexibility to adjust policies by software for different requirements. For example, a hard disk may be installed in a mission-critical data center where performance is much more important than power saving. The same disk can also be install on a home computer whose user wants to balance performance and power saving. Because hardware should provide mechanism only, it is preferred to implemented policies in software.

### 3.1.2 Power Management by Applications

Even in software, power management can be implemented in different layers. Application programs generate requests; therefore, they have the best knowledge about when requests are generated. Microsoft's OnNow allows application programs to set hardware power states [61]. This approach also has limitations. First, different applications may set the same device to different power states. Second, it is complicated to write application programs that determine power states. Third, it is impractical to rewrite all legacy programs and add power management features.

### 3.1.3 Power Management by Operating Systems

We think power managers should be implemented in operating systems to conquer the problems mentioned above. Operating systems have the flexibility to adjust policies for different environments and serve requests from all application programs. Operating systems can be structured into different layers, including device drivers, process managers, schedulers, and so on. Each layer corresponds to a requester model explained in Section 1.3. Device drivers can detect request arrival; this assumes the single-requester model. Process managers consider multiple requesters with their creation and termination. The information about process states and priorities improve prediction accuracy. Moreover, process schedulers determine the time when a process executes and when they generate requests. Schedulers affect the arrival of requests and thus the length of idle periods. Power management by process managers and schedulers will be discussed in more details in the next two chapters. This chapter concentrates on power management by device drivers.

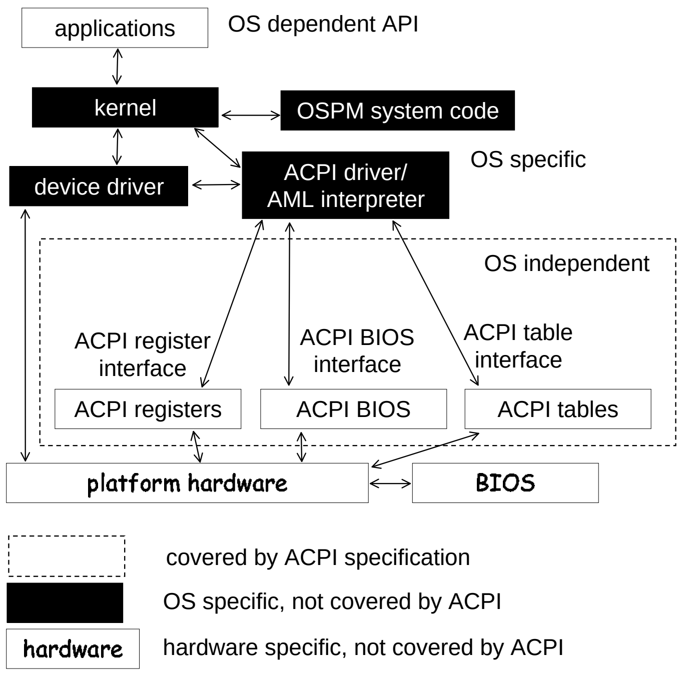


Figure 3.2: ACPI structure

### 3.1.4 Advanced Configuration and Power Interface

Using software to control power states requires collaboration with the hardware. In particular, hardware has to support programming interfaces and allow software to change the power states. Two standards are widely used for power management: *advanced power management* (APM) [1] [6] and *advanced configuration and power interface* (ACPI) [3].

ACPI is an open specification of the interface between hardware and software for power management. Its sponsors include major computer vendors, both hardware and software, such as Intel, Microsoft, Compaq, and Toshiba. Before ACPI was developed, APM was widely used. APM was mostly implemented in *basic input-output systems* (BIOS); it was unable to adapt for fast changing hardware devices, such as plug-and-play features. ACPI was proposed after APM and was intended to replace APM.

ACPI enables operating-system directed configuration and power management

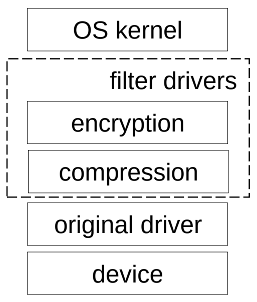


Figure 3.3: Filter drivers encrypts and compresses data.

(OSPM). Migrating power management from BIOS to software has multiple advantages: (1) it can implement advanced and sophisticated policies (2) it is easier to upgrade for better policies (3) it has more information about application programs (4) it avoids conflicts from different device vendors. ACPI requires collaboration between hardware and software to manage power. Figure 3.2 shows the structure of ACPI. Hardware resides at the bottom of this figure; above the hardware, there is a layer of ACPI registers and BIOS as the interface to operating systems. At the top, there are applications that generate requests. Microsoft's *OnNow* [61] provides an application programming interface (API) to manager power on ACPI-compliant systems. ACPI has been recently ported to Linux kernel V2.4 [4].

## 3.2 Implementing Policies in Device Drivers

ACPI does not specify where to implement power management. Many policies use the single-requester model and need to distinguish only busy and idle periods. Such information is available at device drivers; consequently, these policies can be implemented in device drivers. Device drivers are structured as layers so that new functionality can be easily added; this is similar to network protocol layers. In Windows, a *filter driver* can enhance or add features to the original device driver; for example, a filter driver can compress or encrypt all data sent to the device [76]. Figure 3.3 depicts the structure of these filter drivers. Using filter drivers for system-level power management was first proposed in [54].

When a filter driver is loaded into Windows, it first creates a driver object by

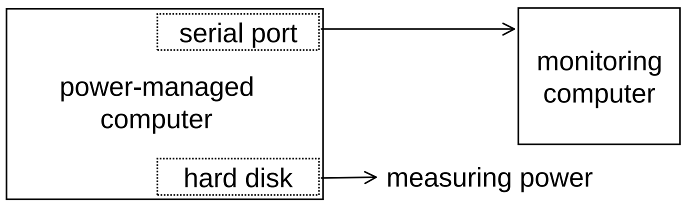


Figure 3.4: connect a monitoring computer through the serial port

calling `IoCreateDevice`; `IoCreateDevice` is a function provided by Windows. Then, it calls `IoAttachDeviceToDeviceStack` to attach itself on the top of the original driver. The communication between the original driver and OS kernel is intercepted by the filter driver. The following assignment will intercept read commands from Windows kernel

```
DriverObject->MajorFunction[IRP_MJ_READ]  = FilterRead;
```

`DriverObject->MajorFunction[IRP_MJ_READ]` is a function pointer; `FilterRead` is a function to handle the read commands from kernel. A typical structure of `FilterRead` is to perform necessary processing, such as compression, and then pass the command to the lower-layer driver. If the filter driver implements a policy, power manager can calculate the length of the previous idle period when `FilterRead` is invoked.

Communication between a driver and OS kernel is performed by creating an IO request packet (IRP). The filter receives an IRP, performs necessary operations, and then passes it to the original driver by calling `IoCallDriver`. The filter driver can also create a new IRP by calling `IoAllocateIrp` and pass it to the lower driver. In Windows, power management commands require special IRP's; hence, they have to be created by calling `PoRequestPowerIrp`. This function specifies the device, the new power state, and a callback function that is invoked after this power IRP is processed. In Windows, device drivers can "print" through a serial port; Figure 3.4 shows such a connection. Using a monitoring computer, we can record information without writing it to the hard disk on the power-managed computer. This method reduces the interference of power management on the original workloads.

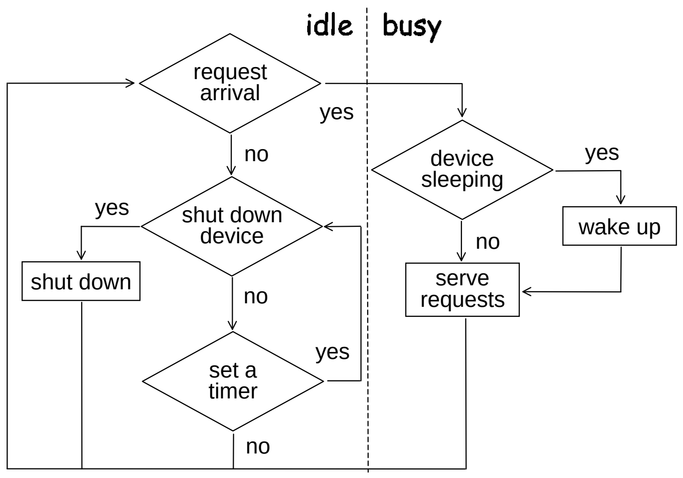


Figure 3.5: flow of a filter driver for power management

Figure 3.5 shows the flow of a typical filter driver that implements power management. When the device is idle, it stays on the the left side of the figure. Some policies reevaluate shutdown decisions periodically and require timers. Request arrival is an event issued by the kernel; hence, the filter driver is invoked only when requests arrive or triggered by the timer. In Windows, users can specify the timeout values for hard disks and monitors; the minimum value is one minute. When a policy is implemented as a filter driver, this limitation does not exist. A power manager can shut down a device immediately after it becomes idle.

## 3.3 System-Level Power Management Policies

This section explains the policies compared later in this chapter. General comparisons of system-level power management policies are available in [11] [53].

### 3.3.1 Oracle Power Manager

Before comparing policies, it is important to understand the baseline. Power management is a prediction problem; a perfect power manager knows exactly when requests arrive. Such a manager is called an *oracle* power manager. This manager shuts down

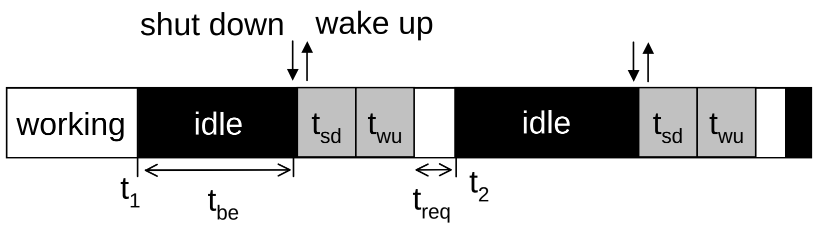


Figure 3.6: worst case power consumption for 2-competitive policy

a device when an idle period is longer than the break-even time of this device; it keeps the device in the working state during short idle periods. This is the best power saving achievable by power management. In reality, such an oracle does not exist; it can be simulated *off-line* by analyzing a trace that records requests.

## 3.3.2 Competitive Timeout Policy

Suppose a device consumes average power $p$ when it is managed by an oracle manager. A "c-competitive" policy causes at most $c \cdot p$ average power, here $c > 1$. Competitive policies make shutdown decisions at run-time without knowing when future requests will arrive; in other words, competitive policies are "on-line".

**Theorem 2** *Assume that shutdown and wakeup commands are atomic: once a shutdown command is issued, the device has to take $t_{sd}$ time (defined in Table 2.1) and consume $e_{sd}$ energy even if it receives a wakeup command immediately. If a timeout policy sets the timeout value to the break-even time of a device, it can achieve 2-competitiveness*[1]*.*

*Proof*

Consider a timeout policy whose timeout value is the break-even time of the device being managed. Figure 3.6 depicts an "adverse" scenario that causes the most power consumption. In this figure, white blocks indicates that the device is busy; black blocks indicate that the device is idle. Gray blocks show when the device changes power states. This policy shuts down the device after it is idle for $t_{be}$, the break-even

[1]This result was shown in [45] for a different setting. We adjust the proof in the context of power management.

time of the device. The worst case occurs if requests arrive right after the device is shut down. Namely, the lengths of the idle periods are $t_{be} + \epsilon$ where $\epsilon$ is a very short duration. The device cannot save energy because it does not have a chance to stay in the low-power sleeping state. Moreover, it has to consume the state-transition energy.

An oracle policy has two choices: either keeps the device in the working state, or shuts down the device immediately after it becomes idle. Because the length of the idle period is very close to the break-even time of the device, the two choices will "break-even" and have the same power consumption. In both cases the average power consumption is $p_w$.

Since the timeout policy cannot foresee future requests, it shuts down the device after the device has been idle for $t_{be}$. However, the device has to wake up immediately to serve requests. Because the device is kept in the working state during idleness, its energy consumption is $p_w \times t_{be}$. It also consumes state-transition energy: $e_{sd} + e_{wu}$ during $t_{sd} + t_{wu}$. Let $t_{req}$ be the time to serve requests before the device becomes idle again. The energy between $t_1$ and $t_2$ is

$$e = p_w \times (t_{be} + \epsilon) + e_{sd} + e_{wu} + p_w \times t_{req} \tag{3.1}$$

Remember that $e_o$ and $t_o$ are the transition energy and time overhead: $e_o = e_{sd} + e_{wu}$ and $t_o = t_{sd} + t_{wu}$. The average energy is

$$\frac{e}{t_{be} + \epsilon + t_{sd} + t_{wu} + t_{req}} = \frac{p_w \times (t_{be} + \epsilon) + e_o + p_w \times t_{req}}{t_{be} + \epsilon + t_o + t_{req}} \tag{3.2}$$

Suppose $\epsilon$ and $t_{req}$ are very small. The average energy can be simplified to

$$\frac{p_w \times t_{be} + e_o}{t_{be} + t_o} \tag{3.3}$$

According to (2.2), we can replace $t_{be}$ by $\frac{e_o - p_s \times t_o}{p_w - p_s}$ and compute the energy during $[t_1, t_2]$:

$$\frac{2p_we_o - p_wp_st_o - e_op_s}{e_o + p_wt_o - 2p_st_o} \tag{3.4}$$

Since $p_w > p_s \geq 0$, we can find an upper bound of the numerator

$$\begin{aligned} & 2p_we_o - p_wp_st_o - e_op_s \\ & < 2p_we_o - p_wp_st_o \\ & < 2p_we_o - 2p_wp_st_o \\ & = 2p_w(e_o - p_st_o) \end{aligned} \tag{3.5}$$

Similarly, we can find a lower bound for the denominator:

$$\begin{aligned} & e_o + p_wt_o - 2p_st_o \\ & = e_o - p_st_o + (p_w - p_s)t_o \\ & > e_o - p_st_o \end{aligned} \tag{3.6}$$

Now, we can find an upper bound of the average power:

$$\frac{2p_we_o - p_wp_st_o - e_op_s}{e_o + p_wt_o - 2p_st_o} < \frac{2p_w(e_o - p_st_o)}{e_o - p_st_o} = 2p_w \tag{3.7}$$

The power is less than $2p_w$ in the worst case. This is less than twice of the power consumption if the device is managed by the oracle manager. Consequently, this timeout policy achieves 2-competitiveness by setting the timeout value to the break-even time of the managed device. $\diamondsuit$

### 3.3.3 Adaptive Timeout Policies

These policies adjusts timeout values dynamically. Let's call the value $\tau$. The policy presented in [25] considers the length of the previous idle period. If it is short, $\tau$ increases; otherwise, $\tau$ decreases. In our comparison, the parameters were set to

$(\alpha_m, \beta_m, \rho) = (1.5, 0.5, 0.1)$ [2] and $\tau$ was thirty seconds initially.

Another approach considers the length of a busy period instead [52]. If a busy period is short, $\tau$ decreases; otherwise, it increases. The initial value of $\tau$ was two minutes and the sampling rate was one Hz with two seconds for the adjustment factor.

The third policy updates $\tau$ asymmetrically: increasing $\tau$ by one second or decreasing it by half [31]. Our experiments limited $\tau$ of this policy between one second and two minutes.

### 3.3.4 Exponential Average Predictive

In [41], the authors observe that the length of a future idle period can be accurately predicted by the length of the previous idle period and the prediction of this period. Mathematically, let $t_{predicted}[i]$ and $t_{actual}[i]$ be the predicted and the actual lengths of the $i^{th}$ idle period. The length of the $(i+1)^{th}$ idle period can be approximated by

$$t_{predicted}[i+1] = a \cdot t_{actual}[i] + (1-a) \cdot t_{predicted}[i] \qquad 0 \le a \le 1 \tag{3.8}$$

This is "discounted average" because the effect of the most recent idle period is discounted by factor $a$ while the previous prediction is discounted by $1-a$. Since $t_{predicted}[i]$ is calculated in the same way, we can expand the equation as follows:

$$\begin{aligned} t_{predicted}[i+1] &= a \cdot t_{actual}[i] + (1-a) \cdot t_{predicted}[i] \\ &= a \cdot t_{actual}[i] + (1-a) \cdot (a \cdot t_{actual}[i-1] + (1-a) \cdot t_{predicted}[i-1]) \\ &= a \cdot t_{actual}[i] + (1-a)a \cdot t_{actual}[i-1] + (1-a)^2 \cdot t_{predicted}[i-1] \\ &\dots \\ &= (1-a)^{i+1} t_{predicted}[0] + \sum_{k=0}^{i} a(1-a)^k t_{actual}[i-k] \end{aligned} \tag{3.9}$$

This is an average of previous idle periods with exponential weights; hence, it is

[2]Please refer to the paper for the definitions of these parameters.

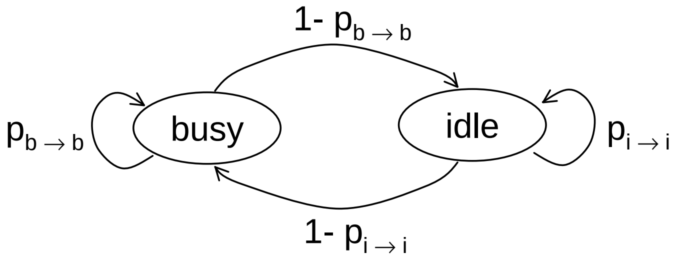


Figure 3.7: basic stochastic model

also called "exponential average". If $t_{predicted}$ is larger than the break-even time, the device is shut down. This policy restrains $t_{predicted}[i+1]$ such that it cannot exceed $c \cdot t_{idle}[i]$ where $c$ is a constant greater than one. In our experiments, we use 0.5 for $a$ and 2 for $c$ as suggested in [41]. All idle periods shorter than one tenth second are ignored so that $t_{predicted}$ is not affected by these very short idle periods. They are discarded at run-time because $t_{acutal}[i]$ is known before computing $t_{predicted}[i+1]$.

### 3.3.5 Learning Tree

Adaptive learning trees (LT) transform sequences of idle periods into discrete events and store them into tree nodes [20]. This algorithm predicts idle periods using finite-state machines similar to branch prediction used in microprocessors and selects a path which resembles previous idle periods. At the beginning of an idle period, it determines an appropriate sleeping state; this algorithm is capable of controlling multiple sleeping states.

### 3.3.6 Stationary Discrete-Time Stochastic Policy

In [62], the authors suggest using stationary stochastic processes to model request arrival and power-state transitions. This discrete-time policy slices time into discrete units: $t$, $2t$, $3t$ ... A simple stationary model of a requester is illustrated in Figure 3.7. In this figure, if the device is busy at $nt$, there is $p_{b\to b}$ probability that the device will remain busy at $(n+1)t$; the device will become idle at $(n+1)t$ with probability $1-p_{b\to b}$. Similarly, the power-manageable device has $p_{i\to i}$ probability to remain idle

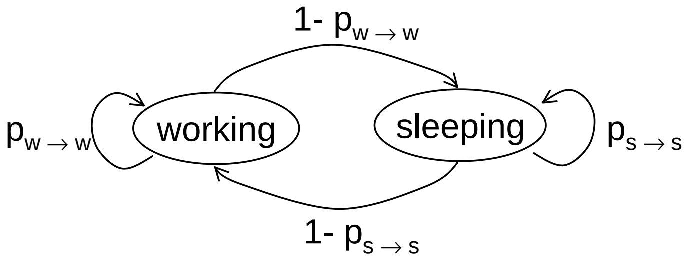


Figure 3.8: state transition probabilities of a device

if it is originally idle. This policy uses a stationary model by assuming that $p_{b \to b}$ and $p_{i \to i}$ are constants. The transitions of power states are also modeled as a stochastic process, like the example shown in Figure 3.8. When the device is in the working state at $nt$, it will remain in the working state at $(n+1)t$ with probability $p_{w \to w}$; it has $1 - p_{w \to w}$ probability to enter the sleeping state. If the device is originally sleeping, it has probability $p_{s \to s}$ to remain sleeping.

We can compute the optimal state-transition probabilities, $p_{w \to w}$ and $p_{s \to s}$, based on the values of $p_{b \to b}$ and $p_{i \to i}$ and the device parameters, including $p_w$, $p_s$, and $t_{be}$. This approach differs from previous policies into two major aspects. First, it builds mathematical models for request generation and power-state transitions; they are represented by stochastic processes. Second, the state-transition probabilities are optimal solutions, not heuristics. One disadvantage of this approach is the need of priori knowledge about request characteristics. This policy has to know $p_{b \to b}$ and $p_{i \to i}$ in advance for computing $p_{w \to w}$ and $p_{s \to s}$.

### 3.3.7 Non-Stationary Discrete-Time Stochastic Policy

The previous policy has one major shortcoming: it cannot adjust $p_{w \to w}$ and $p_{s \to s}$ if the probability of request arrival changes. This was remedied in [19]. The policy first performs off-line optimization for a set of arrival probabilities. For each arrival probability, there is a corresponding shutdown probability. At run-time, the policy maintains a sliding window that encloses recent requests; these requests are used to predict the future arrival rate. There are two possible scenarios. First, if this probability has been optimized off-line, the power manager uses the pre-computed

| policy | features | target devices |
|---|---|---|
| [45] (CA) | competitive | spin-block |
| [25] (ATO1) | adaptive timeout | hard disk |
| [31] (ATO2) | adaptive timeout | hard disk |
| [52] (ATO3) | adaptive timeout | hard disk |
| [41] (EA) | exponential average | telnet |
| [20] (LT) | learning tree | hard disk |
| [62] (DM) | discrete-time Markov | hard disk |
| [19] (NS) | non-stationary stochastic | hard disk |
| [78] (SM) | time-index semi-Markov | hard disk |

Table 3.1: summary of policies compared

shutdown probability. Second, if this probability does not occur in the off-line optimization, the policy linearly interpolates pre-computed shutdown probabilities to obtain the shutdown probability for the current arrival rate.

### 3.3.8 Continuous-Time Stochastic Policies

Discrete-time stochastic policies have one common drawback: they have to periodically reevaluate whether to shut down or wake up devices even when the devices are sleeping and there are no requests. They create timers in the control flow of filter driver in Figure 3.5. To eliminate such unnecessary computation, continuous-time stochastic models are used [66] [79] [80]. This model was later extended as a time-indexed semi-Markov model [78]. It uses two types of probability distributions: (1) Pareto distributions to model the probability that a device becomes busy from idleness: the probability $1 - p_{i \to i}$ (2) exponential distributions for the device to remain busy ($p_{b \to b}$). In this policy, if the power manager initially decides to keep a device in the working state and the idle period is actually long, the algorithm reevaluates the decision so that the power manager can still shut down the device. If the device is already sleeping, no extra computation is needed. Therefore, this algorithm saves power and has low computation overhead.

## 3.4 Policy Comparison

We use a desktop and a laptop computers to conduct our experiments. On the desktop computer, we connect current meters to the power cores (12V and 5V) of a 3.5" hard disk. The setup of the laptop computer is explained with details in Appendix A.1; the same environment is also used for Chapter 4 and Chapter 5.

### 3.4.1 Workload Generation

We did not use performance benchmarks such as SPEC [74] because they measure peak performance and no device should sleep. Instead, a filter driver was used to collect user traces as the workloads. It recorded disk access traces of two users by using the serial-link connection shown in Figure 3.4. The users were developing C programs and making presentation slides. If the time between two requests was less than one millisecond, they were considered as a single long request. Each request was recorded by its time and duration. These traces include disk accesses from user requests and operating system activities. Then the traces were replayed; they took approximately eleven hours.

### 3.4.2 Computation for Power Managers

The amount of computation was quantified by recording the time spent in power managers in the following way:

```
PowerManager
{
    /* before doing anything */
    find the starting time;
    /* perform necessary operations */
    find the ending time;
    this time in power manager = ending time - starting time;
    total time in power manager += this time in power manager;
}
```

We used the performance counter provided by the processor; it could measure time at a high precision (less than one tenth of a microsecond). Our analyses show that all policies spend less than 1% of processor time in power managers; hence, the computation overhead of power managers is not a concern for personal computers, either desktops or laptops.

### 3.4.3 Power and Performance

This section presents the average power of each policy. The performance measures the total number of shutdowns. Interactive performance will be presented in the next section. Device parameters were summarized in Table 2.1. Five terms are used to compare policies:

- power consumption ($p$), unit: Watt. In this comparison, we ignore the power to serve requests; instead, we consider only the power in the working state and the sleeping state. When the power of a device is larger than its working state power, $p_w$, we replace it with $p_w$. This is because power management does not affect the power while serving requests; power management reduces power only when a device is idle.
- number of shutdowns ($n_{sd}$).
- average time in sleeping state for each shutdown ($t_{ss}$), unit: second.
- average time before shutdown ($t_{bs}$), unit: second
- number of wrong shutdowns ($n_{wd}$).

If the device does not sleep longer than its corresponding minimum-sleeping time ($t_{ms}$), the shutdown is defined as a wrong shutdown. Wrong shutdowns actually waste energy. The time before shutdown is defined as the time when a device is idle and before a shutdown command is issued by the power manager. The device wastes power in this duration. The time before shutdown ($t_{bs}$) is the duration when the device is idle but remains in the working state; the device wastes power during this

| policy | $p$ | $n_{sd}$ | $n_{wd}$ | $t_{ss}$ | $t_{bs}$ |
|---|---|---|---|---|---|
| off-line | 1.64 | 164 | 0 | 166 | 0 |
| SM | 1.92 | 156 | 25 | 147 | 18.2 |
| CA | 1.94 | 160 | 15 | 142 | 17.6 |
| NS | 1.97 | 168 | 26 | 134 | 18.7 |
| timeout 30 seconds | 2.05 | 147 | 18 | 142 | 30.0 |
| LT | 2.07 | 379 | 232 | 62 | 5.7 |
| ATO3 | 2.09 | 147 | 26 | 138 | 29.9 |
| ATO1 | 2.19 | 141 | 37 | 135 | 27.6 |
| ATO2 | 2.22 | 595 | 430 | 41 | 4.1 |
| timeout 120 seconds | 2.52 | 55 | 3 | 238 | 120.0 |
| DM | 2.60 | 105 | 39 | 130 | 48.9 |
| EA | 2.99 | 595 | 503 | 30 | 7.6 |
| always-on | 3.48 | - | - | - | - |

Table 3.2: compare policies for 3.5" disk

period. For a timeout policy, $t_{bs}$ equals the timeout value. Among the five quantities, it is preferred to have larger $t_{ss}$ and smaller values for the other four quantities.

Tables 3.2 and 3.3 compare these policies. The first row contains the minimum power consumption without performance degradation; it is generated off-line with full knowledge about future requests (oracle). The last row shows the power consumption if no power management is applied. This table shows that policies SM, CA and NS can save nearly 50% of power on both platforms. Even though they have close power consumption on the mobile disk, they differ significantly in performance. CA and SM have more than twice wrong shutdowns ($n_{wd}$) compared with NS. For policies with similar power consumption, performance is an important factor for evaluation. Figure 3.9 compares the power, number of shutdowns, and percentages of wrong shutdowns of the 2.5" disk. The power and number of shutdowns are normalized relative to the oracle power manager.

### 3.4.4 Comparing Interactive Performance

For a policy, the total waiting time is proportional to the total number of shutdowns ($n_{sd}$). However, even for two policies with similar values of $n_{sd}$, a user may notice

| policy | $p$ | $n_{sd}$ | $n_{wd}$ | $t_{ss}$ | $t_{bs}$ |
|---|---|---|---|---|---|
| off-line | 0.33 | 250 | 0 | 118 | 0 |
| SM | 0.40 | 326 | 76 | 81 | 8.0 |
| NS | 0.43 | 191 | 28 | 127 | 13.4 |
| CA | 0.44 | 323 | 64 | 79 | 5.4 |
| LT | 0.46 | 437 | 217 | 56 | 6.1 |
| ATO1 | 0.47 | 273 | 73 | 88 | 12.4 |
| EA | 0.50 | 623 | 427 | 37 | 3.0 |
| timeout 30 seconds | 0.51 | 139 | 7 | 157 | 30.0 |
| ATO3 | 0.52 | 196 | 48 | 109 | 24.5 |
| DM | 0.62 | 173 | 54 | 102 | 35.2 |
| ATO2 | 0.64 | 881 | 644 | 19 | 2.3 |
| timeout 120 seconds | 0.67 | 55 | 0 | 255 | 120.0 |
| always-on | 0.95 | - | - | - | - |

Table 3.3: compare policies for 2.5” disk

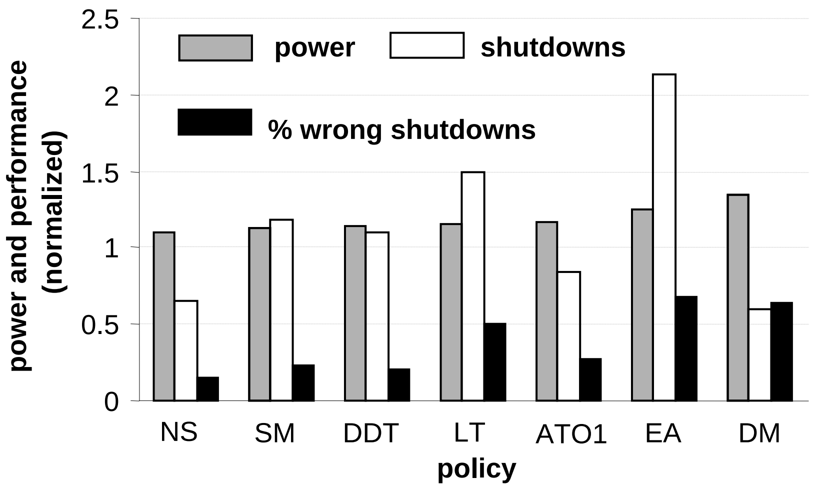


Figure 3.9: normalized comparison of power and performance

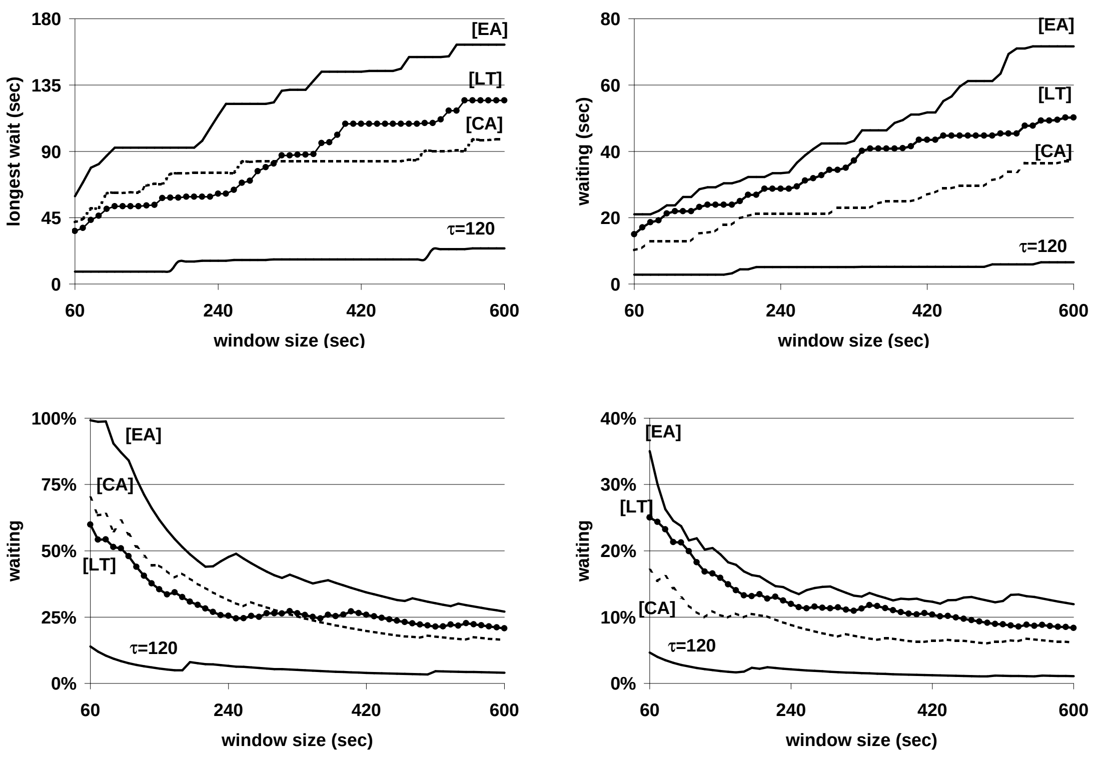


Figure 3.10: worst-case waiting time on 3.5" (left) and 2.5" (right) disks

substantial difference in their performance. Some polices make delays concentrated within short time periods. Such periods are frustrating to users.

Figure 3.10 draws the worst-case waiting time, defined in (2.8), for $wd$ between one to ten minutes. It shows that, in the worst case, CA requires users to wait for 98 seconds in a 10-minute duration on the desktop hard disk. The bottom of the figure is the waiting time by percentage. When the window size increases the percentage of waiting time decreases for all algorithms. When the window size is small, such as one minute, some algorithms may require users to wait for more than 50% of the time. This demonstrates the importance to measure the worst-case performance for small $wd$. Traditional performance metrics using the total waiting time cannot provide enough information for determining user perception of performance degradation. Figure 3.10 has several "jumps" as $wd$ increases because the worst-case waiting time

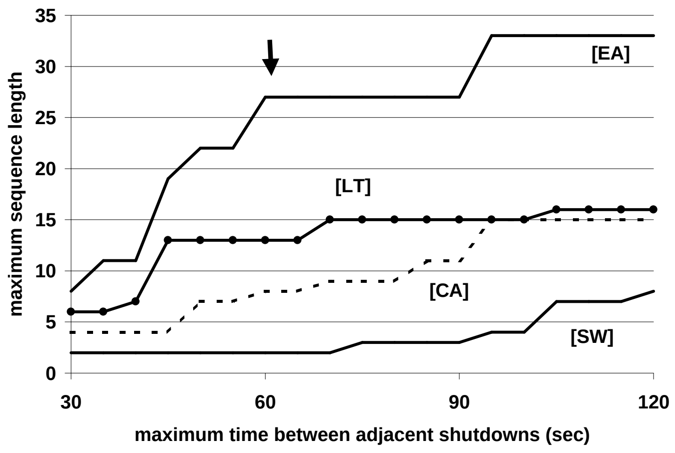


Figure 3.11: maximum length of shutdown sequences

may change from one window to another. This figure also shows that the waiting time is considerably shorter on the 2.5" hard disk.

Figure 3.11 plots the longest shutdown sequence defined in (2.9). In this figure, the horizontal axis is the threshold value and the vertical axis is the lengths of sequences. The arrow indicates that EA has a sequence of 27 waiting periods with less than one minute between two shutdowns. Users perceive delays every minute or even more frequently for 27 times.

## 3.4.5 State Transition Delay

For a hard disk, the wakeup delay is not a constant; instead, it distributes widely. Figure 3.12 are histograms of the distributions of wakeup delays. For the 3.5" disk, the average is 6.97 seconds and the standard deviation is 0.65 second or 9.25%. For the 2.5" disk, it is even more widely distributed. Such variation may be attributed to the arm positions when a wakeup command is issued. The measurement results show that transition delays are not exponentially distributed as modeled by some stochastic policies.

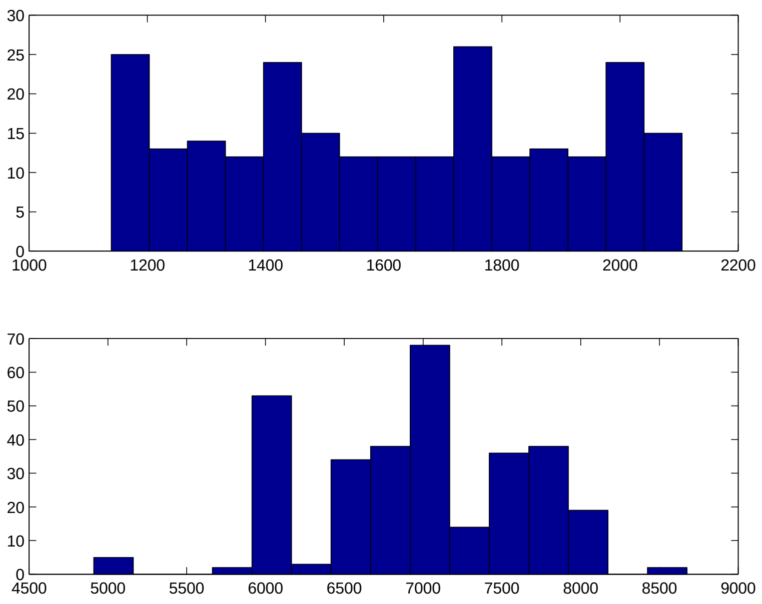


Figure 3.12: wakeup delay (millisecond) for 2.5" (top) and 3.5" disks

### 3.4.6 Learning Period of the Exponential Average Policy

The exponential average policy was not designed for a device with long break-even time; it is understandable that it does not perform well in our experiments. The policy makes several wrong shutdowns within a short time period after a long idle period. This policy predicts the length of an idle period based on the actual length of the previous idle period. If the previous idle period is exceptionally long, the predicted length is also long. Suppose $t_{actual}[i]$ is very large, then $t_{predicted}[i+1]$ is large because $t_{predicted}[i+1] = a \cdot t_{actual}[i] + (1-a) \cdot t_{predicted}[i]$. Even if $t_{actual}[i+1]$ is small, $t_{predicted}[i+2]$ is still large because $t_{predicted}[i+2] = a \cdot t_{actual}[i+1] + (1-a) \cdot t_{predicted}[i+1] \approx (1-a) \cdot t_{predicted}[i+1]$.

Let $a$ be 0.5, $t_{actual}[i]$ be ten minutes, and the break-even time be ten seconds; $t_{predicted}[i+1]$ is at least five minutes. Since $t_{predicted}[i+1]$ is larger than the break-even time, the power manager shuts down the device. Suppose $t_{actual}[i+1]$ is very short. The predicted length for the next idle period, $t_{predicted}[i+2]$, is two and half minutes and still larger than the break-even time. The device will be shut down again. In fact, it will be shut down in the first five idle periods regardless of the actual lengths of them. This policy takes logarithmic idle periods to "learn" that recent idle periods are short and to correct its prediction. In order to remedy this problem, a desirable algorithm should change its prediction sooner once successive wrong shutdowns happen. We do not use predictive wakeup suggested in [41] because it consumes 96% more energy on the disk without significant performance improvement.

### 3.4.7 Memory Requirements

The non-stationary stochastic policy computes in advance the shutdown probabilities for different request arrival rates [19]. It is a tradeoff how many arrival rates are computed. On one extremes, it can compute the optimal probability for all possible arrival rates. At run-time, the power manager only needs to look up a table to find the optimal shutdown probability for a given request arrival rate. However, this generates a large table and the power manager has to store the table in memory. On the other extreme, it can compute only a few rates and linearly interpolates the shutdown probability. While this reduces the memory requirements, it has two other problems. First, it needs more computation at run time. Second, the interpolated probability is only an approximation of the optimal probability. In our experiments, the manager uses more than 50KB memory to store the pre-computed probability to achieve satisfactory power saving. While this is not a problem on personal computers with several megabytes of memory, it can be a concern for systems with tighter memory constraints. The learning tree policy may also grow the tree arbitrarily large at run time. Our experiments limited the tree depth to ten levels so it had at most $2^{10} - 1 = 1023$ tree nodes.

### 3.4.8 Workload Characterization

Some policies characterize workloads in advance. Specifically, SM and DM have two steps: in the first step, they analyze request patterns and find optimal state transition probabilities; in the second step, they control the power states of a device. The first step is performed *off-line* so they must have prior knowledge about request patterns. SM improves DM by using continuous-time request arrival model and adopting a better overhead model. Other policies do not need to analyze requests in advance. For example, NS analyzes request characteristics *on-line* in the sliding window; consequently, this method is more robust when request patterns change.

### 3.4.9 Limitation of Driver-Based Policies

The policies studied in this chapter require only information about request arrival. In order to save more power, power managers need to incorporate more information. The next chapter will discuss how to use process information to predict idle periods more accurately.

## 3.5 Chapter Summary

This chapter presents a framework to implement power-management policies as filter drivers in Microsoft Windows. We used filter drivers to implement various policies. These policies observe request arrival and determine the power states of the device. We quantitatively compare the power saving and performance impact of these policies and point out their advantages and limitations.

# Chapter 4

# Process-Based Policies

A recent trend in power management is to provide higher-level information to power managers, for example, bringing the awareness of power consumption to application programs [26] [51]. Some methods modify application programs to trade off quality of service for power [27]. The previous chapter showed how to perform power management in a device driver; this is the bottom layer in an operating system, as shown in Figure 3.1. Alternatively, power management can be implemented at a higher layer. At a higher layer, the power manager is closer to application programs and can obtain more information to predict the lengths of idle periods. This chapter explains how to incorporate process information to improve power management.

## 4.1 Processes

When a program executes, a *process* is created. This process occupies memory and takes CPU time; it may also generate IO requests. A process is an instantiation of a program. Figure 4.1 shows the states of a process [72]. 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 and termination), operating systems manage when it occupies a processor, how much memory it possesses, which files it opens, and which

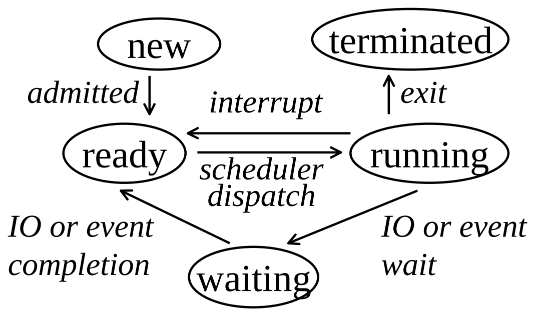


Figure 4.1: process states

| | |
|---|---|
| $pc_i$ | a process |
| $d_i$ | a device |
| $u_i$ | utilization of $d_i$ by all processes |
| $c_i$ | CPU utilization by process $pc_i$ |
| $tbr$ | time between requests |

Table 4.1: symbols and their meanings in this chapter

.

hardware devices it uses. OS kernel has the information about process execution and request generation; such additional information improves the prediction accuracy of power managers.

## 4.2 Estimating 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 (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 $d_i$ and $c_j$ as how often this process runs. The range of $i$ is the number of devices; this is determined by system configuration. The range of $j$ is the number of processes currently under consideration; it is changed at run time due to process creation and termination. All quantities are computed at run time without any modification in application programs. Because it is difficult to model

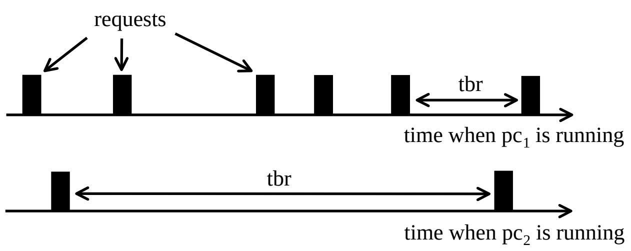


Figure 4.2: time between requests of two processes

request generations mathematically, we use heuristics to estimate device utilization based on per-process information. We will use examples to explain why the heuristics are robust in different scenarios.

### 4.2.1 Device Utilization

Some processes are "CPU-burst", using CPU mostly; some processes are "IO-burst", using IO devices mostly [72]. 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*.

**Example 1** *Figure 4.2 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 is CPU-burst; it rarely generates requests and its tbr is larger.*

*Different programs have distinct request patterns. For example, a file transfer program (*`ftp`*) creates bursty IO requests on a hard disk while a text editor (such as* `emacs`*) creates scattered requests. In one of our measurements, the average tbr for* `ftp` *on a hard disk is much shorter than one second; the average tbr for* `emacs` *is approximately 29 seconds.* $\diamondsuit$

There are various ways to use *tbr* for estimating $u_{i,j}$, for example using the *tbr* between the last two requests or using the running average of the *tbr*'s among all

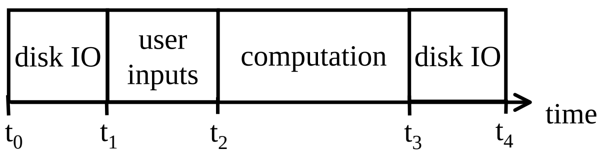


Figure 4.3: a process with four phases

requests. The former considers only one *tbr* while the latter consider all *tbr*'s; neither is appropriate. Using only the last *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 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 $tbr_n$ is the estimated time between requests after these $n$ requests and *tbr* is the latest time between requests (between the $(n-1)^{th}$ and the $n^{th}$ requests). We compute $u_{i,j}$ by this formula:

$$\begin{aligned} tbr_n &= a \cdot tbr + (1-a) \cdot tbr_{n-1} \\ u_{i,j} &= \tfrac{1}{\text{tbr}_n} \end{aligned} \tag{4.1}$$

Next, we explain how to estimate 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 an example of a spreadsheet program with four stages illustrated in Figure 4.3. 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 by requests, the device utilization is overestimated during* $t_1$ *to* $t_3$ *because tbr is not updated.* $\diamond$

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

$$a_{i,j} = e^{-\frac{l_{i,j}}{t_{be,i}}} \tag{4.2}$$

When a process changes from IO-burst to CPU-burst, $l_{i,j}$ is large and $a_{i,j} \ll 1$. When a process changes from CPU-burst to IO-burst, the utilization estimation in (4.1) is not adjusted because $l_{i,j}$ is small and $a_{i,j} \approx 1$. After the adjustment, Equation 4.1 is replaced by the new utilization estimation:

$$w_{i,j} = u_{i,j} \times a_{i,j} \tag{4.3}$$

### 4.2.2 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 spent on this process to compute $c_j$.

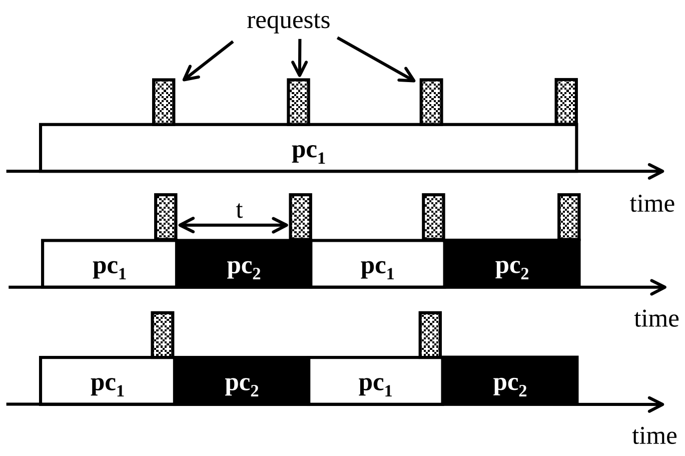


Figure 4.4: three examples of device utilization

$$c_j = \frac{CPUTime(pc_j)}{\sum_{\text{all process } pc_i} CPUTime(pc_i)} \tag{4.4}$$

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. Based on experimental data, we choose one minute as a balance of the two requirements.

### 4.2.3 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:

$$u_i = \sum_{\text{all process } pc_j} w_{i,j} \times c_j \tag{4.5}$$

**Example 3** *Figure 4.4 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, $pc_1$ and $pc_2$, 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 $\frac{1}{t} \cdot 1 = \frac{1}{t}$, $\frac{1}{t} \cdot 0.5 + \frac{1}{t} \cdot 0.5 = \frac{1}{t}$, and $\frac{1}{t} \cdot 0.5 = \frac{1}{2t}$ respectively. This reflects accurately how often the device receives a request.* $\diamondsuit$

### 4.2.4 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

$$u_i < \frac{k}{t_{be,i}} \tag{4.6}$$

where $k$ is the "aggressiveness factor". If $k$ is one, a device is shutdown when the utilization is smaller than $\frac{1}{t_{be,i}}$, or 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 time between requests is longer than $t_{be,i}$. 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 time between requests is shorter than $t_{be,i}$. 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 the average power actually higher. Hence, we suggest a $k$ value equal to or slightly larger than one.

Emphasis should be stressed that our approach is fundamentally different from previous policies that are based on the single-requester model; they do not consider how requests are generated. Our method uses high-level (software) information by distinguishing individual processes.

## 4.3 Experiments

We modified the Linux kernel and device drivers to evaluate this new approach on two IO devices: a hard disk and a network card. The detail of our experimental setup is available in Appendix A.1.

### 4.3.1 Setting Power States

In our implementation, the operating system 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 modified the device drivers so that they exported the commands to the power manager inside the kernel.

### 4.3.2 Workloads

Two 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 transition from idleness to busyness. In [78], the authors discover that Pareto distributions closely approximate the probability that a device changes from idleness to busyness. 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 $0.7/sec$ for $\alpha$ and 0.5 for $\beta$ because they reside in the range presented in [78]. In addition to Pareto distributions, we also consider uniform distributions for comparison. The range of the uniform distribution is zero to ten minutes. 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 minutes later. Each workload runs for two hours.

### 4.3.3 Policy Comparison

We compared out methods with four other policies:

1. no power management
2. three-minute timeout policy
3. 2-competitive policy [45]
4. exponential average policy [41]
5. process-based policy (presented in this chapter)

We compare them 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. $p$: average power, unit: Watt. The measurements include the energy for serving requests; therefore, $p$ could be higher than $p_w$. Including the power for serving requests provides a clearer comparison when we consider the effect on battery lives in portable systems.
2. $t_{ss}$: average time in the sleeping state for each shutdown, unit: second
3. $t_t$: total time during state transitions, unit: second
4. $n_{sd}$: number of shutdowns
5. $n_{wd}$: number of wrong shutdowns.

It is desirable to have low power ($p$), overhead ($t_t$ and $n_{sd}$), low error rate ($n_{wd}$), and long sleeping time ($t_{ss}$).

| workload | | policy | $p$ | $t_{ss}$ | $t_t$ | $n_{sd}$ | $n_{wd}$ |
|---|---|---|---|---|---|---|---|
| | | 1 | 0.91 | 0 | - | - | - |
| | | 2 | 0.89 | 44 | 53 | 5 | 0 |
| 1 | user | 3 | 0.58 | 68 | 435 | 41 | 19 |
| | trace | 4 | 0.74 | 19 | 774 | 73 | 42 |
| | | 5 | 0.50 | 42 | 647 | 61 | 17 |
| | | 1 | 0.90 | 0 | - | - | - |
| | | 2 | 0.88 | 16 | 74 | 7 | 4 |
| 2 | Pareto | 3 | 0.57 | 43 | 721 | 68 | 25 |
| | | 4 | 0.51 | 28 | 1113 | 105 | 30 |
| | | 5 | 0.45 | 41 | 954 | 90 | 11 |
| | | 1 | 0.84 | 0 | - | - | - |
| | | 2 | 0.81 | 46 | 106 | 10 | 2 |
| 2 | uniform | 3 | 0.42 | 80 | 551 | 52 | 5 |
| | | 4 | 0.44 | 45 | 837 | 79 | 17 |
| | | 5 | 0.39 | 88 | 530 | 50 | 4 |

Table 4.2: power and performance of a 2.5" hard disk

| workload | | policy | $p$ | $t_{ss}$ | $t_t$ | $n_{sd}$ | $n_{wd}$ |
|---|---|---|---|---|---|---|---|
| | | 1 | 0.77 | 0 | - | - | - |
| | | 2 | 0.58 | 352 | 14 | 5 | 0 |
| 1 | user | 3 | 0.26 | 93 | 149 | 54 | 8 |
| | trace | 4 | 0.57 | 17 | 262 | 95 | 21 |
| | | 5 | 0.27 | 136 | 94 | 34 | 1 |
| | | 1 | 0.77 | 0 | - | - | - |
| | | 2 | 0.77 | 24 | 19 | 7 | 0 |
| 2 | Pareto | 3 | 0.43 | 40 | 283 | 103 | 12 |
| | | 4 | 0.42 | 14 | 605 | 220 | 46 |
| | | 5 | 0.41 | 66 | 157 | 57 | 7 |
| | | 1 | 0.76 | 0 | - | - | - |
| | | 2 | 0.73 | 89 | 11 | 4 | 0 |
| 2 | uniform | 3 | 0.36 | 45 | 206 | 75 | 0 |
| | | 4 | 0.43 | 16 | 497 | 181 | 29 |
| | | 5 | 0.35 | 46 | 190 | 69 | 6 |

Table 4.3: power and performance for a network card

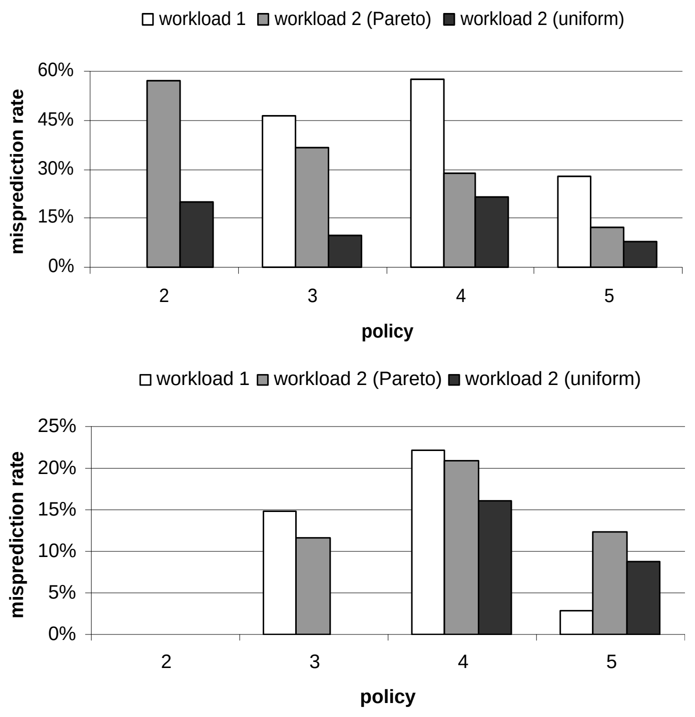
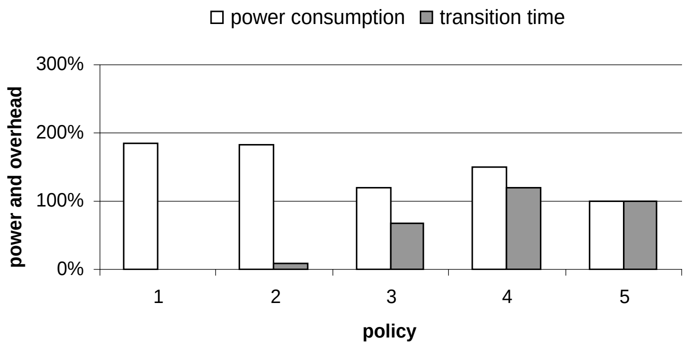


Figure 4.5: misprediction rates, top: hard disk, bottom: network card

## 4.3.4 Power Saving and Performance Impact

Tables 4.2 and 4.3 compare power and performance of different policies; several important facts can be observed.

If a policy saves more power (small $p$), it usually has more shutdowns (large $n_{sd}$); the device spends more time on state transition (large $t_t$). The misprediction rate ($\frac{n_{wd}}{n_{sd}}$) is higher for the disk because it has a longer break-even time. The timeout policy (policy 2) has very low misprediction rates in the network card because it rarely shuts down the device (small $n_{sd}$). Even though the misprediction rate is low, the overall power is high. The 2-competitive method (policy 3) has comparable power saving with our method (policy 5) for the network card. However, the 2-competitive policy has higher misprediction rates. Even though power saving depends on workloads and devices, our method consistently achieves nearly 50% power saving for both devices on all workloads. Other policies have large variations in their power saving. For example, policy 4 saves 20% to 48% power.



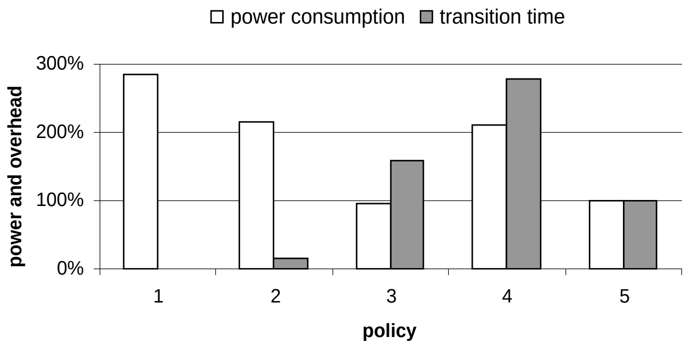


Figure 4.6: power and overhead for workload 1, hard disk (top) and network card (bottom)

Figures 4.6 is the power ($p$) and transition time ($t_t$) of different policies; shorter bars (less power and transition overhead) are preferred. This figure is normalized related to our method.

### 4.3.5 Parameter Setting

Three parameters affect power and performance: discount factor $a$ in Equation (4.1), window size in Equation (4.4), and aggressiveness $k$ in Equation (4.6). In general, if a power manager responds to a potentially long idle period more quickly, it may save more power. Meanwhile, it can cause more shutdowns and degrade performance more seriously. Consequently, these parameters trade off power with performance. A power manager responds more quickly under the following conditions:

- large $a$. When $a$ is large, more weight is put on the latest $tbr$ and $u_{i,j}$ changes more quickly.
- small window size. When the window is smaller, the denominator is smaller and $c_j$ changes more quickly.
- 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 minute for the window size, and 1 for $k$ in generating Tables 4.2 and 4.3.

## 4.4 Chapter Summary

This chapter proposes a new approach for power management by exploiting the information available from operating system kernel. It distinguishes requesters as individual processes and considers the processor utilization of each process for estimating device utilization. Experimental results demonstrate that, with the additional information, this method is more effective in power reduction with better performance.

# Chapter 5

# Low-Power Scheduling

Process schedulers determine when a process executes and directly affects the lengths of idle periods. In this chapter, we will demonstrate how process scheduling can facilitate power management.

## 5.1 Motivating Example

### 5.1.1 Concept of Jobs

Each process can be divided into smaller units, called *jobs* [17]. In a computer, processes are well-defined by the operating system [75]. Unlike "process", there is no widely accepted definition for "job". In this thesis, a job is defined as a unit to finish a specific task and its starting time can be specified by the program.

**Example 1** *When a user executes* `ftp`*, a process is created to execute this program. Downloading a file is a job. Since this process may download multiple files,* `ftp` *can have more than one 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.* ◊

### 5.1.2 Jobs Created by Timers

In Section 1.3.4, we explain that requests can be created by timers. If a request is created by a timer, its arrival time is known in advance. Namely, we can predict precisely when the device will become busy again and determine whether to shut down the device for power management.

In UNIX, creating a job by a timer needs three steps: (1) calling `setitimer` to create a timer (2) registering a callback function for the `SIGALRM` signal (3) executing the callback function when this signal is issued. The task performed by the callback function is considered as a job because it is scheduled by the timer. Requests generated by this job is predictable. 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.

### 5.1.3 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 totally 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* [17]. 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 Figure 5.1.

Another type of constraints is *timing constraints.* The timing constraint of a job is a *deadline*; the job has to finish before the deadline. Deadlines can be classified into three categories: firm, soft, and on-time [17]. Figure 5.2 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

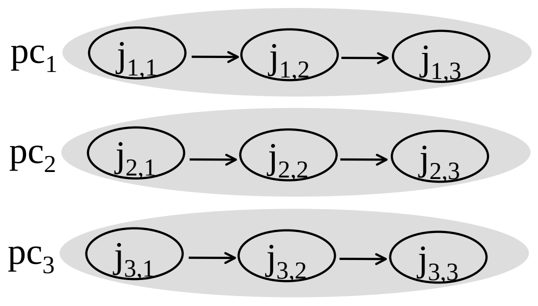


Figure 5.1: precedence of three independent processes

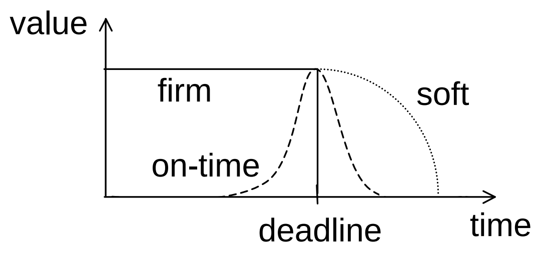


Figure 5.2: three types of deadlines

smoothly after the deadline. If a job has an on-time constraint, it should finish near the deadline, neither too early nor too late. As explained earlier, a timer is used to create a request in the future. When the timer expires, a callback function in invoked. A timer is an "on-time" constraint; it should expire precisely at the specified time.

### 5.1.4 Scheduling Jobs for Power Management

Suppose only three jobs, $j_{1,1}$, $j_{1,2}$, and $j_{2,3}$ need a specific device. For simplicity, we assume that it takes $t$ to execute each job. Figure 5.3 shows two possible execution orders. A black rectangle indicates that this job needs the device. One major 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 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. Even if the break-even time is shorter than $2t$, the second schedule is still preferred because state-transition overhead occurs only once.

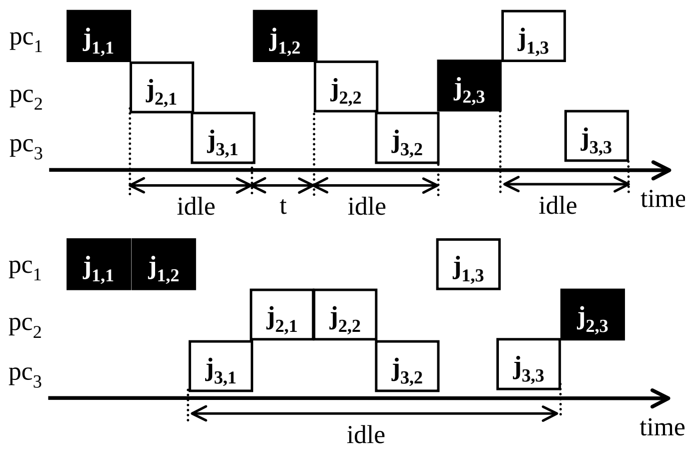


Figure 5.3: two schedules of three independent processes

### 5.1.5 Scheduling in Inactive Systems

In personal computers, some future IO requests are predictable. For example, text editors (such as `Word`) often have “autosavers” that save the contents periodically. An email reader (such as `Netscape`) retrieves email from a mail server and store these mails on a local hard disk. Both `Word` and `Netscape` 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 generated by these two programs are arranged so they arrive at approximately the same time, the disk can remain idle and sleep for longer durations.

## 5.2 Off-line Scheduling

Our method is based on on-line scheduling. 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 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 predict user behavior. We analyze off-line

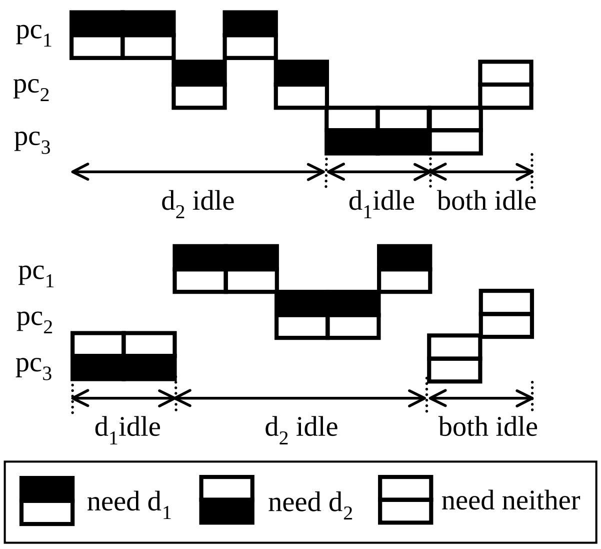


Figure 5.4: three processes using two devices

scheduling first because it sets the basis for understanding on-line scheduling. When there are multiple devices, even off-line scheduling is a complex problem. This can be best illustrated by an example.

**Example 2** *Consider three independent processes ($pc_1$, $pc_2$, and $pc_3$) using two devices ($d_1$ and $d_2$). Each process has three jobs; each job may use $d_1$, $d_2$, both, or neither. The job-device relationship is expressed by a required device set ($\mathcal{RDS}$). Suppose the $\mathcal{RDS}$ of each job is expressed in Table 5.1 and each job takes $t$ to execute. Figure 5.4 shows two schedules of these nine jobs. In the first schedule, $d_2$ is idle for $5t$ first, busy for $2t$, and idle again for another $2t$; 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.* $\diamondsuit$

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

Table 5.1: devices required by each job

### 5.2.1 Problem Formulation

Consider a set of $n$ jobs: $\mathcal{J} = \{j_1, j_2, \ldots, j_n\}$ on a single-processor system with $m$ devices: $\mathcal{D} = \{d_1, d_2, \ldots, d_m\}$. It takes $ex_i$ to execute job $j_i$. 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 constraints (timing and precedence). 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: $d_i$, $i = 1, 2, \ldots, m$. The energy of device $d_k$ is divided into three parts:

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

We use $p_{w,k}$ as the power consumption of device $d_k$ when it is in the working state; $p_{s,k}$ is the power when $d_k$ is sleeping. To compute $e_{busy,k}$, 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 $[t_i, t_{i+1})$.

$$e_{busy,k} = \sum_{\substack{i \in [1,n] \\ \text{such that } r_{s_i,k}=1}} p_{w,k} \cdot (t_{i+1} - t_i) \tag{5.1}$$

| | |
|---|---|
| $j_k$ | a job, one process has one or many (possibly infinite) jobs |
| $ex_i$ | execution time of job $j_i$ |
| $r_{a,b}$ | job $j_a$ uses device $d_b$ |
| $e_{o,i}$ | state-transition energy overhead of device $d_i$ |
| $\mathcal{D}$ | a set of devices being power managed |
| $\mathcal{J}$ | a set of jobs |
| $\mathcal{S}$ | a schedule of jobs |
| $\mathcal{RDS}$ | required device set |

Table 5.2: symbols and their meanings in this chapter

.

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 as

- a sequence of jobs, $j_{s_w}$, $j_{s_{w+1}}$, $\ldots$, $j_{s_x}$, that do not use $d_k$; namely, $r_{s_w,k} = r_{s_{w+1},k} =, \ldots, = r_{s_x,k} = 0$
- $d_k$ is used before this sequence: $r_{s_{w-1},k} = 1$
- $d_k$ 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 $e_{sleep,k}$. Suppose $idle_k$ is an idle period of $d_k$; the length of this period is $\mid idle_k \mid$ ($\mid idle_k \mid = t_{x+1} - t_w$). When $\mid idle_k \mid$ is larger than $t_{be,k}$, $d_k$ sleeps to save power. In order to compute $e_{sleep,k}$, we find all idle periods that are longer than $t_{be,k}$. Let $\mathcal{IS}_k$ be the set of idle periods that are longer than $t_{be,k}$: $\mathcal{IS}_k = \{ idle_k : \mid idle_k \mid > t_{be,k} \}$. The energy $e_{sleep,k}$ is the energy during these long idle periods. Because $d_k$ changes power states, $e_{sleep,k}$ includes the state-transition energy, $e_{o,k}$.

$$e_{sleep,k} = \sum_{is \in \mathcal{IS}_k} (p_{s,k} \cdot \mid is \mid + e_{o,k}) \tag{5.2}$$

where $\mid is \mid$ is the length of the corresponding idle period.

| device | $p_w$ | $p_s$ | $t_o$ | $e_o$ | $t_{be}$ |
|---|---|---|---|---|---|
| $d_1$ | 5 | 1 | $2t$ | $22t$ | $5t$ |
| $d_2$ | 3 | 2 | $t$ | $8t$ | $6t$ |

Table 5.3: device parameters for Figure 5.4

Finally, we consider $e_{idle,k}$ 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: $\mathcal{IW}_k = \{ \ idle_k : \mid idle_k \mid \leq t_{be,k} \ \}$.

$$e_{idle,k} = \sum_{iw \in \mathcal{IW}_k} p_{w,k} \cdot \mid iw \mid \tag{5.3}$$

where $\mid iw \mid$ is the length of the corresponding idle period.

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

$$e = \sum_{k=1}^{m} e_{busy,k} + e_{sleep,k} + e_{idle,k} \tag{5.4}$$

**Example 3** *Let's revisit the two schedules in Figure 5.4. Suppose the parameters of these two devices are shown in Table 5.3. We can compute the energy for the two schedules. It is easier to computer the energy for the first schedule because all idle periods of each device are shorter than the corresponding break-even time; neither device enters the sleeping state. The energy is $(5 + 3) \cdot 9t = 72t$. For the second schedule, the energy for $d_1$ is $5 \cdot 9t = 45t$; the energy for $d_2$ is $3 \cdot 2t = 6t$ for the first $2t$. Since the idle period is longer than the break-even time of $d_2$, it enters the sleeping state. The energy is $e_{o,2} + p_{s,2} \cdot (idle\ time - t_{o,2}) = 8t + 2 \cdot (7t - t) = 20t$. The total energy is $45t + 6t + 20t = 71t$; this is less than the energy of the first schedule. Therefore, the second schedule is more energy-efficient.* $\diamond$

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

## 5.3 On-Line Scheduling

Off-line scheduling is NP-complete even with no timing or precedence constraints. Since our target is interactive systems, on-line scheduling is necessary. This section presents heuristics for low-power scheduling on interactive systems.

### 5.3.1 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. For example, fixed-priority scheduling statically assigns priorities to jobs based on their urgency. Figure 5.5 shows the flow of a Linux scheduler [9]. 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. If there is no interrupt, the scheduler checks whether any timer expires. 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 to execute. If no user process is ready to execute, the schedule selects a special low-priority process called "idle" process; this process is essentially an infinite loop.

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 Figure 5.5. The extension is divided into two parts: (1) it wakes up a sleeping device before scheduling a job that requires this device. (2) it arranges execution orders to facilitate power management.

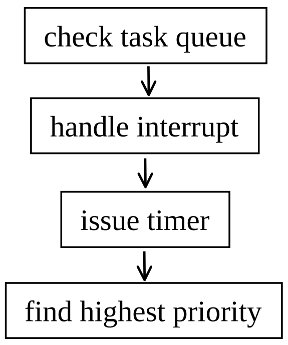


Figure 5.5: steps of a Linux scheduler

### 5.3.2 Predictive Wakeup

If a job generates requests for a sleeping device, this job has to wait for the wakeup delay. Ideally, the device should wake up before the job starts execution to eliminate waiting; this is called "predictive wakeup" [84]. A device should wake up just before requests arrive. Waking up too early wastes energy; waking up too late does not eliminate waiting. Traditional methods for predictive wakeup have low prediction accuracy because they adopt the single-requester model and mix requests from all processes. Our experiments show that the method proposed in [41] actually increase energy due to low prediction accuracy.

In order to perform predictive wakeup, 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. If a process used a device during its last execution, the scheduler predicts that the process will use the same device. The advantage is that no user program needs to be modified; the disadvantage is that the prediction may be wrong, especially when a process changes its behavior from computation-intensive to IO-intensive 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

```
PredictiveWakeup ($\mathcal{J}$ := jobs to schedule)
begin
    sort $\mathcal{J}$ by their timer values;
    $j$ := a job in $\mathcal{J}$ with the earliest timer;
    for each $d_k$ in $\mathcal{RDS}(j)$
        if ($d_k$ is sleeping) and
        (timer($j$) - $now \leq t_{wu,k}$)
            wake up $d_k$;
end
```

Figure 5.6: predictive wakeup

information for predictive wakeup [51]. We propose a method to augment the timer interface for predictive wakeup. 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 5.1.5, 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, five minutes, download)` *and* `RequireDevice (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. Figure 5.6 shows a pseudocode of predictive wakeup.

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

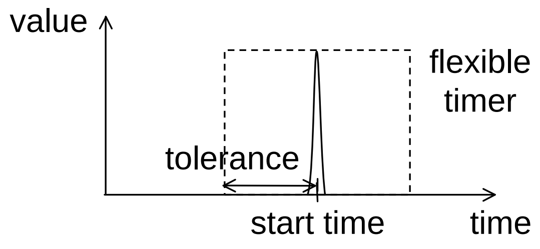


Figure 5.7: flexible timer

the request. Predictive wakeup improves performance but does not save power. The real benefit will be clearer after we explain how to schedule jobs in Section 5.3.4.

### 5.3.3 Flexible Timers

Some jobs have the flexibility to start execution before or after their timer expires (such as the situation in 5.1.5). 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
```

The concept of flexible is illustrated by modifying Figure 5.2. In Figure 5.7, 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.

### 5.3.4 Scheduling Jobs for Power Reduction

The callback function of a timer creates a job; `RequireDevice` also specifies which devices will be used. Together, they specify a set of jobs and the required devices. The

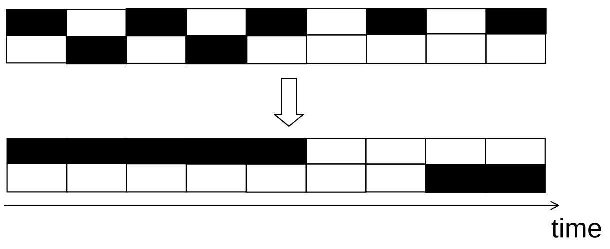


Figure 5.8: group jobs according to their device requirements

system call `RequireDevice` 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** *Figure 5.8 is an example of scheduling for power management on two devices. The meaning of each rectangle is explained earlier in Figure 5.4. At the top of Figure 5.8, 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.* ◇

Figure 5.9 outlines the heuristic 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 $\mathcal{RDS}(\mathcal{J}_i)$. The length of a group, say $\mathcal{J}_i$, is the sum of execution time of all jobs in this group:

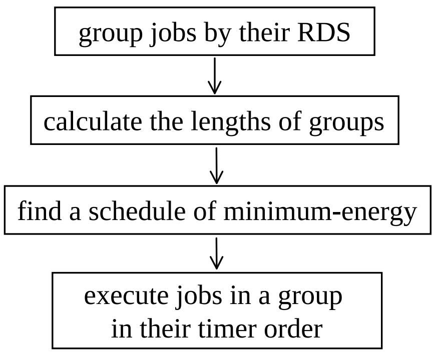

group jobs by their RDS
calculate the lengths of groups
find a schedule of minimum-energy
execute jobs in a group in their timer order

Figure 5.9: low-power scheduling

| job | $j_1$ | $j_2$ | $j_3$ | $j_4$ | $j_5$ | $j_6$ | $j_7$ | $j_8$ | $j_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{RDS}$ | $d_1$ | $d_1$ | $d_1$ | $d_1$ | $d_1$ | $\phi$ | $d_2$ | $d_2$ | $\phi$ |

Table 5.4: devices requirements for Example 6

$$\mid \mathcal{J}_i \mid = \sum_{\forall j_x \in \mathcal{J}_i} ex_x \tag{5.5}$$

Here $ex_x$ is the execution time of $j_x$. Let $\mathcal{LPGS}$ 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 (5.1) - (5.4). For example, $\mid \mathcal{J}_i \mid$ corresponds to the execution time of one job $(t_{i+1} - t_i)$ in (5.1); $e_{busy,k}$ is calculated by

$$e_{busy,k} = \sum_{d_k \in \mathcal{RDS}(\mathcal{J}_i)} p_{w,k} \cdot \mid \mathcal{J}_i \mid \tag{5.6}$$

Energy $e_{sleep,k}$ and $e_{idle,k}$ can be computed in a similar way. After computing the energy of each schedule, the scheduler finds one schedule with the minimum energy. Figure 5.10 shows a pseudocode of the scheduler.

**Example 6** *Consider nine jobs waiting for execution. The devices required by each*

```
/* initialization */
$\mathcal{LPGS}$ := empty; /* low-power group schedule */
/* end of initialization */

...... /* the steps in Figure 5.5; extension starts below */
if (new jobs are created)
    group jobs by their $\mathcal{RDS}$'s;
    calculate the length of each group;
    /* find one schedule first */
    $\mathcal{S}$ := one schedule of these groups;
    $eng$ := energy of $\mathcal{S}$;
    $\mathcal{LPGS}$ := $\mathcal{S}$;
    /* check whether there are better schedule */
    for each schedule of these groups
        $ethis$ := the energy of this schedule;
        if ($ethis < eng$) /* a better schedule */
            $eng$ := $ethis$;
            $\mathcal{LPGS}$ := this schedule;
```

Figure 5.10: scheduling jobs by their $\mathcal{RDS}$ groups

*job is shown in Table 5.4. These nine jobs belong to three $\mathcal{RDS}$'s: $\{d_1\}$, $\{d_2\}$, and $\phi$. The first group, $\mathcal{RDS}_1 = \{d_1\}$, has five jobs: $\{j_1, j_2, j_3, j_4, j_5\}$. The second group, $\mathcal{RDS}_2 = \{d_2\}$, has two jobs $\{j_7, j_8\}$. The third group, $\mathcal{RDS}_3 = \phi$, 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 that the devices have parameters shown in Table 5.3.*

*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.* $\diamondsuit$

Although it seems that Figure 5.10 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. (2) The scheduler considers only ready jobs and there are small number of ready jobs at each moment.

### 5.3.5 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(,` $t_e$`,` $t_t$`,` $t_s$`, )` where $t_e$, $t_t$, and $t_s$ are the execution time, tolerance, and starting time. The scheduler will start the job no later than $t_s + t_t - t_e$.

### 5.3.6 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** *Figure 5.11 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)*

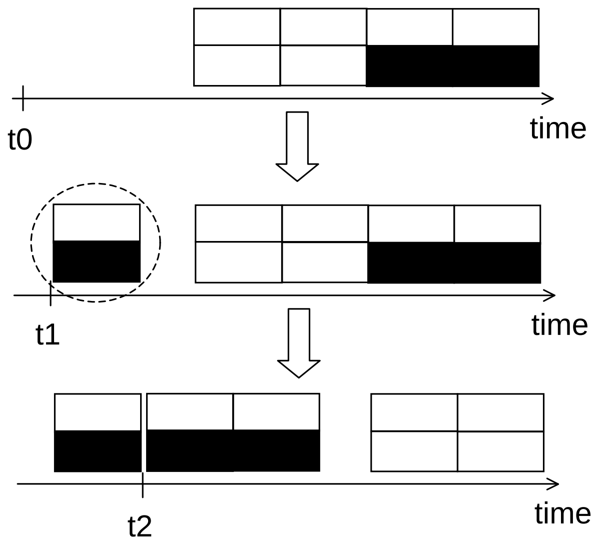


Figure 5.11: group jobs according to their device requirements

*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.* $\diamondsuit$

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

## 5.4 Experiments

Recall the system layers in Figure 3.1: scheduling provides more information for power management. We enhanced the power manager presented in 4.3 to include scheduling information.

A workload is used for comparison; it generates periodic requests with timers. We consider both fixed timers (created by `setitimer`) and flexible timers (created by `RequireDevice`). There are six processes; three use `setitimer` and the other three use `RequireDevice`. Timers expire between one to five minutes. The tolerance is

| policy | $p$ | $t_{ss}$ | $t_t$ | $n_{sd}$ | $n_{wd}$ |
|---|---|---|---|---|---|
| 1 | 0.88 | 0 | - | - | - |
| 2 | 0.87 | 0 | 0 | 0 | 0 |
| 3 | 0.45 | 105 | 392 | 37 | 13 |
| 4 | 0.64 | 14 | 1198 | 113 | 65 |
| 5 | 0.37 | 72 | 583 | 55 | 9 |
| 6 | 0.35 | 85 | 530 | 50 | 0 |
| 7 | 0.25 | 220 | 265 | 25 | 0 |

Table 5.5: power and performance of a 2.5" hard disk

one minute. In addition to the policies used in 4.3.3, we added two policies: (6) process-based with predictive wakeup (7) process-based with scheduling.

Tables 5.5 and 5.6 compare the power and performance of these policies. Policy 6 does not change the time when requests are generated but it reduces misprediction ($sd_w$). Policy 7 changes the time when requests are generated and significantly reduces power, up to 72% on the network card.

Figure 5.13 shows the distributions of idle periods with and without low-power scheduling. Circles represent the lengths of idle periods without low-power scheduling; squares represent the lengths with low-power scheduling. This figure shows that circles are more widely spread. In particular, the circles show some "medium-length" idle periods between 60 seconds to 180 seconds. 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 seconds);

| policy | $p$ | $t_{ss}$ | $t_t$ | $n_{sd}$ | $n_{wd}$ |
|---|---|---|---|---|---|
| 1 | 0.76 | 0 | - | - | - |
| 2 | 0.76 | 0 | 0 | 0 | 0 |
| 3 | 0.31 | 33 | 302 | 110 | 9 |
| 4 | 0.39 | 41 | 225 | 82 | 22 |
| 5 | 0.30 | 104 | 96 | 35 | 3 |
| 6 | 0.29 | 80 | 140 | 51 | 1 |
| 7 | 0.19 | 228 | 73 | 25 | 0 |

Table 5.6: power and performance for a network card

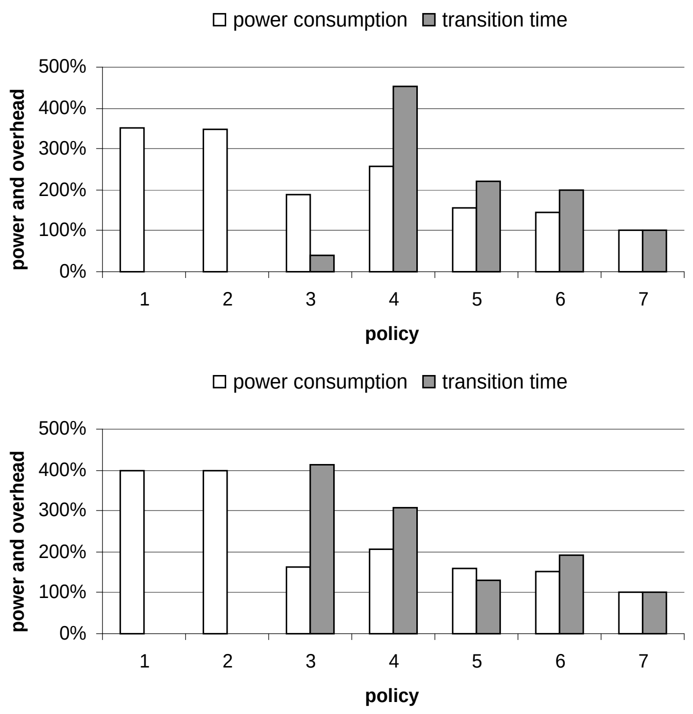


Figure 5.12: power and overhead, hard disk (top) and network card (bottom)

power managers use long idle periods to save power.

When power management is applied to a notebook computer, the battery life increases. Figure 5.14 compares the battery capacity. Power management increases the battery lifetime by approximately four minutes; this is 8.5% of the original lifetime. The increase is limited due to a large baseline power consumption of the rest of the computer, including the processor and other electronic components.

## 5.5 Chapter Summary

This chapter explains how process schedulers can collaborate with power managers. Since schedulers determine when processes execute, schedulers control the durations of idle periods. Schedulers cluster idle periods to make them continuous and long so that power management is applicable. Combining power management with scheduling requires additional information from application programs. We present a new system

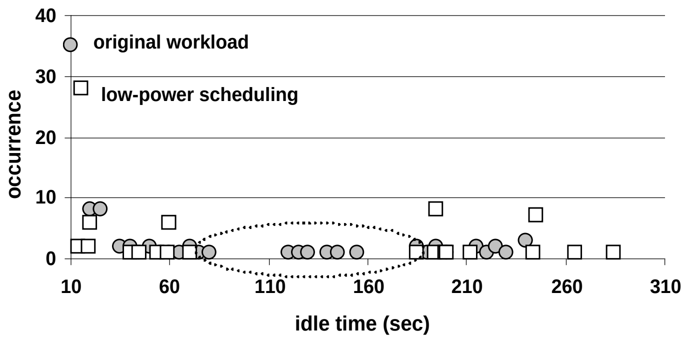


Figure 5.13: Low-power scheduling reduces medium-length idle periods (60 to 180 seconds).

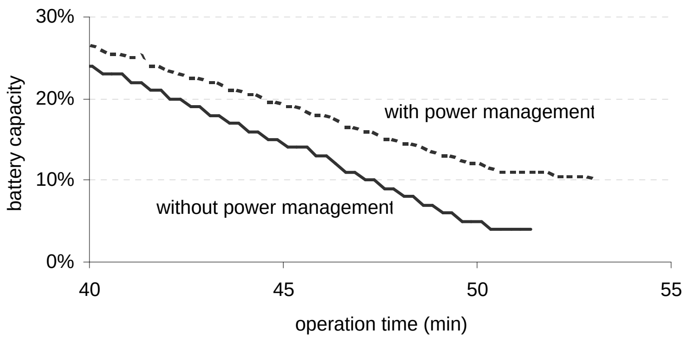


Figure 5.14: Power management increases the battery lifetime.

call that allows application programs to specify device requirements, the time these devices are needed, and the acceptable variations of the actual starting time.

# Chapter 6

# Frequency Scaling on Processors

Previous chapters consider IO devices with only two power states. This chapter discusses managing processor power with multiple active states; different techniques are required to manage multiple power states. The states are differentiated by their performance and power. The processors have better performance when they consume more power. Processors can *slow down* when their utilization is low even though these processors are not idle. This is different from the shutdown techniques explained in earlier chapters. When a device is shut down, it cannot serve any request. When a processor is slowed down, it can still execute instructions at a lower speed. Instead of predicting idleness, power managers estimate the utilization of processors. This chapter focuses on reducing power consumption for "mixed workloads": multimedia applications that require constant output rates and sporadic jobs that need prompt responses.

Section 2.3 explains how frequency scaling can reduce power consumption. Scaling may have negative impact on timing-sensitive programs, for example, failing to meet the output rate for a multimedia program. Scaling frequencies and voltages while meeting timing constraints is an active research topic in recent years [5] [40] [47] [55] [57] [56] [60] [64] [71] [77].

Our method divides multimedia programs into stages and inserts buffers between them. Data buffering has three purposes: (1) to support constant output rates (2) to allow frequency scaling for energy reduction (3) to shorten the response times of

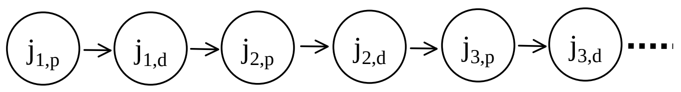


Figure 6.1: Processing and displaying form a pipeline.

sporadic jobs.

This chapter constructs frequency-assignment graphs; each vertex represents the current state of the buffers and the frequencies of the processor. We develop an efficient graph-walk algorithm that assigns frequencies. In this graph, each vertex represents the current state of the buffers, the processor frequencies, and how the buffers are filled (or drained). We present an efficient method to compute optimal solutions by graph walking. This method was implemented on a StrongARM-based hand-held computer. Our experimental results show that inserting buffers can achieve nearly optimal power saving with only a few discrete frequencies.

## 6.1 Buffer Insertion

### 6.1.1 Buffers in a Pipeline

An MPEG player can be divided into *stages*, such as decoding and displaying. These stages form a *pipeline*. Let $j_{i,p}$ and $j_{i,d}$ be the jobs to process (i.e. decode) and to display the $i^{th}$ frame. A frame has to be decoded before being displayed; this is a precedence constraint: $j_{i,p} \rightarrow j_{i,d}$. If there is no additional storage between the two stages, no frame can be processed before the previous frame is displayed. This requires $j_{i-1,d} \rightarrow j_{i,p}$. Figure 6.1 is the precedence relationship for such a pipeline.

If there is additional storage space (buffers) between the two stages, a frame can be processed even if the previous frame has not been displayed. Hence, $j_{i-1,d}$ does not have to precede $j_{i,p}$. The precedence relationship is changed, as shown in Figure 6.2. This figure assumes that the video data have to be processed sequentially; consequently, job $j_{i,p}$ precedes job $j_{i+1,p}$. Also, frames should be output sequentially so $j_{i,d} \rightarrow j_{i+1,d}$. After buffers are inserted between the stages, there are multiple options to arrange the execution order of these jobs. For example, $j_{2,p}$ can execute

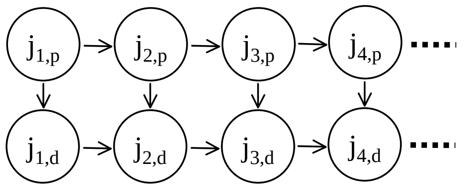


Figure 6.2: Inserting buffers changes the precedence relationship.

before $j_{1,d}$.

## 6.1.2 Energy Reduction with Buffers

A MPEG player has to maintain a constant output rate: displaying a frame every $t$ unit of time; $t$ is called a *period.* For a movie with 30 frames per second, $t$ is 33 milliseconds. Figure 6.3 (a) shows this requirement: $j_d$ executes once every period. If a processor's frequency can be set to any value, it consumes the minimum energy when the processor takes exactly $t$ to process and display one frame [43]. There is no slack time as shown in Figure 6.3 (b).

However, if a processor has only finite frequencies and this optimal frequency is unavailable, the processor has to run at a higher frequency. Since this frequency is higher than necessary, the processor consumes more power; the processor is idle after processing and displaying one frame as shown in Figure 6.3 (c). The processor cannot enter a lower frequency because it will fail to provide the required output rate; Figure 6.3 (d) illustrates this situation.

One solution is to insert buffers between jobs so that the processor can process more frames at a higher frequency while maintaining the same output rate. When enough frames have been processed and stored in the buffers, the processor retrieves processed frames from the buffer to maintain the output rate. Since the processor does not have to process images, it can enter a lower frequency and still meet the output rate requirement. Figure 6.4 depicts this approach. In this figure, the height means the processor frequency. Four frames are processed in the first two and half periods; then, the processor is scaled down to a lower frequency. Before the buffers become empty, the processor enters the higher frequency and refills the buffers. Buffers are

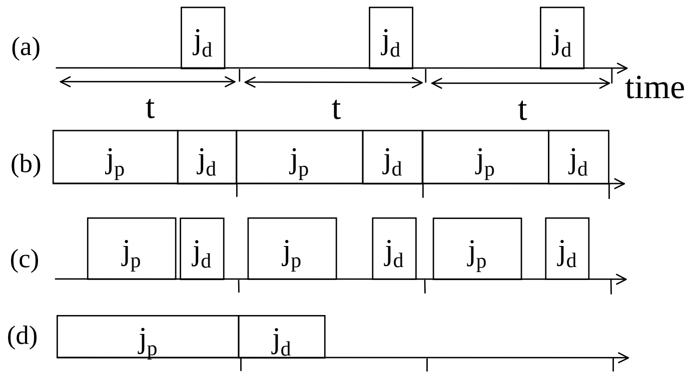


Figure 6.3: (a) constant output rate for display (b) scaling frequency to avoid slack time (c) discrete frequencies cause idleness and waste energy (d) scale to a lower frequency and miss the output rate

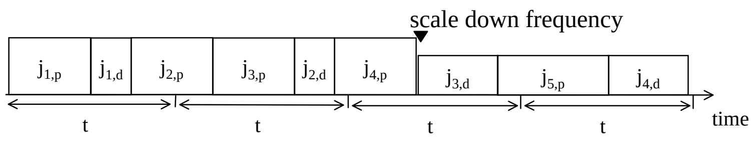


Figure 6.4: process more than one frame at the higher frequency then scale to the lower frequency

used to reduce the power in pipelines [37] [15] [18], or to smoothen run-time variations [42]. Previous studies do not consider the advantages of buffers on processors with finite frequencies.

### 6.1.3 Reducing Response Time

Other than reducing power, buffer insertion can also improve the performance for sporadic jobs without disrupting other jobs. Imagine that a user moves the mouse cursor and clicks one button at the end of the $12^{th}$ period as shown in Figure 6.5. This command can be divided into two jobs: $j_{r,1}$ and $j_{r,2}$. The first job draws the movement of the mouse cursor; the second job processes the command invoked by the click.

Figure 6.5 shows two scenarios: with and without a buffer. In (a), no additional frame is buffered; $j_p$ has to execute once every period. Consequently, only $j_{r,1}$ can

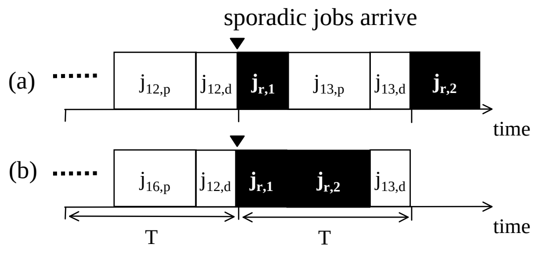


Figure 6.5: (a) no buffer (b) buffer additional frames to reduce the response time of a sporadic job

execute during the 13th period; $j_{r,2}$ has to wait until the 14th period. In contrast, (b) buffers four additional frames ($j_{16,p}$ executes at the 12th period); both $j_{r,1}$ and $j_{r,2}$ can execute during the 13th period. As a result, the user can see the response of this command in the 13th period in (b). Buffering reduces the response time of a sporadic command.

### 6.1.4 Memory Requirements of Buffers

Even though buffering images requires additional memory, a typical computer has enough memory to buffer multiple frames. For example, palm-size computers often have more than 8MB memory. A frame of $240 \times 160$ pixels with 256 colors per pixel requires $240 \times 160$ bytes, or 38KB. Four hundred kilobytes are enough to buffer ten frames; this is only five percent of the available memory.

## 6.2 Related Work

Scaling techniques can be classified into two categories according to whether they consider timing constraints. The first category does not guarantee meeting constraints. In [82], the authors propose several methods that periodically estimate process utilization and adjust the power states. Simulations of various techniques are presented in [32] and [64]. In [77], the authors model the arrival of jobs as random processes. While this approach can meet timing constraints statistically, it does not guarantee to always meet the constraints.

The second category considers timing constraints. In [40], the authors use off-line analysis to determine whether it is possible to meet hard deadlines and to assign the power states. Linear programming methods are proposed in [43] [56] to find optimal voltages / frequencies for processors with discrete power states while meeting deadlines.

Some techniques have been implemented on real systems. In [5] [36] [65], the authors use StrongARM-based systems to demonstrate the effectiveness of scaling and point out some limitations in implementation.

Our work differs from existing approaches in the following ways

1. It assigns frequencies to processors that have finite power states.
2. Data buffers are inserted into the program that needs a constant output rate.
3. An efficient graph-based method is presented to assign frequencies for reducing energy.
4. This method can handle workloads with very long time horizons.
5. The response times of sporadic jobs are shortened without affecting the output rate.
6. It calculates the minimum buffer sizes to meet the timing constraints of sporadic jobs.

## 6.3 Assumptions

We make the following assumptions to simplify the formulation of the problem.

- The processor has only discrete and finite frequencies.
- Data processing is sequential on a single processor; there is no forward data dependence. For example, we do not consider B frames in MPEG. One example of video without forward dependence is motion JPEG.

- The processor changes frequencies only at the beginning of a period of length $t$.
- The total energy is determined by frequencies only; we will use formula (2.21) as the goal to represent energy. We consider the average power for a given duration. Since the integration of power over time is the energy, minimizing energy is equivalent to minimizing power. We use energy and power interchangeably unless it is necessary to distinguish them.
- The jobs in the multimedia program are atomic and their execution cannot cross period boundaries.
- The computational work of a job is measured by the number of operations. One operation takes one time unit at unit frequency. Hence, the execution time of the same job increases linearly to the reciprocal of the frequency.
- The number of operations for a specific job is constant.
- Jobs can be scheduled at the highest frequency.
- It takes no time to start executing a jobs; there is no context switching overhead.
- Buffers are not shared among jobs.

These assumptions may be removed as extensions of the work presented in this chapter.

## 6.4 Analytical Model by Integer Programming

With the assumptions stated above, the power reduction problem can be modeled as an *integer linear programming* problem. For complete comparison, we show the details of such modeling before transforming it into a graph walking problem in the next section. This section derives an analytical model for energy minimization under performance and resource constraints. The formulation becomes more general (and more complex) in each subsection by considering additional factors.

### 6.4.1 Two Frequencies and Two Jobs

This section assumes (1) there are two jobs (2) the processor has only two frequencies (3) it takes no time to change frequencies. We will remove these assumptions later.

The processor has two frequencies: $\phi_0$ and $\phi_1$. An MPEG movie has $n$ frames to display in $n$ periods. The length of a period is $t$. For each frame, there are two jobs: processing ($j_p$) and displaying ($j_d$). Each frame has to be processed before being displayed ($j_p \rightarrow j_d$).

The $i^{th}$ period is during $(i-1, i]t$. Let $\mathcal{N} = \{1, 2, 3, ..., n\}$. We use $x_i \in \{0, 1\}$, $i \in \mathcal{N}$, to indicate the frequency during the $i^{th}$ period. If $x_i = 0$, the processor runs at $\phi_0$; if $x_i = 1$, the processor runs at $\phi_1$. The frequency during the $i^{th}$ period is $\phi_1 \cdot x_i + \phi_0 \cdot (1 - x_i)$. Let $a_i$ be the number of frames processed during the $i^{th}$ period. If $a_i = 1$, one frame is processed during this period; $a_i$ is a non-negative integer. Suppose $w_p$ and $w_d$ are the number of operations for processing and displaying one frame. It takes $\frac{w_p}{\phi_0}$ to process one frame at frequency $\phi_0$.

During the $i^{th}$ period, the total number of operations is $a_i \cdot w_p$ for processing and $w_d$ for displaying. They have to finish within a period; therefore

$$\frac{a_i \cdot w_p + w_d}{\phi_0 + x_i \cdot (\phi_1 - \phi_0)} \leq t \qquad \forall i \in \mathcal{N} \tag{6.1}$$

It can be rewritten as

$$a_i \cdot w_p + w_d \leq t \cdot (\phi_0 + x_i \cdot (\phi_1 - \phi_0)) \qquad \forall i \in \mathcal{N} \tag{6.2}$$

The number of frames processed up to $kt$ ($k \in \mathcal{N}$) is the sum of frames processed in each period: $\sum_{i=1}^{k} a_i$. At least $k$ frames have to be processed before $kt$ because $k$ frames have been displayed at $kt$. This can be expressed by the following constraint:

| system parameters | | |
|---|---|---|
| $s$ | integer | number of frequencies |
| $\mathcal{S}$ | set | $\{1, 2, ..., s\}$ |
| $q$ | integer | index of frequency, $q \in \mathcal{S}$ |
| $\phi_q$ | | one of the available frequencies, $\phi_1 > \phi_2, ..., > \phi_s$ |
| $\Delta$ | | time to change frequencies |
| $b_l$ | | size of the buffer between $j_l$ and $j_{l+1}$ |
| workload parameters | | |
| $t$ | | length of a period, such as 33 ms for an MPEG frame |
| $n$ | integer | number of frames |
| $\mathcal{N}$ | set | $\{1, 2, ..., n\}$ |
| $i$ | integer | index of period, $i \in \mathcal{N}$ |
| $m$ | integer | number of jobs |
| $\mathcal{M}$ | set | $\{1, 2, ..., m\}$ |
| $l$ | integer | index of job, $l \in \mathcal{M}$ |
| $w_l$ | | number of operations of job $j_l$ |
| decision variables | | |
| $a_{i,l}$ | integer | number of executions of $j_l$ in the $i^{th}$ period, $a_{i,l} \geq 0$ |
| $y_{i,q}$ | binary | frequency in the $i^{th}$ period |
| determined by decision variables | | |
| $\delta_i$ | binary | frequency changed in the $i^{th}$ period |
| $\mathbf{y}_i$ | vector | frequency in the $i^{th}$ period, $\mathbf{y}_i =< y_{i,1}, y_{i,2}, ..., y_{i,s} >$ |

Table 6.1: symbols and meanings

$$\sum_{i=1}^{k} a_i \geq k \qquad \forall k \in \mathcal{N} \tag{6.3}$$

At $kt$, $(\sum_{i=1}^{k} a_i)$ frames have been processed but only $k$ frames are displayed; the additional frames are stored in a buffer. Suppose one frame takes one unit of space and the buffer size is $b$. The following constraint restricts the number of frames processed so that they do not overflow the buffer at $kt$.

$$(\sum_{i=1}^{k} a_i) - k \leq b \qquad \forall k \in \mathcal{N} \tag{6.4}$$

As explained in Section 2.3, the total energy for $N$ frames is proportional to

$$\sum_{i=1}^{N} (\phi_0 + x_i \cdot (\phi_1 - \phi_0)) \tag{6.5}$$

This is the cost function for minimization. The problem of energy minimization is to find a policy that assigns frequencies (i.e. the value of $x_i$ for $i \in \mathcal{N}$) and execution orders (i.e. the value of $a_i$) in order to minimize the total energy, while meeting all constraints. The performance constraint is expressed by (6.2). The resource constraint is the limited buffer shown in (6.4). Finally, the precedence constraint requires that (6.3) be satisfied.

In summary, this is an integer linear programming problem (ILP). The parameters depend on the processor ($\phi_0$ and $\phi_1$), the system ($b$), and the workload ($w_p$, $w_d$, $n$, and $t$). Our goal is to minimize

$$\min \sum_{i=1}^{N} (\phi_0 + x_i \cdot (\phi_1 - \phi_0)) \tag{6.6}$$

under the following constraints:

$$\begin{aligned} & a_i \cdot w_p + w_d - t \cdot (\phi_0 + x_i \cdot (\phi_1 - \phi_0)) \leq 0 && \forall i \in \mathcal{N} \\ & \sum_{i=1}^{k} a_i - k \geq 0 && \forall k \in \mathcal{N} \\ & (\sum_{i=1}^{k} a_i) - k - b \leq 0 && \forall k \in \mathcal{N} \end{aligned} \tag{6.7}$$

While this formulation may appear as an overkill for minimizing the energy for a processor running two jobs at one of two possible frequencies, we use it as the foundation for handling more complex and realistic situations.

**Example 1** *Suppose* $t = 6$, $\phi_0 = 2$, $\phi_1 = 1$, $n = 4$, $b = 1$, $w_p = 4$, *and* $w_d = 2$. *There are four inequalities from (6.2):*

$$\begin{aligned} &4a_1 + 2 \leq 6(2 - x_1) \\ &4a_2 + 2 \leq 6(2 - x_2) \\ &4a_3 + 2 \leq 6(2 - x_3) \\ &4a_4 + 2 \leq 6(2 - x_4) \end{aligned} \tag{6.8}$$

*In order to ascertain that one frame is available for display in each period, equation (6.3) requires*

$$\begin{aligned} &a_1 \geq 1 \\ &a_1 + a_2 \geq 2 \\ &a_1 + a_2 + a_3 \geq 3 \\ &a_1 + a_2 + a_3 + a_4 \geq 4 \end{aligned} \tag{6.9}$$

*Since the buffer can accommodate only one frame, equation (6.4) restricts*

$$\begin{aligned} &a_1 - 1 \leq 1 \\ &a_1 + a_2 - 2 \leq 1 \\ &a_1 + a_2 + a_3 - 3 \leq 1 \\ &a_1 + a_2 + a_3 + a_4 - 4 \leq 1 \end{aligned} \tag{6.10}$$

*The cost function by equation (6.6) is*

$$\begin{aligned} &\min 6[(2 - x_1) + (2 - x_2) + (2 - x_3) + (2 - x_4)] \\ &\Rightarrow \min[8 - (x_1 + x_2 + x_3 + x_4)] \end{aligned} \tag{6.11}$$

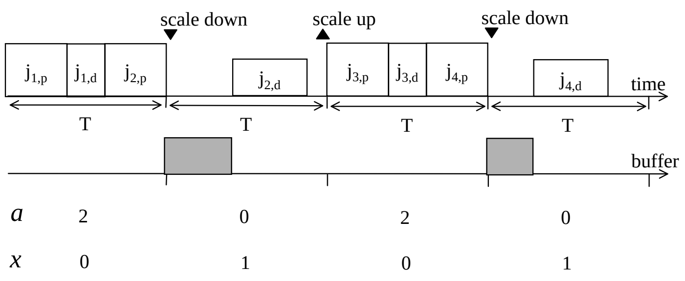


Figure 6.6: The processor changes frequencies every period.

*The minimum energy can be obtained by setting* $x_1 = x_2 = x_3 = x_4 = 1$. *The processor always stays in the lower frequency,* $\phi_1$. *One frame is processed each period:* $a_1 = a_2 = a_3 = a_4 = 1$. $\diamondsuit$

**Example 2** *Consider* $w_p = 5$. *In this case,* $x_1 = x_2 = x_3 = x_4 = 1$ *is no longer a valid solution because the constraints (6.8) and (6.9) are violated. The minimum energy can be obtained by setting* $x_1 = x_3 = 0$ *and* $x_2 = x_4 = 1$. *Two frames are processed in the first and third periods and no frame is processed in the second and fourth periods:* $a_1 = a_3 = 2$, $a_2 = a_4 = 0$. *The processor changes frequencies every period as shown in Figure 6.6.*

*Note that it is prohibited to process four frames in the first two periods by setting* $x_1 = x_2 = 0$, $x_3 = x_4 = 1$, $a_1 = a_2 = 2$, *and* $a_3 = a_4 = 0$, *because this violates the second inequality of the buffer size constraint in (6.10).*

*As a comparison, consider a processor that can continuously scale its frequencies. In this case, the minimum energy is obtained by finding a single frequency* $f_o$ *such that it takes* $t$ *to finish the operations needed for exactly one frame [43]. The value of* $f_0$ *can be calculated by the following equation.*

$$t \cdot f_o = w_p + w_d \tag{6.12}$$

$\diamondsuit$

**Example 3** *If* $w_p = 5$ *and* $w_d = 3$, *equation (6.8) is rewritten as*

$$\begin{aligned} 5a_1 + 3 &\leq 6(2 - x_1) \\ 5a_2 + 3 &\leq 6(2 - x_2) \\ 5a_3 + 3 &\leq 6(2 - x_3) \\ 5a_4 + 3 &\leq 6(2 - x_4) \end{aligned} \tag{6.13}$$

*The processor has to stay at $\phi_0$ for all four frames because the only integer solution is $x_1 = x_2 = x_3 = x_4 = 0$ and $a_1 = a_2 = a_3 = a_4 = 1$.* $\diamondsuit$

### 6.4.2 Multiple Jobs

We can generalize the formulation to handle multiple jobs. In the simplified example of an MPEG player, multiple jobs are generated by partitioning $j_p$ into smaller jobs, such as reading and decoding. These jobs follow precedence constraints: for any frame, reading must execute first, then decoding, and finally displaying. However, a frame can be decoded before the previous frame is displayed. The formulation developed in this section can be applied to any program with periodic on-time constraints. The program is divided into multiple jobs and each job executes multiple times. The precedence constraints are determined by the program.

Suppose there are $m$ jobs: $\mathcal{J} = (j_1, j_2, ..., j_m)$; $j_m$ has to execute once every period. Let $\mathcal{M}$ be $\{1, 2, ..., m\}$. We use $j_{i,l}$ for the $i^{th}$ execution of $j_l$, $l \in \mathcal{M}$. For any $l \in \{1, 2, 3, ..., m-1\}$, job $j_{i,l}$ has to execute before $j_{i,l+1}$, where $i \in \mathcal{N}$. Figure 6.7 shows the precedence relationship between these jobs. Let $w_l$ be the number of operations performed by $j_l$. Let $a_{i,l}$ be the number of executions of $j_l$ during the $i^{th}$ period; $a_{i,l} = 1$ means that $j_l$ executes once in this period. Since $j_m$ executes exactly once each period, $a_{i,m} = 1$ for any value of $i \in \mathcal{N}$. The total number of operations performed in the $i^{th}$ period is $\sum_{l=1}^{m} a_{i,l} \cdot w_l$. All operations have to finish within the period, therefore

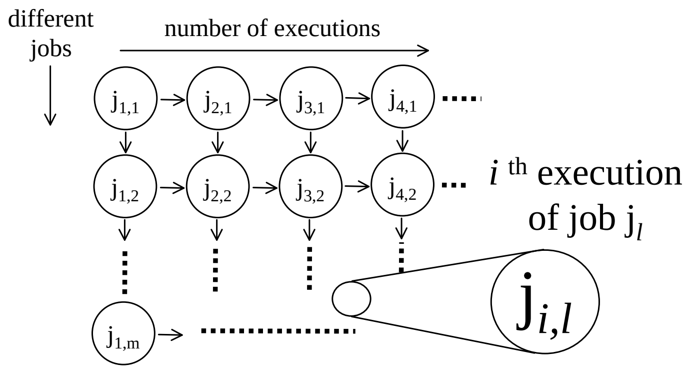


Figure 6.7: precedence of multiple jobs

$$\sum_{l=1}^{m} a_{i,l} \cdot w_l \leq t \cdot (\phi_0 + x_i \cdot (\phi_1 - \phi_0)) \qquad \forall i \in \mathcal{N} \tag{6.14}$$

The number of frames processed by job $j_l$ up to $kt$ is $\sum_{i=1}^{k} a_{i,l}$. The following constraint allows one frame to be displayed each period.

$$\sum_{i=1}^{k} a_{i,1} \geq \sum_{i=1}^{k} a_{i,2} \geq \ldots \geq \sum_{i=1}^{k} a_{i,m} = k \qquad \forall k \in \mathcal{N} \tag{6.15}$$

At time $kt$, job $j_l$ has executed $\sum_{i=1}^{k} a_{i,l}$ times and job $j_{l+1}$ has executed $\sum_{i=1}^{k} a_{i,l+1}$ times. If the former is larger, the additionally processed data are stored in a buffer. Let $b_l$ be the size of the buffer between $j_l$ and $j_{l+1}$. The following constraint avoids buffer overflow:

$$\sum_{i=1}^{k} a_{i,l} - \sum_{i=1}^{k} a_{i,l+1} \leq b_l \qquad \forall k \in \mathcal{N} \text{ and } \forall l \in \mathcal{M} - \{m\} \tag{6.16}$$

The goal is finding $x_i$ and $a_{i,l}$ to minimize energy (6.6) under the timing, precedence, and resource constraints (6.14) to (6.16).

### 6.4.3 Multiple Frequencies

Consider a processor with $s$ discrete frequencies: $\{\phi_1, \phi_2, ... ,\phi_s\}$. Let $\mathcal{S}$ be $\{1, 2, ..., s\}$. There are two ways to express the frequency during a period: (1) an integer whose value is between 1 and $s$ (2) a vector of $s$ binary elements and the $q^{th}$ element is one if and only if the frequency is $\phi_q$. We choose the second method because it simplifies the formulation when we consider scaling overhead later. Suppose $f_i$ is the frequency during the $i^{th}$ period. Let $\mathbf{y}_i$ be a vector with $s$ elements: $\mathbf{y}_i =< y_{i,1}, y_{i,2}, ..., y_{i,s} >$ to represent $f_i$. Each element is a binary variable ($y_{i,q} \in \{0, 1\}$). If the frequency is $\phi_q$, $y_{i,q} = 1$; otherwise, $y_{i,q} = 0$, here $q \in \mathcal{S}$. The value of $f_i$ can be expressed by

$$f_i = \sum_{q=1}^{s} y_{i,q} \cdot \phi_q \tag{6.17}$$

The following constraint allows the processor to be in one and only one frequency during any period.

$$\sum_{q=1}^{s} y_{i,q} = 1 \qquad i \in \mathcal{N} \tag{6.18}$$

The execution time constraint in (6.14) is rewritten as

$$\sum_{l=1}^{m} a_{i,l} \cdot w_l \leq f_i \cdot t = t \cdot (\sum_{q=1}^{s} y_{i,q} \cdot \phi_q) \qquad i \in \mathcal{N} \tag{6.19}$$

The cost function in (6.6) is rewritten as

$$\min \sum_{i=1}^{N} f_i = \sum_{i=1}^{N} \sum_{q=1}^{s} y_{i,q} \cdot \phi_q \tag{6.20}$$

**Example 4** *Consider a processor with three frequencies: $\{4, 2, 1\}$ for three jobs: $j_1$,*

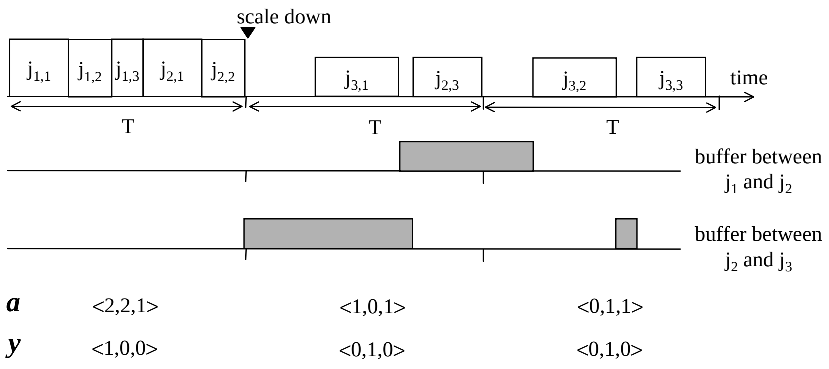


Figure 6.8: lowest-energy solution for Example4

| frequency | execution time | | | total |
|---|---|---|---|---|
| | $j_1$ | $j_2$ | $j_3$ | |
| 4 | 3 | 2 | 1 | 6 |
| 2 | 6 | 4 | 2 | 12 |
| 1 | 12 | 8 | 4 | 24 |

Table 6.2: execution time at different frequencies for Example4

*$j_2$, and $j_3$. Their numbers of operations are $w_1 = 12$, $w_2 = 8$, and $w_3 = 4$. The length of a period is 11. There are two buffers, $b_1$ and $b_2$; each can accommodate one frame. There are three frames, $n = 3$. Table 6.2 shows the execution time of each job at different frequencies. Since $t = 11$ and it takes 12 time units to execute three jobs at frequency 2, the processor must run at the highest frequency in the first period; therefore, $y_{1,1} = 1$, $y_{1,2} = y_{1,3} = 0$.*

*Figure 6.8 shows the solution for the lowest energy. In this figure, a is the sequence of $a_{i,l}$ for the $i^{th}$ period; similarly, y is the sequence of $y_{i,q}$. For example, in the first period, $a =< 2,2,1 >$; this means $a_{1,1} = a_{1,2} = 2$ and $a_{1,3} = 1$. Also in the first period, $y = \{1,0,0\}$; this indicates that $y_{1,1} = 1$ and $y_{1,2} = a_{1,3} = 0$. The frequency is $4 \cdot 1 + 2 \cdot 0 + 1 \cdot 0 = 4$. In this figure, the frequencies at the three periods are 4, 2, and 2 respectively.*

*Figure 6.9 shows another solution; in this solution, the frequencies are 4, 1, and 4. The first solution requires less energy because $4 + 2 + 2 < 4 + 1 + 4$. Notice that these two solutions have different requirements for buffers.* $\diamondsuit$

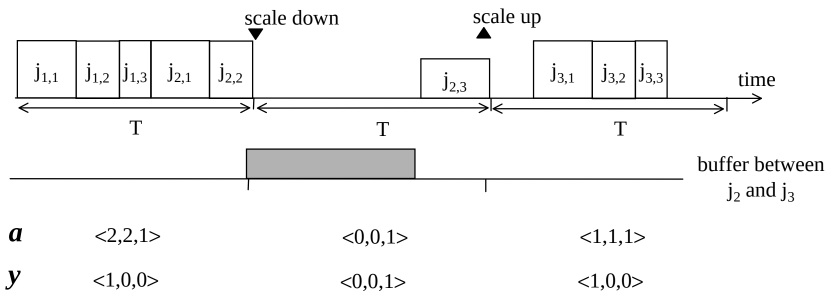


Figure 6.9: another feasible solution for Example4

## 6.4.4 Scaling Overhead

Suppose changing frequencies takes $\Delta$ time regardless of the original and new frequencies; the processor cannot execute any job while scaling frequencies. In [36], the authors report 200 microsecond for changing frequencies on a StrongARM processor. This is less than 1% of a period (33 millisecond); consequently, we assume that frequency change can finish within one period.

We refine the definition of $y_{i,q}$: $y_{i,q} = 1$ if the frequency is $\phi_q$ at the end of the $i^{th}$ period. Remember that $\mathbf{y}_i$ is a vector with $s$ binary elements to represent $f_i$. If, and only if, $\mathbf{y}_i \neq \mathbf{y}_{i-1}$, the processor changes frequencies at the beginning of the $i^{th}$ period. We use a binary variable, $\delta_i \in \{0, 1\}$, to indicate whether the processor changes frequencies at $(i-1)t$. If the frequency is changed, $\delta_i$ is one; otherwise, $\delta_i$ is zero. When the first period starts, the processor does not change frequencies; thus, we set $\mathbf{y}_0$ to be $\mathbf{y}_1$.

The value of $\delta_i$ is computed through a vector of $s$ elements: $\delta_{i,1}, \delta_{i,2}, ..., \delta_{i,s}$. Each element is a binary variable. The value of $\delta_{i,q}$ is the *exclusive or* of $y_{i-1,q}$ and $y_{i,q}$. It can be computed by the following four linear inequalities. The first two inequalities make $\delta_{i,q}$ one if $y_{i-1,q}$ and $y_{i,q}$ are different. The third inequality makes $\delta_{i,q}$ zero if both $y_{i-1,q}$ and $y_{i,q}$ are one. Finally, the fourth inequality make $\delta_{i,q}$ zero if both $y_{i-1,q}$ and $y_{i,q}$ are zero.

$$\begin{cases} \delta_{i,q} \geq y_{i,q} - y_{i-1,q} \\ \delta_{i,q} \geq y_{i-1,q} - y_{i,q} \\ \delta_{i,q} + y_{i,q} + y_{i-1,q} \leq 2 \\ \delta_{i,q} \leq y_{i,q} + y_{i-1,q} \end{cases} \qquad \forall i \in \mathcal{N}, \forall q \in \mathcal{S} \tag{6.21}$$

The value of $\delta_i$ is the sum of these $s$ elements. If $\mathbf{y}_i$ is different from $\mathbf{y}_{i-1}$, two elements of the vector are one. Consequently, $\delta_i$ needs the coefficient $\frac{1}{2}$.

$$\delta_i = \frac{1}{2}\sum_{q=1}^{s}\delta_{i,q} \tag{6.22}$$

**Example 5** *Suppose the frequency of the first three periods are $< 4, 2, 2 >$ in Example 4. The three frequencies are represented by three $\mathbf{y}$ vectors: $\mathbf{y}_1 =< 1, 0, 0 >$, $\mathbf{y}_2 =< 0, 1, 0 >$, and $\mathbf{y}_3 =< 0, 1, 0 >$. The exclusive or of $\mathbf{y}_1$ and $\mathbf{y}_2$ is $< 1, 1, 0 >$. Therefore, $\delta_2 = \frac{1}{2}(1 + 1) = 1$. The frequency is changed in the second period. The exclusive or of $\mathbf{y}_2$ and $\mathbf{y}_3$ is $< 0, 0, 0 >$; $\delta_3$ is zero and the frequency does not change in the third period.* $\diamondsuit$

During the $i^{th}$ period, $\delta_i\Delta$ is used for frequency scaling and $t - \delta_i \cdot \Delta$ is left for processing jobs. All operations have to finish in this period as expressed by the following inequality,

$$\sum_{l=1}^{m} a_{i,l} \cdot w_l \leq (t - \delta_i \cdot \Delta)(\sum_{q=1}^{s} y_{i,q} \cdot \phi_q) \qquad \forall i \in \mathcal{N} \tag{6.23}$$

The cost function is the same as (6.20).

In summary, the problem is to minimize energy for a processor with $s$ frequencies for $m$ jobs:

$$\min \sum_{i=1}^{n} \sum_{q=1}^{s} y_{i,q} \cdot \phi_q \tag{6.24}$$

under the following constraints:

$$\begin{array}{ll}
\sum_{l=1}^{m} a_{i,l} \cdot w_l - (t - \delta_i \cdot \Delta)(\sum_{q=1}^{s} y_{i,q} \cdot \phi_q) \leq 0 & \forall i \in \mathcal{N} \\
(\sum_{i=1}^{k} a_{i,l} - \sum_{i=1}^{k} a_{i,l+1}) - b_l \leq 0 & \forall k \in \mathcal{N}, \forall l \in \mathcal{M} - \{m\} \\
\sum_{i=1}^{k} a_{i,1} \geq \sum_{i=1}^{k} a_{i,2} \geq \ldots \geq \sum_{i=1}^{k} a_{i,m} = k & \forall k \in \mathcal{N} \\
\delta_i = \frac{1}{2} \sum_{q=1}^{s} \delta_{i,q} & \forall i \in \mathcal{N} \\
\begin{cases} \delta_{i,q} \geq y_{i,q} - y_{i-1,q} \\ \delta_{i,q} \geq y_{i-1,q} - y_{i,q} \\ \delta_{i,q} + y_{i,q} + y_{i-1,q} \leq 2 \\ \delta_{i,q} \leq y_{i,q} + y_{i-1,q} \end{cases} & \forall i \in \mathcal{N}, \forall q \in \mathcal{S}
\end{array} \tag{6.25}$$

**Example 6** *Consider Example 2 again with $n = 4$. If $\Delta = 0$, the minimum energy is 10 by setting the frequencies to $< 4, 1, 4, 1 >$. When $\Delta$ is nonzero, this assignment is invalid because there is insufficient time to execute both $j_1$ and $j_2$ twice in the third period after the frequency. If $\Delta = 1$, the minimum energy is 11 when the frequencies are $< 4, 1, 4, 2 >$.*

### 6.4.5 Summary

This section formulates the power reduction problem as integer linear programming. This is a general formulation to minimize the energy of a processor with finite frequencies for running a program that has timing and resource constraints. As we have shown, the formulation can consider different variations of the same problem, including multiple frequencies and scaling overhead. We can also formulate the problem to find the buffer requirements when the available energy is fixed. A large volume of literature has been devoted to solving integer programming more efficiently [14] [44] [46] [69] [85]. In the next section, we present a different approach to solve the problem based on graph-walking techniques. Our method can significantly reduce the computation for finding optimal solutions; in particular, it efficiently finds frequency assignment for many periods (large $n$). Furthermore, our method can handle sporadic jobs more easily.

## 6.5 Frequency Scaling by Graph Walking

The energy-minimization problem has additional structures that allow us to solve it more efficiently. In fact, there are only finite choices in each period; eventually, the assignments of $y$ and $a$ in (6.25) will be cyclic when $n$ is large. In this section, we explain how to find such a cycle. This section is divided into three parts. First, we construct a finite directed graph to represent all frequency assignments. Second, we show that there is a repeating subwalk in any long walk of the graph. Finally, we demonstrate how to use the graph to find frequency assignments for energy minimization.

### 6.5.1 Assignment Graph

An assignment graph contains all possible choices for frequency assignments. Each vertex encodes the current state; it contains the frequency of the processor, the amount of data stored in buffers, and how the data will be consumed or refilled. In other words, the graph represents the state space of frequency assignments.

#### Vertices

Let $G = (\mathcal{V}, \mathcal{E})$ be a directed graph: $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Each vertex encodes the states of the buffers, the frequency, and the execution of each job. A vertex is identified by a vector of $2m-1$ elements: $(\beta_1, \beta_2, ..., \beta_{m-1}, f, \alpha_1, \alpha_2, ..., \alpha_{m-1})$,

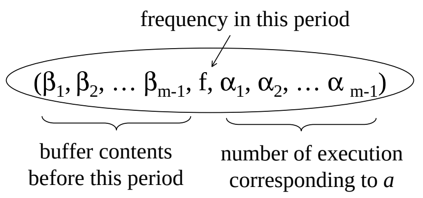

frequency in this period
(β1, β2, ... βm-1, f, α1, α2, ... α m-1)
buffer contents before this period
number of execution corresponding to a

Figure 6.10: encoding of a vertex

here $m$ is the number of jobs. Figure 6.10 illustrates the encoding of a vertex. In this encoding, $\beta_l$ ($l \in \{1, 2, ..., m-1\}$) indicates the amount of data stored in buffer $b_l$ before the period; $f$ is the frequency in this period (or the frequency after a frequency change). The value of $\alpha_l$ indicates how many times $j_l$ executes; it corresponds to $a_{i,l}$ defined in the previous section. All $\beta$'s and $\alpha$'s are integers. The value of $i$ is calculated by traversing vertices, as explained later. Each vertex $v$ has a cost $c(v)$. The cost is the frequency of the vertex, or $c(v) = f$; all costs are positive. Since a vertex represents a period, we can use "vertex" and "period" interchangeably.

**Example 7** *For Example 2, the processor has two frequencies so $f$ can be 1 or 2. The buffer can be filled or empty, so $\beta$ is 0 or 1. If the processor is at frequency 1, $j_1$ and $j_2$ cannot execute within one period; consequently, $\alpha$ is zero when $f$ is 1. The assignment graph includes five vertices:* $(1, 2, 0)$, $(1, 1, 0)$ $(0, 2, 1)$, $(1, 2, 1)$, *and* $(0, 2, 2)$. $\diamondsuit$

Now, we compute an upper bound of the number of vertices. First, we consider the possible values of $\beta_1$. Before entering a period, the buffer between $j_1$ and $j_2$ may have zero, one, two, ..., up to $b_1$ items; there are $b_1 + 1$ possible values for $\beta_1$. Similarly, $\beta_2$ has $b_2 + 1$ possible values and $\beta_l$ has $b_l + 1$ possible values. The processor has $s$ frequencies so $f$ has $s$ choices. At frequency $\phi_1$, job $j_1$ can execute at most

$$\lfloor \frac{t \cdot \phi_1 - w_m}{w_1} \rfloor \tag{6.26}$$

| | |
|---|---|
| $c(v)$ | cost of vertex $v$, the frequency of $v$ |
| $v_1 \Rightarrow v_2$ | $v_1$ and $v_2$ are connected, or $(v_1, v_2) \in \mathcal{E}$ |
| $\mathcal{W}$ | a walk, general format: $< v_1, v_2, ...., v_n >$ |
| $c(\mathcal{W})$ | cost of walk $\mathcal{W}$ |
| $\mathcal{W}_n(v)$ | a minimum-cost walk from $v$ and visits $n$ vertices |
| $\mathcal{W}(v_a, v_b)$ | a minimum-cost walk between vertices $v_a$ and $v_b$ |
| $\star$ | concatenate two walks |
| $\varphi(v)$ | unused operation of vertex $v$ |

Table 6.3: symbols and meanings for assignment graphs

times. This is an upper bound for $\alpha_1$[1]; we call it $\overline{\alpha_1}$. The value of $\alpha_1$ is between zero and $\overline{\alpha_1}$ so there are $\overline{\alpha_1} + 1$ options. We can find upper bounds for other $\alpha$'s in the same way. The following formula is an upper bound of the size of an assignment graph.

$$\begin{aligned}(b_1+1) \times (b_2+1) \times ... \times (b_{m-1}+1) \times s \times (\overline{\alpha_1}+1) \times (\overline{\alpha_2}+1) ... \times (\overline{\alpha_{m-1}}+1) \\ = s \times \prod_{l=1}^{m-1} (b_l+1)(\overline{\alpha_l}+1)\end{aligned} \tag{6.27}$$

This is a loose upper bound because we have not removed invalid vertices. There are three types of invalid vertices; they violate timing, resource, or precedence constraints.

**Example 8** *For Example 4, at frequency* $1$, $j_1$ *and* $j_2$ *cannot execute in a single period because this violates the timing constraint specified by (6.19). The vertex* $(\bullet, \bullet, 1, 1, 1)$ *is invalid regardless of the value for* $\bullet$. *For the same example, vertex* $(0, 1, 4, 2, 2)$ *starts with one frame in buffer* $b_2$ *and executes* $j_2$ *twice. Since only one frame is consumed by* $j_3$*, two frames have to be stored in buffer* $b_2$ *at the end of this period. However,* $b_2$ *can store only one frame; therefore,* $(0, 1, 4, 2, 2)$ *overflows the buffer and is an invalid vertex. Vertex* $(0, 0, 4, 0, 2)$ *violates the precedence constraint because* $j_2$ *executes twice but the buffer between* $j_1$ *and* $j_2$ *is empty and* $j_1$ *does not execute.* $\diamondsuit$

[1]It is an upper bound because the range for $\alpha_1$ may reduce at a lower frequency.

Timing constraints require the processor to operate at a frequency high enough to finish all scheduled jobs. The timing constraint of vertex $(\beta_1, \beta_2, ..., \beta_{m-1}, f, \alpha_1, \alpha_2, ..., \alpha_{m-1})$ is specified below; this is equivalent to the constraint specified in (6.19).

$$\sum_{l=1}^{m} \alpha_l \cdot w_l \leq f \cdot t \tag{6.28}$$

Resource constraints state that buffers cannot overflow. Before entering this period, there are $\beta_l$ items (or frames) in the buffer between $j_l$ and $j_{l+1}$. In this period, $j_l$ executes $a_l$ times and $j_{l+1}$ executes $a_{l+1}$ times. Before leaving this period, $\beta_l + \alpha_l - \alpha_{l+1}$ items must be stored in this buffer and it cannot exceed the buffer size. This it is equivalent to the constraint specified in (6.16):

$$\beta_l + \alpha_l - \alpha_{l+1} \leq b_l \qquad l \in \mathcal{M} - \{m\} \tag{6.29}$$

We define $\alpha_m$ as one because one frame is displayed each period. Finally, precedence constraints prevent buffer underflow. Since $j_{l+1}$ executes $\alpha_{l+1}$ times, there must be enough data either from the buffer or produced by $j_l$. We rewrite the constraint in (6.15) for the vertices in the assignment graph:

$$\beta_l + \alpha_l \geq \alpha_{l+1} \qquad l \in \mathcal{M} - \{m\} \tag{6.30}$$

Removing invalid vertices takes linear time, $O(\mathcal{V})$, because each vertex has to be examined only once.

**Example 9** *In Example 4, either buffer can accommodate one item, so* $(b_1 + 1) = (b_2 + 1) = 2$*. There are three frequencies. Job* $j_1$ *can execute at most* $\frac{11 \times 4 - 4}{12} = 3$ *times and* $j_2$ *can execute at most* $\frac{11 \times 4 - 4}{8} = 5$ *times. Therefore, the graph has at most* $2 \times 2 \times 3 \times 4 \times 6 = 288$ *vertices by equation (6.27). After removing invalid vertices, there are only 21 valid vertices. The graph for Example 2 has at most* $2 \times 2 \times 3 = 12$

*vertices by (6.27); it actually has 5 valid vertices.* $\diamondsuit$

### Starting Vertices

Since all buffers are empty at the beginning, the first $m-1$ elements in the encoding must be zero. Let $(0, 0, ..., 0, f, \alpha_1, \alpha_2, ..., \alpha_{m-1})$ be the first vertex; job $j_l$ executes $\alpha_l$ times. The values for $f$ and $\alpha$'s have to satisfy the following conditions; these are conditions in (6.25) except that $\delta$ is always zero for the first period.

$$\begin{aligned}
&f \cdot t \geq w_m + \sum_{l=1}^{m-1} \alpha_l \cdot w_l \\
&\alpha_l - \alpha_{l+1} \leq b_l \qquad\qquad l \in \{1, 2, ..., m-2\} \\
&\alpha_1 \geq \alpha_2 \geq ... \geq \alpha_{m-1} \geq 1
\end{aligned} \tag{6.31}$$

Any vertex that satisfies these conditions can be a starting vertex. After removing invalid vertices, any vertex with the $(0, 0, ..., 0, f, \alpha_1, \alpha_2, ..., \alpha_{m-1})$ format is a valid starting vertex.. We will use $v^*$ as one starting vertex. If a vertex cannot be reached from any starting vertices, this vertex is eliminated from the assignment graph.

### Edges

An edge $(v_1, v_2) \in \mathcal{E}$ connects two vertices $v_1$ and $v_2$; it indicates a possible transition from state $v_1$ to another state $v_2$ after one period. A transition from $v_1$ to $v_2$ is represented as $v_1 \Rightarrow v_2$. We call $v_1$ a *precedessor* of $v_2$ and $v_2$ a *successor* of $v_1$. There is at most one edge between two vertices.

Some transitions are prohibited. There are two types of invalid transitions. The first type violates continuity conditions. Suppose $v_1 = (\beta_1, \beta_2, ..., \beta_{m-1}, f, \alpha_1, \alpha_2, ..., \alpha_{m-1})$, $v_2 = (\beta'_1, \beta'_2, ..., \beta'_{m-1}, f', \alpha'_1, \alpha'_2, ..., \alpha'_{m-1})$, and $v_1 \Rightarrow v_2$. The continuity conditions require that the data stored in the buffers remain the same after leaving $v_1$ and before entering $v_2$. Before entering $v_1$, there are $\beta_l$ items in the $l^{th}$ buffer. During $v_1$, $\alpha_l$ more items are added to the buffer and $\alpha_{l+1}$ of these items are consumed by job $j_{l+1}$. Consequently, there are $\beta_l + \alpha_l - \alpha_{l+1}$ items left before entering $v_2$. The following formula expresses this continuity condition:

$$\beta_l' = \beta_l + \alpha_l - \alpha_{l+1} \qquad l \in \{1, 2, ..., m-1\} \tag{6.32}$$

The second type violates timing constraints. Consider Example 6 that includes scaling overhead. It takes one time unit to change the frequencies; therefore, $j_1$ and $j_2$ cannot execute twice at frequency 4 if the previous frequency is different from 4. In other words, $(0,1,1,0,0) \Rightarrow (0,0,4,2,2)$ is an invalid transition.

**Example 10** *In Example 4,* $(0,0,4,2,2) \Rightarrow (1,0,2,0,1)$ *is an invalid transition. Before visiting the first vertex, buffer* $b_1$ *is empty. The first vertex executes both* $j_1$ *and* $j_2$ *twice so* $b_1$ *is still empty. However, the following vertex starts with non-empty buffer* $b_1$*. This violates continuity principle.*

*The assignment graph for Example 4 has 21 vertices and 106 edges. Each vertex has 5 edges in average. For Example 6, the graph has 21 vertices and 100 edges. The second example has fewer edges due to the scaling overhead. For Example 4,* $(1,0,2,0,1) \Rightarrow (0,0,4,2,2)$ *is a valid transition, but it is invalid in Example 6.* $\diamondsuit$

## Merging Vertices

After constructing the graph, we can further reduce its size by merging vertices. Two vertices can merge if they have identical predecessors and successors. This happens when two vertices differ only in their frequencies; the merged vertex uses the lower frequency because it suffices to use the lower frequency. For example, $(0,1,2,0,0)$ can merge with $(0,1,1,0,0)$ in Example 4. Since they have the same inward and outward edges, they perform identical operations. Consequently, it is unnecessary to use a higher frequency if a lower frequency is sufficient.

**Example 11** *The assignment graph for Example 4 has 14 vertices and 50 edges after merging equivalent vertices. In this example, vertex merging reduces the number of vertices by one third and the number of edges by more than a half. The graph for Example 6 has 21 vertices and 100 edges before the merge and 18 vertices and 77 edges after the merge.* $\diamondsuit$

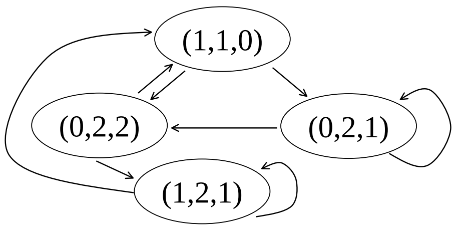


Figure 6.11: assignment graph for Example 2

**Example 12** *Figure 6.11 shows the assignment graph for Example 2 after removing invalid vertices and merging equivalent vertices. This graph has one vertex with frequency* 1 *and three vertices with frequency* 2*. Vertex* $(0, 2, 1)$ *can reach itself; this means the processor can keep running at frequency* 2 *and executing* $j_p$ *once every period. All successors of* $(1, 1, 0)$ *have frequency* 2*. This means that the processor can run at frequency* 1 *for only one period; then, it has to run at frequency* 2 *for at least one period before entering* $(1, 1, 0)$ *again.* $\diamondsuit$

### Walks

A *walk* $\mathcal{W}$ of a graph is a sequence of vertices $\mathcal{W} =< v_1, v_2, ...., v_n >$ such that $(v_i, v_{i+1}) \in \mathcal{E}$ for any $i \in \{1, 2, ..., n - 1\}$. A walk is a sequence of assignments of frequencies $(f)$ and executions $(\alpha)$ by the vertices. Walk $\mathcal{W}$ *visits* a vertex $v$ if $v$ appears in the sequence. Vertices $v_2$, $v_3$, ... , $v_{n-1}$ are *intermediate* vertices in the walk. We define the length of a walk as the number of vertices in the sequence[2], or $n$. A *closed walk* of $v_1$ starts and ends at the same vertex: $v_1 = v_n$ [8] [24]. If $(v_i, v_i) \in \mathcal{E}$, the walk $< v_i, v_i >$ is called a *loop*. A *subwalk* is a walk contained in a longer walk; for example, $< v_i, v_{i+1}, ..., v_j >$ is a subwalk of $< v_1, v_2, ...., v_n >$ if $1 \leq i \leq j \leq n$. This subwalk starts from $v_i$, ends at $v_j$, and visits $j - i + 1$ vertices. A walk is a *path* if all vertices are distinct [8] [24]. Graph walking has been applied to a wide range of problems, such as finding the resistance in an electric network and the locations for servers [21] [86]. Two walks can be concatenated. We use $\star$ as the concatenation operator: walk $W_1 =< v_1, v_2, ..., v_n >$ is concatenated with walk $W_2 =< u_1, u_2, ..., u_m >$, written as $W_1 \star W_2$. The result is a longer walk,

[2]Some texts use the number of edges as the length of a walk.

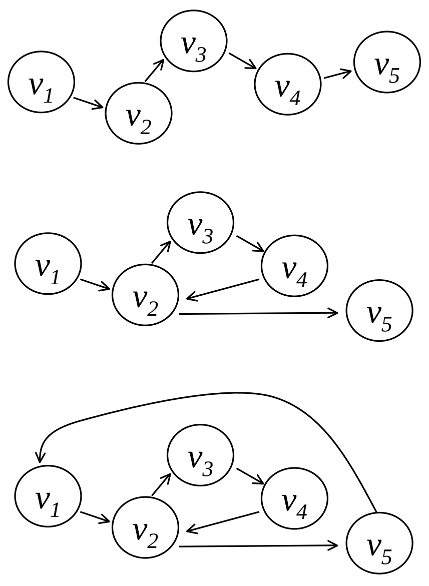


Figure 6.12: examples of walks

$W_1 \star W_2 =< v_1, v_2, ..., v_n, u_1, u_2, ..., u_m >$. They can concatenate if $(v_n, u_1) \in \mathcal{E}$.

**Example 13** *Figure 6.12 are three examples of walks. The first is a walk from $v_1$ to $v_5$. The second walk, $< v_1, v_2, v_3, v_4, v_2, v_5 >$ contains a closed walk, $< v_2, v_3, v_4, v_2 >$. The third walk $< v_1, v_2, v_3, v_4, v_2, v_5, v_1 >$ contains two closed walks*[3]*, one starting from $v_1$ and the other starting from $v_2$.* $\diamondsuit$

Figure 6.11 has a loop of vertex $(0, 2, 1)$. This is not incidental. Because we assume jobs can be scheduled at the highest frequency, there is always a loop of vertex $(0, 0, ...., \phi_1, 1, 1, ...)$ here $\phi_1$ is the highest frequency. This is equivalent to the pipeline without a buffer as shown in Figure 6.1. This loop means executing each job once per period and storing no additional data in the buffers.

## Cost of a Walk

The cost of a walk is the sum of the cost of each vertex. If a vertex is visited twice, the cost is added twice. The cost of $\mathcal{W}$ is

[3]We assume the walk stops after visiting $v_1$ twice

$$c(\mathcal{W}) = \sum_{i=1}^{n} c(v_i) \tag{6.33}$$

The average cost of a walk is defined as

$$\frac{c(\mathcal{W})}{n} = \frac{1}{n}\sum_{i=1}^{n} c(v_i) \tag{6.34}$$

A minimum-cost walk can be defined for two different conditions:

1. a walk starting from a given vertex ($v_1$) and visiting a given number ($n$) of vertices. This walk is represented as $\mathcal{W}_n(v_1) =< v_1, v_2, ..., v_n >$. The ending vertex ($v_n$) is not specified.

2. a walk starting from a given vertex ($v_a$) and ending at another given vertex ($v_b$). We use $\mathcal{W}(v_a, v_b) =< v_a, v_1, ..., v_n, v_b >$ to represent such a walk. The number of visited vertices is not specified.

The costs of these two types of walks are written as $c(\mathcal{W}_n(v_1))$ and $c(\mathcal{W}(v_a, v_b))$.

**Example 14** *Figure 6.11 has several walks, including*

- *two loops:* $(0,2,1) \Rightarrow (0,2,1)$ *and* $(1,2,1) \Rightarrow (1,2,1)$
- $(1,1,0) \Rightarrow (0,2,2) \Rightarrow (1,1,0)$
- $(1,1,0) \Rightarrow (0,2,1) \Rightarrow (0,2,2)$
- $(1,1,0) \Rightarrow (0,2,2) \Rightarrow (1,2,1)$

*Among these walks,* $(1,1,0) \Rightarrow (0,2,2) \Rightarrow (1,1,0)$ *has the minimum average cost.* $\diamondsuit$

### 6.5.2 Energy Minimization by Assignment Graphs

#### Minimum-Cost Subwalks

Because a walk is an assignment of frequencies, a minimum-cost walk is equivalent to an assignment that minimizes the energy consumption. This section finds a minimum-cost walk for $n$ periods, namely visiting $n$ vertices.

**Theorem 1** *Suppose $\mathcal{W}_n(v_1) =< v_1, v_2, ..., v_n >$ is a minimum-cost walk from $v_1$ and visits $n$ vertices. A subwalk $< v_i, v_{i+1}, ..., v_j >$ of $\mathcal{W}_n(v_1)$ is a minimum-cost walk from $v_i$ to $v_j$ and visits $j - i + 1$ vertices.*

This can be proven by contradiction. The cost of $\mathcal{W}_n(v_1)$ is $\sum\limits_{k=1}^{n} c(v_k)$, or $\sum\limits_{k=1}^{i-1} c(v_k) + \sum\limits_{k=i}^{j} c(v_k) + \sum\limits_{k=j+1}^{n} c(v_k)$. If the walk $< v_i, v_{i+1}, ..., v_j >$ is not minimum-cost, then we can find another walk $< v_i, v'_{i+1}, ..., v'_{j-1}, v_j >$ that starts from $v_i$, ends at $v_j$, visits the same number of vertices, and has a lower cost: $\sum\limits_{k=i+1}^{j-1} c(v'_k) < \sum\limits_{k=i+1}^{j-1} c(v_k)$. Replacing $< v_i, v_{i+1}, ..., v_j >$ by $< v_i, v'_{i+1}, ..., v'_{j-1}, v_j >$ will reduce the cost of the original walk $\mathcal{W}_n(v_1)$; this contradicts the premise that $\mathcal{W}_n(v_1)$ is a minimum-cost walk. Therefore, $< v_i, v_{i+1}, ..., v_j >$ is a minimum-cost walk from $v_i$ to $v_j$ and visits $j - i + 1$ vertices. $\diamondsuit$

This theorem is similar to shortest subpaths for computing a shortest path between two vertices in a graph [22]. A minimum-cost walk differs from a shortest path in three ways:

- It specifies the number of vertices, not the ending vertex
- The cost is determined by vertices in formula (6.33), not the weights on edges.
- It allows visiting the same vertex multiple time.

The methods presented in [22] compute shortest paths between two given vertices. Because the power-reduction problem is different, we approach this problem by modifying the methods in [22].

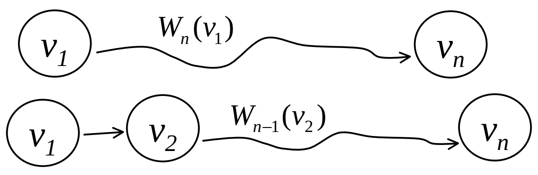


Figure 6.13: divide $\mathcal{W}_n(v_1)$ into two subwalks

### Finding Subwalk Recursively

Suppose $\mathcal{W}_n(v_1)$ is $< v_1, v_2, ..., v_n >$, then $c(\mathcal{W}_n(v_1)) = c(v_1) + c(v_2) + ... + c(v_n)$. By definition, $\mathcal{W}_1(v_1)$ is $< v_1 >$ and $c(\mathcal{W}_1(v_1))$ is $c(v_1)$. For $\mathcal{W}_2(v_1)$, the minimum cost is to find vertex $v_2$ such that $(v_1, v_2) \in \mathcal{E}$ and $c(v_1) + c(v_2)$ is the minimum:

$$\min c(\mathcal{W}_2(v_1)) = \min_{v_2 \in \mathcal{V}, (v_1,v_2) \in \mathcal{E}} c(v_1) + c(v_2) \tag{6.35}$$

Similarly, the minimum cost for $\mathcal{W}_3(v_1))$ can be found by

$$\min c(\mathcal{W}_3(v_1)) = \min{}^{(v_1,v_2)\in\mathcal{E}}_{(v_2,v_3)\in\mathcal{E}} c(v_1) + c(v_2) + c(v_3) \tag{6.36}$$

We can divide $\mathcal{W}_n(v_1) =< v_1, v_2, ..., v_n >$ into two walks: $< v_1 >$ and $< v_2, v_3, ..., v_n >$, here $(v_1, v_2) \in \mathcal{E}$. The cost of $\mathcal{W}_n(v_1)$ can be computed by

$$\begin{aligned} c(\mathcal{W}_n(v_1)) &= \sum_{i=1}^{n} c(v_i) \\ &= c(v_1) + \sum_{i=2}^{n} c(v_i) \\ &= c(v_1) + c(\mathcal{W}_{n-1}(v_2)) \qquad (v_1, v_2) \in \mathcal{E} \end{aligned} \tag{6.37}$$

Figure 6.13 illustrates this concept. This is a recursive relation; each time, we reduce the length of the walk by one vertex. In this recursion, $v_2$ is used to reduce the length of walk $\mathcal{W}_n(v_1)$. Equation (6.37) computes $c(\mathcal{W}_n(v_1))$ by reducing the length of the walk through the recursive relation. Since there may be multiple choices for

$v_2$, it takes exponential time to find a minimum-cost walk by equation (6.37) [35]: $O(c^n)$, $c$ is the average number of successors of each vertex and $c \geq 1$. We need a more efficient method.

### Memorization of Subwalks

We can reduce the time complexity by memorizing shorter walks to construct long walks. In other words, if we already know the minimum-cost walk $\mathcal{W}_{n-1}(v_2)$, there is no need to compute it again, based on the principle of optimal subwalks proved in Section 6.5.2. Memorization eliminates computing the same subwalks multiple times. The algorithm in Figure 6.14 computes a minimum-cost walk of $\mathcal{W}_n(v_1)$ by memorizing shorter walks. For each iteration of *wlength*, at most $|\mathcal{V}|^2$ vertices are visited; the execution time is $O(n|\mathcal{V}|^2)$. Memorization reduces the complexity from exponential of $n$ to linear of $n$.

## 6.5.3 Efficient Assignments

Even though MinimumCostWalk has complexity $O(n|\mathcal{V}|^2)$, there are still two problems. First, the time is linear of $n$ even though the graph size is independent of $n$. Second, it computes $\mathcal{W}_i(v)$ for every value of $i$ while we are interested in $i = n$ only. Because assignment graphs are finite, we can compute minimum-cost walks even more efficiently for large $n$. This section explains how to find minimum-cost walks efficiently when $n \gg |\mathcal{V}|$.

### Minimum-Cost Walks between Two Vertices

Based on the Floyd-Warshall algorithm for finding shortest paths in a graph [22], we can find a minimum-cost walk between vertex $v_a$ and vertex $v_b$. The algorithm is called MinimumCostWalk2V as shown in Figure 6.15. This algorithm has complexity $O(|\mathcal{V}|^3)$ because of the nested iteration.

```
MinimumCostWalk(**input** graph: $G = (\mathcal{V}, \mathcal{E})$, integer: $n$)
/*  $\mathcal{W}_i(v)$: minimum-cost walk from $v$ with length $i$
    $wcost(v, i)$: cost of $\mathcal{W}_i(v)$
    $n$: maximum length of walks */
**begin**
    /* initialization */
    **for each** $v \in \mathcal{V}$
        $wcost(v, 1) := c(v)$;
        $\mathcal{W}_1(v) := < v >$;

    **for** $(wlength := 2; wlength \leq n; wlength ++)$
        **for each** $v_1 \in \mathcal{V}$
            $wcost(v_1, wlength) := \infty$; /* initialize */
            **for each** $v_2 \in \mathcal{V}$ **and** $(v_1, v_2) \in \mathcal{E}$
            $newcost := c(v_1) + wcost(v_2, wlength - 1)$;
            **if** $(wcost(v_1, wlength) > newcost)$
                $wcost(v_1, wlength) := newcost$;
                $\mathcal{W}_{wlength}(v_1) := < v_1 > \star \mathcal{W}_{wlength-1}(v_2)$;
                /* $\star$: concatenation of two walks */
**end**
```

Figure 6.14: find minimum-cost walks by formula (6.37)

## Pigeonhole Principle

Suppose there are $n$ pigeons and $m$ holes. We want to assign these pigeons to the holes. If there are more pigeons than holes ($n > m$), at least one hole must have two or more pigeons. This is called the *pigeonhole principle* [33].

Suppose there are $m$ balls labeled as $1, 2, ..., m$ stored in a box. Every minute, we select one ball from the box, record its number, and put it back to the box. After $n$ minutes, we have seen $n$ balls. If $n > m$, one number between 1 and $m$ must occur two or more times, according to the pigeonhole principle.

If a walk visits $n$ vertices and $n$ is larger than the number of vertices in the graph ($n > |\mathcal{V}|$), at least one vertex must be visited twice or more: there is at least one closed walk.

## Redefining Closed Walk

A closed walk has the format $< v_1, v_2, ..., v_n >$ where $v_n = v_1$. In the rest of this chapter, we restrict closed walks so that $v_1$ is visited exactly twice and no other vertex is visited twice or more: $v_i \neq v_j$ if $1 \leq i < j \leq n$ except $i = 1$ and $j = n$. We call such closed walks $\mathcal{CWALK}$s. According to the pigeonhole principle, any closed walk in $G = (\mathcal{V}, \mathcal{E})$ visits at most $|\mathcal{V}| + 1$ vertices ($v_1$ is counted twice).

**Example 15** *In Figure 6.12, the third walk contains two closed walks; specifically, $< v_1, v_2, v_3, v_4, v_2, v_5, v_1 >$ is a closed walk and it contains another closed walk $< v_2, v_3, v_4, v_2 >$. With the new restriction, no closed walk may contain another closed walk. Consequently, the closed walk $< v_1, v_2, v_3, v_4, v_2, v_5, v_1 >$ is no longer a valid closed walk.* $\diamondsuit$

Figure 6.16 is an algorithm for finding all $\mathcal{CWALK}$s that have minimum average costs. The average cost of a walk is defined by Equation (6.34). If two closed walk have the same average cost, the algorithm keeps the shorter one. This algorithm first finds all minimum-cost walks of lengths up to $|\mathcal{V}| + 1$. Then, it determines whether the walk is closed and computes the average cost; finally, it keeps only closed walks with minimum average costs. Since the jobs can be scheduled, there is at least one

```
MinimumCostWalk2V(**input** graph: $G = (\mathcal{V}, \mathcal{E})$)
/*  $\mathcal{W}(v_a, v_b)$: minimum-cost walk from $v_a$ to $v_b$
    $\mathcal{W}^-(v_a, v_b)$: $\mathcal{W}(v_a, v_b)$ without visiting the last vertex, $v_b$
        if $\mathcal{W}(v_a, v_b) =< v_a, v_1, v_2, ..., v_n, v_b >$ then
        $\mathcal{W}^-(v_a, v_b) =< v_a, v_1, v_2, ..., v_n >$
    $mcost(v_a, v_b)$: cost of $\mathcal{W}(v_a, v_b)$
    $n\mathcal{W}(v_a, v_b)$: number of vertices in $\mathcal{W}(v_a, v_b)$ */
**begin**
    **for each** $v_a \in \mathcal{V}$
        **for each** $v_b \in \mathcal{V}$
            **if** $(v_a, v_b) \in \mathcal{E}$
                $\mathcal{W}(v_a, v_b)$: $= < v_a, v_b >$;
                $mcost(v_a, v_b)$: $= c(v_a) + c(v_b)$;
                $n\mathcal{W}(v_a, v_b)$: $= 2$;
            **else**
                $\mathcal{W}(v_a, v_b)$: $= <>$;
                $mcost(v_a, v_b)$: $= \infty$;
                $n\mathcal{W}(v_a, v_b)$: $= \infty$;
    **for each** $v_c \in \mathcal{V}$ /* intermediate vertex */
        **for each** $v_a \in \mathcal{V}$
            **for each** $v_b \in \mathcal{V}$
                $costpassc := mcost(v_a, v_c) + mcost(v_c, v_b) - c(v_c)$;
                /* subtract $c(v_c)$ because it is counted twice */
                **if** $(mcost(v_a, v_b) > costpassc)$
                    $mcost(v_a, v_b) := costpassc$;
                    $n\mathcal{W}(v_a, v_b) := n\mathcal{W}(v_a, v_c) + n\mathcal{W}(v_c, v_b) - 1$;
                    $\mathcal{W}(v_a, v_b)$: $= \mathcal{W}^-(v_a, v_c) \star \mathcal{W}(v_c, v_b)$;
**end**
```

Figure 6.15: find minimum-cost walks between two vertices

```
FindClosedWalk(input graph: $G = (\mathcal{V}, \mathcal{E})$)
/* $\mathcal{CWALK}(v)$: closed walk of $v$ with minimum average cost
   $cwcost(v)$: the cost of $\mathcal{CWALK}(v)$
   $nwalk(v)$: the length of $\mathcal{CWALK}(v)$ */
begin
   MinimumCostWalk($G$, $|\mathcal{V}| + 1$);
   for each $v \in \mathcal{V}$
      /* initialization */
      $\mathcal{CWALK}(v)$ := empty;
      $cwcost(v) := \infty$;

      for ($wlength := 2$; $wlength \leq |\mathcal{V}| + 1$; $wlength$ ++)
         $v' :=$ last vertex of $\mathcal{W}_{wlength}(v)$;
         if $(\frac{cwcost(v)}{nwalk(v)} > \frac{wcost(v,wlength)}{wlength})$ /* smaller average cost */
            if ($v = v'$) /* a closed walk */ and
            ($\mathcal{W}_{wlength}(v)$ visits $v$ exactly twice) and
            (no other vertex is visited more than once)
               $\mathcal{CWALK}(v) := \mathcal{W}_{wlength}(v)$;
               $cwcost(v) := wcost(v, wlength)$;
               $nwalk(v) := wlength$;
            else
               $\mathcal{CWALK}(v)$ := empty;
               $cwcost(v) := \infty$;
end
```

Figure 6.16: find closed walks of minimum average costs

trivial solution: a loop of vertex $(0, 0, ...., \phi_1, 1, 1, ...)$, here $\phi_1$ is the highest frequency. Since $n = |\mathcal{V}| + 1$, it takes $O(|\mathcal{V}|^3)$ for calling Minimum-Cost Walk. For each vertex, $wlength$ changes from 2 to $|\mathcal{V}| + 1$ and it takes $wlength$ to compute the average cost of $\mathcal{W}_{wlength}(v)$. It takes $O(|\mathcal{V}|^2)$ time for each vertex. Hence, this algorithm takes $O(|\mathcal{V}|^3)$ time.

## Walks of Infinite Length

After finding the minimum-cost $\mathcal{CWALK}$s, it is easy to find an infinite-length walk with the minimum average cost. When $n$ approaches infinity, a minimum-cost walk

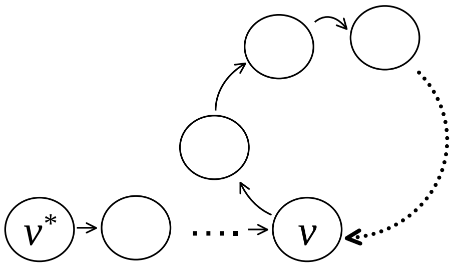


Figure 6.17: A walk of infinite length must repeat a closed walk indefinitely.

starts from one starting vertex, defined in Section 6.5.1, reaches a closed walk and repeats this closed walk. Figure 6.17 illustrates such a walk. This closed walk is chosen because it has the minimum average cost, defined as

$$\min_{v \in \mathcal{V}} \frac{cwcost(v)}{nwalk(v)} \tag{6.38}$$

If the vertex $v$ is not a starting vertex, we can find a minimum-cost walk that connect one $v^*$ to $v$ by MinimumCostWalk2V. Since FindClosedWalk takes $O(|\mathcal{V}|^3)$, it takes $O(|\mathcal{V}|^3)$ to find a walk of infinite length with the minimum average cost. Because the walk is infinite, the "initial cost" from $v^*$ to $v$ can be ignored. A natural question is whether this closed walk is reachable from a starting vertex. Since the jobs can be scheduled at the highest frequency, there must be an infinite walk available. In particular, there exists one trivial solution by taking the loop of the vertex $(0, 0, ...., \phi_1, 1, 1, ...)$, here $\phi_1$ is the highest frequency. There may be other solutions that satisfy all constraints and require lower power consumptions. Our method finds these solutions with time complexity $O(|\mathcal{V}|^3)$; this is independent of $n$.

From the pigeonhole principle, frequency assignments must form a closed walk for a workload with an infinite time horizon. We need to emphasize that our method does not have to know the length of the closed walk in advance. In contrast, using integer linear programming (ILP) has to determine this length in advance and select $n$ large enough for the equations and inequalities in (6.25).

Since a typical MPEG movie contains thousands of frames, it is reasonable to

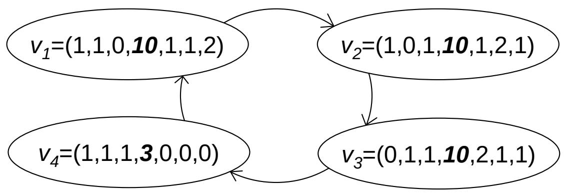

$v_1$=(1,1,0,**10**,1,1,2)
$v_2$=(1,0,1,**10**,1,2,1)
$v_3$=(0,1,1,**10**,2,1,1)
$v_4$=(1,1,1,**3**,0,0,0)

Figure 6.18: the walk for four jobs with 80% processor utilization

approximate it with infinite frames. In reality, no movie can have infinite frames. Appendix C explains how to find a long but finite walk that has the minimum cost. A finite walk is different in three ways: (1) the initial cost needs to be considered (2) it has a "tail" that may not be a complete closed walk (3) the cost of the tail needs to be considered.

**Example 16** *A processor has five frequencies: 10, 7, 5, 4, and 3; a program keeps the processor 80% utilized at frequency 10. There are four jobs with equal numbers of operations: each job takes 20% time in a period. Without any buffer, the processor is idle 20% time in each period. If there is one buffer between two jobs, a low-power schedule is presented in Figure 6.18. In each period at frequency 10, one of the buffers is filled; then, the processor runs at frequency 3 to reduce energy. The average frequency is 8.25, or 3% above optimal. This walk also shows that some frequencies (7,5, and 4) are not used.* ◇

## 6.6 Respone Time of Sporadic Jobs

The previous section considered periodic jobs and showed how frequency scaling and data buffering can reduce the energy consumption. The optimal assignments of frequencies are determined by a graph-based algorithm. In this section, we explain how to compute the response time of a sporadic job in the presence of periodic jobs. We show how to calculate the response time of a sporadic job if it arrives at a period represented by a vertex in a walk. For simplicity, this section ignores scaling overhead. We also assume that a sporadic job completes before another sporadic job arrives.

The following scenario is an example to illustrate the mixture of periodic and sporadic jobs. When a user is watching an MPEG movie, the movie creates periodic jobs. Occasionally, the user may move the mouse to a slider and adjust the volume; this movement creates a sporadic job. A desirable outcome consists of three parts (1) the sporadic job is processed promptly (2) the frame rate of the MPEG movie remains constant (3) the power consumption is minimized.

When a sporadic job arrives, the processor has to execute additional operations. These operations can be executed in two ways. First, the sporadic job is executed with only the "spare" operations in each period. Alternatively, if buffers are nonempty, data can be retrieved from the buffers and some jobs do not have to execute. By draining the buffers, the sporadic job can finish earlier.

### 6.6.1 Unused Operations

Some time periods may have "unused" operations because these operations are not used to execute any of job $j_1$ to $j_{m-1}$. We use $\varphi(v)$ as the number of unused operations of vertex $v$. For vertex $v = (\beta_1, \beta_2, ..., \beta_{m-1}, f, \alpha_1, \alpha_2, ..., \alpha_{m-1})$, $\varphi(v)$ can be found by the following formula

$$\varphi(v) = t \cdot f - (w_m + \sum_{l=1}^{m-1} \alpha_l \cdot w_l) \tag{6.39}$$

In this formula, $t \cdot f$ is the total number of allowed operations and $(w_m + \sum_{l=1}^{m-1} \alpha_l \cdot w_l)$ is the number of operations needed to execute the jobs required in this period. Their difference determines how many additional operations can be conducted in this period.

**Example 17** *In this example, we represent unused operations as the percentage of a period at the highest frequency. If the processor is completely idle while running at the highest frequency, the unused operation is 100%. If the processor is idle but runs at half of the highest frequency, the unused operation is 50%.*

*Let's reconsider Example 16. If there is no sporadic job, the minimum energy is achieved by repeating a closed walk with four vertices. This solution is shown in Figure*

*6.18. In the period represented by vertex $v_2 = (1,0,1,10,1,2,1)$, $j_1$ and $j_3$ execute once because $\alpha_1 = \alpha_3 = 1$. In the same period, $j_2$ executes twice because $\alpha_2 = 2$. The fourth job always executes once each period. Since each job takes 20% time, the total time required to execute these four jobs is $20\% + (1+2+1) \times 20\% = 100\%$. Consequently, $\varphi(v_2) = 0$ and no additional operation can be executed.*

*For the period represented by $v_4$, only $j_4$ executes because $\alpha_1 = \alpha_2 = \alpha_3 = 0$. Notice that the frequency is only 30% of the highest frequency; $\varphi(v_4) = 30\% - 20\% = 10\%$. Because this is insufficient for any of $j_1$, $j_2$, or $j_3$, it is unused. However, this period can execute a sporadic job if it needs half of operations of $j_4$.* $\diamondsuit$

Consider a sporadic job that needs $w_r$ operations. Suppose the MPEG player still maintains the same output rate; then, the sporadic job can execute only by the unused operations in each vertex. The sporadic job can finish in one period if the amount of unused operations is larger than this job's number of operations, or $\varphi(v) \geq w_r$.

Suppose the sporadic job arrives at the beginning of a walk of $n$ vertices: $\mathcal{W} =< v_1, v_2, ...., v_n >$. The sporadic job can finish within $n$ periods if there are enough unused operations in these vertices. This condition can be expressed by the following inequality:

$$\sum_{i=1}^{n} \varphi(v_i) \geq w_r \tag{6.40}$$

## 6.6.2 Effects of Buffers

Equation (6.39) does not consider how data buffering reduces the response time of a sporadic job. To maintain the constant output rate, namely $j_m$ executes once each period, the data required by $j_m$ may be obtained in one of the two ways: (1) from the buffer between $j_m$ and $j_{m-1}$ or (2) generated by job $j_{m-1}$ in the same period. In other words, $j_{m-1}$ does not have to execute in this period if the buffer between $j_m$ and $j_{m-1}$ is nonempty. Let $v$ be a vertex representing this period. Condition (1) means $\beta_{m-1} > 0$ and condition (2) means $\alpha_{m-1} > 0$. Consequently, $\beta_{m-1} + \alpha_{m-1}$ must be at least one so that $j_m$ can execute in this period. This requirement can be written as

$\beta_{m-1} + \alpha_{m-1} > 0$.

We can generalize this relationship. Suppose job $j_l$ executes in a period. Job $j_{l-1}$ must execute in the same period if the buffer between $j_l$ and $j_{l-1}$ is empty ($\beta_{l-1} = 0$). We define an indicator function $\gamma_l$ to determine whether job $j_l$ has to execute:

$$\gamma_l = \begin{cases} 1 & \text{if } \gamma_{l+1} = 1 \text{ and } \beta_l = 0 \\ 0 & \text{otherwise} \end{cases} \qquad l \in \mathcal{M} - \{m\} \tag{6.41}$$

We define $\gamma_m = 1$ since job $j_m$ executes once every period. Because $\gamma$'s are the minimum requirements to keep the output rate, $\gamma_l$ must be smaller or equal to $\alpha_l$, or $\gamma_l \leq \alpha_l$. During this period, the minimum number of operations to sustain the constant output rate is

$$w_m + \sum_{l=1}^{m-1} w_l \cdot \gamma_l \tag{6.42}$$

Let $\varphi_b(v)$ be the maximum number of operations available for a sporadic job when the effects of buffers are considered.

$$\varphi_b(v) = t \cdot f - (w_m + \sum_{l=1}^{m-1} w_l \cdot \gamma_l) \tag{6.43}$$

Since $\gamma_l \leq \alpha_l$, $\varphi_b(v)$ must be larger than or equal to $\varphi(v)$. In order words, buffering allows the processor to spend more time on the sporadic job. for The response time can be computed using the procedure presented in Figure 6.19. It first checks whether there are enough unused operations ($\varphi(v)$) in one period. Then, it checks whether this job can finish in one period by draining the buffers ($\varphi_b(v)$). If neither is successful, it recursively computes the response time by adding one period each time.

```
ResponseTime(**input** vertex: $v = (\beta_1, ..., \beta_{m-1}, f, \alpha_1, ..., \alpha_{m-1})$, job: $w_r$)
/* find how many periods ($t$) are needed to finish the sporadic
   job that needs $w_r$ operations */
begin
    compute $\varphi(v)$ by equation (6.39);
    if $\varphi(v) \geq w_r$ /* enough unused operations */
        return $t$;
    compute $\varphi_b(v)$ by equation (6.43);
    if $\varphi_b(v) \geq w_r$
        find one next vertex $v'$ by the continuity condition (6.32);
        return $t$;
    /* the job will take more than $t$ to execute */
    /* find the response time recursively */
    find one next vertex $v'$ by the continuity condition (6.32);
    return $t$ + ResponseTime($v'$, $w_r - \varphi_b(v)$);
end
```

Figure 6.19: find the response time of a sporadic job

**Example 18** *Consider Example 16 again for computing the response time of a sporadic job. Suppose a sporadic job needs one period at frequency 10 to complete. Without buffers, it takes five periods to complete this job because the processor can spend only 20% of time in each period on this job.*

*Now, let's consider how buffering reduces the response time. For vertex $v_1$ in Figure 6.18, $\gamma_3$ is one but $\gamma_1$ and $\gamma_2$ equal zero. Since only $j_3$ and $j_4$ have to execute, the processor can spend 60% time in this period for a sporadic job. Because $j_1$ and $j_2$ do not execute, the next vertex is different from $v_2$. Using the continuity conditions, we find one vertex to follow $v_1$; it is $(1, 0, 0, 10, 0, 1, 1)$ as shown in Figure 6.20. This vertex can spend 40% time executing the sporadic job. The sporadic job finishes in two periods; this is 60% reduction from five periods. Similarly, we can compute the response time of the sporadic job if it arrives at $v_4$. It takes three periods as shown in Figure 6.20; this is 40% improvement from the original five periods.*

*It takes two periods to finish the sporadic job if it arrives at $v_2$ or $v_3$. On average, the response time of the sporadic job is $\frac{2\times3+3}{4} = 2.25$ periods; this is 55% improvement from the original five periods.* $\diamondsuit$

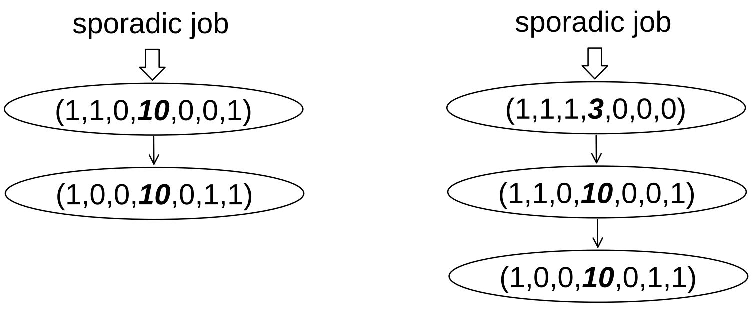


Figure 6.20: A sporadic job finishes in two periods if it arrives at $v_1$ (left). It takes three periods if it arrives at $v_4$ (right).

## 6.6.3 Timing Constraints of Sporadic Jobs

The previous section analyzed the average response time of a sporadic job. Using the same technique, we can determine whether it is possible to meet the timing constraint of a sporadic job. The timing constraint is the maximum acceptable execution time after a sporadic job arrives.

**Example 19** *For Example 18, if the timing constraint is four periods, the assignment in Figure 6.18 can meet this constraint. On the other hand, if it is two periods, this assignment cannot satisfy the constraint.* $\diamondsuit$

In order to decide whether it is possible to finish a sporadic job, $j_r$, within $n$ time periods, we have to find the shortest response time of $j_r$. The response time is the shortest when all buffers are full and the processor is running at the highest frequency. Thus, we assume all buffers are full and the frequency is the highest when $j_r$ arrives. We also assume that no sporadic job arrives before another sporadic job completes (because they are "sporadic").

In Equation (6.43), the maximum number of operations occurs when $\gamma_1 = \gamma_2 = ... = \gamma_{m-1} = 0$:

$$\max \varphi_b(v) = t \cdot \phi_1 - w_m \tag{6.44}$$

Here $t$ is the length of a period, $\phi_1$ is the highest frequency, and $w_m$ is the number of operations executed by job $j_m$.

If $\frac{w_r}{n} > t \cdot \phi_1 - w_m$, then it is impossible to meet the timing constraint because there are insufficient operations in each period. Allocating more memory for buffers will not solve the problem; the only solution is to find a faster processor with higher $\phi_1$.

The value of $\varphi_b(v)$ is $t \cdot \phi_1 - w_m$ only if the buffer between $j_{m-1}$ and $j_m$ has data. Since the buffer size is $b_{m-1}$, the buffer will become empty in $b_{m-1}$ periods after $j_r$ arrives. If $n \leq b_{m-1}$ and $\frac{w_r}{n} \leq t \cdot \phi_1 - w_m$, then $j_r$ can finish in $n$ periods. Job $j_{m-1}$ does not have to execute during these $n$ periods and $j_m$ can still execute once each period.

When $n > b_{m-1}$, the buffer between $j_{m-1}$ and $j_m$ becomes empty before $j_r$ completes. Thus, $j_{m-1}$ must execute $n - b_{m-1}$ times so that $j_m$ can execute once each period. The maximum value of $\varphi_b(v)$ drops to $t \cdot \phi_1 - (w_m + w_{m-1})$ after $b_{m-1}$ periods. Consequently, when $n \leq b_{m-1} + b_{m-2}$, $j_r$ can finish in $n$ periods if

$$w_r \leq b_{m-1} \times (t \cdot \phi_1 - w_m) + (n - b_{m-1}) \times (t \cdot \phi_1 - w_m - w_{m-1}) \tag{6.45}$$

As $n$ becomes larger, more and more buffers become empty and the values of $\varphi_b(v)$ decrease; this condition is illustrated in Figure 6.21. Following this analysis, we can derive the condition to finish $j_r$ within $n$ periods after $j_r$ arrives:

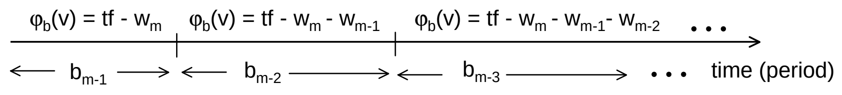


Figure 6.21: $\varphi_b(v)$ decreases as more buffers become empty.

$$\begin{array}{lll}
\text{if} & n \le b_{m-1} & w_r \le n(t\cdot\phi_1 - w_m) \\
\text{if} & b_{m-1} < n \le b_{m-1} + b_{m-2} & w_r \le b_m(t\cdot\phi_1 - w_m) + (n - b_{m-1}) \times (t\cdot\phi_1 - w_m - w_{m-1}) \\
\ldots & & \\
\text{if} & \sum_{l=k}^{m-1} b_l < n \le \sum_{l=k-1}^{m-1} b_l & w_r < \sum_{l=k}^{m-1} b_l(t\cdot\phi_1 - \sum_{p=l+1}^{m} w_p) + (n - \sum_{l=k}^{m-1} b_l)(t\cdot\phi_1 - \sum_{p=k}^{m} w_p) \\
\text{if} & n > \sum_{l=1}^{m-1} b_l & w_r < \sum_{l=1}^{m-1} b_l(t\cdot\phi_1 - \sum_{p=l+1}^{m} w_p) + (n - \sum_{l=1}^{m-1} b_l)(t\cdot\phi_1 - \sum_{l=1}^{m} w_l)
\end{array} \tag{6.46}$$

where $1 < k < m$.

The conditions in (6.46) indicate that enlarging the last buffer (larger $b_{m-1}$) is most effective because it is multiplied by the largest coefficient, $t \cdot \phi_1 - w_m$. This analysis further suggests how to calculate the minimum number of items stored in each buffer. In fact, buffers do not have to always remain full. As long as these conditions are satisfied, $j_r$ is guaranteed to finish within $n$ periods. This sets the lower limits of the buffer sizes. So far, we have been assuming that the items stored in different buffers takes the same memory size. When they require different memory sizes, it is necessary to consider how to allocate memory for different buffers. For example, suppose the same amount of memory can increase $b_1$ by one or $b_{m-2}$ by two. From the conditions above, we know that increasing $b_{m-2}$ is more effective in reducing the the response time of a sporadic job. In contrast, if the same memory can increase $b_1$ by two or $b_{m-2}$ by one, we need to compare their coefficients in (6.46) to decide which is more effective for guaranteeing the response time of $j_r$. No existing scaling technique is able to analyze the relationship between buffer sizes and the response time of a sporadic job. After the buffer sizes change, the method presented in Section

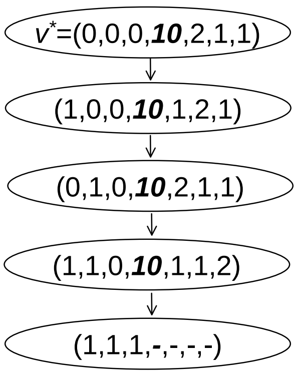


Figure 6.22: fill the buffers from a starting vertex

6.5 efficiently finds a minimum-power frequency assignment when the buffer sizes change.

In the assignment graph, the $\beta$ values of a vertex represent the numbers of items stored in buffers. Since (6.46) specifies the minimum number for each buffer, a vertex should be removed if its $\beta$ values are too small. After excluding these vertices, a minimum-cost walk can meet all constraints of the mixed workloads and also achieves the minimum energy consumption.

There is, however, an initial walk after the system starts and the buffers are being filled; during this period the timing constraint of a sporadic job cannot be met.

**Example 20** *In Example 16, when the processor runs at the highest frequency, it can execute one job twice and the other jobs once in each period. Figure 6.22 shows the walk that represents how buffers are filled from a starting vertex.* $\diamondsuit$

The minimum time period to fill all buffers can be calculated by finding a shortest path from one starting vertex $v^*$ to a vertex whose encoding is $(b_1, b_2, ..., b_{m-1}, \bullet, \bullet, ..., \bullet)$. Algorithms for finding shortest paths between vertices can be found in [22].

In summary, this section describes how to compute the response times of sporadic jobs based on the assignment graphs developed in Section 6.5. We explain how to take advantages of the buffered data to reduce the response times without affecting the on-time constraints of periodic jobs. We also analyze whether it is possible to satisfy the timing constraint of a sporadic job.

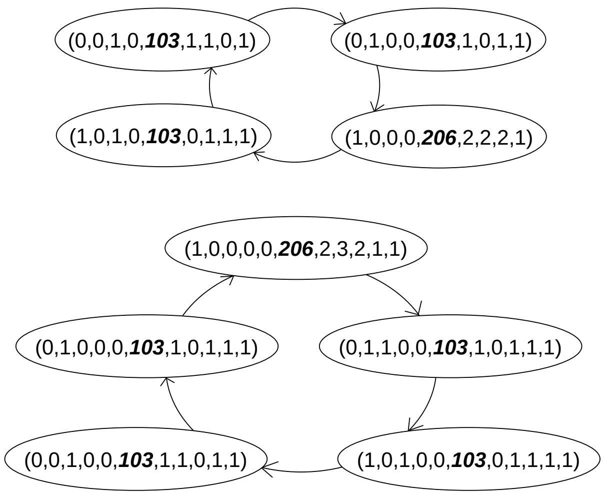


Figure 6.23: schedules for five and six jobs with 60% processor utilization

## 6.7 Experiments

The details of our experimental setup are described in Appendix A.2.

### 6.7.1 Synthesized Workload

A synthesized workload is used to compare the power consumption in the following scenarios. We use the procedures presented in Section 6.5 to find a minimum-cost walk for each scenario.

- 3 to 6 jobs. The last job has to execute once every period.
- 2 to 5 frequencies. We start with 206 and 103 MHz; then, we add 59, 147, and 89 MHz.
- 40% to 70% processor utilization at the highest frequency. In this chapter, utilization always means the utilization at the highest frequency (206MHz in our setup).

Each period is one second and one buffer is inserted between two jobs. Figure 6.23 shows the schedules for five and six jobs with three frequencies (206, 103, and

59 MHz) when the processor utilization is 60%. Since the processor is 60% busy, it cannot stay at 103 MHz or 59 MHz indefinitely; otherwise, it would violate the timing constraints. The frequency in each period is written in **boldface**. Notice that 59 MHz is not used even though it is available.

Figure 6.24 depicts the measured power consumption in three cases: no frequency scaling, scaling down during idleness, and scaling using our method. These data were obtained with 70% processor utilization. As can be seen at the top of the figure, the system consumes less power when the processor is idle. The boundaries of periods are clearly visible. If we scale down the frequency during idleness, period boundaries are also distinguishable. The power consumption in the "valleys" are deeper, indicating lower consumption during idleness. However, the power remains virtually unchanged when the processor is busy. Finally, the bottom of this figure is the power consumption of our method. We can see that the period boundaries are now blurred; this is because our method rearranges the execution order of jobs. Some jobs execute at a higher frequency; some other jobs execute at a lower frequency. This figure shows clearly that our method has lower average power; it is approximately 1.6 W. This is nearly 40% reduction among the scalable range ($\frac{1.89-1.6}{1.89-1.17} = 40\%$).

Because Figure A.8 and Figure A.9 show almost linear scaling in power and performance, we can predict the power consumption accurately for different scenarios. The average error is 2.5% and the maximum error is 8%. Figure 6.25 depicts the predicted and measured power with four and five frequencies. The horizontal axes are the processor utilization at 206 MHz and the vertical axes are the power consumption. The squares and diamonds represent the predicted power consumption. The lines connect the measurement results. The triangles are the optimal solutions (minimum power) if the processor's frequency can be continuously scaled. Squares, diamonds and triangles almost overlap on each other in the figure because their values are very close.

The minimum power is computed as follows. Let $p_{59}$ (1.167W) and $p_{206}$ (1.886W) be the power at 59 and 206 MHz respectively. The power is a linear interpolation using the following formula:

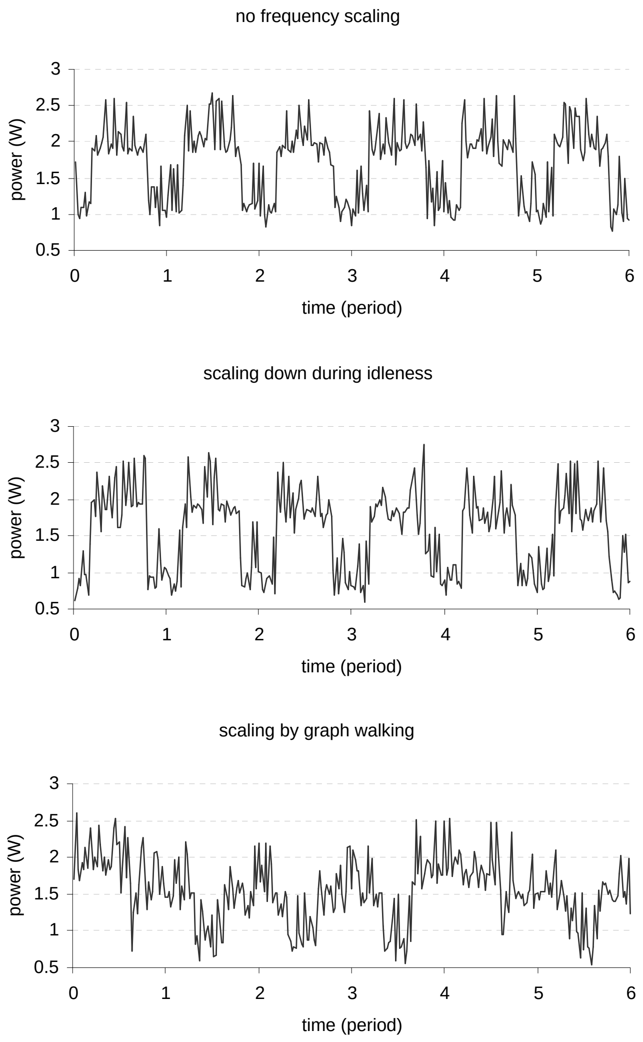
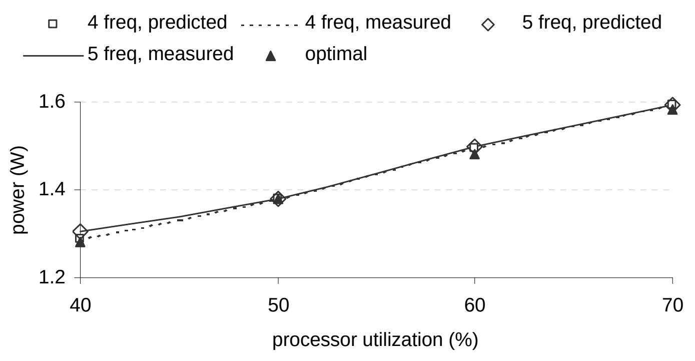


Figure 6.24: no frequency scaling (top), scale down during idleness (middle), and using graph walking technique with buffer insertion (bottom). Our method saves 40% power in the scalable range.



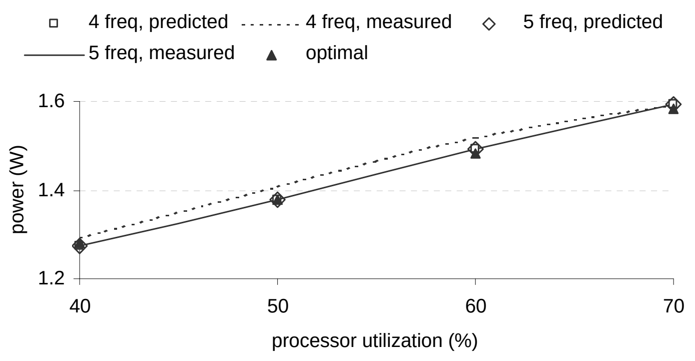


Figure 6.25: estimated, measured, and optimal power for three (top) and four jobs (bottom)

$$p_{59} + (p_{206} - p_{59})\frac{utilization - \frac{59}{206}}{1 - \frac{59}{206}} \tag{6.47}$$

Figure 6.25 shows that four frequencies (206, 147, 103, and 59 MHz) are sufficient to achieve almost the minimum power, computed by (6.47).

### 6.7.2 Reducing Power for Playing MPEG Video

An MPEG player differs from the previous synthesized workload in two major ways: the execution time varies from frame to frame and it has many IO operations. We characterize an MPEG player ported to Assabet. At 206MHz, the program can display approximately 20 frames. We divided the program into three stages: reading data, decoding data, and displaying images. To conquer the variation of execution time, we assign fixed duration to each stage: (1) 10 milliseconds to reading one frame (2) 25 milliseconds to decode one frame (3) 10 milliseconds to display one frame. Together, they take 45 milliseconds. Our target frame rate is 15 frames per second so the processor utilization is 68% (67 milliseconds for one frame, $\frac{45}{67} = 68\%$). We compared the power consumption in the following scenarios. The measurement is obtained using the same setup illustrated in Figure A.6; all values include the power of a network card.

- no frequency scaling, frame rate controlled by busy waiting. The system consumes 2.45 W.
- no frequency scaling, frame rate controlled by calling `usleep` [4]. The power consumption is 2.30W.
- frequency scaling with 2 frequencies: 206 and 103 MHz. A buffer with three slots is inserted between stages. The power consumption is 2.16W.
- frequency scaling with 3 frequencies: 206, 103, and 59 MHz. The power consumption is 2.16 W. Frequency 59MHz is not used because it does not help reduce the power consumption.
- frequency scaling with 4 frequencies: 206, 147, 103, and 59 MHz. The processor stays at 147MHz and the power is 2.12W. This is 46% reduction in the scalable range ($\frac{2.45-2.12}{1.89-1.17} = 46\%$).

The first case keeps the processor busy while waiting for the beginning of the next period. The following code illustrates such busy waiting to control the frame rate:

[4]This function suspends the execution of the calling process; the unit is microsecond.

```
..... after finish one frame
while (target finish time > current time)
  { update current time; }
target finish time += one period;
```

In contrast, the second scenario calls `usleep` if there is slack time:

```
..... after finish one frame
slack time = target finish time - current time;
if (slack time > 0)
  { usleep(slack time); }
target finish time += one period;
```

When a process calls `usleep`, this process voluntarily suspends its execution. If no process needs the processor, the processor becomes idle and consumes less power. In our measurement, approximately 0.15W power is saved by calling `usleep`. We call this *intra-period* power saving because it saves power inside each period. In contrast, our method uses buffers across the boundaries of periods to save power; this can be considered as an *inter-period* power-saving technique. Combining our method with the intra-period technique saves 0.33W, 46% in the scalable range or 14% ($\frac{0.33}{2.45} = 14\%$) of the original power. This is very close to 0.33W predicted by formula (6.47). Notice that after inserting buffers, the power is reduced by 0.29W even when there are only two frequencies. This example shows that adding buffers is more effective than adding available frequencies to the processor.

## 6.7.3 Buffer Size

Figure 6.25 shows that our predicted power consumption is very close to the measured values. In the rest of this section, we use the same method to estimate the power consumption for different buffer sizes, job sizes, and arrival rates of sporadic jobs.

The experimental results in Section 6.7.1 show that after inserting buffers, a few frequencies are sufficient to save significant amount of power. We use the same workload to study the effect of buffer sizes. We consider five frequencies (206, 147, 103,

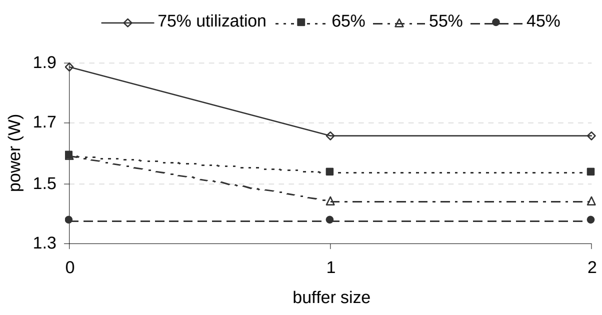


Figure 6.26: power consumption with different buffer sizes

89, 59 MHz), four jobs with equal number of operations, and 45% to 75% processor utilization at 206 MHz. Figure 6.26 compares the power consumption with different buffer sizes. In all cases, inserting one or two buffers between two jobs have identical effects. For 55%, 65%, and 75% utilization, adding one buffer between two jobs reduces the power because the processor can switch to low frequencies. When the utilization is 45%, adding buffers has no effect because the system consumes the minimum power if the processor stays at 103 MHz. This figure shows that adding one buffer between two jobs is very effective; adding two buffers has no additional advantages. This example suggests that buffer insertion does not need substantial amount of memory.

### 6.7.4 Job Size

The synthesized workload in Section 6.7.1 assumes all jobs need the same number of operations. Dividing a program into equal-size stages can be difficult; for example, the MPEG player in 6.7.2 is naturally divided into three stages and they take different amounts of time. Suppose the amount of operations from all jobs ($\sum_{l=1}^{m} w_l$) is a constant. Dividing these operations to different ways may cause different power consumption. Consider the following ways to divide the operations. We are interested in finding the best one for power saving.

1. $w_1 = 7w_2$, $w_2 = w_3 = w_4$

2. $w_1 = 4w_4$, $w_2 = 3w_4$, $w_3 = 2w_4$

3. $4w_1 = w_4$, $3w_1 = w_2$, $2w_1 = w_3$

4. $w_1 = w_2 = w_3 = w_4$

5. $w_1 = w_4 = 4w_2$, $w_2 = w_3$

6. $w_2 = w_3 = 4w_1$, $w_1 = w_4$

Figure 6.27 depicts the relationship of these cases. The widths indicate the relative numbers of operations in each case. We consider five frequencies of the processor; one buffer is inserted between two jobs. Figure 6.28 is ratio of power consumption related to the first case. For 60% utilization (white bars), the first case consumes more power than the other cases. This is because $w_1$ is so large that the processor cannot execute $j_1$ twice in one period to fill the first buffer. Because the first buffer is not filled, the processor cannot scale down the frequency. This case causes the most power consumption. The other cases can fill the buffers and reduce the power. For 80%, however, only case 3 and case 4 can use the buffers to save power. This example suggests the following rule: it is preferred that earlier jobs, such as $j_1$ and $j_2$, need fewer operations so that the buffers can be filled. If $j_1$ needs too many operations, then the first buffer cannot be filled and all the other buffers are unused. If the buffers are unused, they cannot facilitate power saving.

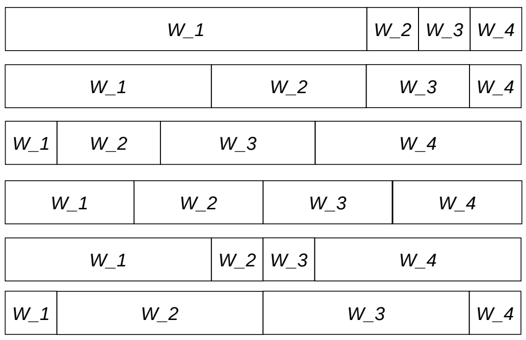


Figure 6.27: different ways to divide operations into four jobs

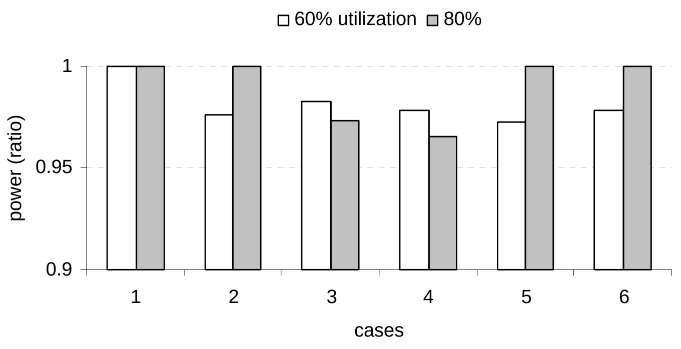


Figure 6.28: power consumption of job sizes in Figure 6.27

### 6.7.5 Arrival Rate of Sporadic Jobs

If sporadic jobs arrive frequently, the average power consumption is higher. This section discusses how the arrival rate of sporadic jobs affects the average power.

Figure 6.29 is a redraw of the five-job case in the top of Figure 6.23. Consider a sporadic job that takes one period at the peak frequency (206 MHz) to complete. Suppose this sporadic job arrives at the period represented by $v_2$. We can follow the procedure explained in Example 18 to compute the response time of the sporadic job. The result is shown in Figure 6.30. After running at 206 MHz for three periods, the sporadic job completes and the processors returns the closed walk shown in Figure 6.29 starting from $v_1$. Even though the closed walk was "interrupted" by the sporadic job at $v_2$, the frequency assignment does not necessarily continue from $v_3$ after processing the job. If there is no sporadic job, the average power in the next four period from $v_2$ is 1.51W. In contrast, the sporadic job causes the average power in the same time interval to rise to 1.76W.

The same method can be applied to compute the response time of the sporadic job

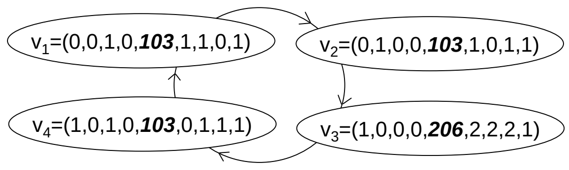


Figure 6.29: closed walk for five jobs with 60% processor utilization, redraw of Figure 6.23

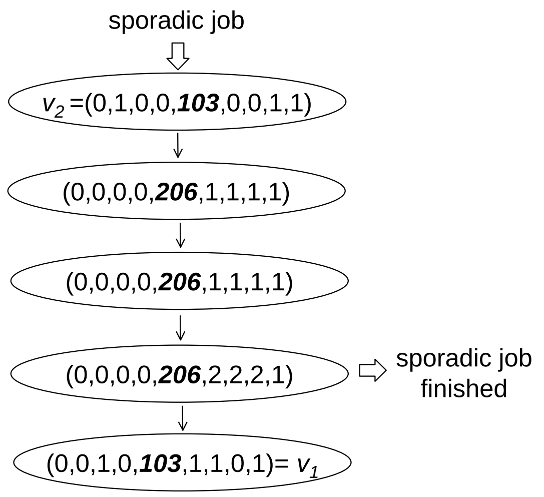


Figure 6.30: process a sporadic job that arrives at the period represented by $v_2$

if it arrives at the period represented by $v_1$, $v_3$, or $v_4$. We can also find the additional energy required to process this sporadic job. After processing the sporadic job, the frequency assignment will eventually return to the closed walk in Figure 6.29, but it does not always starts from $v_1$.

Let's assume that the time interval between two sporadic jobs is long enough so that the processor can return to the original closed walk in Figure 6.29. Figure 6.31 shows the power consumption for different arrival rates of the sporadic job. The horizontal axis is the average number of periods between two sporadic jobs; the vertical axis is the average power consumption. This figure shows that the average power reduces rapidly as the time between two sporadic jobs increases. Since sporadic jobs are "sporadic" and the time interval between them should generally be large, their effects on power is insignificant.

### 6.7.6 Timing Constraints and Maximum Operations of Sporadic Jobs

Section 6.6.3 derives the relationship between buffer sizes and the response time of a sporadic job. We use a synthesized workload described in Section 6.7.1 to pictorially illustrate the conditions. There are four jobs and each takes 20% time in a period

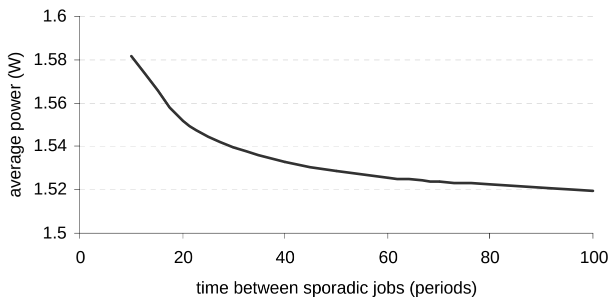


Figure 6.31: average power consumption for different intervals between two sporadic jobs

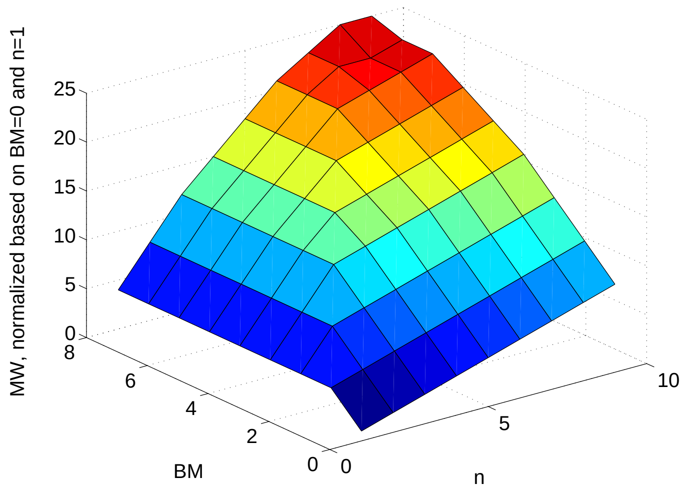


Figure 6.32: maximum number of operations of a sporadic job: $n$ is the timing constraint and $BM$ is the total buffer memory sizes.

when the processor runs at the highest frequency. Without any buffers, there is 20% processor time unused in each period.

Let $BM$ be the total memory allocated for buffers: $BM = \sum_{l=1}^{m-1} b_l$. Let $MW(n)$ be the maximum number of operations a sporadic job can complete within $n$ periods. If $BM$ is zero, there is no buffer so $MW(1)$ is 20% of a period. We use 20% of a period as the normalization base. Figure 6.32 shows $MW$ for different $n$'s and $BM$'s. In this figure, we can observe that $WM$ is the largest when $BM$ and $n$ have compatible values. This can be understood by examining the conditions of (6.46) in Section 6.6.3. When $n$ is small and $BM$ is sufficiently large, adding more buffers will not increase $MW$; this corresponds to the first condition in (6.46) and the lower left side of the surface in Figure 6.32. In contrast, when $BM$ is too small, $MW$ increases linearly with $n$ because buffers become empty before the sporadic job completes. This situation corresponds to the last condition in (6.46) and the lower right side of the surface.

### 6.7.7 Summary

We use the graph-based algorithm presented in Section 6.5 to find frequency assignments for different scenarios. Our experiments show that inserting buffers effectively reduces power consumption even for a processor with only a few frequencies. Inserting buffers can achieve nearly the minimum power with four frequencies. We also demonstrate that adding one buffer between two jobs is sufficient in many cases; consequently, buffers do not need substantial amount of memory. We provide a guideline for dividing operations into multiple jobs. Finally, we show that sporadic jobs have negligible effects on power increases.

## 6.8 Chapter Summary

This chapter addresses power reduction by frequency scaling for mixed workloads. We explain existing methods based on integer linear programming and point out the need for efficient solutions. Our method inserts buffers between jobs and builds an assignment graph; each vertex encodes the current states of the buffers and the frequency of the processor. We develop a graph-based method that has complexity $O(|\mathcal{V}|^3)$ where $|\mathcal{V}|$ is the number of vertices of the state-space graph. We use a frequency-scalable system to demonstrate the effectiveness of our method. By inserting buffers, we can achieve nearly optimal power saving using only four frequencies. Furthermore, this method is able to dramatically reduce the response time of sporadic jobs. This method saves approximately 46% power of an MPEG player, after excluding the non-scaleable base power. We also analyze the effects of buffer sizes and how to divide programs into multiple jobs.

# Chapter 7

# Conclusion

Reducing power consumption has become a major design goal in all types of electronic systems. Most of these systems do not have to operate at their peak performance continuously. The run-time variation of workloads provides opportunities to reduce power consumption by power management. Power managers predict future utilization of hardware components and determine their power states. Because of state-transition overhead, power management reduces energy only when the overhead can be compensated by the saved energy. For an IO device, this condition is expressed by its break-even time.

Operating systems can be structured into layers. In each layer, different information is available to power managers. When power managers have additional information, they can predict utilization more accurately with better power saving and performance. Furthermore, process schedulers can arrange the execution orders of processes to create long and continuous idle periods during which devices can be shut down to save energy. In addition to saving power, it is also important to reduce the execution time of power managers. An efficient graph-based method can significantly reduce the time required to find optimal frequency assignments.

## 7.1 Thesis Summary

Operating systems provide important information for power management. When power managers has more information about the processes that generate requests, power managers can improve energy saving while maintaining satisfactory performance.

A Windows-based framework is presented for implementing policies to control the power states of IO devices. Because device drivers have enough information for most previously proposed policies, these policies can be implemented in our framework. We compare the power saving, computation requirements, and performance impact of these policies. Since device drivers do not have detailed information about individual processes, these policies are not taking the full advantages of the information available from operating systems.

A process-based policy is proposed to distinguish individual processes for predicting the idleness of IO devices. Different processes have different device utilization and create requests only when they execute. Hence, it is essential to separate processes and to estimate how often they execute. By collaborating with process managers, power managers predict idleness more accurately and save more power with smaller performance degradation.

The third part of this thesis explains how to further improve power saving by combining process scheduling and power management. Process schedulers determine the execution order of processes and affect the lengths of idle periods. We propose an application programming interface (API) that allows application programs to indicate their future device requirements. A heuristic is presented to group processes and create idle periods longer than the break-even time of the power-managed device.

Finally, a graph-based method is used to assign processor frequencies efficiently after inserting data buffers in multimedia programs. This method constructs a space-search graph; each vertex indicates the states of the buffers and the frequency of the processor. Data buffering has three purposes: maintaining constant output frame rates, saving energy by frequency scaling, and reducing the response time of sporadic jobs. An algorithm is proposed to find optimal solutions efficiently and to estimate the response time of sporadic jobs.

The methods presented in this thesis have been implemented on different interactive systems. Experimental results demonstrate better power saving and performance.

## 7.2 Future Work

This work can be extended in several directions. First, some parameters are chosen statically, for example, $a$ in Section 4.2.1 and $k$ in Section 4.2.4. Further investigation is needed to determine whether dynamically adjusting their values can provide more power saving and better performance. Second, the policies presented in this thesis manage the power states of distinct IO devices independently. It is possible that the idle periods of some devices are highly correlated so the power state of one device provides better indication of future idleness of another device. Third, compilers can analyze programs and provide valuable information for dynamic power management. It is not yet well understood how to effectively combine power-aware operating systems and compilers.

This thesis focuses on reducing the power of IO devices and processors. In a complete system, there are other components that consumer large portions of power; examples are memory and display. They have unique characteristics and require different techniques for energy reduction. Specifically, memory has short wakeup delays but requires "warm-up" time. After a block of memory is woken up, it has empty contents and data need to be filled. Filling data into the newly awaken memory units may consume even more energy. Furthermore, before shutting down a memory unit, its contents have to be written back to the next level in the memory hierarchy and this consumes energy. Thus, different sets of rules are needed to manage the power states of memory. Managing the power of display also has unique challenges because display cannot be shut down while the system is interacting with a user.

Many portable systems use wireless networks to connect to the Internet; this provides opportunities for energy reduction. Most network servers are connected to power grids and are not constrained by battery lifetimes. Migrating tasks from battery-powered portable systems to servers may lengthen the battery lifetimes on portable systems. It remains a research area to design widely-acceptable protocols that can migrate tasks for energy efficiency.

Multiple-processor systems have appeared in today's desktop computers. As hardware technology improves, it is possible to have multiple processors on a laptop computer in the near future. The policies presented in this thesis are not designed to manage multiple identical devices or processors. For those systems, power managers have to collaborate with load-balancing managers, namely "power-aware load balancing", to achieve the best power efficiency and desirable performance.

Finally, there is a need to design policies that consider overall systems and multiple heterogeneous devices simultaneously. In order to facilitate the exploration of design choices, we foresee the necessity to develop system-level tools for policy design and tuning.

# Appendix A

# Measuring Power

## A.1 Power of IO Devices

Measuring the power of IO devices can be challenging. In notebook computers, many devices are directly plugged into the main boards to obtain power as well as signals. Some other devices acquire power through very short cables. For example, inside a Sony VAIO notebook computer, the power cables for the hard disk is shorter than one inch. On a desktop computer, most IO cards are connected through PCI or similar interfaces. Measuring their power requires cutting the wires on the mother boards or the contacts of these cards.

### A.1.1 Hardware Connection

In order to reflect the current technology, it is essential to measure the power of commercially available devices, instead of proprietary devices. Devices were chosen for four reasons: (1) they consume large power (2) they are not always used when a computer is operating (3) they have large break-even times (4) it is possible to measure the power fed into the device only.

Unfortunately, commercial devices do not have probing points to measure power; therefore, it is necessary to "cut the wires". To conquer these difficulties, we measured the two types of devices: a device with power cables or a device with PCMCIA

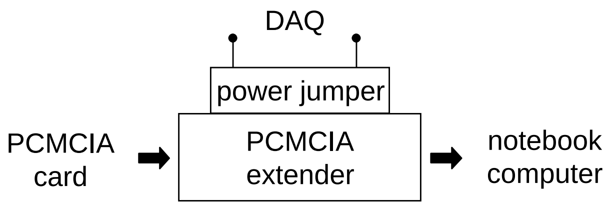


Figure A.1: connect DAQ to measure the power of a PCMCIA card

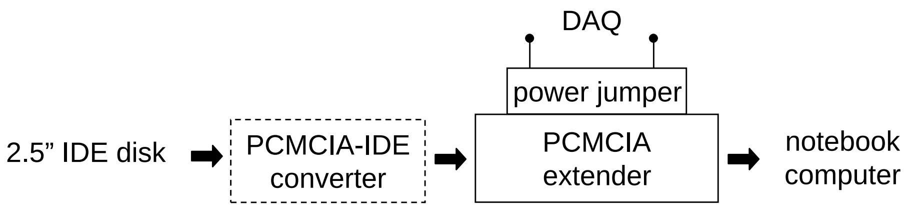
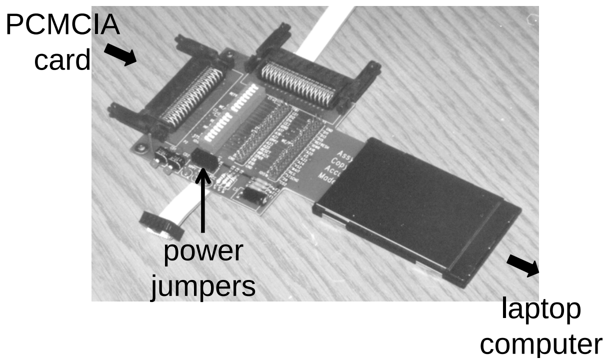


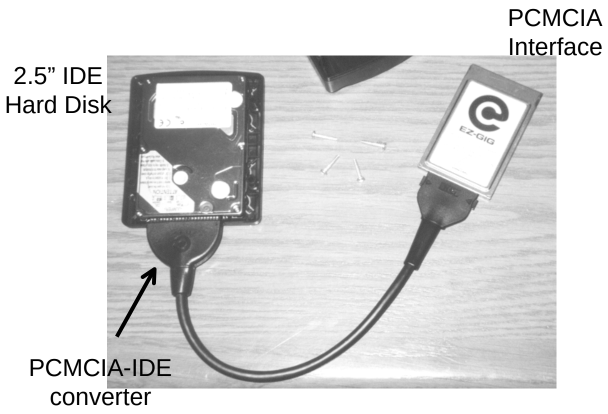
Figure A.2: measure the power of a 2.5" hard disk

interfaces. A typical 3.5" hard disk needs two power sources: 12V and 5V. Their power cables are usually more than five inches long and can be connected to meters.

In order to measure the power of a PCMCIA-based device, we used a PCMCIA extender manufactured by Accurite. The extender is designed for developing and debugging PCMCIA devices; it has jumpers for measuring the signals as well as the power of a PCMCIA device. We connected the power jumpers to a data acquisition card (DAQ) by National Instruments to measure the power flowing into a PCMCIA device; this DAQ can take up to one million readings per second. Figure A.1 illustrates this connection. This method can measure the power of any PCMCIA device. Two PCMCIA devices were considered: a network interface card (NIC) and a 2.5" hard disk. To measure the power of a 2.5" hard disk, we used a PCMCIA-IDE converter produced by Appicorn to connect the disk to to a laptop computer, as shown in Figure A.2. Figure A.3 is the photos of the experimental setup.

Figure A.4 shows the steady-state power of the Hitachi hard disk. Since the disk is controlled through the PCMCIA interface, it is completely turned off in the sleeping state. Consequently, it consumes virtual no power even though it consumes 0.125W in the sleeping state by the vendor's specification.





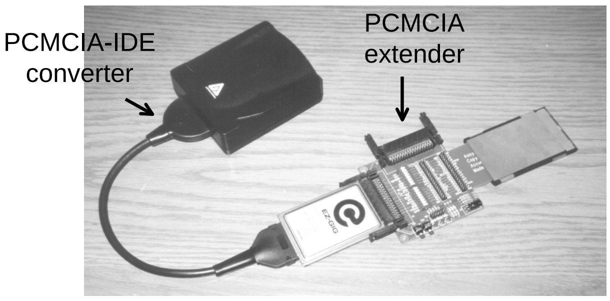


Figure A.3: top: PCMCIA extender; middle: PCMCIA-IDE converter and a 2.5” hard disk; bottom: a hard disk connected to the extender

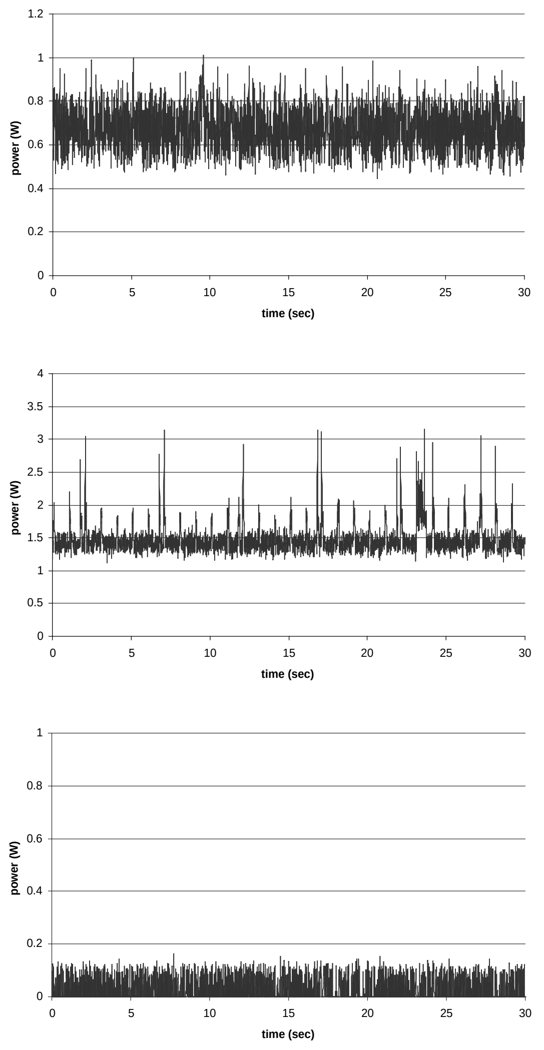


Figure A.4: power consumption in three conditions: idle, writing, and sleeping

| | | |
|---|---|---|
| vendor | Hitachi | IBM |
| model | DK23AA-12 | DARA-212000 |
| capacity | 12 GB | 12 GB |
| peak power | 4.5W | 4.7 W |
| seek | 2.3 W | 2.3W |
| read | 2.15 W | 2.0 W |
| write | 2.1 W | 2.1 W |
| sleep | 0.125 W | 0.1 W |
| standby | 0.25 W | 0.25 |
| performance idle | NA | 1.85 W |
| active idle | NA | 0.85 W |
| low-power idle | NA | 0.65 W |

Table A.1: specifications of two hard disks

### A.1.2 Device Variations

Devices of the same functionality may have wide variations in their power consumption. Table A.1.2 compares the specifications of two 2.5” hard disk drives; one disk has three additional power states. Because ACPI supports at most five power states, some of these additional states are not software controllable. Furthermore, the power in the “sleeping” state is not always the same as the “sleep power” specified by the manufacturer.

Figure A.5 is the transitions of two 2.5” disks; one is Hitachi DK23AA-60 disk and the other is Fujitsu MHF2043AT disk. The Hitachi disk takes longer and consumes more energy during transitions. From this figure, we can see even the same type of devices have large variations in their transition overhead. The Hitachi disk was used to generate the experimental data in this thesis.

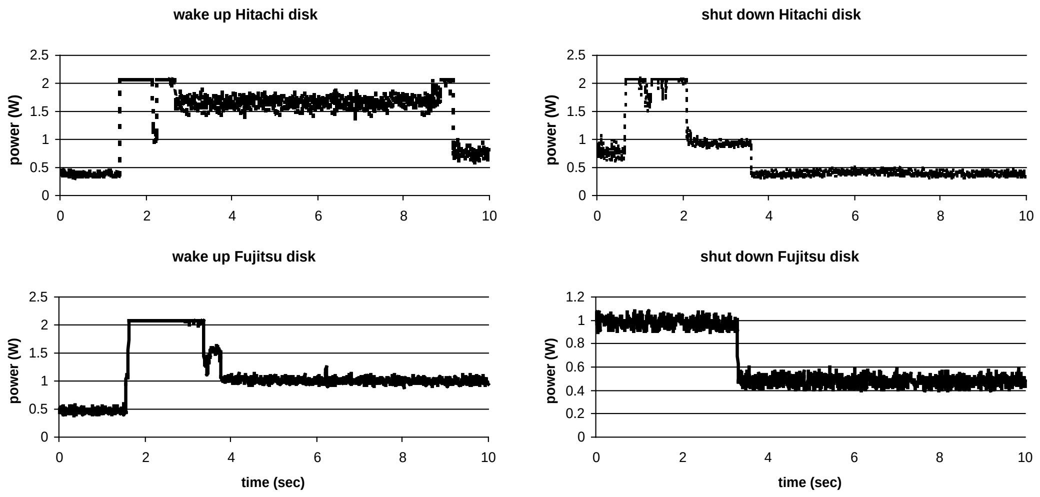


Figure A.5: state transitions of two 2.5” disks

## A.2 Power of a Frequency-Scalable System

In this section, we first describe our experimental environment. Then, we measure the power saving of a synthesized workload. We modify an MPEG player to scale frequencies for power reduction. We also discuss how buffer sizes and job sizes affect power saving.

### A.2.1 Experimental Setup

We set up a system to measure power saving by frequency scaling. The system is composed of a palm-size computer using Intel's StrongARM processor [2] (also called *Assabet*). The processor has eleven frequencies, between 59 and 206 MHz. It has a 320×240 touchscreen, 16MB SDRAM, and a Compact Flash interface; this interface can be used for networking. This system runs Linux ported for ARM processor [7]. We used a National Instrument Data Acquisition Card (DAQ) to measured the DC current from the AC/DC adapter; this is the total power consumption of the whole system and directly affects battery lifetime. Figure A.6 illustrates the setup for our experiments. Assabet supports frequency scaling but it does not support voltage

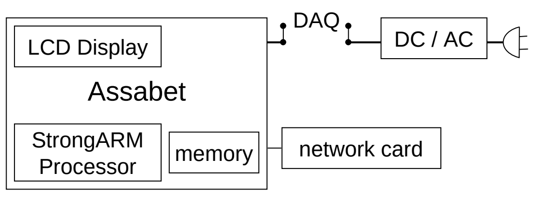


Figure A.6: setup for our experiments

scaling. It takes approximately 2 microseconds to change frequencies[1]. Assabet can also connect to a companion board, called *Neponset*, that provides interfaces for PCMCIA, USB, a serial port, and audio input/output.

Figure A.8 is the power consumption at different frequencies. We kept the processor busy by running an infinite loop. When the processor is busy, the system consumes 1.89W at 206 MHz and 1.17W at 59MHz; this is 0.72 W or 38% reduction of power consumption. When the processor is idle, it consumes 1.22W at 206 MHz and 0.97W at 59MHz; the power reduction is 0.25W. There is a baseline power that cannot be reduced by frequency scaling, such as the power for the LCD display. Figure A.9 compares the performance at different frequencies. The performance scales up almost linearly with frequencies.

[1]The implementation in Linux 2.4.1 recalibrates a software timer each time the clock frequency changes. This recalibration takes up to 150 milliseconds. However, such recalibration is not always necessary.

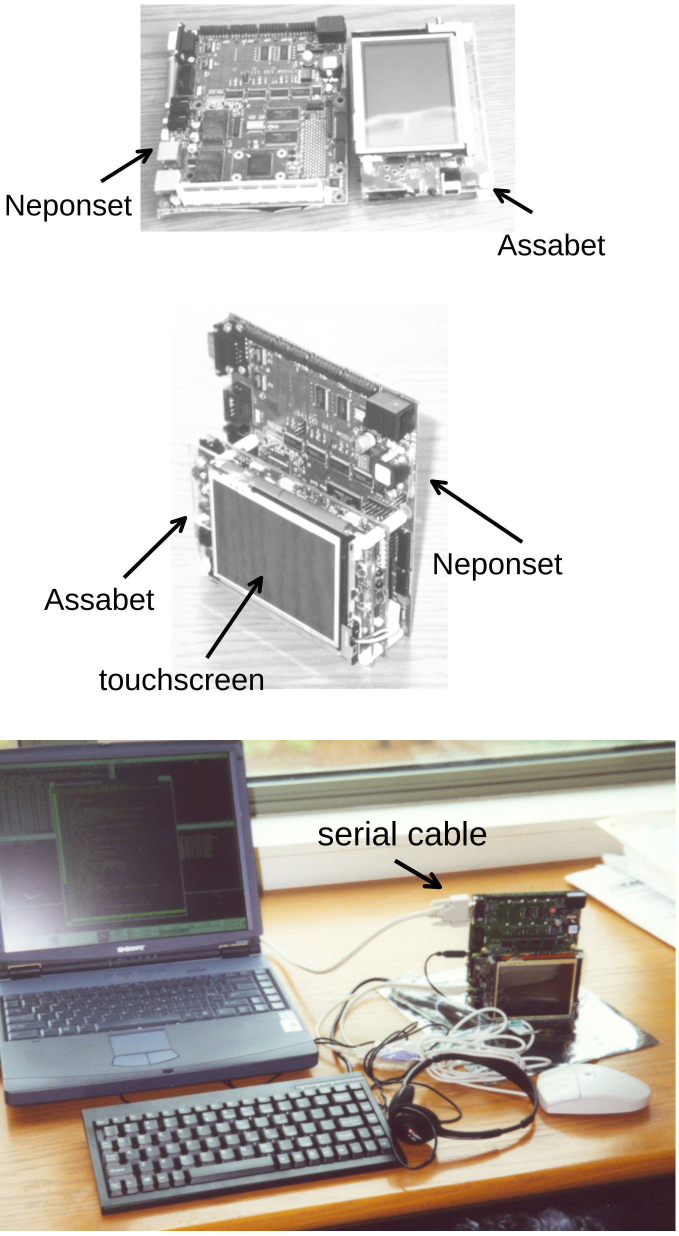


Figure A.7: top: Assabet and Neponset; middle: Assabet and Neponset connected; bottom: connected to a laptop computer through a serial cable

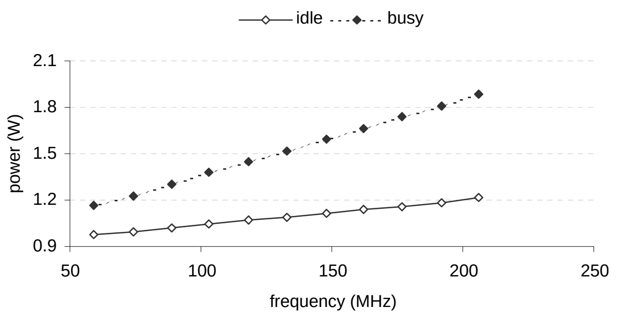


Figure A.8: power consumption at different frequencies

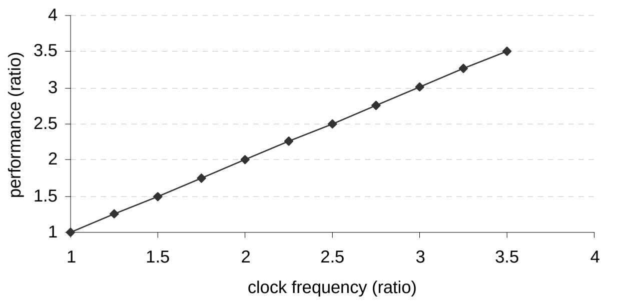


Figure A.9: performance at different frequencies

# Appendix B

# 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 (5.4). 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 [30]. 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 (5.4) 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. Even without timing and precedence constraints, low-energy scheduling is still an NP-complete problem as proved below.

## B.1 Simplification Assumptions

It is NP-complete to answer whether a schedule exists. 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 be composed 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; an idle period is "wasted" if it is short. 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.2 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

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

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

$$sw = \sum_{k=1}^{m} sw_k \tag{B.2}$$

## B.3 Problem Statement

Under the simplification in B.1, the low-energy scheduling problem is equivalent to finding a schedule such that the total number of switches is the minimum.

$$\min \sum_{k=1}^{m} \sum_{i=1}^{n-1} r_{s_i,k} \oplus r_{s_{i+1},k} \tag{B.3}$$

## B.4 Distance Between Jobs

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

$$dt_{x,y} = \sum_{k=1}^{m} r_{x,k} \oplus r_{y,k} \tag{B.4}$$

**Example 4** *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 B.1.*

*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*

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

Table B.1: job-device relationship

*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 $\mathcal{M}_{3\times3}$ to encodes these distances; $m_{x,y}$ is the distance between $j_x$ and $j_y$. The distance matrix of these three jobs is*

$$\begin{bmatrix} 0 & 2 & 1 \\ 2 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{B.5}$$

*$\mathcal{M}$ 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.* $\diamondsuit$

## B.5 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 5** *Figure B.1 shows the distance graph of the jobs the previous example.*

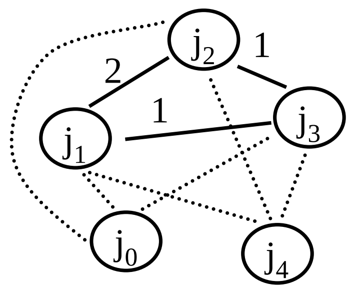


Figure B.1: graph of jobs and their distances

*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 ([ND22, p.211] in [30]). Consequently, the simplified scheduling problem is NP-complete.

# Appendix C

# Long and Finite Walks

Section 6.5 explained how to find a minimum-cost walk for a workload that has infinite time horizon. Because an MPEG movie usually has thousands of frames, it is valid to approximate the movie as infinite frames. When a workload is infinite, we can ignore the initial cost in the walk that determines frequency assignment; this initial cost is the cost of the walk from a starting vertex to a minimum-cost closed walk. In reality, no workload can have infinite time horizon. For a finite-length workload the initial cost cannot be ignored; also, there may be a "tail" that does not form a complete closed walk.

Recall that $G = (\mathcal{V}, \mathcal{E})$ is an assignment graph for frequency scaling. When the number of vertices in a walk exceeds the number of vertices in $G$, the walk must contain at least one closed walk, according to the pigeonhole principle. Figure C.1 illustrates such a long walk.

Even though a long walk $\mathcal{W}$ must contain one closed walk, the pigeonhole does not explain whether $\mathcal{W}$ may contain multiple non-overlapping closed walks. In fact, it is possible that a minimum-cost long walk contains multiple closed walks. However, we can always find another walk whose cost is also minimum but has a special format. This special format divides $\mathcal{W}$ into three parts as shown Figure C.1. The first part starts from one starting vertex $v^*$ and ends at the beginning of the first closed walk; the second part repeats this closed walk; the third part leaves this closed walk and finishes $\mathcal{W}$. We can prove that the length of the third part is always less than the

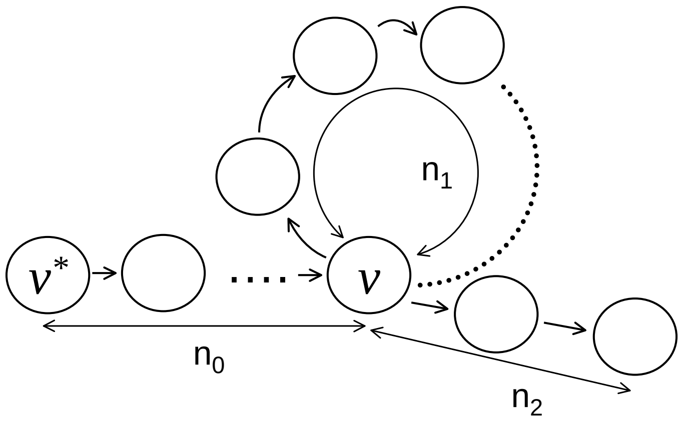


Figure C.1: a walk with one closed walk

length of the closed walk.

**Theorem 2** *If a finite walk contains a closed walk constructed by FindClosedWalk, there is a walk of equal or smaller cost such that the subwalk after leaving the closed walk is shorter than the length of the closed walk. Suppose the walk in Figure C.1 is a minimum-cost walk of a finite length and it contains a closed walk of $v$ with length $n_1$. There is a minimum-cost walk such that the subwalk after leaving the closed walk is shorter, namely, $n_2 < n_1$.*

*Proof*

We prove it by contradiction. If $n_2 > n_1$, there is a walk from $v$ of length $n_2$ with a lower average cost than the closed walk, or $\frac{\mathcal{W}_{n_2}(v)}{n_2} < \frac{cwcost(v)}{n_1}$. Otherwise, this minimum-cost walk should repeat the closed walk more times until $n_2$ is less than $n_1$. However, this is impossible because FindClosedWalk finds only closed walks that have the minimum average cost among all walks from $v$. Since the original walk contains a closed walk of $v$, the walk must have a lower (or equal) average cost than $\mathcal{W}_{n_2}(v)$. If $n_2 > n_1$, the original walk is not minimum cost and this violates the premise. Hence, $n_2$ cannot be larger than $n_1$. $\diamond$

Notice that this theorem does not claim that all minimum-cost walks have such a format; instead, it guarantees that among all same-length walks of the minimum cost, there is one walk with this format. It is possible that a walk has a different format with the same cost.

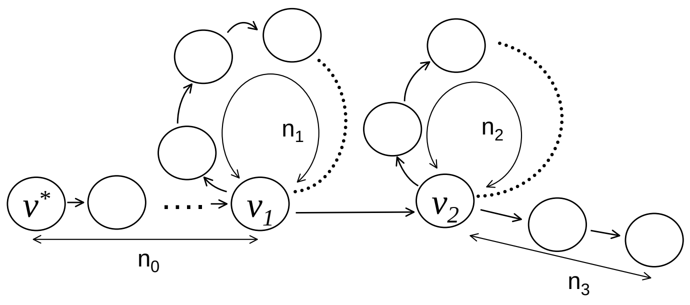


Figure C.2: a walk with multiple closed walks

**Corollary 3** *If a minimum-cost walk has multiple closed walk, there is a walk of the same cost such that the length of the subwalk after leaving the first closed walk is shorter than the length of the first closed walk.*

For example, a minimum-cost walk has two closed walks as shown in Figure C.2. Let $n_1$ and $n_2$ be the lengths of the first and second closed walks. If this walk repeats the second walk $l_2$ times, then $l_2 \times (n_2 - 1) + n_3 - 1 < n_1$. Here we need to subtract one from $n_2$ because the length of a closed walk counts the starting and ending vertices ($v_2$) twice. When this closed walk repeats multiple times, the same vertex should be counted only once.

The above theorem states that the "tail" after leaving the closed walk is shorter than the length of the closed walk. We can find a minimum-cost walk by dividing it into three subwalks: before entering a closed walk, the closed walk, and after leaving the closed walk as illustrated in Figure C.1. The lengths of the first and the third subwalks ($n_0$ and $n_2$) must be less than $|\mathcal{V}|$ by the pigeonhole principle.

In order to find a minimum-cost finite-length walk, our algorithm first checks whether $n$ is small. For a small $n$, a minimum-cost walk does not necessarily contain a closed walk. Such a walk can be found directly by MinimumCostWalk. For a larger $n$, the algorithm finds a closed walk that has a minimum cost. Since the first and the third subwalks are shorter than $|\mathcal{V}|$, they can be found by MinimumCostWalk. Figure C.3 shows the algorithm; it compares which closed walks produce the minimum cost. For each vertex that has a closed walk, it finds a minimum-cost walk from a starting vertex. The length of this walk is $n\mathcal{W}(v^*, v)$; this is $n_0$ in Figure C.1. Then, it computes $lr$; this is the number of times the closed walk repeats.

$$lr = \lfloor \frac{n - n_0}{n_1 - 1} \rfloor \tag{C.1}$$

Finally, it computes the length of the walk after leaving the closed walk.

$$n_2 = (n - n_0) \mod (n_1 - 1) \tag{C.2}$$

The cost of this walk is

$$mcost(v^*, v) + lr \times (cwcost(v) - c(v)) + wcost(v, n_3 - 1) \tag{C.3}$$

The complexity of this algorithm is $O(|\mathcal{V}|^3)$. When $n$ is small, LongFinite-Walk takes $O(|\mathcal{V}|^2 n)$, the same as MinimumCostWalk. When $n$ is larger than $|\mathcal{V}|$, LongFiniteWalk calls MinimumCostWalk, MinimumCostWalk2V, and FindClosed-Walk; their complexity is $O(|\mathcal{V}|^3)$. Then, LongFiniteWalk considers every closed walk reachable from a starting vertex; this takes $O(|\mathcal{V}|^2)$ iterations. Consequently, LongFiniteWalk takes $O(|\mathcal{V}|^3)$; this is independent of $n$. The minimum-cost walk can be constructed by applying *repeated squaring* of the closed walk [22]; this takes $O(\log lr)$.

```
LongFiniteWalk(**input** graph: $G = (\mathcal{V}, \mathcal{E})$, integer: $n$)
/* $mincost$: minimum cost of a walk visiting $n$ vertices */
**begin**
    $mcost := \infty$;
    /* if $n$ is small, find a minimum-cost walk directly */
    **if** $(n < |\mathcal{V}| + 1)$
        MinimumCostWalk($G$, n);
        **for each** $v^*$ /* starting vertex */
            **if** $mincost > wcost(v^*, n)$
                $mincost := wcost(v^*, n)$;
        **return** $mincost$;

    /* $n$ is large */
    MinimumCostWalk($G$, $|\mathcal{V}| + 1$);
    MinimumCostWalk2V($G$);
    FindClosedWalk($G$);
    **for each** $v^*$
        **for each** $v \in \mathcal{V}$
            **if** ($\mathcal{C}walk(v)$ **not** empty)
                $lr := \lfloor \frac{n - n\mathcal{W}(v^*, v)}{nwalk(v) - 1} \rfloor$;
                $n_3 := (n - n\mathcal{W}(v^*, v)) \mod (nwalk(v) - 1)$;
                $newcost := mcost(v^*, v) + lr \times (cwcost(v) - c(v)) + wcost(v, n_3 - 1)$;
                **if** $(mincost > newcost)$
                  $mincost := newcost$;
    **return** $mincost$;
**end**
```

Figure C.3: find minimum-cost walks

# Bibliography

[1] Advanced Power Management Overview. developer.intel.com/IAL/powermgm/apmovr.htm.

[2] StrongARM Development Kit. http://developer.intel.com/design/strong/.

[3] ACPI. www.teleport.com/˜acpi.

[4] ACPI4Linux. phobos.fs.tum.de/acpi/index.html.

[5] Andrea Acquaviva, Luca Benini, and Bruno Riccó. An Adaptive Algorithm for Low-Power Streaming Multimedia Processing. In *Design Automation and Test in Europe*, pages 273–279, 2001.

[6] APM 1.2. www.microsoft.com/HWDEV/busbios/amp_12.htm.

[7] ARM Linux. http://www.arm.linux.org.uk/.

[8] R. Balakrishnan and K. Ranganathan. *A Textbook of Graph Theory*. Springer, 2000.

[9] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, 2 edition, 1997.

[10] L. Benini, G. Castelli, A. Macii, M. Poncino, and R. Scarsi. A Discrete-Time Battery Model for High-Level Power Estimation. In *Design Automation and Test in Europe*, pages 35–39, 2000.

[11] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Transactions on VLSI Systems*, 8(3), June 2000.

[12] Luca Benini, Alessandro Bogliolo, Giuseppe Andrea Paleologo, and Giovanni De Micheli. Policy Optimization for Dynamic Power Management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6):813–833, June 1999.

[13] Luca Benini and Giovanni De Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer, 1997.

[14] John R. Birge and Francois Louveaux. *Introduction to Stochastic Programming*. Springer, 1997.

[15] Jason J. Brown, Danny Z. Chen, Garrison W. Greenwood, Xiaobo Hu, and Richard W. Taylor. Scheduling for Power Reduction in a Real-Time System. In *International Symposium on Low Power Electronics and Design*, pages 84–87, 1997.

[16] Thomas D. Burd and Robert W. Brodersen. Design Issues for Dynamic Voltage Scaling. In *International Symposium on Low Power Electronics and Design*, pages 9–14, 2000.

[17] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.

[18] Lama H. Chandrasena and Michael J. Liebelt. A Rate Selection Algorithm for Quantized Undithered Dynamic Supply Voltage Scaling. In *International Symposium on Low Power Electronics and Design*, pages 213–215, 2000.

[19] Eui-Young Chung, Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. Dynamic Power Management for Non-Stationary Service Requests. In *Design Automation and Test in Europe*, pages 77–81, 1999.

[20] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. Dynamic Power Management Using Adaptive Learning Tree. In *International Conference on Computer-Aided Design*, pages 274–279, 1999.

[21] Don Coppersmith, Peter Doyle, Prahaka Raghavan, and Marc Snir. Random Walks on Weighted Graphs and Applications to On-Line Algorithms. *Journal of the Association for Computing Machinery*, 40(3):421–453, July 1993.

[22] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[23] Robert P. Dick, Ganesh Lakshminarayana, Anand Raghunathan, and Niraj K. Jha. Power Analysis of Embedded Operating Systems. In *Design Automation Conference*, pages 312–315. 2000.

[24] Reinhard Diestel. *Graph Theory*. Springer, 1997.

[25] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive Disk Spin-Down Policies for Mobile Computers. In *Computing Systems*, volume 8, pages 381–413, 1995.

[26] Carla Schlatter Ellis. The Case for Higher-Level Power Management. In *Workshop on Hot Topics in Operating Systems*, pages 162–167, 1999.

[27] Jason Flinn and M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *ACM Symposium on Operating Systems Principles*, pages 48–63, 1999.

[28] Jason Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, 1999.

[29] Arnaud Forestier and Mircea R. Stan. Limits to Voltage Scaling from the Low Power Perspectiv. In *Symposium on Integrated Circuits and Systems Design*, pages 365–370, 2000.

[30] Michael R. Garey and Deavid S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Complteness.* W.H. Freeman and Company, 1979.

[31] Richard Golding, Peter Bosch, and John Wilkes. Idleness is not Sloth. In *USENIX Winter Conference*, pages 201–212, 1995.

[32] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. In *ACM International Conference on Mobile Computing and Networking*, pages 13–25, 1995.

[33] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[34] Paul Greenawalt. Modeling Power Management for Hard Disks. In *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 62–6, 1994.

[35] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics.* Addison-Wesley, 2 edition, 1989.

[36] Dirk Grunwald, Philip Levis, Keith I. Farkas, Charles B. Morrey III, and Michael Neufeld. Policies for Dynamic Clock Scheduling. In *Symposium on Operating system Design and Implementation*, pages 73–86, 2000.

[37] Vadim Gutnik and Anantha P. Chandrakasan. Embedded Power Supply for Low-Power DSP. *IEEE Transactions on VLSI Systems*, 5(4):425–435, 1997.

[38] Willian R. Hamburgen, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timonthy Mann, and Keith I. Farkas. Itsy: Stretching the Bounds of Mobile Computing. *Computer*, 34(4):28–36, April 2001.

[39] John Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2 edition, 1996.

[40] Inki Hong, Miodrag Potkonjak, and Mani B. Srivastava. On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *International Conference on Computer-Aided Design*, pages 653–656, 1998.

[41] Chi-Hong Hwang and Allen CH Wu. A Predictive System Shutdown Method for Energy Saving of Event Driven Computation. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):226–241, 2000.

[42] Chaeseok Im, Huiseok Kim, and Soonhoi Ha. Dynamic Voltage Scheduling Techniques for Low Power Multimedia Applications using Buffers. In *International Symposium on Low Power Electronics and Design*, pages 34–39, 2001.

[43] Tohru Ishihara and Hiroto Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *International Symposium on Low Power Electronics and Design*, pages 197–202, 1998.

[44] Peter Kall and Stein W. Wallace. *Stochastic Programming.* John Wiley & Sons, 1997.

[45] A.R. Karlin, M.S. Manasse, L.A. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Nonuniform Problems. *Algorithmica*, 11(6):542–571, June 1994.

[46] Andrey I. Kibzun and Yuri S. Kan. *Stochastic Programming Problems.* John Wiley & Sons, 1996.

[47] C.M. Krishna and Yann-Hang Lee. Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems. In *Real-Time Technology and Applications Symposium*, pages 156–165, 2000.

[48] P. Krishnan, P.M. Long, and J.S. Vitter. Adaptive Disk Spindown via Optimal Rent-to-Buy in Probabilistic Environments. *Algorithmica*, 23(1):31–56, January 1999.

[49] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *USENIX Winter Conference*, pages 279–292, 1994.

[50] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Power-Aware OS for Interactive Systems. *IEEE Transactions on VLSI Systems*, page to appear.

[51] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Requester-Aware Power Reduction. In *International Symposium on System Synthesis*, pages 18–24, 2000.

[52] Yung-Hsiang Lu and Giovanni De Micheli. Adaptive Hard Disk Power Management on Personal Computers. In *Great Lakes Symposium on VLSI*, pages 50–53, 1999.

[53] Yung-Hsiang Lu and Giovanni De Micheli. Comparing System-Level Power Management Policies. *IEEE Design & Teest of Computers*, 18(2):10–19, March-April 2001.

[54] Yung-Hsiang Lu, Tajana Šimunić, and Giovanni De Micheli. Software Controlled Power Management. In *International Workshop on Hardware/Software Codesign*, pages 157–161, 1999.

[55] Jiong Luo and Niraj K. Jha. Power-Conscious Joint Scheduling of Periodic Task Graphs and Aperiodic Tasks in Distributed Real-Time Embedded Systems. In *International Conference on Computer-Aided Design*, pages 357–364, 2000.

[56] A. Manzak and C. Chakrabarti. Variable Voltage Task Scheduling for Minimizing Energy or Minimizing Power. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 3239–3242, 2000.

[57] Thomas L. Martin and Daniel P. Siewiorek. The Impact of Battery Capacity and Memory Bandwidth on CPU Speed-Setting: A Case Study. In *International Symposium on Low Power Electronics and Design*, pages 200–205, 1999.

[58] Giovanni De Micheli and Luca Benini. System Level Power Optimization: Techniques and Tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, April 2000.

[59] Lode Nachtergaele, Vivek Tiwari, and Nikil Dutt. System and Architecture-Level Power Reduction of Microprocessor-Based Communication and Multimedia Applications. In *International Conference on Computer-Aided Design*, 2000.

[60] Takanori Okuma, Tohru Ishihara, and Hiroto Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. In *International Symposium on System Synthesis*, pages 24–29, 1999.

[61] OnNow. www.microsoft.com/hwdev/onnow/.

[62] Giuseppe Andrea Paleologo, Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. Policy Optimization for Dynamic Power Management. In *Design Automation Conference*, pages 182–187, 1998.

[63] Massoud Pedram and Qing Wu. Design Considerations for Battery-Powered Electronics. In *Design Automation Conference*, pages 861–866, 1999.

[64] Trevor Pering, Tom Burd, and Robert Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In *International Symposium on Low Power Electronics and Design*, pages 76–81, 1998.

[65] Johan Pouwelse, Koen Langendoen, and Henk Sips. Energy Priority Scheduling for Variable Voltage Processors. In *International Symposium on Low Power Electronics and Design*, pages 28–33, 2001.

[66] Qinru Qiu and Massoud Pedram. Dynamic Power Management Based on Continuous-Time Markov Decision Processes. In *Design Automation Conference*, pages 555–561, 1999.

[67] Jan M. Rabaey and Massoud Pedram, editors. *Low Power Design Methodologies*. Kluwer, 1996.

[68] Dinesh Ramanathan and Rajesh Gupta. System Level Online Power Management Algorithms. In *Design Automation and Test in Europe*, pages 606–611, 2000.

[69] Sheldon Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, 1983.

[70] Youngsoo Shin and Kiyoung Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Design Automation Conference*, pages 134–139, 1999.

[71] Youngsoo Shin, Kiyoung Choi, and Takayasu Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. In *International Conference on Computer-Aided Design*, pages 365–368, 2000.

[72] Abraham Silberschatz and Peter B Galvin. *Operating System Concepts.* Addison-Wesley, 4 edition, 1994.

[73] Mani B. Srivastava, Anantha P. Chandrakasan, and Robert W. Brodersen. Predictive System Shutdown and Other Architecture Techniques for Energy Efficient Programmable Computation. *IEEE Transactions on VLSI Systems*, 4(1):42–55, March 1996.

[74] Standard Performance Evaluation Corporation. www.spec.org.

[75] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

[76] Peter G. Viscarola and W. Anthony Mason. *Windows NT Device Driver Development.* Macmillan Technical Publishing, 1999.

[77] Tajana Šimunić, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Michel. Dynamic Voltage Scaling for Portable Systems. In *Design Automation Conference*, pages 524–529, 2001.

[78] Tajana Šimunić, Luca Benini, Peter W Glynn, and Giovanni De Micheli. Dynamic Power Management for Portable Systems. In *International Conference on Mobile Computing and Networking*, pages 11–19, 2000.

[79] Tajana Šimunić, Luca Benini, and Giovanni De Micheli. Event-Driven Power Management of Portable Systems. In *International Symposium on System Synthesis*, pages 18–23, 1999.

[80] Tajana Šimunić, Haris Vikalo, Peter W Glynn, and Giovanni De Micheli. Dynamic Power Management of Laptop Hard Disk. In *Design Automation and Test in Europe*, 2000.

[81] Tajana Šimunić, Haris Vikalo, Peter W Glynn, and Giovanni De Micheli. Energy Efficient Design of Portable Wireless Systems. In *International Symposium on Low Power Electronics and Design*, pages 49–54, 2000.

[82] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[83] Neil H.E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design.* Addison Wesley, 1993.

[84] John Wilkes. Predictive power conservation. Technical report, Hewlett-Packard, HPL-CSP-92-5, 1992.

[85] Laurence A. Wolsey. *Integer Programming.* John Wiley& Sons, 1998.

[86] A.H. Zemanian. Wandering Through Infinity. In *IEEE International Symposium on Circuits and Systems*, pages 1749–1750, 1992.