Requester-Aware Power Reduction

Yung-Hsiang Lu, †Luca Benini, Giovanni De Micheli
CSL, Stanford University, USA. {luyung, nanni}@stanford.edu
† DEIS, Università di Bologna, Italy. lbenini@deis.unibo.it

Abstract

Typically, power reduction is conducted by hardware techniques, such as varying clock frequencies and/or supply voltages. However, hardware devices consume power to serve the requests from software programs. Consequently, it is essential to consider software for power reduction. This paper proposes "requester-aware" power reduction through the collaboration with programs. Experimental results show that this approach can save nearly 70% power with negligible performance degradation.

1. Introduction

Power reduction has become a major goal in designing electronic systems. For portable systems, low power means longer battery life or lighter battery weight. For desktop computers or servers, low power reduces electric bills and improves reliability.

Many run-time power-reduction techniques have been proposed. Dynamic power management is a power-reduction technique that puts idle hardware devices into low-power *sleeping states* to reduce power consumption [3]. A device is *idle* if it has no request to serve; it can sleep to save power. When new requests arrive, the device wakes up entering a *working state* to serve these requests. Requests are generated by programs (also called *requesters*). For example, when a program reads or writes a file on a hard disk, it generates IO requests on the hard disk. Similarly, when a program sends a packet through networks, it generates a request on the network card.

Power state changes have overhead: additional energy consumption and delay [9]; therefore, a device should sleep only when the overhead can be justified by the amount of energy saved. *Power managers* (PM) determine power states according to certain rules (also called *policies*). In the past, power management was mainly implemented in two ways. In the first approach, power managers observe requests at devices to predict future workloads; they are implemented in hardware or device drivers without direct interaction with requesters [3]. On the other extreme, programs can directly control power states through Microsoft's *OnNow* [10] application programming interface (API).

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
ISSS 2000, Madrid, Spain
© 2000 IEEE 1080-1082/00 $10.00

We believe neither extreme is appropriate. This paper proposes one approach between the two extremes. It considers how requests are generated— by running programs. Programs can specify what devices are needed before generating requests. Power states are affected, but not controlled, by individual programs. Making programs aware of power management is suggested in [5]; however, no study has been devoted to examining appropriate interaction between power managers and programs. This paper has three major contributions. First, we explain the advantages to distinguishing individual requesters. Second, we propose a performance-based API which allows programs to indicate their device requirements without detailed hardware information. Finally, we build an experimental environment to demonstrate the effectiveness of this approach. Experimental results show that 70% power of a wireless Ethernet card can be saved with only 3% performance degradation.

2. Power Managers and Requesters

Power managers and requesters can interact in three different ways: no direct interaction, requester-controlled power management, and a new approach between these two extremes, presented in this paper.

2.1. Autonomous Power Management

Most power management techniques fall into the first category. They observe requests at the managed devices and predict the length of future idleness to determine power states [3]. They change power states "autonomously" without direct interaction with requesters; they use an abstract model with a single requester that generates all requests. Figure 1 depicts this approach.

In reality, however, requests may be generated by multiple requesters. For example, requests for a network interface card (NIC) may come from different programs, such as `ftp`, `telnet`, or `netscape`. They have different power consumption patterns and performance requirements. For example, `ftp` creates bursty requests but `telnet` usually creates sparse requests. Furthermore, programs can finish and exit; when no program requires a device, the device can sleep immediately. Without information about requesters, power managers may either waste power to maintain performance unnecessarily or cause delays to save power.

2.2. Requester-Controlled Power Management

The second category is mainly limited to ACPI-compliant devices. ACPI (Advanced Configuration and Power Interface [1]) is proposed by Intel and four other companies for operating-system controlled power management. ACPI is an interface between software and hardware; it replaces Advanced Power Management (APM) by adding devices' capability of being woken up through software. ACPI also allows a device to have multiple sleeping states, distinguished by the amount of power consumed and the time to wake up.

Microsoft's *OnNow* [10] and *ACPI4Linux* [2] support power management for ACPI-compliant devices. OnNow also provides an API for programs to specify power states directly as illustrated in Figure 2. Programs can keep a device in the working state by calling `SetThreadExecutionState`; programs can also wake up a device using `RequestDeviceWakeup`, or receive notice about power-state changes by `WM_POWERBROADCAST`.

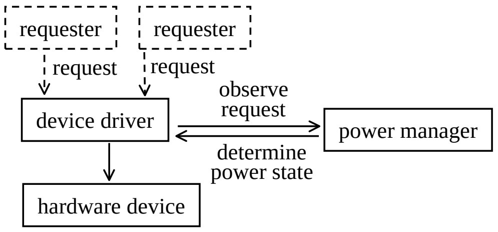

Figure 1: The power manager does not distinguish requesters as indicated by the dotted lines.

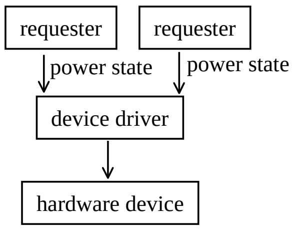

Figure 2: Requesters control power states directly.

OnNow's "power-state based" API has several drawbacks. First, there are no well-defined rules to distinguish power states. Clear distinction is necessary for programs to decide which power states to set. However, clear distinction is impractical due to wide ranges of device parameters. For example, the wakeup delay on a 3.5" hard disk is four times longer than the delay on a 2.5" hard disk [9]. Second, different programs may set the same device to different power states and damage hardware. Finally, technology improvement is changing hardware parameters. If the state-transition energy and delay are negligible for a program, it should not be concerned about the power states of the device.

3. Requester Process

Since there are problems in either approach, we propose a new method between the two extremes. We explain the necessity to consider individual requesters; we use *processes* to distinguish requesters. Power states are affected, but not directly controlled, by running processes. There are multiple advantages in our approach. First, processes are real instantiation of requesters. Processes can be created or terminated; when no process uses a device, the device can sleep immediately. Second, process schedulers arrange the order of execution, hence the generation of requests. Furthermore, processes can specify their device requirements through a function call presented in this paper.

3.1. Power Management and Process Management

When a program executes, it creates one or multiple processes. A process is an instantiation of a program. The process is terminated when the program finishes. When a process terminates, it cannot generate requests any more. A typical computer contains nearly a dozen

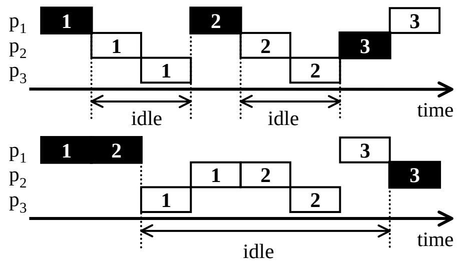

Figure 3: two schedules of three programs. The second schedule reorders execution to make a long, continuous idle period.

of devices, such as a network card, a speaker, a microphone, a hard disk drive, a floppy drive, a DVD drive, a USB controller, and so on. Some devices can be used by only a few programs. When no program uses any of these devices, they can sleep to save power.

Distinguishing requesters by their processes is first proposed in [8]; the study shows that more than 50% power saving compared to traditional aggregate-view timeout power management. Processes provide valuable information for power managers such as the priority, CPU utilization, and timing constraints. When a process runs at high priority or has high CPU utilization, it is likely to generate requests faster than a low priority or low CPU-utilization process. When a process has timing constraints, power management has to consider the impact of state-transition delay on this process. If a power manager does not distinguish individual processes, it may be too aggressive in saving power and fail to meet the requirements of some processes, or be too conservative and waste power.

3.2. Power Management and Process Scheduling

As explained earlier, power-state changes have associated overhead. Specifically, when a device stays in the sleeping state too short, the saved energy cannot compensate state-change energy. A quantitative measurement of the minimum length of sleeping to save energy is called the *minimum sleeping time* (T_{ms}). Power management reduces energy consumption only if a device can stay in the sleeping state longer than T_{ms}; T_{ms} is a device-specific parameter, independent of workloads.

Process execution orders are controlled by *schedulers*. Figure 3 is an example of three running processes (P_1, P_2, and P_3); two of them (P_1 and P_2) generate requests for a device. A block indicates that a process is running. If the process generates requests for a device during this period, the block is filled; an unfilled block indicates that the process does not generate requests for this device. In this figure, each process has multiple blocks ordered as as 1,2, and 3. Schedulers can arrange the order between independent processes to adjust the length of an idle period; schedulers cannot rearrange these blocks and change the order within each process. In the first schedule of this example, the third block of P_2 (filled block number 3) executes earlier than the third block of P_1 (unfilled block number 3). Their execution order is changed in the second schedule. On the other hand, the third block of P_1 (unfilled block number 3) cannot execute earlier than the second block of P_1 (filled block number 2) because they belong to the same process. Scheduling affects power management in two ways:

1. Scheduling can arrange process execution so that idle periods are clustered instead of scattered. Power management is advantageous only when a device can sleep longer than its T_{ms}. Contiguous and long idle periods make power management applicable [4].

2. Even when the original scattered idle periods are long enough to save power, clustered idle periods reduce the number of power-state transitions and state-transition overhead.

In Figure 3, the second schedule is preferred because the idle period is long and continuous. A "power-friendly" scheduler for multiple devices is presented in [7]; it reduces up to 33% power and 40% state transitions.

Schedulers have to know the devices used by each process in advance. This can be achieved in several ways, such as using the recent history to predict future device usage based on the principle of "locality". Another approach is to allow programs to specify their requirements as explained next.

4. Requester Device Requirements

There are three ways to predict whether a device will be used in the future. The first is an aggregate view explained in 2.1; requests are are considered to come from an abstract requester. The second approach predicts devices usage of each process [8]. This section presents the third approach: requesters actively specify which devices will be used through an application programming interface (API). Power managers use such information for determining appropriate power states. We use the conventional term “API” even though it is not restricted to application programs (running at user space); privileged programs running in kernel space can also use the same interface.

Power management, like virtual memory management, should be transparent to most programs. Programs do not determine which virtual pages reside in physical memory; nor should they actively control the power states of devices. Operating systems provide each program with an illusion of large continuous memory addressing space, even though memory is divided into segments and pages. Swapping and demand paging are hidden from most programs. Another example of such illusion is file system; hardware details such as disk plates and cylinders are hidden. A logic abstraction, called file system, is presented to programmers. Similarly, power-state changes should be hidden from programs. Programmers do not have to handle power state change events; therefore, we claim that commands such as `RequestDeviceWakeup` or `SetThreadExecutionState` in OnNow are unnecessary.

Some programs may, however, anticipate predictable performance from hardware devices. Power managers maintain an illusion that these devices are always in their working states ready to serve requests. Programmers do not need to control hardware power states; instead, they can indicate the requirements of hardware devices regardless of the their power states. In contrast to power-state-based API in OnNow, we propose “performance-based” API as an interface between power management and programs.

One function is sufficient for most programs:

```
RequireDevice(device, type, period, wait)
device: hardware device
type:   always / periodic / once / delete
period: requirement period (ms)
wait:   time allowed to wait (ms)
```

This function has four parameters. The first parameter specifies which device is needed. The second parameter specifies how this device is used: always needed when the program is running, periodically, just once, or delete the previous performance specification. The third parameter is the period; it is used only when the second parameter is `periodic`. Finally, the fourth parameter specifies how long the program can wait; a large number indicates that this program can tolerate longer wait. We provide some examples below.

4.1. Periodic Requests

A text editor automatically saves the current content to the hard disk every five minutes (300 second); its performance requirement can be specified as

```
RequireDevice(HardDisk, periodic, 300000, 500)
```

4.2. File Downloading

A user downloads a file from the Internet. When the user clicks “download” from a browser, the browser forks an `ftp` process. File transfer needs both the network card and the hard disk. Because the `ftp` program fetches only one file and then exits, the browser specifies the performance requirement for these two devices.

```
RequireDevice(HardDisk, once, - , 100)
RequireDevice(Network, once, - , 100)
```

Alternatively, the `ftp` can specify

```
RequireDevice(HardDisk, always, - , 100)
RequireDevice(Network, always, - , 100)
```

Note that the first method is preferred because the browser can specify the performance requirement then fork a `ftp` process. If a device needs to be woken up, the wakeup delay can overlap with the process creation time and reduce the overall waiting after clicking the “download” button.

4.3. Internet Streaming Video

In contrast to `ftp`, Internet streaming video continuously downloads images and sounds. For example, `realplayer` shows movies on computer screens. In order to reduce storage space, streaming video programs downloads data to refill their buffers while video and audio are shown. They often generate periodic or nearly periodic network requests. The requests have real-time constraints. If images do not arrive in time, users will see blank screens. These programs can specify their performance requirements as

```
RequireDevice(Network, periodic, 2000 , 10)
```

These examples show that this API is very flexible; it is also simple to understand. Programmers do not have to consider power states and other hardware details.

5. Put It All Together

Figure 4 shows requester-aware power management. The process management provides information about requester creation and termination. The scheduler adjusts execution orders based on the requests reported from a device driver and the performance requirements specified by requesters. The lengths of idle periods are reported to the power manager. The power manager uses all information to determine an appropriate power state.

6. Experiments and Results

We set up an experimental environment on a notebook computer to understand power consumption patterns of individual devices for running different programs. In this paper, we explain one set of experiments. The notebook uses a WaveLan wireless ethernet PC-card; it supports IEEE 802.11 standard [11]. The card and the computer are connected through a Twin 3300-4EXACR PC-card extender; it provides probing points for measuring power. We use a National Instruments data acquisition (DAQ) card to record the power consumption; it can read sixteen power sources simultaneously and up to one million samples per second. Figure 5 shows the experimental environment. Unlike the power profiling presented in [6], our measurement setup is purely hardware-based; there is no software overhead.

We use a one-hour three-phase (I, II, and III) workload; a requester is created for each phase. Phase I sends a file of 50MB; phase II receives a file of the same size. In phase III, a program periodically requests for acknowledgment from a remote machine, similar to `ping`. The period is ten seconds and this repeats fifty times. The network card is idle for five minutes between two adjacent phases and after phase III. The purpose of the first two phases is to measure the bandwidth; the third phase measures the round-trip time (RTT).

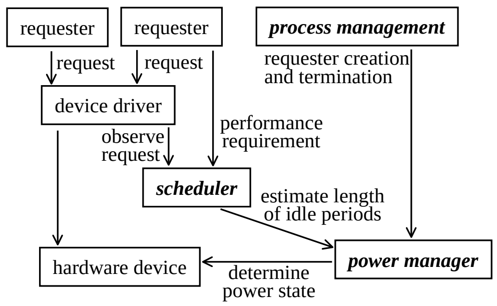

Figure 4: Process management and scheduler provide valuable information for power manager.

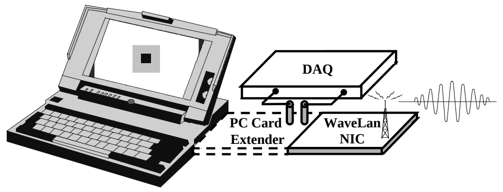

Figure 5: experiment setup

We consider four power management schemes: no power management, power management in IEEE 802.11, using process information (3.1), and using scheduling and our API (3.2 and 4). In IEEE 802.11 standard, a wireless client has to communicate with the access point (wireless server) every 0.1 second to obtain a beacon [11]; the card can sleep for 0.1 second if it is idle. The third scheme turns off the card between two adjacent phases and after the workload finishes. Finally, in the fourth scheme, the card is turned on and off in phase III using our API. We implement the last two schemes in the kernel of Linux 2.2.

Table 1 shows our experimental results. Smaller power and RTT are desirable while larger bandwidth is preferred. The card consumes 859 mW on average in scheme 1. The bandwidth is 180 KB/s for sending and 167 KB/s for receiving; the average RTT is 18.9 millisecond. When we use the IEEE 802.11 power management, the average power reduces to 290 mW. However, the performance degrades up to 50% for sending. This power management scheme does not consider requesters and treats each network packet independently. Figure 6 shows that scheme 2 shuts down the card repetitively during phase I; this scheme takes much longer to finish phase I while schemes 1 and 3 finish phase I earlier (at around 320 second). For scheme 2, RTT increases significantly because the card needs to wait for beacons, in both sending and receiving.

In contrast, our schemes (3 and 4) put the card into the high performance state when a requester is actively sending or receiving packets; it turns off the card when the requester terminates. This scheme reduces nearly 70% power and provides high performance. The bandwidth is slightly lower (3%) compared to the first scheme due to the wake-up delay of the card. No change in the requester program is needed for the third scheme. Finally, the fourth scheme further reduces power in phase III for periodic requests. The power manager

scheme	power	send	receive	RTT
1. no power management	859 mW	180 KB/s	167 KB/s	18.9 ms
2. IEEE 802.11 power management	290 mW	89 KB/s	143 KB/s	327 ms
3. requester-aware (3.1)	277 mW	178 KB/s	162 KB/s	18.9 ms
4. requester-aware (3.1) + scheduling (3.2) + API (4)	270 mW	178 KB/s	162 KB/s	18.9 ms

Table 1: experimental results

wakes up the card periodically before the scheduler selects this requester. This is feasible because the requester uses our API to specify its periodic pattern. This reduces power while maintaining small RTT. Figure 7 shows the power consumption of four schemes during phase III. Scheme 4 saves 64.8% power in phase III compared to scheme 1.

7. Conclusion

This paper examines the interaction between power management and requesters. Requester-specific information helps power managers find power-saving opportunities. Power management, like memory management, should be transparent to programs. We propose performance-based API that allows requesters to specify their device requirements without overloading programmers with excessive hardware details. Experimental results show that nearly 70% power can be reduced by considering requester information.

8. Acknowledgments

This work was supported in part by MARCO/DARPA Gigascale Silicon Research Center and in part by NSF under contract CCR-9901190.

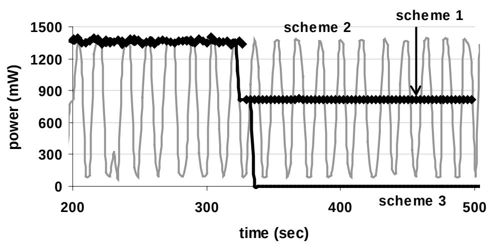

Figure 6: power consumption during phase I

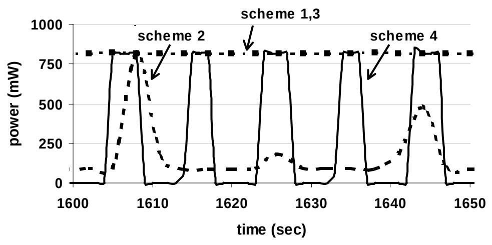

Figure 7: power consumption during phase III

9. References

[1] ACPI. http://www.teleport.com/˜acpi.

[2] ACPI4Linux. http://phobos.fs.tum.de/acpi/index.html.

[3] 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.

[4] 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.

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

[6] 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.

[7] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Low-Power Task Scheduling for Multiple Devices. In *International Workshop on Hardware/Software Codesign*, pages 39–43, 2000.

[8] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Operating-System Directed Power Reduction. In *International Symposium on Low Power Electronics and Design*, 2000.

[9] Yung-Hsiang Lu, Eui-Young Chung, Tajana Šimunić, Luca Benini, and Giovanni De Micheli. Quantitative Comparison of Power Management Algorithms. In *Design Automation and Test in Europe*, pages 20–26, 2000.

[10] OnNow. http://www.microsoft.com/hwdev/onnow/.

[11] 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*, 2000.