Extending Mobile Computer Battery Life through Energy-Aware Adaptation

Jason Flinn

CMU-CS-01-171

December 2001

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
M. Satyanarayanan, Chair
Todd Mowry
Dan Siewiorek
Keith Farkas, Compaq Western Research Laboratory

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Materiel Command (AFMC) under contracts F19628-93-C-0193 and F19628-96-C-0061, DARPA, the Space and Naval Warfare Systems Center (SPAWAR) / U.S. Navy (USN) under contract N660019928918, the National Science Foundation (NSF) under contracts CCR-9901696 and ANI-0081396, IBM Corporation, Intel Corporation, AT&T, Compaq, Hughes, and Nokia.

Keywords: Energy-aware adaptation, application-aware adaptation, power management, mobile computing, ubiquitous computing, remote execution

Abstract

Energy management has been a critical problem since the earliest days of mobile computing. The amount of work one can perform while mobile is fundamentally constrained by the limited energy supplied by one's battery. Although a large research investment in low-power circuit design and hardware power management has led to more energy-efficient systems, there is a growing realization that more is needed—the higher levels of the system, the operating system and applications, must also contribute to energy conservation.

This dissertation puts forth the claim that *energy-aware adaptation*, the dynamic balancing of application quality and energy conservation, is an essential part of a complete energy management strategy. Energy-aware applications identify possible tradeoffs between energy use and application quality, but defer decisions about which tradeoffs to make until runtime. The operating system uses additional information available during execution, such as resource supply and demand, to advise applications which tradeoffs are best.

This dissertation first shows how one can measure the energy impact of the higher levels of the system. It describes the design and implementation of PowerScope, an energy profiling tool that maps energy consumption to specific code components. PowerScope helps developers increase the energy-efficiency of their software by focusing attention on those processes and procedures that are responsible for the bulk of energy use.

PowerScope is used to perform a detailed study of energy-aware adaptation, focusing on two dimensions: reduction of data and computation quality, and relocation of execution to remote machines. The results of the study show that applications can significantly extend the battery lifetimes of the systems on which they execute by modifying their behavior. On some platforms, quality reduction and remote execution can decrease application energy usage by up to 94%. Further, the study results show that energy-aware adaptation is complementary to existing hardware energy-management techniques.

The operating system can best support energy-aware applications by using *goal-directed adaptation*, a feedback technique in which the system monitors energy supply and demand to select the best tradeoffs between quality and energy conservation. Users specify a desired battery lifetime, and the system triggers applications to modify their behavior in order to ensure that the specified goal is met. Results show that goal-directed adaptation can effectively meet battery lifetime goals that vary by as much as 30%.

Acknowledgments

I may perhaps be unusual in that I found the writing of my dissertation to be a very enjoyable task. I think that this is in large part due to the tremendous people with whom I have worked during this project.

My advisor, Satya, made invaluable contributions not just to this dissertation, but also to my development as a research professional. He exhibited a constant optimism in the ultimate usefulness and success of my research that helped to sustain me as I worked through the hard problems. He was also remarkably successful in keeping an eye on the "big picture" and preventing me from getting mired in the details. Perhaps his most valuable contribution, though, lies in teaching me how to express my ideas through both the spoken and written word.

I am indebted to the other members of my thesis committee, Keith Farkas, Todd Mowry, and Dan Siewiorek, for helping me select interesting research directions to explore. Their input led to the development of Spectra, which turned out to be one of the more fun projects in the dissertation.

I have built upon the work of the many members of the Odyssey group. Brian Noble not only laid the foundation for this work by developing Odyssey; he also served as a role model during my first years at CMU. Dushyanth Narayanan, Eric Tilton, and Kip Walker were my brothers-in-arms in the coding trenches—we spent many late nights running experiments, getting demos to run, and building a working system. The newer students in our group, Rajesh Balan and SoYoung Park, have helped me refine my ideas and evaluate my work. It is always a pleasure when one can work with such a talented group of people.

My former 8208 officemates, Hugo Patterson, David Petrou, and Sanjay Rao, served as sounding-boards for many crazy ideas. Bob Baron and Jan Harkes provided a tremendous amount of technical advice about kernel internals and the Coda file system. Eyal de Lara provided a great deal of help with the Puppeteer system.

I'd like to especially thank my father for encouraging me to go back to graduate school, and my mother for reminding me of the importance of having fun. My friends from Penn, Ben Matelson, John Mayne, Patrick O'Donnell, Ted Restelli, and Geoff Taubman, have provided a support network that has withstood the test of time. I'd also like to thank the many new friends I have made here at CMU, including, but not limited to: Rajesh Balan, Angela and Joe Brown, Chris Colohan, Charlie Garrod, Chris Palmer, Carrie Sparks, Greg Steffan, Kip Walker, and Ted Wong.

Contents

List of Figures

Chapter 1

Introduction

Energy is a vital resource for mobile computing. The amount of work one can perform while mobile is fundamentally constrained by the limited energy supplied by one's battery. Unfortunately, despite considerable effort to prolong the battery lifetimes of mobile computers, no silver bullet for energy management has yet been found. Instead, there is growing consensus that a comprehensive approach is needed—one that addresses *all* levels of the system: circuit design, hardware devices, the operating system, and applications.

This dissertation puts forth the thesis that *energy-aware adaptation*, the dynamic balancing of energy conservation and application quality, is an essential part of a comprehensive energy management solution. Occasionally, energy usage can be reduced without affecting the perceived quality of the system. More often, however, significant energy reduction perceptibly impacts system behavior. The effective design of mobile software thus requires striking the appropriate balance between application quality and energy conservation.

It is incorrect to make static decisions that arbitrate between these two competing goals. Dynamic variation in time operating on battery power, hardware power requirements, application mix, and user specifications all affect the optimum balance between quality and energy conservation. Energy-aware adaptation surmounts these difficulties by making decisions dynamically. Applications statically specify *possible* tradeoffs, but defer decisions about which tradeoffs to make until execution. The system uses additional information available during execution, such as resource supply and demand, to advise applications which tradeoffs are best.

This chapter begins with an overview of previous approaches to energy management. It then provides a more detailed vision of energy-aware adaptation and presents the thesis statement. It concludes by presenting a road map for the rest of the dissertation.

1.1 Energy management in mobile computing

Energy management can be viewed as a resource constraint problem. When a computing device is mobile, the supply of energy in its battery must be sufficient to meet the energy demands of the work it will perform before being reconnected to an external power source. Thus, if one wishes to accomplish more work while mobile, one must increase energy supply or decrease demand.

Attacking the supply side of the problem has proven difficult. Historically, battery technology has improved very slowly over time [62]. Further, the need for mobility requires computing systems to be as small and light as possible. Since batteries represent a significant portion of the size and weight of mobile devices, one cannot increase battery size without also increasing these undesirable properties.

Attacking the demand side of the problem has historically proven more fruitful. Advances in low-power circuit design have led to the development of energy-efficient hardware components. For example, the Transmeta Crusoe processor [41] and Bluetooth network technology [30] are both designed to reduce the energy needs of mobile devices.

Research in hardware power management has led to further energy reductions. Ideally, power-managed components expend energy only when they are performing useful work. When not being used, they enter power-saving states which greatly lower power dissipation. Examples of hardware power management are voltage-scaling processors [71, 72, 97], wireless network protocols [43, 44], and disk spin-down algorithms [17, 16, 57].

Unfortunately, advances in low-power circuit design and hardware power management have not been enough to meet the growing energy demands of mobile computers. Partly, this is because lower-level strategies can not capitalize on opportunities for energy management presented by applications and the operating system. Without knowledge of application intent, it is impossible to prioritize activities and save energy by performing only the most important ones. Further, hardware power management strategies must be conservative. Since hardware drivers cannot assess the impact of performance degradation on applications, they reduce energy usage only when the performance impact is almost certain to be negligible.

In recent years, there has been a growing realization that the higher levels of the system, the operating system and applications, must be involved in energy management [18, 66, 89]. This dissertation focuses on these levels and proposes energy-aware adaptation as the key mechanism for implementing higher-level energy management.

1.2 Energy-aware adaptation

Simply stated, energy-aware adaptation is the dynamic balancing of quality and energy conservation. One aspect of quality is *data fidelity*, the degree to which data presented at a client matches the ideal reference copy at a server. Fidelity is a type-specific notion since different kinds of data can be degraded using a variety of type-specific algorithms. For example, a client playing video data could switch to a lower frame rate to save energy when battery life is critical. Yet another aspect of quality is *computational fidelity*, the degree to which the output of a computation matches the highest-quality output that could be produced.

Performance is also an aspect of quality. For example, consider an application which has the ability to execute a portion of its functionality on a remote server. Remote execution can often reduce the energy usage of the mobile client by decreasing the utilization of the CPU and other hardware components. However, remote execution can also lead to increased execution time if a large amount of communication is needed. In such a scenario, energy-aware adaptation is needed to balance the competing goals of performance and energy conservation.

Energy-aware applications statically determine the possible tradeoffs between quality and energy conservation, but defer decisions about which of these tradeoffs to make. During their execution, the system provides support for making these decisions by monitoring energy supply and demand, providing a history of past energy usage, and soliciting user preferences. The system uses this information to provide dynamic advice to applications about which tradeoffs they should make.

1.3 The thesis

Energy-aware adaptation is the focus of this dissertation's thesis:

A collaborative relationship between the operating system and applications can effectively reduce the energy usage of mobile computers. Energy-aware adaptation allows this collaboration to dynamically balance application quality and energy conservation. It is feasible to construct such a system with only modest modification to existing application source code.

1.4 Road map for the dissertation

The rest of this document validates the thesis. The next chapter begins by setting the context for this work. It proposes metrics for evaluating the effectiveness of energy management and discusses the energy-use characteristics of mobile systems. It also describes the Odyssey platform for mobile computing, a framework that will be used to provide operating system support for energy-aware applications.

Chapter 3 describes PowerScope, a tool for measuring software energy usage. PowerScope is an energy profiler—it attributes energy consumption to specific code components of applications and the operating system. By focusing attention on those code components most responsible for energy usage, PowerScope helps developers make their software more energy-efficient. In the context of this dissertation, PowerScope provides the measurement infrastructure necessary to study the effectiveness of energy-aware adaptation.

Chapters 4 and 5 evaluate the feasibility of energy-aware adaptation. They show that applications can modify their behavior to significantly extend the battery lifetimes of the systems on which they execute. Further, they reveal that the benefits of energy-aware adaptation are often very predictable, and that energy-aware adaptation is complementary to existing hardware energy-management techniques. Chapter 4 studies four applications running on the Linux operating system: a video player, a speech recognizer, a map viewer, and a Web browser. Chapter 5 extends these results to shrink-wrapped applications running on closed-source operating systems. It shows how a middleware-based proxy approach can add energy-awareness to Microsoft's PowerPoint 2000 application.

Chapter 6 describes operating system support needed to effectively support energy-aware applications. It introduces goal-directed adaptation, a feedback technique that allows the system to adjust for the current importance of energy conservation. Users specify a goal for battery lifetime, and the system attempts to ensure that the goal is met by guiding applications to adapt their behavior. Then, the chapter shows how the system can improve the effectiveness of goal-directed adaptation by maintaining a history of application energy usage. It describes how the history of energy usage allows the system to support a wider range of adaptation policies and react more agilely to changes in energy supply and demand.

Chapter 7 shows how remote execution represents an additional dimension of energy-aware adaptation. It describes Spectra, a system which enables applications to save energy by partly executing on remote computers. Spectra balances the the competing goals of performance, energy conservation, and application quality in deciding where applications can best locate functionality. It reflects both application resource demand and current resource availability by monitoring CPU, network, energy, and file cache state on local and remote machines, and by using goal-directed adaptation to determine the relative importance of energy conservation.

Related work is discussed in Chapter 8. Chapter 9 concludes the dissertation with a summary of the key contributions. It also discusses future research directions generated by this dissertation.

Chapter 2

Background

This chapter describes the background context of the dissertation. The next section provides an overview of the metrics that will be used to evaluate the effectiveness of energy management strategies. Section 2.2 first explores the diversity of form factors and the energy usage characteristics of mobile computers. It then provides specific details about the two primary platforms that will be used for evaluation: the IBM 560X laptop computer and Compaq's Itsy pocket computer. Section 2.3 describes the Odyssey platform for mobile computing. Odyssey provides the basic building blocks necessary to implement system support for energy-aware adaptation.

2.1 Energy metrics

An ideal battery can be modeled as a finite store of energy. If a battery-powered device expends some amount of energy to perform an activity, the energy supply available for other activities is reduced by that amount. The power usage of a device is its instantaneous rate of energy usage. Power is expressed in units of Watts, while energy is expressed in Joules (Watt-seconds).

For discrete activities such as performing a fixed amount of computation or browsing a Web page, *energy usage* is the best metric for evaluating the impact on battery lifetime. For continuous activities such as displaying streamed video data or backlighting a display, *average power usage* is a more appropriate metric.

When measuring impact on battery lifetime, it is important to capture the energy usage of an entire mobile computing system rather than the isolated energy usage of individual components such as the processor or network interface. A strategy which decreases one component's energy usage may increase the energy usage of other components. For example, network power management can increase the total energy used to transfer a file; although network energy use decreases, other components use more energy because the data takes longer to transfer [21]. Because all hardware components are typically powered by the same battery, strategies that decrease one component's energy needs, but increase total system energy usage are misguided. Unless otherwise noted, the measurements in this dissertation report energy and power usage for the entire mobile computing system under study.

At the next level of detail, it is often useful to characterize the *background power usage* of a device. This is the amount of power dissipated by a mobile computer when no activity of interest to the user is being performed, i.e. while it executes the kernel idle procedure. Most modern processors, including Intel's Pentium and StrongArm chips, provide halt instructions which are called during the idle procedure to minimize power demand. Further, on some mobile laptops, the operating system may use Advanced Power Management (APM) support [35] to place other components in power-savings states. Nevertheless, background power usage can be considerable for devices such as laptop computers. Although components such as the processor and disk enter low power states, they still must be partially powered so that they can be quickly restarted when needed.

Dynamic power usage is the amount of power consumed by an activity above and beyond the background power usage of the device on which it executes. Thus, total power usage is the sum of background and dynamic power usage. Dynamic power usage is a useful metric for estimating the power demand of concurrent activities: the total power usage of two concurrent activities should be roughly equivalent to the sum of the background power usage of the device and the dynamic power usage of the two activities (Section 4.8 explores this issue in more detail). For discrete activities, one can calculate *dynamic energy usage* by multiplying average dynamic power usage by execution time.

The above metrics assume that batteries behave ideally. However, this is rarely true in practice. The most important deviation from ideal behavior is *nonlinearity*—as power draw increases, the total energy that can be extracted from a battery decreases [61]. In addition, batteries may exhibit *recovery*: a reduction in load for a period of time may result in increased capacity. Finally, research has shown that peak power usage can sometimes be a more important factor than average power usage in determining battery capacity [62].

Unless otherwise noted, this dissertation assumes the ideal model for battery behavior. One important reason is simplicity—the impact of nonlinearity, recovery, and peak power usage depend upon the specific characteristics of the mobile system under study, as well as the type of battery technology being employed. Since this dissertation will assess the impact of energy-aware adaptation on a variety of mobile systems, no single model for non-ideal battery behavior will apply. In addition, it is important to note that most of the energy management techniques studied in this dissertation decrease average power use. Thus, the gains reported will be slightly understated due to nonlinear battery behavior.

2.2 Hardware platform characteristics

Mobile computers come in widely varying form factors. High-end laptop computers can weigh over seven pounds with a volume of over 225 cubic inches [33]. In contrast, a typical handheld computer weighs only five ounces with a volume of 7 cubic inches [70]. Current research efforts are reducing mobile computer form factors even further, for example, IBM Research has created a wristwatch computer capable of running Linux [65].

Form factor diversity is generated by a fundamental tradeoff between mobility and functionality. The need for mobility drives manufacturers to create smaller and smaller computing platforms. Size and weight restrictions limit resource availability on these platforms: they have less powerful processors, less storage capacity, and smaller batteries. They therefore can not provide the same level of functionality as their larger counterparts. Since the optimal tradeoff between mobility and functionality is task-dependent, it is reasonable to expect that the current variety of form factors will persist.

Form factor diversity leads to diversity in the energy-use characteristics of mobile devices. Since the battery capacity of small, handheld devices is extremely limited by size constraints, energy-efficiency is typically a primary concern in their design. On the other hand, battery capacity is usually much greater in large devices such as laptop computers—consequently, laptops typically have much higher power consumption than handheld devices.

In this dissertation, I will account for diversity in form factors and energy-use characteristics by validating proposed energy management techniques on two different hardware platforms. These platforms represent two of the most common form factors: laptops and handheld computers. The next two sections describe these platforms: the IBM 560X laptop computer and Compaq's Itsy pocket computer. Section 2.2.3 compares the characteristics of the two platforms.

2.2.1 The IBM 560X laptop computer

The IBM 560X laptop used for evaluation has a 233 MHz Pentium processor and 64 MB of memory. Additionally, either a Lucent 900 MHz or 2.4 GHz WaveLAN PCMCIA card provides 2 Mb/s wireless network access. The use of different network cards reflects changes in my experimental environment over time—the original 900 MHz network was replaced by the 2.4 GHz network. In the dissertation, I will note which network was used for each experiment. Figure 2.1 shows the power usage of several hardware components of the laptop. The measurements were obtained by executing benchmarks that varied the power state of individual hardware components and measuring steady-state power dissipation with a digital multimeter.

As Figure 2.1 shows, background power usage is quite significant: with the CPU idle, the display off, and the network and disk in power-saving states, the laptop draws 5.6 Watts. The processor and display are the most significant power consumers—the processor uses 5.10 Watts to execute a busy-wait loop in which all accesses hit in the L1 cache, and the display consumes from 1.95–4.54 Watts, depending upon screen brightness. The network interface and disk consume less power: 1.46 Watts and 0.88 Watts in their respective idle states.

Component	State	Power (W)
CPU / MMU	CPU Halted	0.00
	Busy Wait	5.10
	Memory Read	3.54
	Memory Write	4.10
Display	Bright	4.54
	Dim	1.95
WaveLAN	Idle	1.46
	Standby	0.18
Disk	Idle	0.88
	Standby	0.24
Other	Idle	3.20

Background power (CPU halted, display dim, WaveLAN & disk standby) = 5.6 Watts.

This figure shows the measured power consumption of components of the IBM 560X laptop. Power usage is slightly but consistently super-linear; for example, the laptop uses 10.28 Watts when the screen is brightest and the disk and network are idle—0.21 Watts more than the sum of the individual power usage of each component. The WaveLAN measurements are for the 900 MHz network card. The last row shows the power used when the display, network, and disk are all powered off. Each value is the mean of five trials—in all cases, the sample standard deviation is less than 0.01 Watts.

Figure 2.1: Power consumption of the IBM ThinkPad 560X

2.2.2 The Itsy pocket computer

The Itsy pocket computer [31] is a high-performance handheld developed by Compaq's Palo Alto Research Labs. Two different Itsy units are used for evaluation: an Itsy v1.5 and an Itsy v2.2. Both models have a StrongArm 1100 processor that can operate at 11 different clock frequencies, ranging from 54.0 MHz to 206.4 MHz, to reduce power demand. Unless otherwise noted, all Itsy measurements in this dissertation use the maximum 206.4 MHz clock frequency. The Itsy v1.5 has 48 MB of DRAM and 32 MB of flash memory—the Itsy v2.2 has 32 MB of DRAM and 32 MB of flash. The Itsy v1.5 is powered by two AAA batteries and contains precision resistors that allow measurement of total power usage as well as the power used by various subsystems. The Itsy v2.2 is powered by a Lithium-Ion rechargeable battery. In addition to precision resistors, it also contains a DS2437 smart battery chip [12] which reports detailed information about battery status and power drain. Both Itsy models lack a wireless network interface—a serial link is used for communication.

Figure 2.2 shows the measured power consumption of several hardware components of the Itsy v1.5. More detailed measurements of the energy characteristics of this platform can be found in [19] and [22]. Viredaz and Wallach have performed detailed power measurements of the Itsy version 2 [93]. Their results show the version 2 power usage is roughly

Component	State	Power (W)
CPU / MMU	CPU Halted	0.00
	Busy Wait	0.43
	Memory Read	0.62
	Memory Write	1.41
Display	Enabled	0.04
UART	Enabled	0.05
	Transmitting	0.12
Other	Idle	0.16

Background power (CPU halted, display and UART enabled) = 0.25 Watts.

This figure shows the measured power consumption of components of the Itsy v1.5. The last row shows the power used when the display and UART are powered off. Each value is the mean of five trials—in all cases, the sample standard deviation is less than 0.01 Watts.

Figure 2.2: Power consumption of the Itsy v1.5

similar to that of the Itsy v1.5.

The background power usage of the Itsy v1.5 is only 0.25 Watts. The CPU is clearly an important power consumer—executing a busy-wait loop consumes an additional 0.43 Watts. The memory subsystem also represents an important source of power demand. The dynamic power used to read data from DRAM memory is 0.62 Watts and the dynamic power needed to write data is 1.41 Watts. The UART (serial network interface) consumes an additional 0.05 Watts when enabled—the UART power drain increases to 0.12 Watts when data is transmitted. The LCD display consumes only 0.04 Watts—the low power consumption can be attributed to the lack of a backlight.

2.2.3 Comparison of platform characteristics

Comparing Figures 2.1 and 2.2, the most striking difference between the two platforms is the order-of-magnitude differential in power demand. The background power usage of the Itsy v1.5 is approximately 22 times less than the background power usage of the IBM 560X. Similarly, the dynamic power needed to execute a busy-wait loop is approximately 12 times less on the Itsy.

It is also clear that the relative range of power demand is much greater for the Itsy v1.5. For example, the ratio of dynamic to background power usage is 5.6 when the write benchmark is executed. For the laptop, the maximum ratio of dynamic to background power is 0.9 (occurring when a busy-wait is executed). Thus, the Itsy is more efficient in its use of energy resources—it expends relatively less power when hardware components are idle.

The relative power expenditure of hardware components varies by platform. For ex-

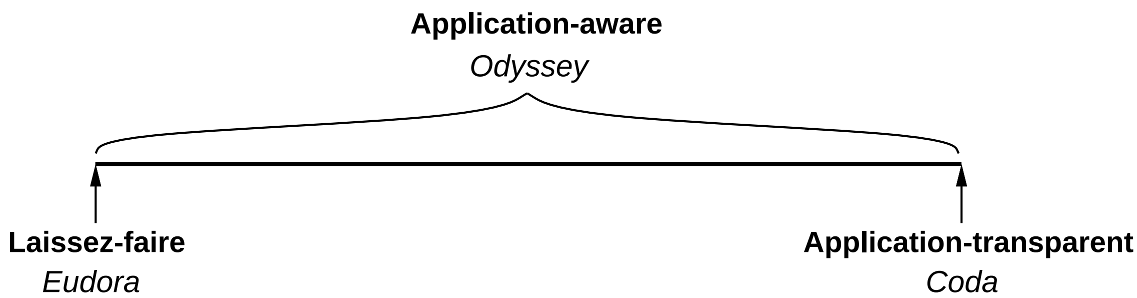


Figure 2.3: Models of adaptation

ample, the memory subsystem is a large power consumer for the Itsy (as shown by the difference between memory write and busy wait power consumption). However, the memory subsystem is a relatively insignificant portion of the laptop's power budget. Similarly, the display represents a relatively more significant portion of the laptop's power budget. An important consequence of this observation is that power tradeoffs between hardware components are platform-specific. One such tradeoff is remote processing, which reduces CPU power demand but increases network power usage. Since the ratio of network to processor power usage differs between the Itsy and the IBM 560X, remote execution will sometimes reduce power usage on one platform but not the other.

2.3 The Odyssey platform for mobile computing

In this dissertation, the Odyssey platform for mobile computing provides the basis for implementing system support for energy-aware adaptation. This section provides a brief overview of the relevant details of Odyssey—a more complete discussion of the design rationale and architecture can be found in [68].

Odyssey provides support for mobile information access through *application-aware adaptation*, a collaborative partnership between the operating system and applications. The system monitors resource levels, notifies applications of relevant changes, and makes resource allocation decisions. The original Odyssey prototype only supported network bandwidth adaptation. This dissertation describes how the infrastructure has been expanded to also support energy-aware adaptation.

Adaptation in Odyssey involves the trading of data or computational quality for resource consumption. For example, a client playing full-color video data from a server could switch to black and white video when bandwidth drops, rather than suffering lost frames. Similarly, a map application might fetch maps with less detail rather than suffering long transfer delays for full-quality maps. Odyssey captures this notion of data degradation through an attribute called *data fidelity*, that defines the degree to which data presented at a client matches the reference copy at a server.

Odyssey also supports applications which can vary the quality of their computations to adjust for variations in resource availability. For example, a speech recognition engine running on a handheld device with little processing power might use a smaller, task-specific vocabulary to provide speech-to-text translations with reasonable latency. Odyssey captures this notion through an attribute called *computational fidelity*, that defines the degree to which the output of the computation matches the highest-quality output that could be produced.

Fidelity is a type-specific notion since different kinds of data and computation can be degraded differently. Fidelity may often be multi-dimensional—for example, a video player may choose to degrade quality by using a greater amount of lossy compression, reducing the size of the video display, or decreasing the video frame rate. Since the minimal level of fidelity acceptable to the user can be both context and application dependent, Odyssey allows each application to specify the fidelity levels it currently supports.

Odyssey is designed to support multiple applications concurrently executing on a mobile client. The need to coordinate resource management across applications mutes the effectiveness of many previous approaches to mobile computing. For example, commercial applications such as Eudora [74] provide vertically integrated support for mobility, in which each application assumes that it has full use of available network bandwidth. Eudora implicitly adapts to network bandwidth by transmitting messages in order of importance. Even a more sophisticated toolkit approach such as Rover [39] only pays minimal attention to resource coordination. Odyssey provides centralized monitoring and coordinated resource management that controls the use of limited resources by applications.

Figure 2.3 places application-aware adaptation in context, spanning the range between two extremes. At one extreme, adaptation is entirely the responsibility of individual applications. This *laissez-faire* approach, used by commercial software packages such as Eudora, avoids the need for system support. But, it fails to address the issue of application concurrency. At the other extreme, *application-transparent adaptation*, the system bears full responsibility for both adaptation and resource management. This approach, exemplified by the Coda file system [40], is especially attractive for legacy applications because they can run unmodified. Application concurrency is well supported, but application diversity is not, since control of fidelity is entirely in the hands of the system.

Odyssey's client architecture is shown in Figure 2.4. Odyssey is conceptually part of the operating system, even though it is implemented in user space for simplicity. The *viceroy* is the Odyssey component responsible for monitoring the availability of resources and managing their use. Code components called *wardens* encapsulate type-specific functionality. There is one warden for each data type in the system. Several applications have been modified to use Odyssey, including a video player, a speech recognizer, a map viewer, a Web browser, and a virtual reality application.

Odyssey provides applications with two separate interfaces. The first interface allows an application to express its resource expectations. If resource levels stray beyond the specified expectations, Odyssey notifies the application through an upcall. The application then adjusts its fidelity to match the new resource level and communicates a new set of

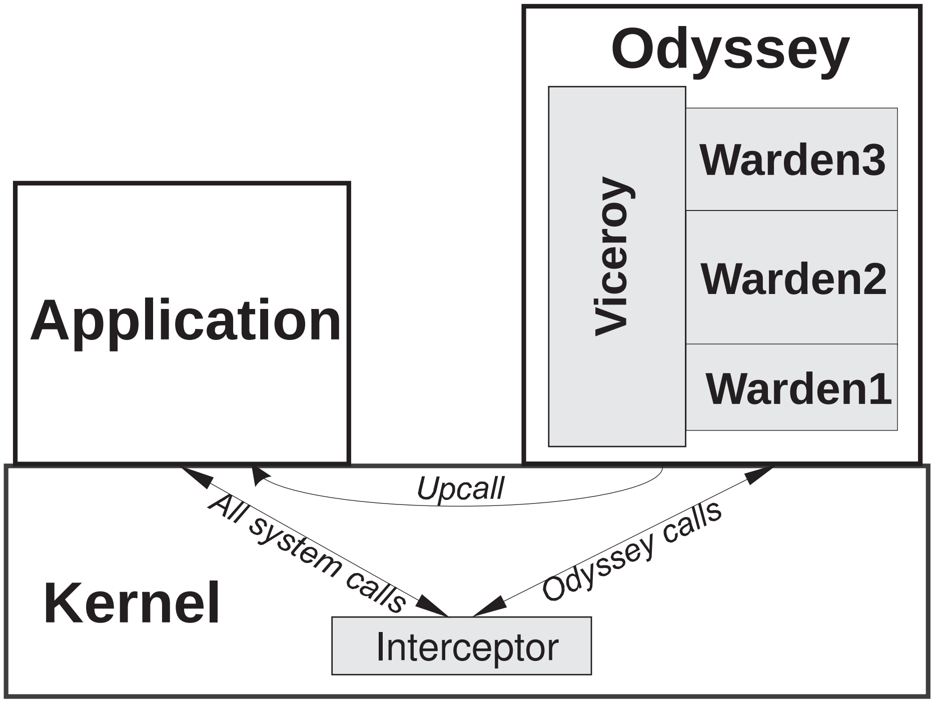


Figure 2.4: Odyssey architecture

expectations to Odyssey. This interface is most appropriate for applications which perform continuous operations, can change fidelity levels dynamically, and understand their own resource requirements.

The second interface allows applications to periodically query Odyssey to determine the fidelity level at which they should operate. This interface is more appropriate for applications which perform discrete operations or do not know their own resource requirements. An application first describes the operation it is about to perform. Odyssey then estimates the resource demand of the application, matches demand to current resource availability, and returns the fidelity level most appropriate for the operation.

Some applications, such as the Odyssey Web browser and map viewer, use a proxy to avoid modifications to application source code. Other applications, such as our video player and speech recognizer, are modified to interact directly with Odyssey. In all cases, the total amount of code that needs to be modified is very small, i.e. less than 1000 lines of code.

In its current instantiation, Odyssey assumes that applications are cooperative. Thus, Odyssey expects that applications will execute at the fidelity it specifies. However, by adding appropriate operating system support, Odyssey could potentially enforce its resource allocation decisions by detecting and penalizing misbehaving applications.

2.4 Summary

This chapter began by discussing the metrics that will be used to evaluate energy management strategies in this thesis. Total energy usage will be used for discrete activities, while average power usage will be used for continuous activities. The chapter then described the two primary hardware platforms for evaluation: the IBM 560X laptop and the Itsy pocket computer. The choice of these platforms reflects the diversity in form factors and energy efficiency in mobile computing.

Finally, this chapter described the Odyssey platform for mobile computing, which will provide the basis for implementing system support for energy-aware adaptation. Odyssey provides support for *application-aware adaptation*, a collaborative partnership between the operating system and applications. Odyssey monitors resource levels, notifies applications of relevant changes, and makes resource allocation decisions. Applications modify data or computational fidelity to adjust their resource demands to meet changing resource availability. The previous version of Odyssey only supported network bandwidth adaptation—this dissertation extends Odyssey to support energy-aware adaptation.

Chapter 3

PowerScope: Profiling application energy usage

One of the keys to progress in energy-efficient software design is the ability to attribute energy consumption to specific software components. Unfortunately, there is currently a dearth of tools which have the ability to measure the energy impact of software. This chapter describes how I have constructed one such tool, called PowerScope, which fills this need by profiling application energy usage.

CPU profilers such as `prof` and `gprof` have proven useful for software performance optimization because they expose code components wasteful of CPU cycles. In a similar fashion, PowerScope helps developers design energy-efficient software by using statistical profiling to map energy consumption to program structure. Using PowerScope, one can determine what fraction of the total energy consumed during a certain time period is due to specific processes in the system. Further, one can drill down and determine the energy consumption of different procedures within a process. By providing such feedback, PowerScope allows attention to be focused on those system components responsible for the bulk of energy consumption. As improvements are made to these components, PowerScope quantifies the benefits and helps expose the next target for optimization. Through successive refinement, a system can be improved to the point where its energy consumption meets design goals. PowerScope also helps developers expose energy-related bugs in their code which are not revealed through traditional testing methodology. For example, a busy-wait loop may have no perceptible performance impact, but PowerScope would reveal its wasteful energy usage.

Section 3.1 discusses the important considerations in the design of an energy profiler. The implementation of PowerScope is detailed in Section 3.2. Section 3.3 evaluates the tool, focusing on two key issues: the accuracy with which PowerScope attributes energy costs to specific processes and procedures, and the overhead of its operation.

3.1 Design considerations

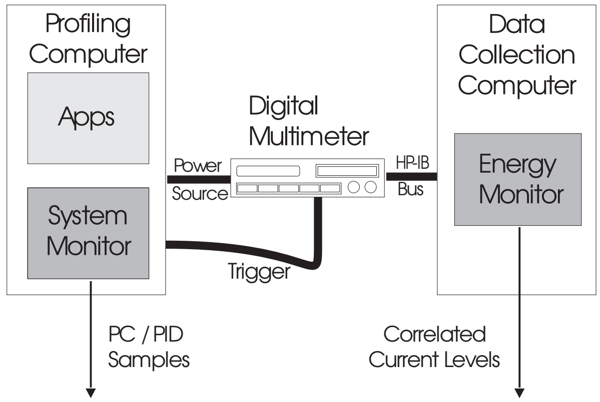
The design of PowerScope follows from its primary purpose: enabling application developers to build energy-efficient software. PowerScope's design scales to complex applications, which may consist of several concurrently executing threads of control, and which may run on a variety of mobile platforms. For both simple and complex applications, PowerScope provides developers detailed and accurate information about energy usage.

The most important consideration in the design of PowerScope is the need to gather sufficient information to produce a detailed picture of application activity. The usefulness of a profiling tool is directly related to how definitively it assigns costs to specific application events. Attributing costs in detail enables developers to quickly focus their attention on problem areas in the code. While it is certainly desirable to map energy costs to specific processes, the added detail of mapping energy costs to procedures within each process can provide valuable information. PowerScope therefore reports both sets of information, attributing energy usage to both processes and to procedures within each process. As will be discussed in Section 3.3.1, the specific hardware characteristics of the system being monitored limit the minimum procedure size that can be accurately profiled.

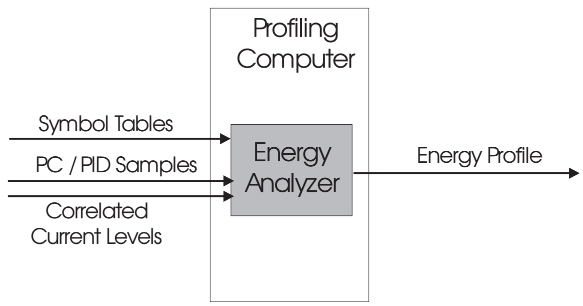
It is also important for PowerScope to monitor the activities and energy use of *all* processes executing on a computer system. Complex applications often consist of several concurrently executing processes. Further, profiling the activity of only a single process omits critical information about total energy usage. For instance, a task which blocks frequently may expend large amounts of energy on the screen, disk, and network when the processor is idle. Asynchronous activity, such as network interrupts, can also account for a significant portion of energy consumption. An energy profiler which monitors energy usage only when a specific process is executing will not account for the energy expended by these activities.

Another consideration in PowerScope's design is that the tool be easily portable between different hardware platforms. The power dissipation characteristics of mobile platforms differ widely, so energy optimizations for one platform may be inappropriate for others. To determine the best design for a particular application, developers may need to profile it on a variety of mobile devices. PowerScope therefore does not require specific hardware to be present on a mobile computer, not does it depend upon platform-specific knowledge such as device power characteristics. This design minimizes the effort required to generate profiles on different hardware devices.

Finally, PowerScope is designed to minimize the overhead that it imposes on the system it is monitoring. This overhead is reflected both in additional CPU usage and in additional energy expended during execution. Because overhead affects the profile results, minimizing the profiling overhead helps maximize the accuracy of the generated profile. The design of PowerScope includes several optimizations, described in the next section, that reduce its impact on the system being profiled.



(a) Data collection



(b) Off-line analysis

This figure shows how PowerScope generates an energy profile. As applications execute on the profiling computer, the System Monitor samples system activity and the Energy Monitor samples power consumption. Later, the Energy Analyzer uses this information to generate an energy profile.

Figure 3.1: PowerScope architecture

3.2 Implementation

3.2.1 Overview

The prototype version of PowerScope, shown in Figure 3.1, uses statistical sampling to profile the energy usage of a computer system. To reduce overhead, profiles are generated by a two-stage process. During the data collection stage, the tool samples both the power consumption and the system activity of the profiling computer. PowerScope then generates an energy profile from this data during a later analysis stage. Because the analysis is performed off-line, it creates no profiling overhead.

During data collection, PowerScope uses two computers: a profiling computer, on which applications execute, and a data collection computer, which is used to reduce overhead. A digital multimeter samples the power consumption of the profiling computer. I require that this multimeter have an external trigger input and output, as well as the ability to sample DC current or voltage at high frequency. The present implementation uses a Hewlett Packard 3458a digital multimeter, which satisfies both these requirements. The data collection computer controls the multimeter and stores current samples.

An alternate implementation would be to perform measurement and data collection entirely on the profiling computer using an on-board digital multimeter with a PCI or PCMCIA interface. However, this implementation makes it very difficult to differentiate the energy consumed by the profiled applications from the energy used by data collection and by the operation of the on-board multimeter. Further, the present implementation makes switching the measurement equipment to profile different hardware platforms much easier.

The functionality of PowerScope is divided among three software components. Two components, the System Monitor and Energy Monitor, share responsibility for data collection. The System Monitor samples system activity on the profiling computer by periodically recording information which includes the program counter (PC) and process identifier (PID) of the currently executing process. The Energy Monitor runs on the data collection computer, and is responsible for collecting and storing current samples. Because data collection is distributed across two monitor processes, it is essential that some synchronization method ensure that they collect samples closely correlated in time. I have chosen to synchronize the components by having the digital multimeter signal the profiling computer after taking each sample.

The final software component, the Energy Analyzer, uses the raw sample data collected by the monitors to generate the energy profile. The analyzer runs on the profiling computer since it uses the symbol tables of executables and shared libraries to map samples to specific procedures. There is an implicit assumption in this method that the executables being profiled are not modified between the start of profile collection and the running of the off-line analysis tool.

3.2.2 The System Monitor

The System Monitor consists of a device driver which collects sample data and a user-level daemon process which reads the samples from the device driver and writes them to a file. The device driver is currently implemented as a Linux loadable kernel module (LKM), allowing PowerScope to run without any modification to kernel source code. Although the System Monitor currently operates only on the Linux operating system, this design approach should enable it to be relatively portable to other operating systems.

The design of the System Monitor is similar to the sampling components of Morph [100] and DCPI [3]. The present implementation samples system activity when triggered by the digital multimeter. Each twelve byte sample records the value of the program counter (PC) and the process identifier (PID) of the currently executing process, as well as additional information such as whether the system is currently handling an interrupt. This assumes that the profiling computer is a uniprocessor—a reasonable assumption for a mobile computer.

Samples are written to a circular buffer residing in kernel memory. This buffer is emp-

```
pscope_init (u_int size);
pscope_read (void* sample,
             u_int size,
             u_int* ret_size);
pscope_start (void);
pscope_stop (void);
```

Figure 3.2: PowerScope API

tied by the user-level daemon, which writes the samples to a file. The daemon is triggered when the buffer grows more than 7/8 full, or by the end of data collection.

The System Monitor records a small amount of additional information that is used to generate profiles. First, it associates each currently executing process with the pathname of an executable. Then, for each executable it records the memory location of each loaded shared library and associates the library with a pathname. For Linux versions 2.1 and greater, the kernel `d_path()` routine is used to associate each process or library with a corresponding pathname. For previous versions of Linux in which this method is unavailable, the System Monitor associates each process or library with a device and inode number. In both cases, this mapping is recorded only once for each library or executable. The information is written to the sample buffer during data collection, and is used during off-line analysis to associate each sample with a specific executable image.

The programming interface shown in Figure 3.2 allows applications to control profiling. The API is implemented as a user-level library which marshals arguments and calls `ioctl` operations on the PowerScope device driver.

The user-level daemon calls `pscope_init()` to set the size of the kernel sample buffer. Since there is a tension between excessive memory usage and frequent reading of the buffer by the daemon, the buffer size has been left flexible to allow efficient profiling of different workloads. The daemon calls `pscope_read()` to read samples out of the buffer. The `pscope_start()` and `pscope_stop()` system calls allow application programs to precisely indicate the period of sample collection. Multiple sets of samples may be collected one after the other; each sample set is delineated by start and end markers written into the sample buffer.

3.2.3 The Energy Monitor

The Energy Monitor runs on the data collection computer and communicates with the digital multimeter. There is no specific operating system requirement for the data collection computer; it currently runs Windows 95 to take advantage of manufacturer-provided device drivers for the multimeter.

The Energy Monitor configures the multimeter to periodically sample the power usage of the profiling computer. The specific method of power measurement depends upon the system being profiled. For many laptop computers, the simplest method is to sample the current drawn through the laptop's external power source. Usually, the voltage variation is extremely small, for example it is less than 0.25% for the IBM 701C and 560X laptops. Therefore, current samples alone are sufficient to determine the energy usage of the system. The battery is removed from the laptop while measurements are taken to avoid extraneous power drain caused by charging. Current samples are transmitted asynchronously to the Energy Monitor which stores them in a file for later analysis.

An alternate method can be employed for systems such as the Compaq Itsy v1.5 pocket computer that provide internal precision resistors for power measurement [92]. For the Itsy, the Energy Monitor configures the multimeter to measure the instantaneous differential voltage, V_{diff}, across a $20\ m\Omega$ precision resistor located in the main power circuit. The instantaneous current, I, can therefore be calculated as $I = V_{diff}/0.02\ \Omega$. Since the voltage being supplied to the computer, V_{supp}, does not vary significantly, these measurements are sufficient to calculate instantaneous power usage, P, as $P = V_{supp} * I$. Further, because the Itsy contains additional internal resistors, the same method can be used to profile the isolated power usage of Itsy subsystems.

The above method is also useful when the maximum current drawn by the profiling computer exceeds the rated capacity of the measurement equipment. In such cases, PowerScope can measure the current drop across a precision resistor inserted between the profiling computer and its external power supply.

Sample collection is driven by the multimeter clock. Synchronization with the System Monitor is provided by connecting the multimeter's external trigger input and output to I/O pins on the profiling computer. The specific pins are platform-specific—for example, I use parallel port pins for the IBM 560X laptop and general purpose I/O pins for the Itsy. Immediately after the multimeter takes a power sample, it toggles the value of an input pin. This causes a system interrupt on the profiling computer, during which the System Monitor samples system activity. Upon completion, the System Monitor triggers the next sample by toggling an output pin (unless profiling has been halted by the `pscope_stop` system call). The multimeter buffers this trigger until the time to take the next sample arrives. This method ensures that the power samples reflect application activity, rather than the activity of the System Monitor.

The original PowerScope design used the clock of the profiling computer to drive sample collection. Although simpler to implement, that design had the disadvantage of biasing the profile values of activities correlated with the system clock. Since PowerScope drives sample collection from the multimeter, the lack of synchronization between the multimeter and profiling computer clocks introduces a natural jitter that makes clock-related bias very unlikely. Using the multimeter clock also allows PowerScope to generate interrupts at a finer granularity then that allowed by using kernel clock interrupts.

An alternative approach would be to trigger interrupts using processor performance counters such as those found on the StrongARM 1100 and Pentium II chips. I rejected this approach due to portability concerns. Some processors, such as the Pentium chip used in the IBM 560X laptop, lack performance counters. Further, methods for accessing performance counters vary by processor family, and thus require architecture-specific code.

The user may specify the sample frequency as a parameter when the Energy Monitor is started. With the multimeter currently being used, the maximum sample frequency is approximately 700 samples per second.

3.2.4 The Energy Analyzer

The Energy Analyzer generates an energy profile of system activity. Recall that total energy usage can be calculated by integrating the product of the instantaneous current, I_t, and voltage, V_t, over time, as follows:

$$E = \int I_t V_t dt \tag{3.1}$$

This value can be approximated by simultaneously sampling both current and voltage at regular intervals of time Δt. Further, in the systems which I have measured, V_t is constant within the limits of accuracy for which I am striving. PowerScope therefore calculates total energy over n samples using a single measured voltage value, V_{meas}, as follows:

$$E \approx V_{meas} \sum_{t=0}^{n} I_t \Delta t \tag{3.2}$$

The Energy Analyzer reads the raw data generated by the monitors and associates each current sample collected by the Energy Monitor with the corresponding sample collected by the System Monitor. It assigns each sample to a process bucket using the recorded PID value. Samples that occurred during the handling of an asynchronous interrupt, such as the receipt of a network packet, are not attributed to the currently executing process but are instead attributed to a bucket specific to the interrupt handler. If no process was executing when the sample was taken, the sample is attributed to a kernel bucket. The energy usage of each process is calculated as in Equation 3.2 by summing the current samples in each bucket and multiplying by the measured voltage (V_{meas}) and the sample interval (Δt).

The Energy Analyzer then generates a summary of energy usage by process, such as the one shown in Figure 3.3(a). Each entry displays the total time spent executing the process, calculated by multiplying the total number of samples that occurred while the process was executing by the sample period. Each entry also displays the total energy usage of the process and its average power usage, which is calculated by dividing energy usage by execution time.

Energy Usage by Process:

Process	Elapsed Time (s)	Total Energy (J)	Average Power (W)
/obj/odyssey/bin/janus	40.521	489.522	12.081
kernel	40.572	301.210	7.424
Interrupts-Wavelan	27.654	296.287	10.714
/obj/odyssey/bin/xanim	18.073	218.458	12.087
/usr/X11R6/bin/XF86_SVGA	13.369	162.659	12.167
/obj/odyssey/bin/viceroy	11.730	141.101	12.029
/obj/odyssey/bin/editor	2.130	25.087	11.776
/usr/bin/netscape3	1.495	17.791	11.901

(a) Partial summary of energy usage by process

Energy Usage Detail for process /obj/odyssey/bin/viceroy

User-level procedures:

Procedure	Elapsed Time (s)	Total Energy (J)	Average Power (W)
Internal_Signal	0.210	2.585	12.327
ExaminePacket	0.165	1.939	11.763
Dispatcher	0.160	1.872	11.693
sftp_DataArrived	0.106	1.285	12.162
IOMGR_CheckDescriptors	0.096	1.159	12.064
IOMGR_Select	0.078	0.955	12.177

(b) Partial detail of process energy usage

This figure shows a sample energy profile for a computer running multiple concurrent applications. Part (a) shows a portion of the summary of energy usage by process. Part (b) shows a portion of the detailed profile for a single process,

Figure 3.3: Sample energy profile

The Energy Analyzer repeats the above steps for each process to determine the energy usage by procedure. The process and shared library information stored by the System Monitor is used to reconstruct the memory address of each procedure from the symbol tables of executables and shared libraries. Then, the PC value of each sample is used to place the sample in a procedure bucket. When the profile is generated, procedures that reside in shared libraries and kernel procedures are displayed separately. Figure 3.3(b) shows a partial profile of one typical process.

3.3 Validation

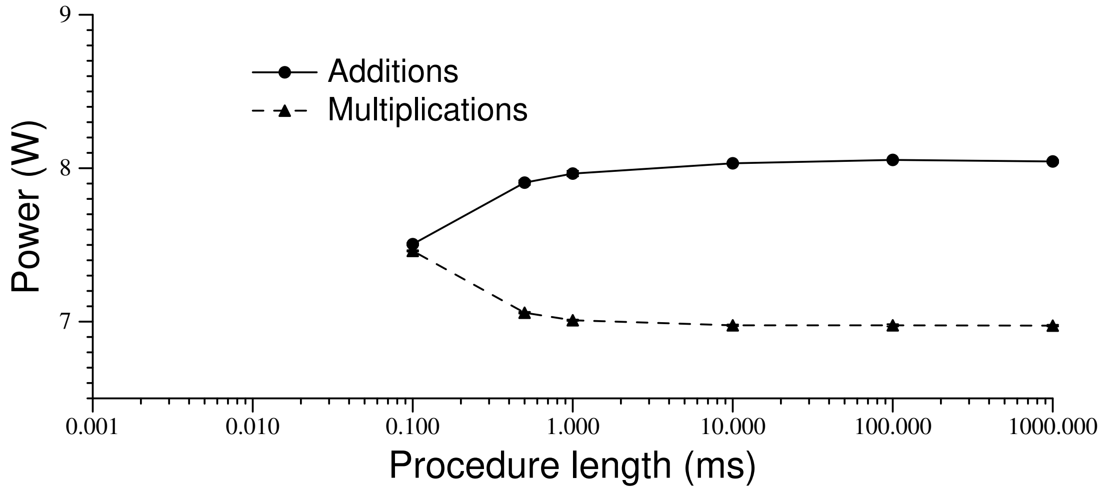
For PowerScope to be effective, it must accurately determine the energy cost of processes and procedures. Further, it must operate with a minimum of overhead on the system being measured to avoid significantly perturbing the profile results.

I created benchmarks to assess how successful PowerScope is in meeting both of these goals. Each benchmark was run on two hardware platforms, the Compaq Itsy v1.5 pocket computer and the IBM ThinkPad 560X laptop computer.

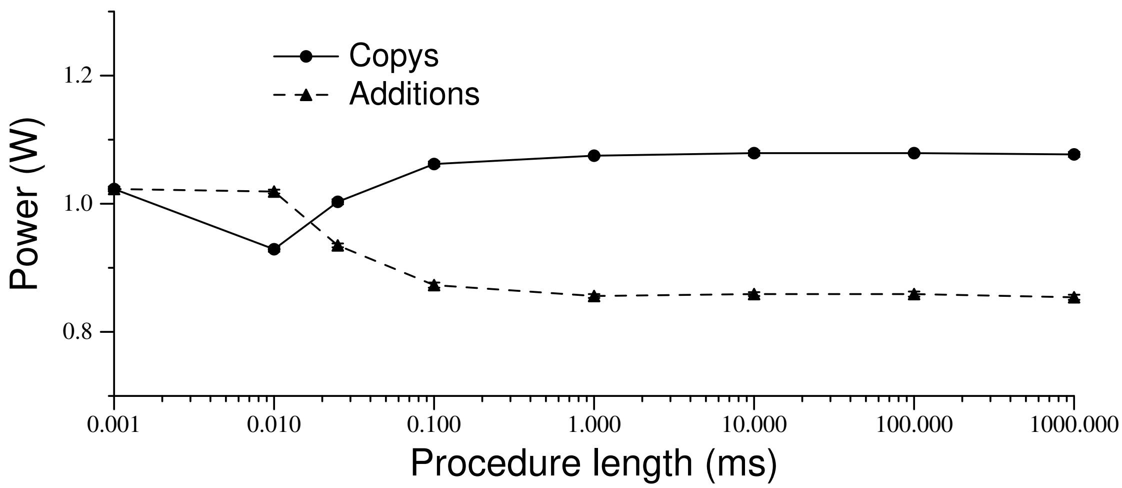
3.3.1 Accuracy

There are several factors which potentially limit PowerScope's accuracy. First, the digital multimeter's power measurements are not truly instantaneous—the multimeter's A/D converter must measure the input signal over a period of time. However, this period, or *integration time*, is normally quite small. In the case of the HP 3458a multimeter used for these experiments, the minimum integration time is only 1.4 μs. Second, there will be some capacitance in the computer system being measured. High-frequency changes in power usage may not be measurable at the point in the circuit where the multimeter probes are attached. Finally, there is a delay between the time when the multimeter takes a measurement and the time when the corresponding kernel sample is taken; this delay includes time to propagate an electrical signal to the profiling computer and time to handle the corresponding hardware interrupt. If a procedure is of sufficiently short duration, a sample taken during its execution may be incorrectly attributed to a procedure which executes later. Combined, these factors limit PowerScope's accuracy—there will be some minimum event duration below which PowerScope will be unable to accurately determine the event's power usage.

I measured the minimum event duration by running a benchmark which alternates execution between two different procedures. Each procedure has a known power usage and runs for a configurable length of time. When these procedures are of sufficiently long duration, for example, one second, PowerScope can accurately determine the power usage of each procedure. However, as the duration of the two procedures is shortened, PowerScope will eventually be unable to successfully determine their individual power usages. To ensure maximum accuracy, I used the highest sampling rate supported by my current measurement equipment for these measurements—approximately 700 samples per second.



(a) PowerScope accuracy for ThinkPad 560X



(b) PowerScope accuracy for Itsy v1.5

This figure shows PowerScope's accuracy as a function of the length of the event being measured. Each graph shows the power usage reported for two different procedures which execute alternately. As the procedure length is reduced, PowerScope is eventually unable to distinguish the individual power usage of the two procedures. The measurements in the top graph were performed on the IBM ThinkPad 560X laptop and the measurements in the bottom graph were performed on the Compaq Itsy v1.5. Each point represents the mean of ten trials—the (barely noticeable) error bars in each graph show 90% confidence intervals. Note that procedure length, on the x-axis, is displayed using a log scale.

Figure 3.4: PowerScope accuracy

Figure 3.4(a) shows the results of running the benchmark on the 560X laptop. The figure shows the power usage of two procedures reported by PowerScope for a variety of procedure durations. The first procedure performs additions in an unrolled loop and has a power usage of 8.04 Watts (measured with a duration of 1 second). The second procedure performs multiplications in an unrolled loop and has a power usage of 6.97 Watts. White it may seem unintuitive that multiplication requires less power than addition, less multiplication instructions execute per unit of time, meaning that the total *energy* needed to perform a multiplication is higher.

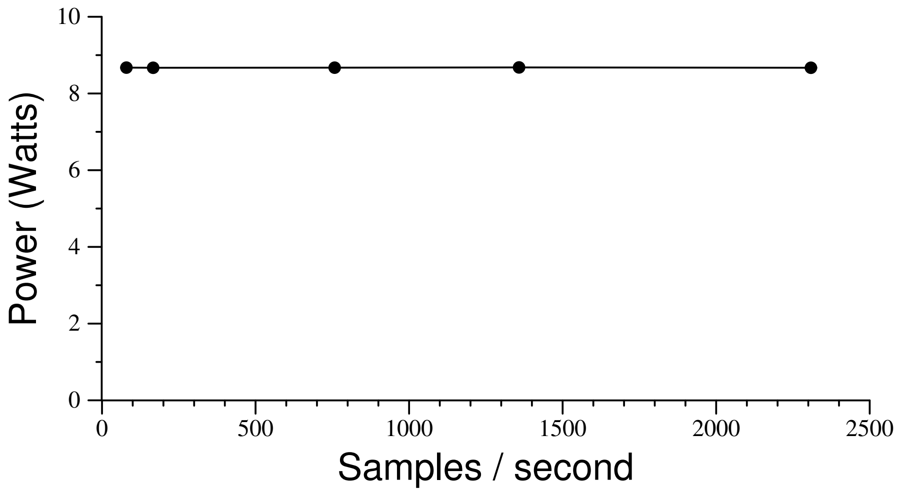
PowerScope correctly reports the individual power usage of each procedure within experimental error for durations of 10 ms. When the procedure length is set to 1 ms., PowerScope reports slightly inaccurate results (within 1% of the correct value). As the procedure duration is further decreased, PowerScope's accuracy also decreases. At a duration of 100 μs., PowerScope is unable to distinguish the power usage of individual procedures. In this case, the limiting factor is probably the capacitance of the laptop computer.

Figure 3.4(b) shows the results of running the benchmark on the Itsy v1.5. Because the power used to perform multiplications on the Itsy is very similar to the power used to perform additions, the benchmark replaces the multiplication procedure with one that copies data from one memory location to another. The copies are performed in an unrolled loop and all memory references hit in the first-level data cache. The power usage of the addition procedure is 0.85 Watts, and the power usage of the copy procedure is 1.08 Watts.

On the Itsy, PowerScope correctly reports individual power usage within experimental error for durations of 1 ms. The reported values are slightly inaccurate with a procedure length of 100 μs. (within 3% of the correct value). Interestingly, at 10 μs., PowerScope reports a higher power usage for the addition procedure than for the copy procedure. Because these results are significant within experimental error, they strongly indicate that the power measurements are being perturbed by the latency between the time when measurements are taken and the time when the System Monitor samples system activity on the profiling computer. Power samples that should be attributed to one procedure are instead being incorrectly attributed to the other.

The preceding measurements cannot detect one possible source of inaccuracy: the effect of variation in the sampling frequency. To quantify this potential effect, I measured the power usage of the IBM560X laptop while it executed the benchmark application for approximately 10 seconds using a procedure duration of 1 ms.. I sampled power usage at various frequencies using the HP3458a multimeter. Since I did not use PowerScope for these experiments, any variation in power usage can be attributed to variation in the sampling frequency.

Figure 3.5 shows the results of these experiments for five different frequencies between 80 and 2309 samples per second (the maximum frequency for the multimeter without PowerScope). The measured power usage varies by less than 1 mW. Thus, the choice of sampling frequency does not significantly impact measurement results.



This figure shows that variation in the sample frequency does not impact power measurements. It shows the power usage of an IBM560X laptop executing a benchmark application, measured at several different sampling frequencies. Each point represents the mean of five trials—90% confidence intervals are not visible on this graph.

Figure 3.5: Effect of variation in the sample frequency

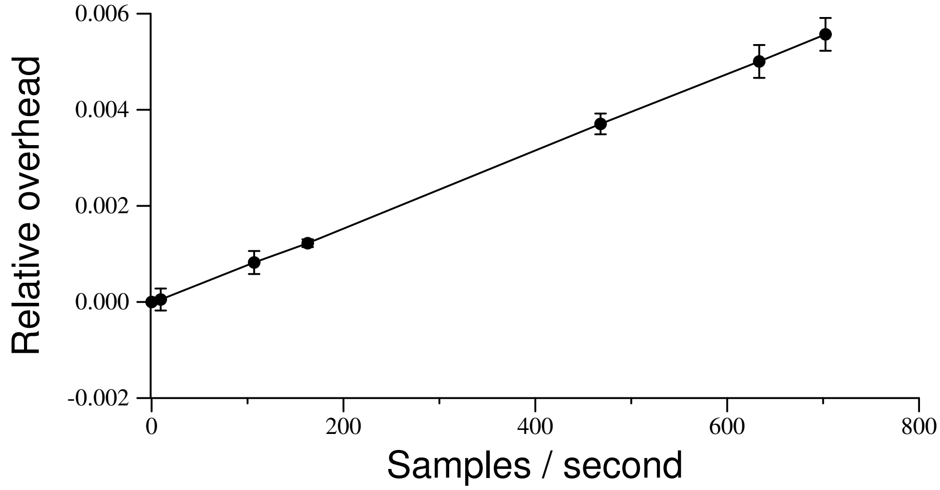
3.3.2 Overhead

Running PowerScope imposes a small overhead on the system being profiled due to the activity of the System Monitor. The impact can be expressed in terms of both CPU usage and additional energy consumption on the profiling computer.

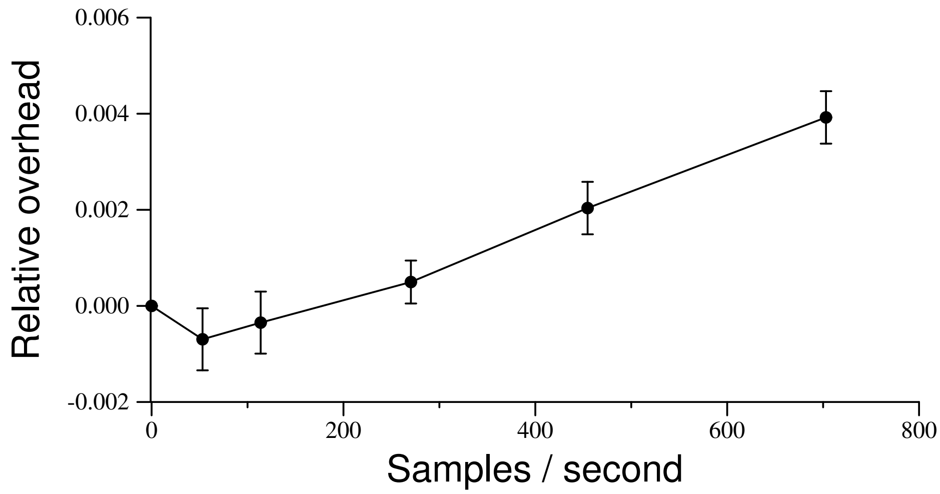
To determine PowerScope's CPU overhead, I measured the execution time of the benchmark described in Section 3.3.1 for a variety of sampling rates, and compared the results to the execution time of the benchmark when PowerScope was not running. For the 560X laptop, latency was measured using the Pentium cycle counter. For the Itsy, the Linux `gettimeofday` system call was used. This benchmark does not include the cost of periodically writing data to a file for long-running profiles. However, because the file write is amortized across a large number of samples, the additional CPU cost should be quite low.

Figure 3.6 shows the results of these experiments for the two platforms. In both cases, PowerScope's CPU overhead is quite low—less than 0.6% at the maximum sampling rate of the multimeter. Note that although two measurements in Figure 3.6(b) show a negative overhead, the upper bound of each measurement's 90% confidence interval is greater than zero, meaning that the anomalies can likely be attributed to experimental error.

To determine PowerScope's energy overhead, I used PowerScope to measure the energy usage of the benchmark described in Section 3.3.1 for a variety of sampling rates. For comparison, I measured the energy consumption of the benchmark without PowerScope by using the digital multimeter to measure the energy consumption of the profiling computer during a period when the benchmark was the only activity executing. Because there is a marked difference between energy consumption depending upon whether or not the bench-



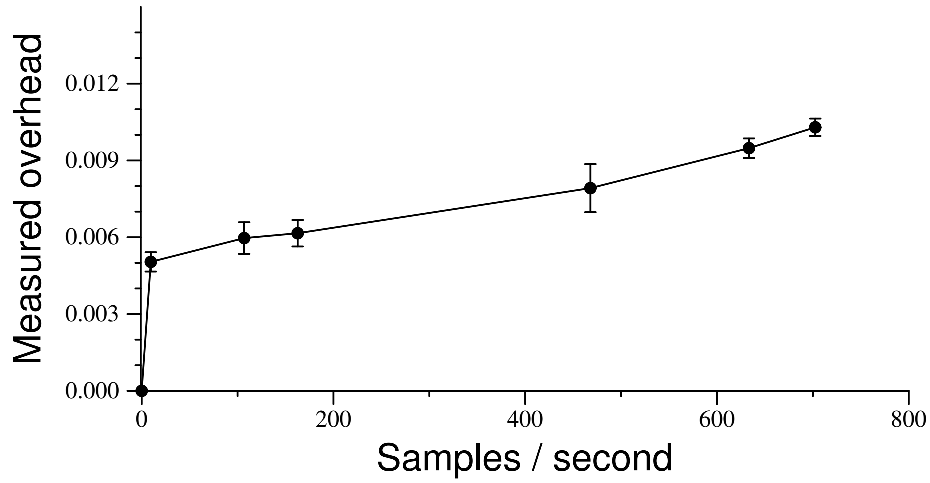
(a) CPU overhead for ThinkPad 560X



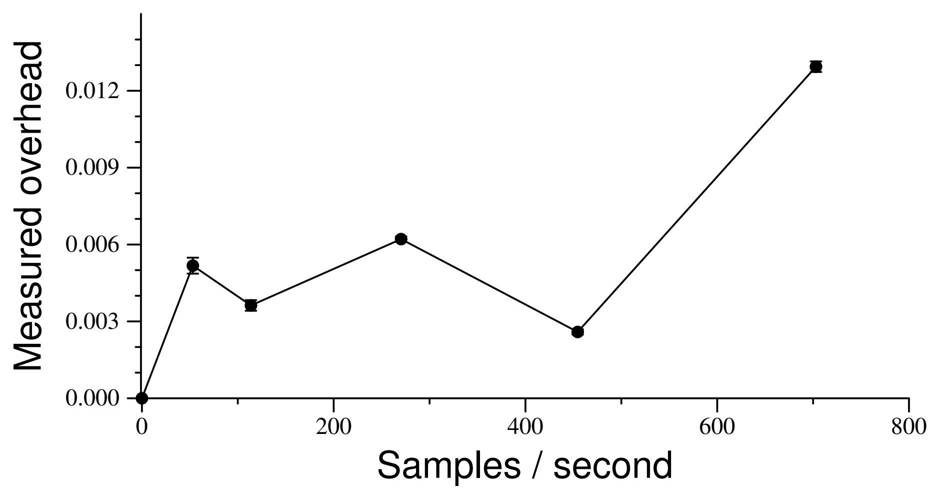
(b) CPU overhead for Itsy v1.5

This figure shows PowerScope's relative CPU overhead as a function of the sampling frequency. Both graphs show the additional time required to complete a fixed amount of calculations while PowerScope is used to profile energy consumption. The measurements in the top graph were performed on the IBM ThinkPad 560X laptop and the measurements in the bottom graph were performed on the Compaq Itsy v1.5. Each point represents the mean of ten trials—the error bars in each graph show 90% confidence intervals.

Figure 3.6: PowerScope CPU overhead



(a) Energy overhead for ThinkPad 560X



(b) Energy overhead for Itsy v1.5

This figure shows PowerScope's measured relative energy overhead as a function of the sampling frequency. Both graphs show the amount of additional energy usage reported by PowerScope relative to baseline energy consumption without PowerScope. The measurements in the top graph were performed on the IBM ThinkPad 560X laptop and the measurements in the bottom graph were performed on the Compaq Itsy v1.5. Each point represents the mean of ten trials—the error bars in each graph show 90% confidence intervals.

Figure 3.7: PowerScope energy overhead

mark is running, it was possible to manually identify those samples collected when the benchmark was executing and to calculate the average power consumption of the benchmark. The energy usage of the benchmark was then calculated by multiplying its average power usage by its measured execution time.

Figure 3.7 shows the results of these experiments for the two platforms. In both cases, PowerScope's energy overhead is low—about 1.0% for the ThinkPad 560X and 1.3% for the Itsy at the maximum sampling rate. Like the CPU benchmark, this value does not include the cost of periodically writing data to a file for long-running profiles.

While the laptop results show that energy overhead increases fairly regularly with the sample rate, the Itsy results are decidedly more irregular. While it is unclear precisely what leads to this effect, it is possible that different sampling frequencies induce slightly different cache effects on the benchmark application. Such cache effects would be much more noticeable on the Itsy since its cache is smaller and memory usage makes up a much larger percentage of its overall power budget.

3.4 Summary

This chapter has described the design and implementation of the PowerScope energy profiler. PowerScope helps developers design energy-efficient software by mapping energy consumption to specific processes executing on a computer system, and to individual procedures within those processes.

There are several important considerations in the design of an energy profiler. First, it should map energy consumption to code components as accurately as possible. Second, it should report the energy usage of *all* activities occurring on the computer system during the profiling period. Third, it should maximize portability to support profiling on a variety of hardware platforms. Finally, it should minimize the amount of overhead imposed during the profiling period.

Evaluation of PowerScope shows that it is successful in meeting these goals. Depending upon the system being profiled, PowerScope can accurately attribute energy to events with durations as small as $100\,\mu$s. Further, profiles such as the one in Figure 3.3 report the energy consumption of all processes that execute during the profiling period. The results in Section 3.3 show that PowerScope can generate profiles on two very different computing platforms. Finally, PowerScope's overhead is quite low. On the platforms evaluated, its CPU overhead was at most 0.6% and its energy overhead was at most 1.3%.

Chapter 4

Energy-aware adaptation

In order to design energy-aware software, it is first necessary to understand how software design choices impact system energy use. This need motivated me to perform a detailed study of energy usage for several applications that might commonly be found in mobile computing environments. In this chapter, I discuss this study, its results, and the implications for energy-aware application and operating system design.

4.1 Goals of the study

The primary goal of the study was to assess the feasibility of energy-aware adaptation. For adaptation strategies to be effective, it is crucial that changes in application behavior yield significant energy savings. If the potential savings are meager, then developers will not modify applications to make them energy-aware. Further, the resulting energy savings should be predictable. The more accurately an adaptive system can project the impact of potential changes, the quicker it can converge upon the optimum balance between application quality and energy conservation.

In this study, the main dimension of energy-aware adaptation is the tradeoff between application fidelity and energy use. Since fidelity is an application-dependent metric, it was necessary to measure several different applications as they executed at different levels of fidelity. The greater the difference in energy use, the more effective energy-aware adaptation is likely to be for a given application.

Although it was not my main focus, I also studied another dimension of energy-aware adaptation: the tradeoff between execution location and energy use. Although it seems intuitive that offloading computation to a server can reduce client energy usage, this is not always the case. Even though the energy used by the client's CPU will decrease, this benefit may be offset by increased network energy use. Alternatively, remote execution may prove to be slower than local execution when communication needs are significant, creating a tradeoff between energy conservation and performance.

A secondary goal of the study was to assess the impact of existing hardware power-management strategies, such as spinning down the hard drive, disabling wireless network receivers, and turning off the display. I measured both the stand-alone impact of these strategies, as well as their impact when combined with energy-aware adaptation. I was especially interested in confirming that the energy savings from energy-aware adaptation could enhance those achievable through hardware power management. Although these distinct approaches to energy savings seem composable, it was important to verify this experimentally.

4.2 Methodology

I measured the energy used by four applications: a video player, a speech recognizer, a map viewer, and a Web browser. My selection of these particular applications was driven by several factors. First, these are applications that are commonly used when mobile. Speech recognition enables hands-free operation, while map viewing assists navigation. It is also reasonable to expect that mobile access of Web and video data will increase as wireless bandwidth becomes more plentiful. Second, each application has at least one definable dimension of fidelity, allowing me to study the relationship between fidelity and energy use. The final factor in selecting these applications was their extensibility. All ran on Linux, and three had freely available source code. The fourth, Netscape, had not yet released source code, but had a well-defined interface for extension. In the study, source-code availability allowed me to use PowerScope to gain a more detailed picture of application energy use. The large degree of extensibility allowed me to easily modify applications to support multiple levels of fidelity. However, making applications energy-aware does not always require application or operating system source code, as will be seen in the next chapter.

I used the IBM 560X laptop as the primary platform for evaluation. Since the video player, map viewer, and Web browser have a considerable amount of platform-specific code, it would have been prohibitively time-consuming to port them to the Itsy. However, the speech recognizer proved relatively easy to port—I therefore measured its energy consumption for both platforms and compared the results.

I first observed the applications as they operated in isolation, and then as they operated concurrently. In each experimental trial, the fidelity of an application was fixed at a constant value.

I explored sensitivity of energy consumption to data fidelity by using four data objects for each application: that is, four video clips, four speech utterances, four maps, and four Web images. I first measured the baseline energy usage for each object at highest fidelity with hardware power management disabled. Secondly, I measured energy usage with hardware power management enabled. Then, I successively lowered the fidelity of the application, measuring energy usage at each fidelity with hardware power management enabled.

This sequence of measurements is directly reflected in the format of the graphs presenting the results: Figures 4.2, 4.5, 4.10, and 4.15. Since a considerable amount of data is condensed into these graphs, I explain their format here even though their individual contents will not be meaningful until the detailed discussions in Sections 4.4 through 4.7.

For example, consider Figure 4.2. There are six bars in each of the four data sets on the x-axis; each data set corresponds to a different video clip. The height of each bar shows total energy usage, and the shadings within each bar show energy usage by software component. The component labeled "Idle" aggregates samples that occurred while executing the kernel idle procedure—effectively a Pentium `hlt` instruction. The component labeled "WaveLAN" aggregates samples that occurred during wireless network interrupts.

For each data set, the first and second bars, labeled "Baseline" and "Hardware-Only Power Mgmt.", show energy usage at full fidelity with and without hardware power management. The difference between the first two bars gives the application-specific effectiveness of hardware power management. Each of the remaining bars shows the energy usage at a different, reduced fidelity level with hardware power management enabled. The difference between one of these bars and the first bar ("Baseline") gives the combined benefit of hardware power management and fidelity reduction. The difference between one of these bars and the second one ("Hardware-Only Power Mgmt.") gives the additional benefit achieved by fidelity reduction above and beyond the benefit achieved by hardware power management.

The measurements for all bars except "Baseline" were obtained while powering down as many hardware components as possible for each application. After ten seconds of inactivity, I transitioned the disk to standby mode. Further, I modified the network communication package used to place the wireless network interface in standby mode except during remote procedure calls or bulk transfers. Finally, I turned off the display during the speech application. Since the other applications were interactive, the display was continuously enabled during their operation.

These are the maximally aggressive power management strategies that could be employed with the computer used in the study. However, more recent mobile computers support further dimensions of power management—for example, several mobile processors can reduce their power and energy requirements by decreasing the CPU clock frequency. Hence, the effectiveness of hardware power management appears to be increasing with time. As it will be shown in this chapter that hardware power management and fidelity reduction are synergistic, it is reasonable to expect that the benefits of fidelity reduction will continue to increase in the future as hardware power management becomes more effective.

4.3 Experimental setup

For this study, I used the ThinkPad 560X described in Section 2.2.1 as the client. The client ran the Linux 2.2 operating system. All servers were 200 MHz Pentium Pro desktop computers with 64 MB of memory. The client communicated with servers over a 2 Mb/s wireless WaveLAN network operating at 900 MHz.

I measured application and system energy use with PowerScope, sampling at a rate of approximately 600 times per second. To avoid confounding effects due to non-ideal battery behavior, the client used an external power supply. Further, to eliminate the effects of charging, the client's battery was removed.

4.4 Video player

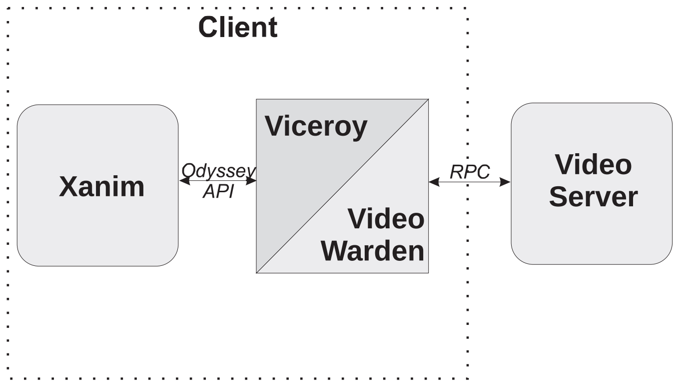


Figure 4.1: Odyssey video player

4.4.1 Description

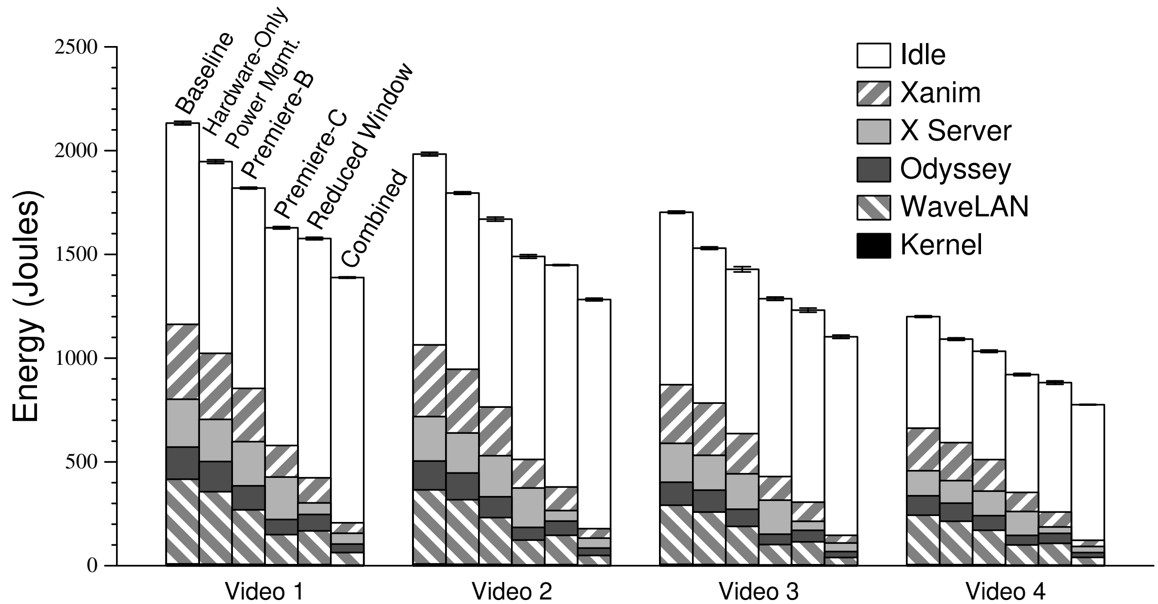
I first measured the impact of fidelity on the video application shown in Figure 4.1. Xanim fetches video data from a server through Odyssey and displays it on the client. It supports two dimensions of fidelity: varying the amount of lossy compression used to encode a video clip, and varying the size of the window in which it is displayed. There are multiple tracks of each video clip on the server, each generated off-line from the full fidelity video clip using Adobe Premiere. They are identical to the original except for size and the level of lossy compression used in frame encoding.

4.4.2 Results

Figure 4.2 shows the total energy used to display four videos at different fidelities. At the baseline fidelity, much energy is consumed while the processor is idle because of the limited bandwidth of the wireless network—not enough video data is transmitted to saturate the processor. Most of the remaining energy is consumed by asynchronous network interrupts, the Xanim video player, and the X server.

For the four video clips, hardware-only power management reduces energy consumption by a mere 9–10%. There is little opportunity to place the network in standby mode since it is nearly saturated. Most of the reduction is due to disk power management—the disk remains in standby mode for the entire duration of an experiment.

The bars labeled Premiere-B and Premiere-C in Figure 4.2 show the impact of lossy compression. Whereas the baseline video is encoded at QuickTime/Cinepak quality level 2, Premiere-B and Premiere-C are encoded at quality levels 1 and 0 respectively. Premiere-C, the highest level of compression, consumes 16–17% less energy than hardware-only



This figure shows the total energy used to display four QuickTime/Cinepak videos from 127 to 226 seconds in length, ordered from right to left above. For each video, the first bar shows total energy usage without hardware power management or fidelity reduction. The second bar shows the impact of hardware power management alone. The next two show the impact of lossy compression. The fifth shows the impact of reducing the size of the display window. The final bar shows the combined effect of lossy compression and window size reduction. The shadings within each bar detail energy usage by software component. Each value is the mean of five trials—the error bars show 90% confidence intervals.

Figure 4.2: Energy impact of fidelity for video playing

power management. Note that these gains are understated due to the bandwidth limitation imposed by the wireless network. With a higher-bandwidth network, I could raise baseline fidelity and thus transmit better video quality when energy is plentiful. The relative energy savings of Premiere-C would then be higher.

By examining the shadings of each bar in Figure 4.2, it can be seen that compression significantly reduces the energy used by Xanim, Odyssey, and the WaveLAN device driver. However, the energy used by the X server is almost completely unaffected by compression. I conjectured that this is because video frames are decoded before they are given to the X server, and the size of this decoded data is independent of the level of lossy compression.

To validate this conjecture, I measured the effect of halving both the height and width of the display window, effectively introducing a new dimension of fidelity. As Figure 4.2 shows, shrinking the window size reduces energy consumption 19–20% beyond hardware-only power management. The shadings on the bars confirm that reducing window size significantly decreases X server energy usage. In fact, within the bounds of experimental error, X server energy consumption is proportional to window area.

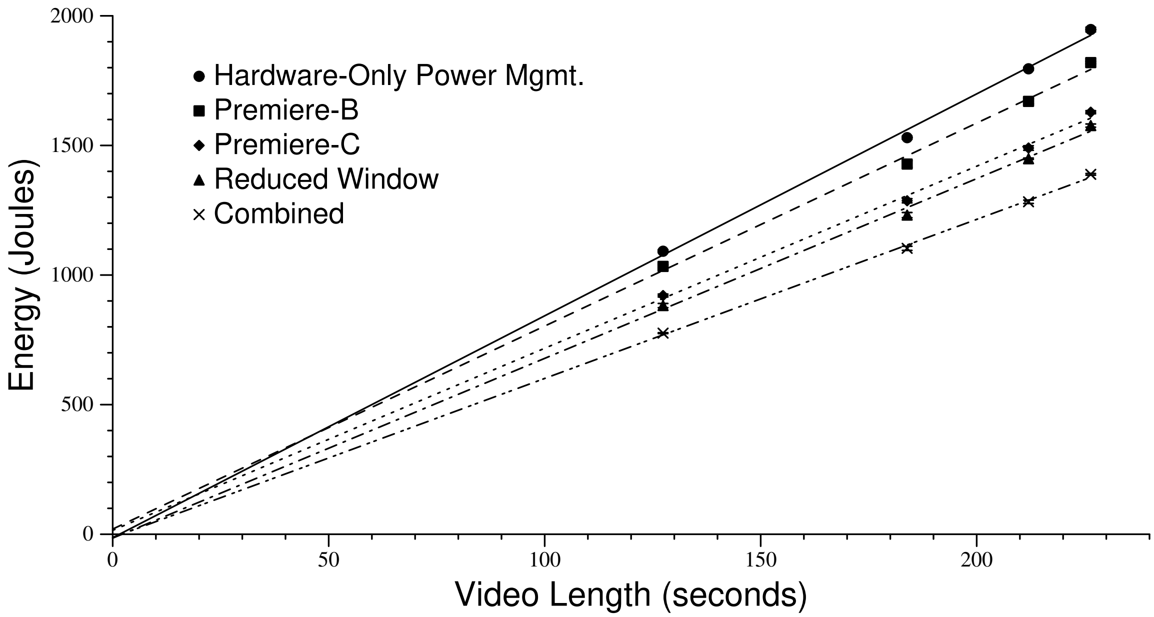


Figure 4.3: Predicting video player energy use

This figure shows the relationship between system energy use, data fidelity, and video length. For each level of data fidelity, four data points show the total energy used to display the four videos from Figure 4.2—the corresponding line represents the best linear fit through these points. All measurements were taken with hardware power management enabled. The (barely noticeable) error bars show 90% confidence intervals for energy use.

Finally, I examined the effect of combining Premiere-C encoding with a display window of half the baseline height and width. This results in a 28–30% reduction in energy usage relative to hardware-only power management. Relative to baseline, using all the techniques (hardware power management, lossy encoding, and reducing the window size) together yields about a 35% reduction.

From the viewpoint of further energy reduction, the rightmost bar of each data set in Figure 4.2 seems to offer a pessimistic message: there is little to be gained by further efforts to reduce fidelity. Virtually all energy usage at this fidelity level occurs when the processor is idle.

Fortunately, this is precisely where advances in hardware power management can be of the most help. For example, consider a modern mobile processor, such as TransMeta's Crusoe chip [41], which can reduce its clock frequency to save power and energy. As the fidelity of the video is reduced, the processor can operate at correspondingly lower clock frequencies since the needed computation per frame is smaller. With such a processor, the total energy used by the lowest fidelity will be significantly reduced.

Figure 4.3 shows video player energy use as a function of video length for five different levels of data fidelity: baseline, Premiere-B, Premiere-C, reduced window size, and

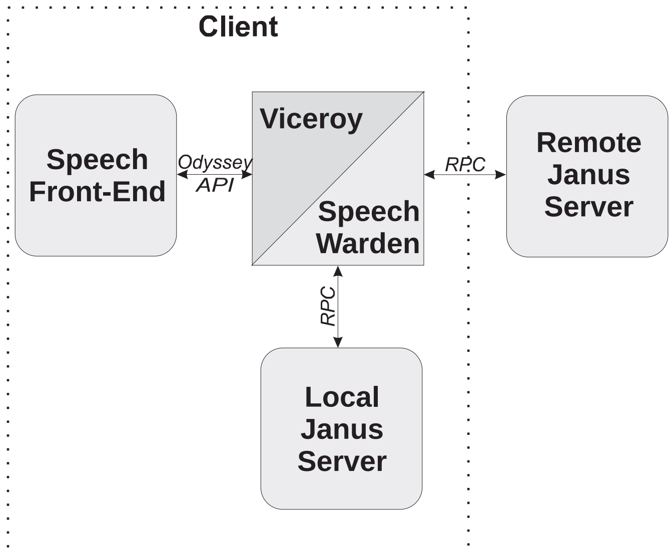


Figure 4.4: Odyssey speech recognizer

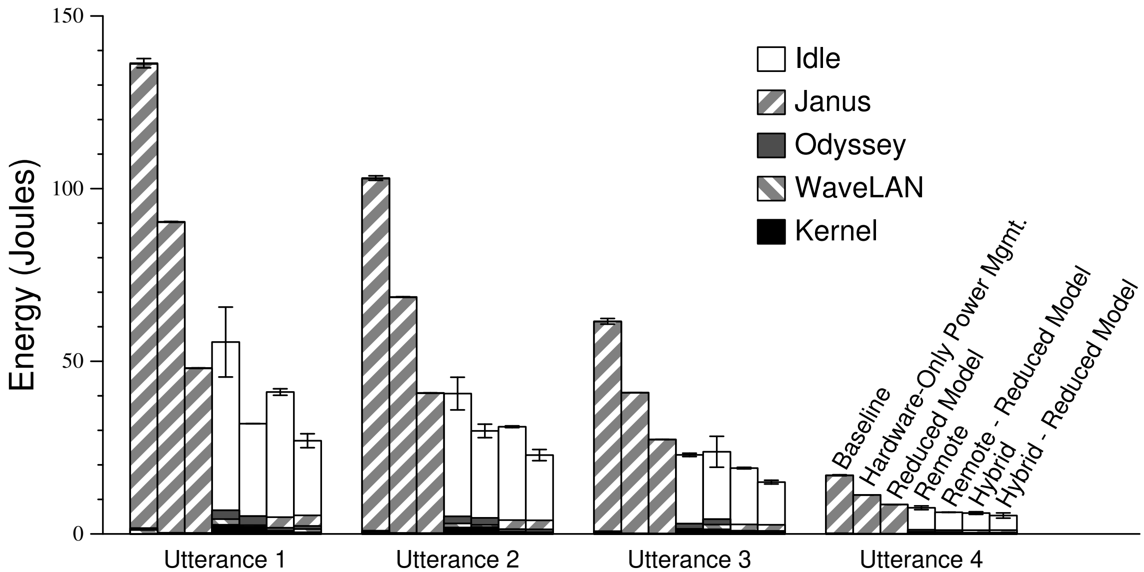
the combination of Premiere-C and reduced window size. In each case, hardware power management is enabled. For each fidelity level, four data points show the total energy used to play each of the videos in the study, and the corresponding line shows the best linear fit through these points. From the data, it is clear that the linear model is a good fit—the coefficient of determination (R^2) is greater than 99% for every fidelity. In addition, the maximum relative error for any data point is 2.3%. Thus, if an adaptive system were to be given a new video and could determine its length, it is reasonable to expect that it would be able to accurately predict the energy needed to display the video at each fidelity. Of course, these results apply directly only to the fidelities investigated in this study; it is possible that different encoding schemes may prove to be less predictable.

4.5 Speech recognizer

4.5.1 Description

The second application is an adaptive speech recognizer. As shown in Figure 4.4, it consists of a front-end that generates a speech waveform from a spoken utterance and submits it via Odyssey to a local or remote instance of the Janus speech recognition system [94].

Local recognition avoids network transmission and is unavoidable if the client is disconnected. In contrast, remote recognition incurs the delay and energy cost of network communication but can exploit the CPU, memory, and energy resources of a remote server that is likely to be operating from a power outlet rather than a battery. The system also supports a hybrid mode of operation in which the first phase of recognition is performed



This figure shows the energy used to recognize four spoken utterances from one to seven seconds in length, ordered from right to left above. For each utterance, the first bar shows energy consumption without hardware power management or fidelity reduction. The second bar shows the impact of hardware power management alone. The remaining bars show the additional savings realized by adaptive strategies. The shadings within each bar detail energy usage by activity. Each measurement is the mean of five trials—the error bars show 90% confidence intervals.

Figure 4.5: Energy impact of fidelity for speech recognition

locally, resulting in a compact intermediate representation that is shipped to the remote server for completion of the recognition. In effect, the hybrid mode uses the first phase of recognition as a type-specific compression technique that yields a factor of five reduction in data volume with minimal computational overhead.

Fidelity is lowered in this application by using a reduced vocabulary and a less complex acoustic model. This substantially reduces the memory footprint and processing required for recognition, but degrades recognition quality. The system alerts the user of fidelity transitions using a synthesized voice. The use of low fidelity is most compelling in the case of local recognition on a resource-poor disconnected client, although it can also be used in hybrid and remote cases.

Although reducing fidelity limits the number of words available, the word-error rate may not increase. Intuitively, this is because the recognizer makes fewer mistakes when choosing from a smaller set of words in the reduced vocabulary. This helps counterbalance the effects of reducing the sophistication of the acoustic model.

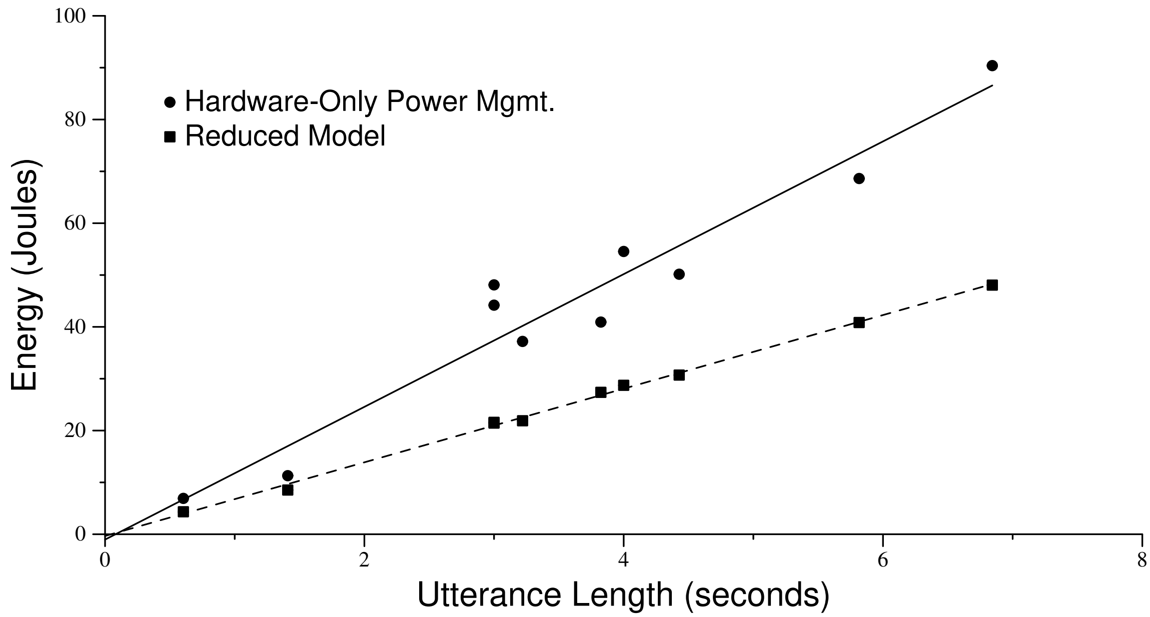


Figure 4.6: Predicting speech recognition energy use

This figure shows the relationship between system energy use, data fidelity, and utterance length. For each level of data fidelity, ten data points show the total energy used to recognize different utterances (including the ones from Figure 4.5)—the corresponding line represents the best linear fit through these points. All measurements were taken with hardware power management enabled. The (barely noticeable) error bars show 90% confidence intervals for energy use.

4.5.2 Results

Figure 4.5 presents measurements of client energy usage when recognizing four pre-recorded utterances using local, remote, and hybrid strategies at high and low fidelity. The baseline measurements correspond to local recognition at high fidelity without hardware power management. Since speech recognition is compute-intensive, almost all the energy in this case is consumed by Janus.

Hardware power management reduces client energy usage by 33–34%. Such a substantial reduction is possible because the display can be turned off and both the network and disk can be placed in standby mode for the entire duration of an experiment. This assumes that user interactions occur solely through speech, and that disk accesses can be avoided because the vocabulary, language model and acoustic model fit entirely in physical memory. More complex recognition tasks may trigger disk activity and hence show less benefit from hardware power management.

Lowering fidelity by using a reduced speech model results in a 25–46% reduction in energy consumption relative to using hardware power management alone. This corresponds to a 50–65% reduction relative to the baseline.

Remote recognition at full fidelity reduces energy usage by 33–44% relative to using hardware power management alone. If fidelity is also reduced, the corresponding savings is 42–65%. These figures are comparable to the energy savings for remote execution reported in the literature for other compute-intensive tasks [69, 77]. As the shadings in the fourth and fifth bars of each data set in Figure 4.5 indicate, most of the energy consumed by the client in remote recognition occurs with the processor idle—much of this is time spent waiting for a reply from the server. Lowering fidelity speeds recognition at the server, thus shortening this interval and yielding additional energy savings.

Hybrid recognition offers slightly greater energy savings than remote recognition: 47–55% at full fidelity, and 53–70% at low fidelity, both relative to hardware-only power management. Hybrid recognition increases the fraction of energy used by the local Janus code; but this is more than offset by the reduction in network transmission and idle time.

Overall, the net effect of combining hardware power management with hybrid, low-fidelity recognition is a 69–80% reduction in energy usage relative to the baseline. In practice, the optimal strategy will depend on resource availability and the user's tolerance for low-fidelity recognition. Chapter 7 explores this issue in greater detail.

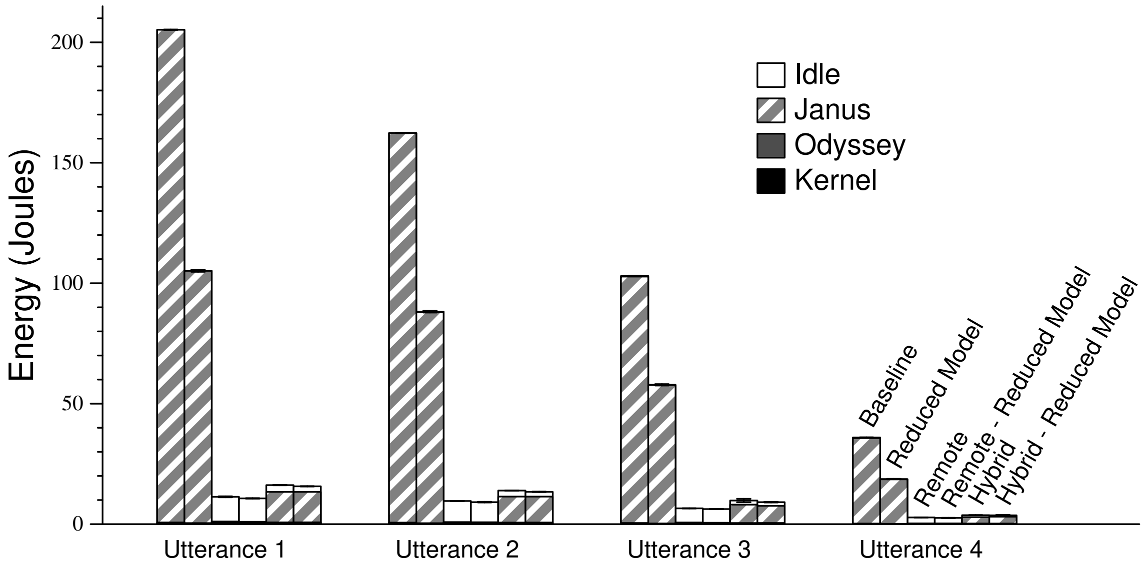
Figure 4.6 shows speech recognizer energy use as a function of utterance length for full and reduced fidelity with hardware power management enabled. For both fidelity levels, each data point shows the total energy used to recognize one of a set of ten spoken utterances—this set includes the four utterances from Figure 4.5. The corresponding lines show the best linear fit through these points. The linear model for the baseline fidelity produces a reasonable fit (93.7% coefficient of determination), while the fit is considerably better for the reduced fidelity (99.8%). At the baseline fidelity, the relative error for six of the ten utterances is less than 10%; at the reduced fidelity, nine utterances have relative error less than 10%. While speech recognition is less predictable than video display, these results are quite encouraging—simple linear models appear to do a reasonably good job of predicting energy use.

4.5.3 Results for Itsy v1.5

I have ported Janus to the Itsy v1.5 pocket computer in order to explore the impact of different hardware platforms on energy-aware adaptation. Figure 4.7 presents measurements of client energy usage for the same four pre-recorded utterances when the speech front-end executes on the Itsy v1.5. Since the Itsy does not have a PCMCIA interface, a serial link was used as the network transport.

Because the Itsy is an experimental platform, there exists less system support for hardware power management than on the ThinkPad laptop. Thus, the results in Figure 4.7 do not reflect the possibility that one could disable the Itsy's display and serial link when they are not in use, or that one could reduce the processor clock frequency when CPU load is low.

Contrasting Figures 4.5 and 4.7 shows that the Itsy consumes more energy than the



This figure shows the energy used to recognize four spoken utterances from one to seven seconds in length, ordered from right to left above. The client machine which executes the recognitions is an Itsy v1.5 pocket computer—the results can be contrasted with Figure 4.5 in which the client machine is an IBM ThinkPad 560 laptop. The first bar shows energy consumption without fidelity reduction. The remaining bars show the additional savings realized by adaptive strategies. The shadings within each bar detail energy usage by activity. Each measurement is the mean of five trials—the (barely noticeable) error bars show 90% confidence intervals.

Figure 4.7: Energy impact of fidelity for speech recognition on the Itsy v1.5

ThinkPad to perform local speech recognition, but significantly less energy to perform hybrid and remote recognition. Although the Itsy is considerably more energy-efficient than the ThinkPad, its StrongArm processor is slower. More significantly, the Janus recognizer performs a large number of floating-point operations, which are emulated in software on the Itsy. These factors combine to make local speech recognition prohibitively slow. Although average power usage is much lower on the Itsy, the total energy needed for local recognition is higher.

Use of the reduced quality speech model for local recognition provides considerable benefit on the Itsy, reducing energy usage from 44–49%. During hybrid and remote execution, use of the reduced speech model has little impact. Although the recognition completes more quickly on the remote server, the Itsy expends very little energy during the additional time it must wait for full-quality recognition because its background power usage is very small.

Remote recognition uses much less energy on the Itsy. The energy savings from remote execution range from 92–94% for full-quality recognition and from 86–90% for reduced-quality recognition. Two factors contribute to this large savings: the lack of floating-point support makes local processing less attractive, and the energy-efficiency of the hardware

Execution Location	Fidelity	ThinkPad 560X Normalized Energy	Itsy v1.5 Normalized Energy
Local	Full	1.00	1.00
Local	Reduced	0.64	0.53
Remote	Full	0.61	0.06
Remote	Reduced	0.47	0.06
Hybrid	Full	0.48	0.09
Hybrid	Reduced	0.36	0.08

This figure compares speech recognition energy usage on the Itsy v1.5 and ThinkPad 560X. Each row shows the average energy used to recognize an utterance on the two platforms with hardware power management enabled. Each data value is the geometric mean of energy usage, normalized to local energy usage at full fidelity, for the four utterances shown in Figures 4.5 and 4.7.

Figure 4.8: Comparison of per-platform speech recognition energy use

platform means that very little energy is expended waiting for remote processing requests to complete.

Hybrid recognition uses more energy than remote recognition on the Itsy, even though the operation completes in less time. The energy impact of performing the first stage of recognition locally is greater than the total amount of energy spent waiting for recognition to complete during fully-remote execution. This has two important implications for remote execution. First, it illustrates that tradeoffs between performance and energy conservation exist for common applications. Second, it shows that remote execution decisions must consider both the client and its available resources when deciding where to locate functionality.

The speech recognition results give some evidence that the effectiveness of energy-aware adaptation will increase as mobile hardware becomes more energy-efficient. Intuitively, modifying application behavior can only affect dynamic power usage—background power usage is fixed by device energy characteristics. Energy-efficient hardware platforms, which have lower background power usage and a greater ratio of maximum dynamic to background power usage, will benefit more. The results in Figures 4.5 and 4.7 bear this out. The maximum energy savings due to fidelity reduction and remote execution is only 64–78% on the ThinkPad, but 92–94% on the more energy-efficient Itsy. However, these results are biased by the Itsy's lack of floating-point support, which contributes to some of the energy differential. Figure 4.8 highlights the difference between the two platforms be showing the energy usage of each fidelity and remote execution location normalized to local energy usage at full fidelity.

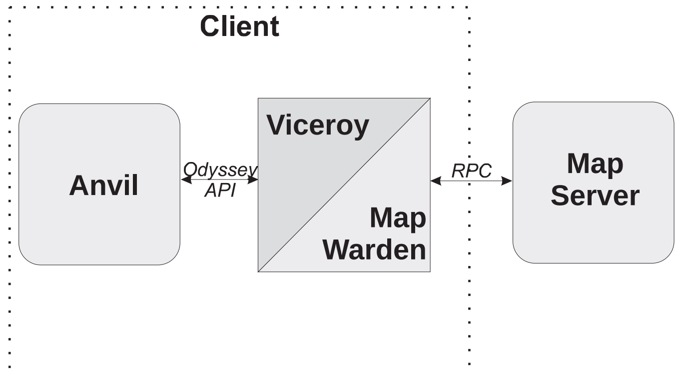


Figure 4.9: Odyssey map viewer

4.6 Map viewer

4.6.1 Description

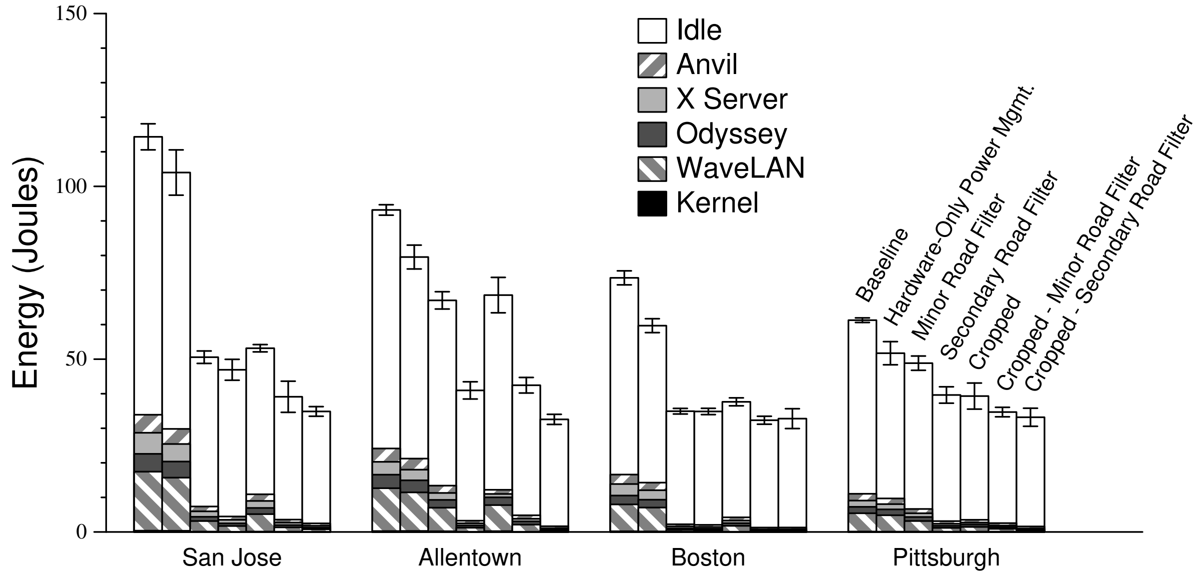
The third application that I measured was an adaptive map viewer named Anvil. As shown in Figure 4.9, Anvil fetches maps from a remote server via Odyssey and displays them on the client. Fidelity can be lowered in two ways: filtering and cropping. Filtering eliminates fine detail and less important features (such as secondary roads) from a map. Cropping preserves detail, but restricts data to a geographic subset of the original map. The client annotates the map request with the desired amount of filtering and cropping. The server performs any requested operations before transmitting the map data to the client.

4.6.2 Results

I measured the energy used by the client to fetch and display maps of four different cities. Viewing a map differs from the two previous applications in that a user typically needs a non-trivial amount of time to absorb the contents of a map after it has been displayed. This period, which I will refer to as *think time*, should logically be viewed as part of the application's execution since energy is consumed to keep the map visible. In contrast, the user needs negligible time after the display of the last video frame or the recognition of an utterance to complete use of the video or speech application.

Think time is likely to depend on both the user and the map being displayed. My approach to handling this variability was to use an initial value of 5 seconds and then perform sensitivity analysis for think times of 0, 10 and 20 seconds. For brevity, Figure 4.10 only presents detailed results for the 5 second case; for other think times, I present only the summary information in Figure 4.11.

The baseline bars in Figure 4.10 show that most of the energy is consumed while the kernel executes the idle procedure; a significant portion of this can be attributed to background power usage during the five seconds of think time. The shadings on the bars indicate



This figure shows the energy used to view U.S.G.S. maps of four cities. For each map, the first bar shows energy usage without hardware power management or fidelity reduction, with a 5 second think time. The second bar shows the impact of hardware power management alone. The remaining bars show the additional savings realized by degrading map fidelity. The shadings within each bar detail energy usage by activity. Each measurement is the mean of ten trials—the error bars are 90% confidence intervals.

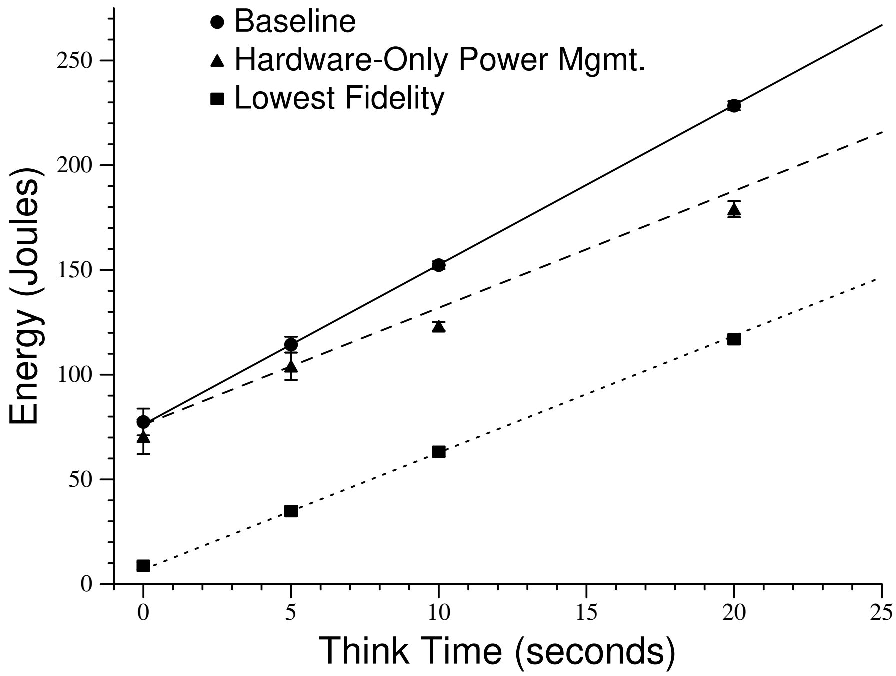
Figure 4.10: Energy impact of fidelity for map viewing

that network communication is a second significant drain on energy. The comparatively larger confidence intervals for this application result from variation in the time required to fetch a map over the wireless network.

Hardware power management reduces energy consumption by about 9–19% relative to the baseline. Although there is little opportunity for network power management while the map is being fetched, the network can remain in standby mode during think time. Since the disk is never used, it can always remain in standby mode.

The third and fourth bars of each data set show the effect of fidelity reduction through two levels of filtering. One filter omits minor roads, while the more aggressive filter omits both minor and secondary roads. The savings from the minor road filter range from 6–51% relative to hardware-only power management. The corresponding figure for the secondary road filter is 23–55%.

The fifth bar of each data set shows the effect of lowering fidelity by cropping a map to half its original height and width. Energy usage at this fidelity is 14–49% less than with hardware-only power management. In other words, cropping is less effective than filtering for these particular maps. Combining cropping with filtering results in an energy savings of 36–66% relative to hardware-only power management, as shown by the rightmost bars of each data set. Relative to the baseline, this is a reduction of 46–70%. There is little savings



This figure shows how the energy used to view the San Jose map from Figure 4.10 varies with think time. The data points show measured energy usage. The solid, dashed and dotted lines represent linear models of energy usage for the baseline, hardware-only power management and lowest fidelity cases. The latter combines filtering and cropping, as in the rightmost bars of Figure 4.10. Each measurement is the mean of ten trials—the error bars are 90% confidence intervals.

Figure 4.11: Effect of user think time for map viewing

left to be extracted through software optimization—almost all the energy is consumed in the idle state.

After examining energy usage with 5 seconds of think time, I repeated the above experiments with think times of 0, 10, and 20 seconds. At any given fidelity, energy usage, E_t increases with think time, t. I expected a linear relationship: $E_t = E_0 + t \cdot P_B$, where E_0 is the energy usage for this fidelity at zero think time and P_B is the background power consumption on the client (5.6 Watts from Figure 2.1).

Figure 4.11 confirms that a linear model is indeed a good fit. This figure plots the energy usage for four different values of think time for three cases: baseline, hardware power management alone, and lowest fidelity combined with hardware power management. The divergent lines for the first two cases show that the energy reduction from hardware power management scales linearly with think time. The parallel lines for the second and third cases show that fidelity reduction achieves a constant benefit, independent of think time. The complementary nature of these two approaches is thus well illustrated by these measurements.

Figure 4.12 shows map viewer energy use as a function of the undistilled map size

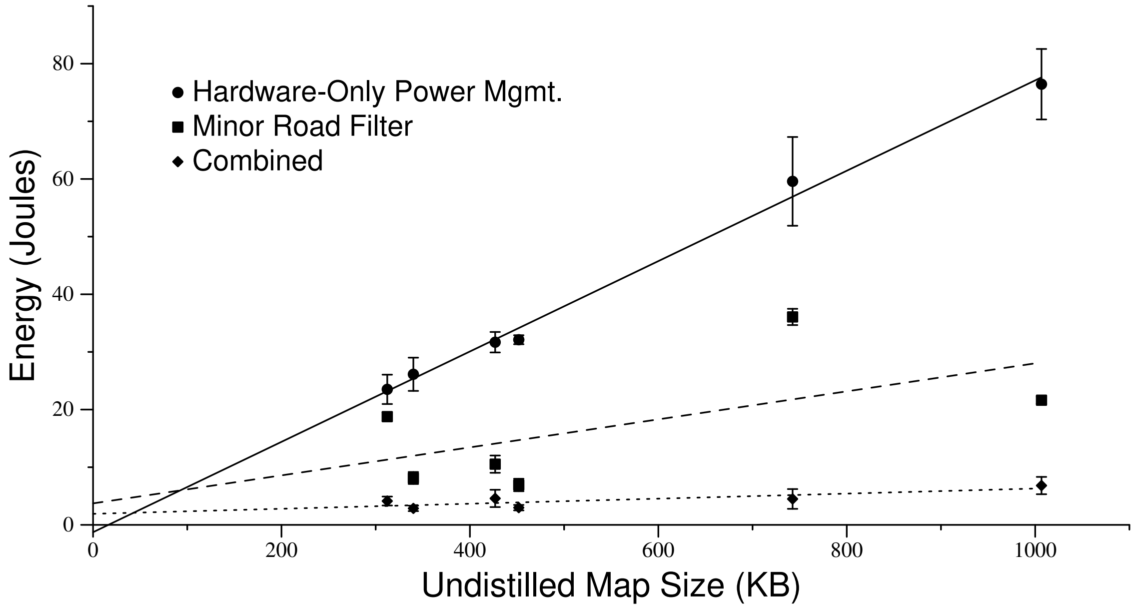


Figure 4.12: Predicting map viewer energy use

This figure shows the relationship between system energy use, data fidelity, and undistilled map size. For each level of data fidelity, ten data points show the total energy used to view six maps (including the ones from Figure 4.10)—the corresponding line represents the best linear fit through these points. All measurements were taken with hardware power management enabled. The error bars show 90% confidence intervals for energy use.

for three fidelity levels: baseline, minor road filtering, and the combination of minor road filtering, secondary road filtering, and cropping. (The remaining fidelity levels are omitted for clarity). For each fidelity, six data points show the total energy used to fetch and display different city maps, including the four from Figure 4.10. Since I have already shown that the energy cost of user think time is quite predictable, these results assume zero think time, and thereby isolate the variance in the energy cost of fetching and displaying maps.

Visual inspection of the linear fits reveals that energy usage corresponds closely to image size only in the baseline case. Whereas the baseline has a coefficient of determination of greater than 99%, the minor road filter and combined fidelities have R^2 values of 36% and 69% respectively. Clearly, the undistilled map size does not remain a good predictor when map fidelity is reduced.

When simple models do not yield good predictions, it is possible that additional information will give more accurate results. In the specific example of the map viewer, the percentage of features omitted by a given filter may vary widely from map to map. However, if the map server were to store summary information with each map listing the occurrence of different feature types, one could anticipate the effectiveness of a filter. This would allow an adaptive system to determine the number of map features that would be fetched at

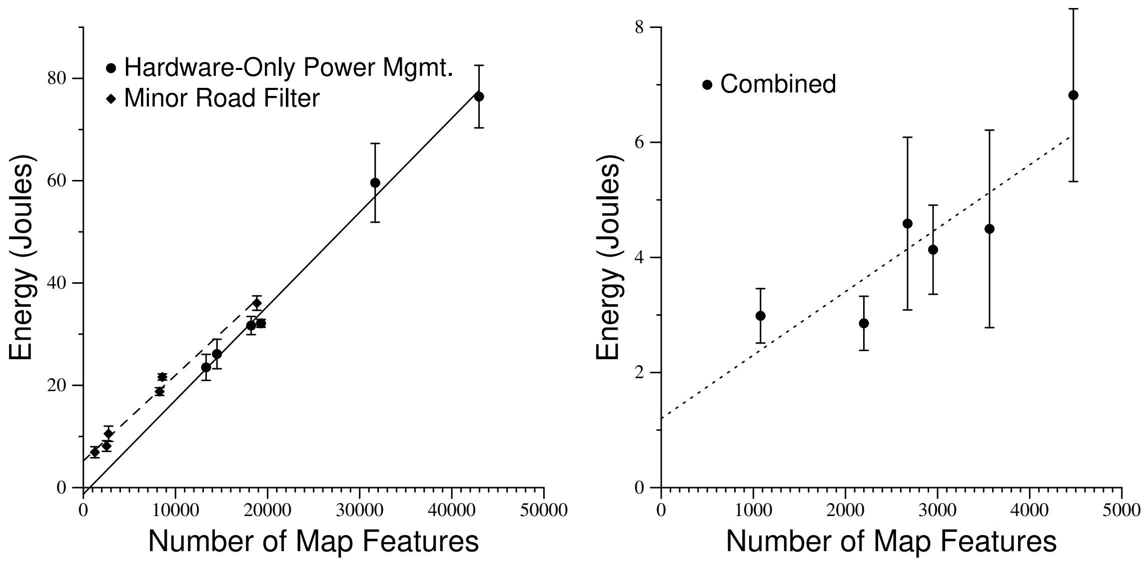


Figure 4.13: Predicting map viewer energy use by number of features

This figure shows the relationship between system energy use, data fidelity, and the number of map features. For each level of data fidelity, ten data points show the total energy used to view the six maps from Figure 4.12—the corresponding line represents the best linear fit through these points. All measurements were taken with hardware power management enabled. The error bars show 90% confidence intervals for energy use.

various levels of filtering.

Figure 4.13 shows the benefit of this additional information. It displays map viewer energy use as a function of fidelity and the number of features fetched. For clarity, the data is displayed in two graphs: the graph on the left shows the baseline and minor road filter fidelities, while the other graph shows the combined fidelity. Knowledge of the number of features that will be omitted by the minor road filter makes predictions much more accurate—the R^2 value for the minor-road filter fidelity is now 99%. Unfortunately, although the linear fit for the combined fidelity improves, it is still quite poor with an R^2 value of 79%. Although the model can anticipate the effect of filtering, it is still unable to cope with variation in the amount of features removed by cropping.

4.7 Web browser

4.7.1 Description

The fourth application is an adaptive Web browser based on Netscape Navigator, as shown in Figure 4.14. In this application, Odyssey and a distillation server located on either side of a variable-quality network mediate access to Web servers. Requests from an unmodified

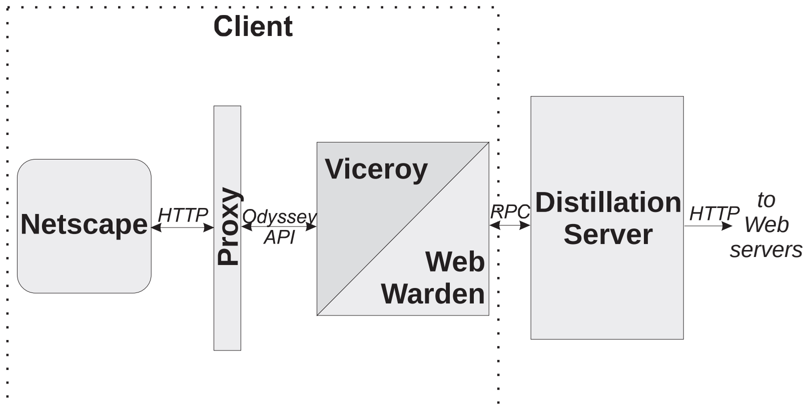


Figure 4.14: Odyssey Web browser

Netscape browser are routed to a proxy on the client that interacts with Odyssey. After annotating the request with the desired level of fidelity, Odyssey forwards it to the distillation server which transcodes images to lower fidelity using lossy JPEG compression. This is similar to the strategy described by Fox et al [23], except that control of fidelity is at the client.

4.7.2 Results

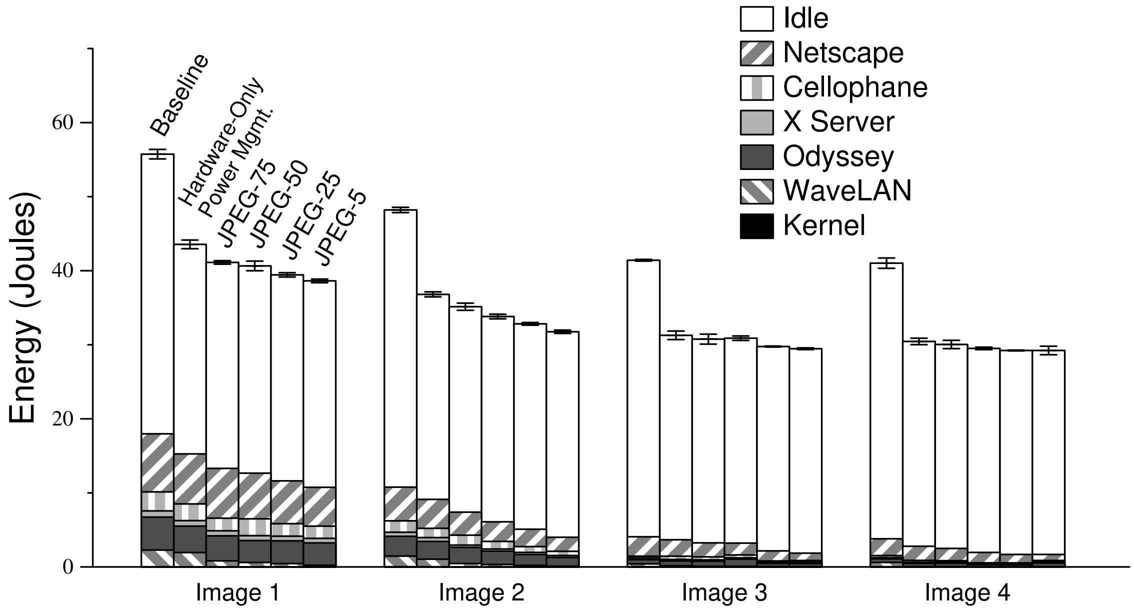
As with the map application, a user needs some time after an image is displayed to absorb its contents. I therefore include energy consumed during user think time as part of the application. I use a baseline value of 5 seconds and perform sensitivity analysis for 0, 10, and 20 seconds.

Figure 4.15 presents measurements of the energy used to fetch and display four GIF images of varying sizes. Hardware power management achieves reductions of 22–26%. The shadings on the first and second bars of each data set indicate that most of this savings occurs in the idle state, probably during think time.

The energy benefits of fidelity reduction are disappointing. As Figure 4.15 shows, the energy used at the lowest fidelity is merely 4–14% lower than with hardware power management alone; relative to baseline, this is a reduction of 29–34%. The maximum benefit of fidelity reduction is severely limited because the relative amount of energy used during think time (28 Watts) is much greater than the energy used to fetch and display an image, (2–16 Watts). Thus, even if fidelity reduction could completely eliminate the energy used to fetch and display an image, energy use would drop only 9–36%.

Although Web fidelity reduction shows little benefit in this study, it may be quite useful in other environments. For example, a more energy-efficient mobile device would use less energy during think time, increasing the possible benefit of fidelity reduction. Similarly, if a high-speed network were unavailable, the energy benefit of distillation would increase.

The effect of varying think time is shown in Figure 4.16. The linear model introduced



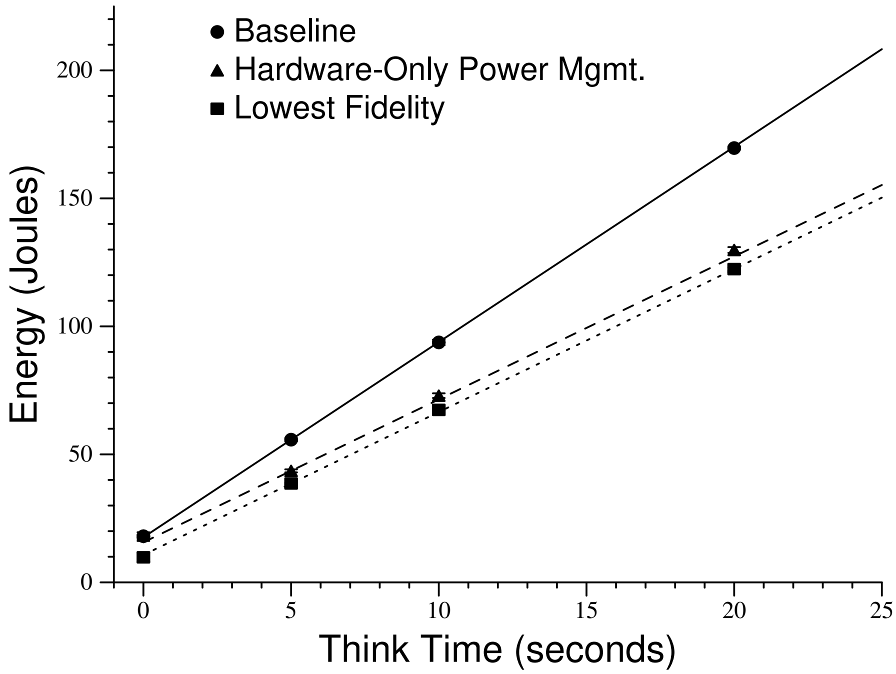
This figure shows the energy used to display four GIF images from 110 B to 175 KB in size, ordered from right to left above. For each image, the first data bar shows energy consumption at highest fidelity without hardware power management, assuming a think time of five seconds. The second bar shows the impact of hardware power management alone. The remaining bars show energy usage as fidelity is lowered through increasingly aggressive lossy JPEG compression. The shadings within each bar detail energy usage by activity. Each measurement is the mean of ten trials—the error bars show 90% confidence intervals.

Figure 4.15: Energy impact of fidelity for Web browsing

in Section 4.6.2 fits observations well for all three cases: baseline, hardware-only power management, and lowest fidelity. The close spacing of the lines for the two latter cases reflects the small energy savings available through fidelity reduction. The divergence of the lines for the first two cases shows the importance of hardware power management during think time.

Figure 4.17 shows Web browser energy use as a function of the undistilled image size for three fidelity levels: baseline, JPEG 50, and JPEG 5. For each fidelity, seven data points show the total energy used to fetch and display different images, including the four from Figure 4.15. As with the map application, these results include no user think-time, thereby isolating the variation in energy used to fetch and display images. For this application, simple linear models yield reasonable predictions: the R^2 values for baseline, JPEG 50, and JPEG 5 are 91%, 93%, and 98% respectively.

A related study of Netscape energy usage [64] looks at the relationship between energy use and image size in more detail. It attributes some of the variation in energy use to the Netscape application, speculating that the scheduling behavior of Netscape's user-level threading package leads to non-deterministic effects. The study also finds that different



This figure shows how the energy used to display Image 1 from Figure 4.15 varies with user think time. The data points on the graph show measured energy usage for user think times of 0, 5, 10, and 20 seconds. The solid, dashed and dotted lines represent linear models of energy consumption for the baseline, hardware-only power management, and lowest fidelity cases. Each measurement represents the mean of ten trials—the error bars are 90% confidence intervals.

Figure 4.16: Effect of user think time for Web browsing

images vary slightly in the amount of compression achieved for equivalent levels of JPEG quality.

4.8 Effect of concurrency

How does concurrent execution affect energy usage? One can imagine situations in which total energy usage goes *down* when two applications execute concurrently rather than sequentially. For example, once the screen has been turned on for one application, no additional energy is required to keep it on for the second. One can also envision situations in which concurrent applications interfere with each other in ways that increase energy usage. For example, if physical memory size is inadequate to accommodate the working sets of two applications, their concurrent execution will trigger higher paging activity, possibly leading to increased energy usage. Clearly, the impact of concurrency can vary depending on the applications, their interleaving, and the machine on which they run.

What is the effect of lowering fidelity? The measurements reported in Sections 4.4

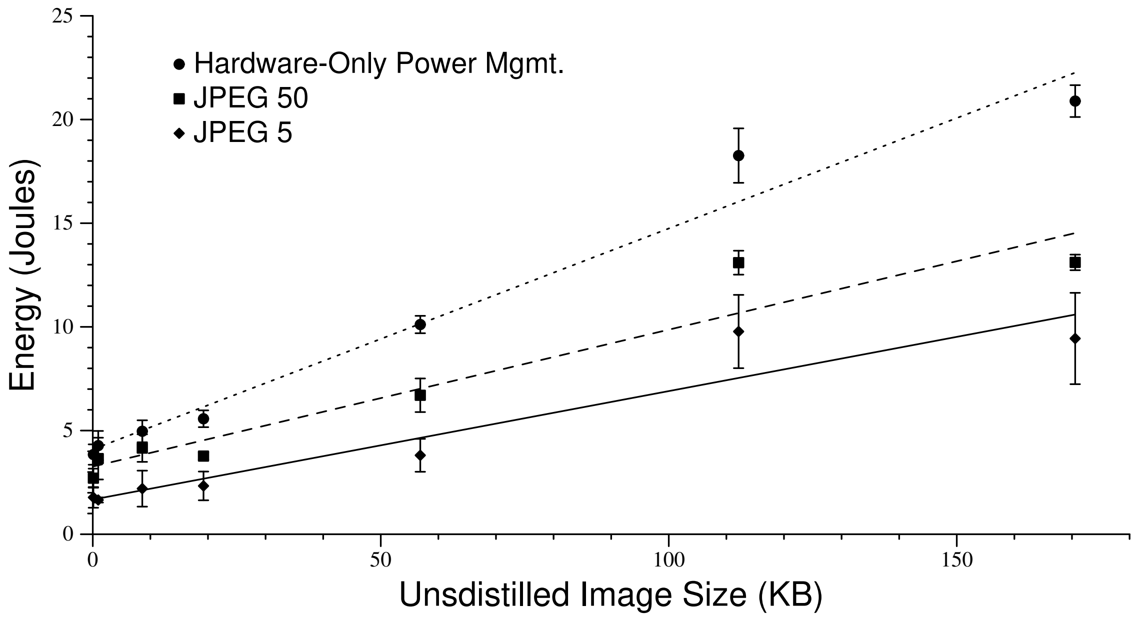


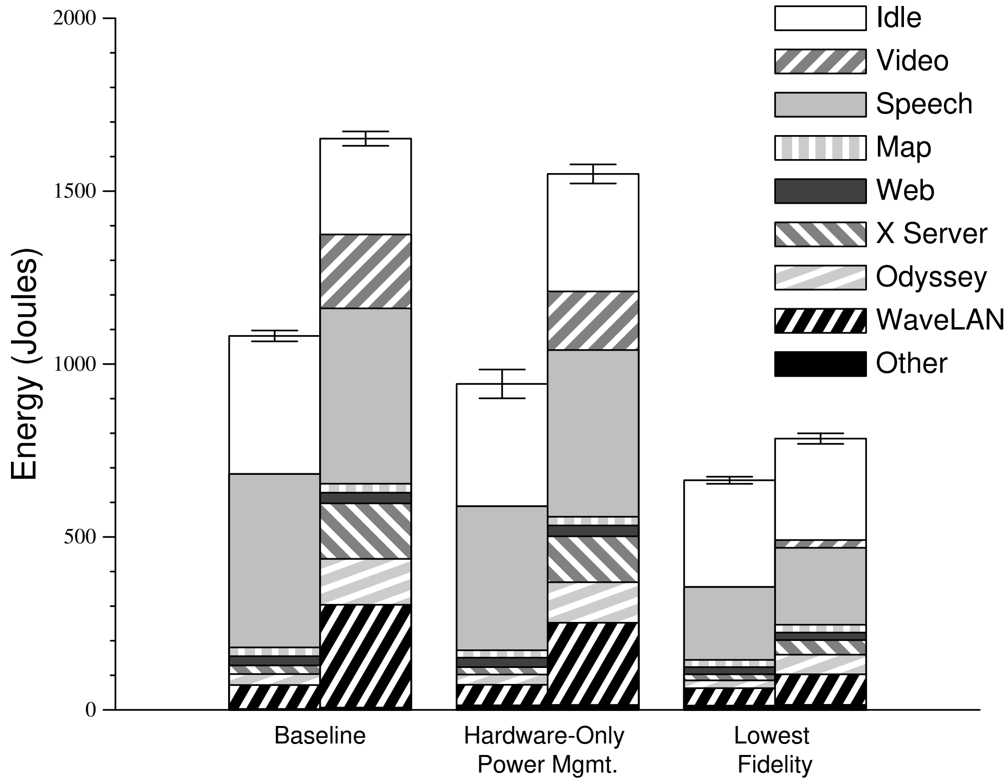
Figure 4.17: Predicting Web browser energy use

This figure shows the relationship between system energy use, data fidelity, and undistilled image size. For each level of data fidelity, six data points show the total energy used to fetch and display different GIF images (including the ones from Figure 4.15)—the corresponding line represents the best linear fit through these points. All measurements were taken with hardware power management enabled. The error bars show 90% confidence intervals for energy use.

to 4.7 indicate that lowering fidelity tends to increase the fraction of energy consumption attributable to the idle state. Concurrency allows background energy consumption to be amortized across applications. It is therefore possible in some cases for concurrency to enhance the benefit of lowering fidelity.

To confirm this intuition, I compared the energy usage of a composite application when executing in isolation and when executing concurrently with the video application described in Section 4.4. The composite application consists of six iterations of a loop that involves the speech, Web, and map applications described in Sections 4.5 to 4.7. The loop consists of local recognition of two speech utterances, access of a Web page, access of a map, and five seconds of think time. The composite application models a user searching for Web and map information using speech commands, while the video application models a background newsfeed. This experiment takes between 80 and 160 seconds.

Figure 4.18 presents the results of the experiments for three cases: baseline, hardware-only power management, and minimal fidelity. In the first two cases, all applications ran at full fidelity; in the third case, all ran at lowest fidelity. For each data set, the left bar shows energy usage for the composite application in isolation, while the right bar shows energy



Each data set in this figure compares energy usage for the composite application described in Section 4.8 in isolation (left bar), with total energy usage when a video application runs concurrently (right bar). Each measurement is the mean of five trials—the error bars are 90% confidence intervals.

Figure 4.18: Effect of concurrent applications

usage during concurrent execution.

For the baseline case, the addition of the video application consumes 53% more energy. But with hardware power management, it consumes 64% more energy. This difference is due to the fact that concurrency reduces opportunities for powering down the network and disk. For the minimum fidelity case, the second application only adds 18% more energy. The significant background power usage of the client, which limits the effectiveness of lowering fidelity, is amortized by the second application. In other words, for this workload, concurrency does indeed enhance the energy impact of lowering fidelity.

Figure 4.19 provides more detail by showing background and dynamic energy use for each application. As described in Section 2.1, I define background energy as the amount of energy that would have been used if the computer had remained idle instead of executing the application. Background energy is application-independent; it represents the cost of operating hardware components in their lowest power states; for example, keeping the display backlit and the disk in standby mode. I define dynamic energy use to be the amount of energy consumed by an application above and beyond its background energy use. Thus,

Scenario	Composite Only Energy (J)			Video Only Energy (J)			Concurrent Energy (J)		
	Bkgd.	Dyn.	Total	Bkgd.	Dyn.	Total	Bkgd.	Dyn.	Total
Baseline	819.9	261.5	1081.4	1148.1	260.5	1408.6	1148.1	503.8	1651.9
Hardware	622.0	320.6	942.6	841.4	435.5	1276.9	841.4	708.4	1549.8
Lowest	471.5	192.6	664.0	505.3	50.1	555.4	505.3	279.2	784.5

This table displays the background and dynamic energy usage for the results shown in Figure 4.18. It shows energy usage for the composite application and background video feed in isolation, and then the energy usage when the two applications execute concurrently. The three rows show energy use at full fidelity with hardware power management disabled, at full fidelity with power management enabled, and at lowest fidelity with power management enabled.

Figure 4.19: Background and dynamic energy use for concurrent applications

dynamic energy use captures the application-specific component of the energy usage.

As Figure 4.19 confirms, dynamic energy use is a much better metric than total energy use when projecting the effect of executing two applications concurrently. The dynamic energy use when the two applications are executed concurrently is roughly equivalent to the sum of each application's dynamic energy use when executed independently. Of course, the correlation is not perfect: the actual combined dynamic energy use varies 4–13% from the sum of the individual dynamic energy usages. This variation can be attributed to several factors, such as the ability to amortize transition costs for hardware components (i.e. spinning up the disk) across multiple applications, and memory effects when the working sets of all applications do not fit in physical memory.

Unfortunately, one would often like to project total rather than dynamic energy usage. Although background power levels are easily determined for a hardware platform, the calculation of background energy requires one to project how long an application will execute. The concurrent execution latency will often depend upon contention for shared resources such as CPU, network, and disk. Therefore, accurately projecting total concurrent energy usage requires knowledge of availability and application use of these shared resources.

4.9 Summary

My primary goal in performing this study was to determine whether lowering data fidelity yields significant energy savings. The results of Sections 4.4 to 4.7 confirm that such savings are indeed available over a broad range of applications relevant to mobile computing. Further, those results show that lowering fidelity can be effectively combined with hardware power management. Section 4.8 extends these results by showing that concurrency can magnify the benefits of lowering fidelity.

In addition, the results of Section 4.5 show that remote resources can sometimes be

Application	Think Time (s.)	Baseline	Hardware Power Mgmt.	Fidelity Reduction	Combined
Video	N/A	1.00	0.90–0.91	0.84–0.84	0.65–0.65
Speech	N/A	1.00	0.66–0.67	0.22–0.36	0.20–0.31
Map	0	1.00	0.80–1.01	0.06–0.13	0.07–0.18
	5	1.00	0.81–0.91	0.38–0.67	0.31–0.54
	10	1.00	0.74–0.84	0.53–0.77	0.42–0.58
	20	1.00	0.76–0.78	0.69–0.89	0.51–0.67
Web	0	1.00	0.85–1.06	0.40–0.75	0.32–0.54
	5	1.00	0.74–0.78	0.88–0.97	0.66–0.71
	10	1.00	0.75–0.78	0.93–0.98	0.70–0.74
	20	1.00	0.74–0.77	0.96–0.99	0.72–0.73

This table summarizes the impact of data fidelity on application energy consumption. Each entry shows the minimum and maximum measured energy consumption on the IBM ThinkPad 560X for four data objects. The entries are normalized to baseline measurements of full fidelity objects with no power management. This data was extracted from Figures 4.2, 4.5, 4.10, and 4.15.

Figure 4.20: Summary of the energy impact of fidelity

profitably employed to reduce the energy usage of a mobile client. However, the results also provide a caution: performing computation remotely does not always lead to reduced client energy use. The hybrid mode of speech recognition uses less energy than the fully remote mode on the ThinkPad because the energy cost of performing a small amount of computation locally is outweighed by the benefit of decreased energy usage for network transmission.

The study also reveals that it is often possible to predict the energy impact of fidelity reduction. For the video, speech, and Web applications, simple linear models based on fidelity and input data size provide good fits for application energy use. This means that an adaptive system could observe an application execute on several data objects at a given fidelity level, construct a simple model, and use that model to project how much energy will be used when the application operates on new objects at that fidelity level.

The map application shows that achieving accurate energy predictions sometimes requires more work. Filtering based upon feature type yields considerable variation in energy reduction. This variation can be accurately modeled, however, if the server stores a summary with each map listing the number of occurrences of each feature type. Cropping introduces still more variation in energy use. Thus, the map application confirms that there can often be a tradeoff between complexity and accuracy in predicting energy use.

At the next level of detail, Figure 4.20 summarizes the results of Sections 4.4 to 4.7. For clarity, the data in each row is normalized to the baseline values.

The key messages of Figure 4.20 are:

- *There is significant variation in the effectiveness of fidelity reduction across data objects.*

  The reduction can span a range as broad as 29% (0.38–0.67 for the map viewer, at think time 5). The video player is the only application that shows little variation across data objects. As mentioned above, this variation is often quite predictable with the use of simple linear models.

- *There is considerable variation in the effectiveness of fidelity reduction across applications.*

  Holding think time constant at 5 seconds and averaging across data objects, the energy usage for the four applications at lowest fidelity is 0.84, 0.28, 0.51 and 0.93 relative to their baseline values. The mean is 0.64, corresponding to an average savings of 36%.

- *Combining hardware power management with lowered fidelity can sometimes reduce energy usage below the product of the individual reductions.*

  This is seen most easily in the case of the video application, where the last column is 0.65 rather than the expected value of 0.76, obtained by multiplying 0.9 and 0.84. Intuitively, this is because reducing fidelity decreases hardware utilization, thereby increasing the opportunity for hardware power management.

Chapter 5

A proxy approach for closed-source environments

The previous chapter showed that energy-aware applications can often significantly extend the battery lifetimes of the laptop computers on which they operate by trading fidelity for reduced energy usage. However, since only multimedia, source-code available applications running on the Linux operating system were studied, it is not clear whether energy-aware adaptation can help the Windows office applications that users commonly run on laptop computers.

Thus, several important questions remain: Can one show significant energy reductions for office applications commonly executed on laptop computers? Is it possible to add energy-awareness without access to application source code? Is this approach valid for applications executing on source-code unavailable operating systems (i.e. Windows)? In this chapter, I will answer these questions by studying the potential benefits of energy-aware adaptation for Microsoft's popular PowerPoint 2000 application.

5.1 Overview

As with many office applications, PowerPoint enables users to include increasing amounts of rich multimedia content in their documents—for example, charts, graphs, and images. Since these objects tend to be quite large, the processor, network, and disk activity needed to manipulate them accounts for significant energy expenditure. Yet, when editing a presentation, a user may only need to modify and view a small subset of these objects. Thus, it may be possible to significantly reduce PowerPoint energy consumption by presenting the user with a distilled version of a presentation: one which contains only the information that the user is interested in viewing or editing.

One can use Puppeteer [14], component-based middleware developed by Rice University, to perform such distillation. Puppeteer takes advantage of well-defined interfaces exported by applications to change behavior without source code modification. Using Puppeteer, one can create a distilled version of a presentation which initially omits all multi-

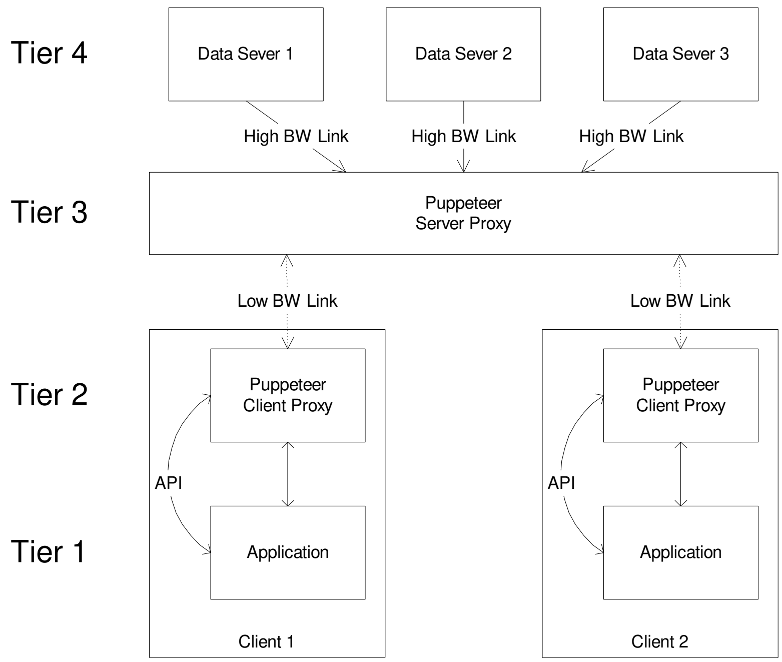


Figure 5.1: Puppeteer architecture

media content not on the first or master slide. Placeholders are inserted into the document to represent omitted objects. Later, if the user wishes to edit or view a component, she may click on the placeholder, and Puppeteer will load the component and dynamically insert it into the document. Thus, when the user edits only a subset of a document, the potential for significant energy savings exists.

In the remainder of this chapter, I explore the feasibility of adding energy-aware adaptation to PowerPoint. The next section provides a more detailed description of Puppeteer, and Section 5.3 describes my energy measurement methodology for the Windows platform. Section 5.4 measures the impact of energy-aware adaptation while loading, editing, and saving PowerPoint documents, and Section 5.5 summarizes the results.

5.2 Puppeteer

Puppeteer adapts the behavior of component-based applications, such as Microsoft's PowerPoint and Internet Explorer, in response to variation in resource availability in mobile environments. While Puppeteer's design goals are similar to Odyssey's, its implementation is quite different. It uses the exported APIs of applications and the structured nature of the documents they manipulate to change application behavior without source-code modification. Puppeteer thus acts as a proxy for an application.

Puppeteer currently supports two forms of adaptation: subsetting and versioning. Subsetting loads only a portion of the elements of a document. Versioning loads different versions of some elements, for example low-resolution distillations of images.

Commercial applications such as Microsoft's Office suite are ideally suited for Puppeteer. These applications have well-defined document structures, allowing Puppeteer to parse documents to implement specific policies such as degrading all images of a certain size. Further, such applications have rich external interfaces that enable Puppeteer to incrementally modify loaded documents. For example, Puppeteer can insert additional data or higher-resolution versions of images using such APIs.

Figure 5.2 shows the Puppeteer architecture. There are four tiers: applications, Puppeteer client proxies, Puppeteer server proxies, and data servers. Data servers are arbitrary repositories of data such as Web servers, file servers, or databases. All communication between applications and data servers pass through the Puppeteer client and server proxies. Applications and data servers remain completely unmodified—adaptation is performed by client and server proxies working in concert.

Puppeteer adjusts to reduced bandwidth availability between a mobile client and data servers by loading degraded versions of documents. Application-specific policies determine which subset of components should be fetched and which should be degraded. Puppeteer parses the document to uncover the structure of the data, fetches selected components at specific fidelity levels, and updates the application with the newly fetched data.

Once a degraded document version is loaded, the application returns control to the user. Although the application believes that it has loaded the original, full-quality document, Puppeteer maintains a list of degraded and omitted components. While editing or viewing a document, a user may double-click on a degraded component or on a placeholder for an omitted component. Puppeteer then fetches the component at its highest quality and inserts it into the document using the application's external programming interface.

Puppeteer is currently being modified to allow users to save changes made to degraded versions of documents. Since Puppeteer understands document structure, it can parse the modified version on the client and send to the server only those components which have been modified. The server proxy can then merge the modified components into the full-quality version and save the resulting document.

5.3 Measurement methodology

All measurements reported in this chapter were collected in a client-server environment similar to the one used in the previous chapter. PowerPoint executes on the client: an IBM 560X laptop, as described in Section 2.2.1. The server is a 400 MHz Pentium II desktop with 128 MB of memory. Both machines run the Windows NT 4.0 operating system. The machines communicate using a 2 Mb/s 2.5 GHz Lucent WaveLan network. I added an additional 32 MB of memory to the client, bringing the total to 96 MB—this was necessary to run PowerPoint, Puppeteer, and Windows NT without thrashing.

Since PowerPoint source code is unavailable, it made little sense to port PowerScope to Windows to perform these measurements. Instead, I simply measured the amount of total energy consumed by the laptop between two points in program execution. The hardware setup used for Windows measurements is identical to that used for PowerScope measure-

Presentation	Document Identifier	Full-Quality Size (MB)	Distilled Size (MB)	Ratio
A	24666	15.02	1.67	0.11
B	24758	11.42	0.47	0.04
C	24890	7.26	0.83	0.11
D	26295	3.11	3.11	1.00
E	26141	2.23	1.32	0.59
F	24773	1.72	0.11	0.07
G	25189	1.07	0.36	0.34
H	26388	0.87	0.75	0.86
I	26140	0.20	0.20	1.00
J	25132	0.08	0.08	1.00

Figure 5.2: Sizes of sample presentations

ments. I attached the probes of the HP3458a digital multimeter to the external power input of the client laptop and removed the battery to eliminate the effects of charging. I also connected the output pin of the client's parallel port to the multimeter's external trigger.

I created a dynamic library that allows applications to precisely indicate the start and end of each measurement. An application first calls the `start_measuring` function, which records the current time and toggles the parallel port pin. Once the pin is toggled, the multimeter samples current levels at its maximum rate of 1357.5 times per second. When the event being measured completes, the application calls the `stop_measuring` function, which returns the elapsed time since the `start_measuring` function was called.

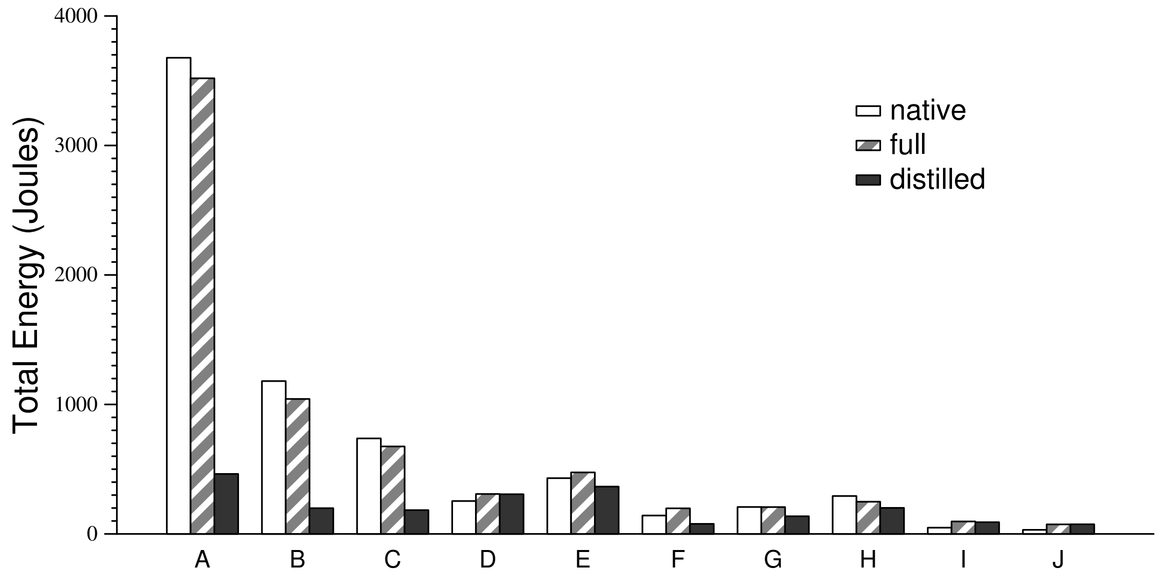
To calculate total energy usage, I first derive the number of samples, n, that were taken before `stop_measuring` was called by multiplying the elapsed measurement time by the sample rate. The mean of the first n samples is the average current level. Multiplying this value by the measured voltage for the laptop power supply yields the average power usage. This is multiplied by the elapsed time to calculate total energy usage.

I assume aggressive power management policies. All measurements were taken using a disk-spindown threshold of 30 seconds (the minimum allowed by Windows NT). The wireless network uses standard 802.11 power management. However, the display is not disabled during measurements since PowerPoint is interactive.

5.4 Benefits of PowerPoint adaptation

5.4.1 Loading presentations

I first examined the potential benefit of loading distilled PowerPoint presentations. I measured the energy used to fetch presentations from the remote server over the wireless network and render them on the client.



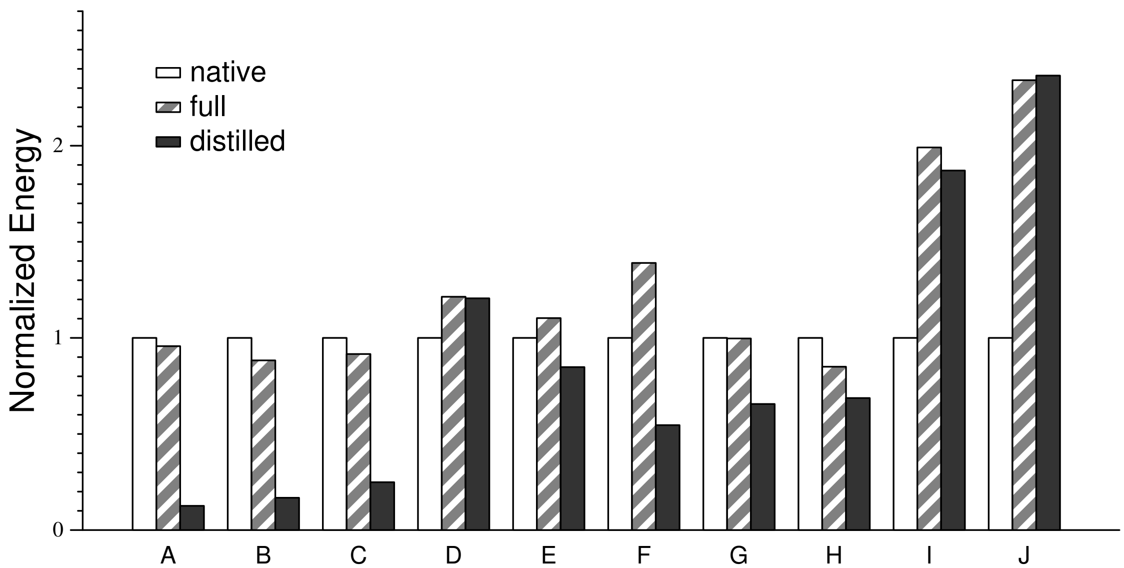
This figure shows the energy used to load ten PowerPoint presentations from a remote server. Native mode loads presentations from an Apache Web server. Full and distilled modes load presentations from a Puppeteer server, with full mode loading the entire presentation and distilled mode loading a lower-quality version. Each bar represents the mean of five trials—90% confidence intervals are sufficiently small so that they would not be visible on the graph.

Figure 5.3: Energy used to load presentations

I chose a sample set of documents from a database of 1900 PowerPoint presentations gathered from the Web as described by de Lara et al. [13]. From the database, I selected ten documents relatively evenly distributed in size. Figure 5.2 shows the sizes of these documents, as well as the size reductions achieved by distillation. For experimental repeatability, it also lists an identifier that uniquely identifies each document within the presentation database. The specific distillation policy used in these experiments omits all multimedia data that is not contained on either the first or master slide. As might be expected, larger documents tend to have the most multimedia content, although there is considerable variation in the data. For three documents (D, I, and J), distillation does not reduce document size at all.

For each document, I first measured the energy used by PowerPoint to load the presentation from a remote server (I will refer to this as “native mode”). In this case, the document is loaded from an Apache Web server. I also investigated the cost of loading documents from a remote NT file system but found that the latency and energy expenditure was significantly greater than using the Web server.

I then measured energy used by PowerPoint when the document was loaded from the same server using Puppeteer. In this case, the document is served from a Puppeteer server proxy. I measured two modes of operation: “full” mode, in which the entire document is



This figure shows the relative energy used to load ten PowerPoint presentations from a remote server, as described in Figure 5.3. For each data set, results are normalized to the amount of energy used to load the document in native mode.

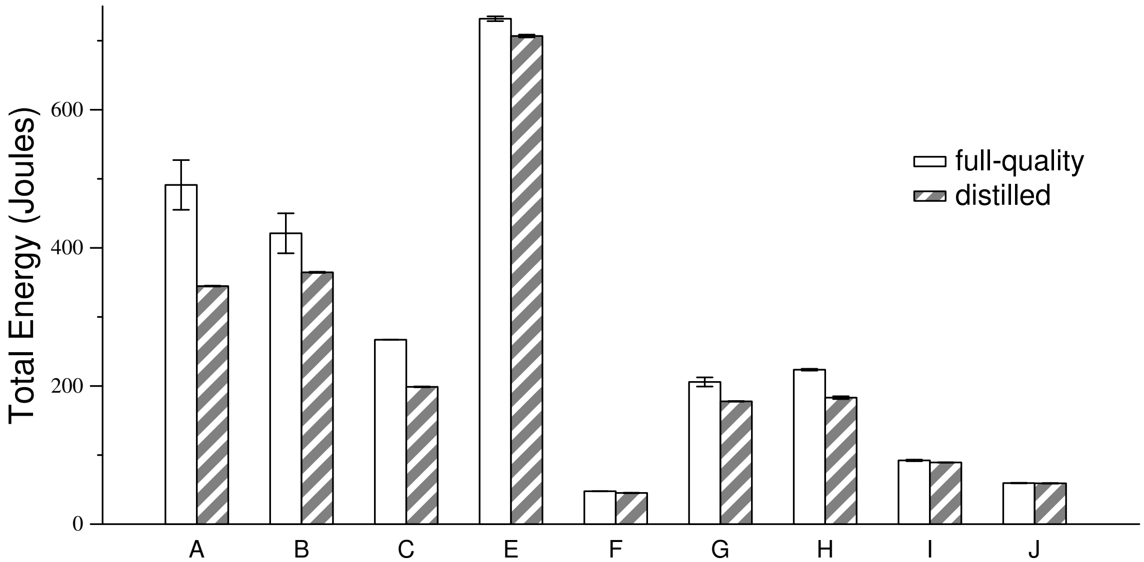
Figure 5.4: Normalized energy used to load presentations

loaded, and "distilled" mode, in which a reduced-quality version is loaded.

Figure 5.3 shows the total energy used to fetch the documents using native, full, and distilled modes. In Figure 5.4, I show the relative impact for each document by normalizing each value to the energy used by native mode. The energy savings achieved by distillation vary widely. Loading a distilled version of document A uses only 13% as much energy as native mode, while distilling document J uses 137% more energy. On average, loading a distilled version of a document uses 60% of the energy used by native mode.

It is interesting to note that full mode can sometimes use less energy to fetch a document than native mode. This is because fetching a presentation with Puppeteer tends to use less power than native mode. Thus, even though native mode takes less time to fetch a document, its total energy usage can sometimes be greater. Without source code, it is impossible to know for certain why Puppeteer power usage is lower than native mode. One possibility is more efficient scheduling of network transmissions.

The results in Figures 5.3 and 5.4 show that while most documents benefit from distillation, some suffer an energy penalty. This indicates that Puppeteer might profit by predicting whether a document will benefit from distillation. If it predicts that a document will benefit, it could distill and fetch it; otherwise, it could forego distillation and fetch the entire document.



This figure shows the amount of energy needed to page through a presentation. For each data set, the left bar shows energy use for a full-quality presentation, and the right bar shows energy use for a reduced-quality version of the same presentation. Document D is omitted from this experiment since it contains only a single slide. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Figure 5.5: Energy used to page through presentations

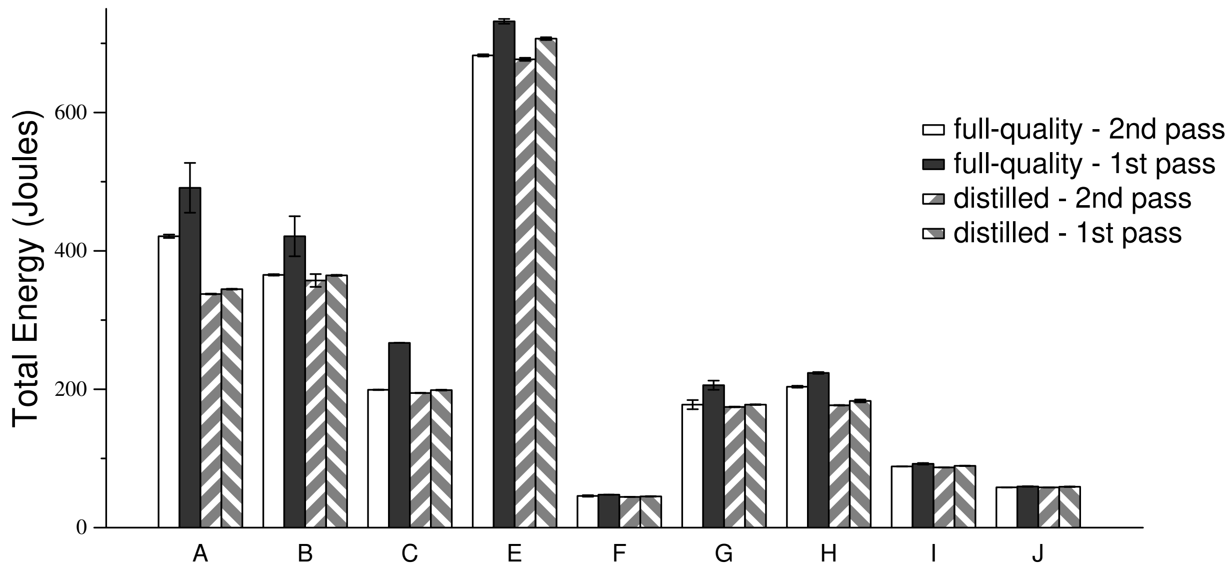
One possible prediction method is to distill only presentations that have a size greater than a fixed threshold, reasoning that small documents are unlikely to contain significant multimedia content. Analysis of the documents used in this study suggests that a reasonable threshold is 0.5 MB. A strategy of distilling only presentations larger than 0.5 MB uses 52% of the energy of native mode to load the ten documents.

Another possible prediction method is to distill only those documents with a percentage of multimedia content greater than a threshold value. As shown in Figure 5.2, distillation does not reduce the sizes of three documents. If Puppeteer does not distill these documents, it uses only 51% of the energy of native mode to fetch the ten documents.

5.4.2 Editing presentations

I next measured how document distillation affects the energy needed to edit a presentation. While it is somewhat intuitive that loading a smaller, distilled version of a document can require less energy, it is less clear that distillation also reduces energy usage while the document is being displayed or edited.

Naturally, energy usage depends upon which activities a user performs. While a definitive measurement of potential savings requires a detailed analysis of user behavior, one can estimate such savings by looking at the energy used to perform common activities.



This figure shows the amount of energy needed to page through a presentation a second time. The energy needed to page through each presentation the first time is also shown for comparison. For each data set, the left two bars show energy use for a full-quality presentation, and the right two bars show energy use for a reduced-quality version. Document D is omitted from this experiment since it contains only a single slide. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

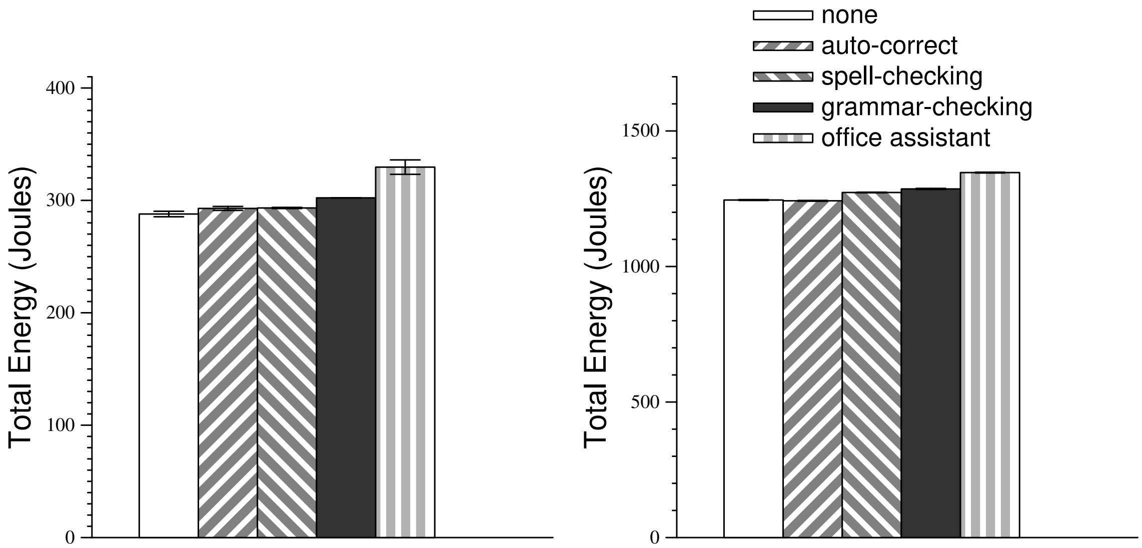
Figure 5.6: Energy used to re-page through presentations

One very common activity is paging through the slides in a presentation. I created a Visual Basic program to simulate the actions of a user performing this activity. The program loads the first slide, then sends `PageDown` keystrokes to PowerPoint until all remaining slides have been displayed. After sending each keystroke, the program waits for the new slide to render, then pauses for a second to simulate user think-time.

I measured the energy used to page through both the full-quality and distilled versions of each document. Figure 5.5 presents these results for nine of the ten presentations in the sample set—presentation D is omitted because it contains only a single slide. As shown by the difference in height between each pair of bars in Figure 5.5, distilling a document with large amounts of multimedia content can significantly reduce the energy needed to page through the document. Energy savings range from 1% to 30%, with an average of 13%.

After PowerPoint displays a slide, it appears to cache data allowing it to quickly re-render the slide, thereby reducing the energy needed for redisplay. This effect is shown in Figure 5.6, which displays the energy used to page through each document a second time.

For ease of comparison, Figure 5.6 also shows the energy needed to page through each document the first time. Comparing the heights of corresponding bars shows that sub-



(a) no interval

(b) 100 ms. interval

This figure shows the amount of energy needed to perform background activities during text entry. The graph on the left shows energy use when text is entered without pause, and the graph on the right shows energy use with a 100 ms. pause between characters. Each bar shows the cumulative effect of performing background activities, so the leftmost bar in each graph was measured while no background activities were being performed, and the rightmost bar in each graph was measured while all background activities were being performed. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Figure 5.7: Energy used by background activities during text entry

sequent slide renderings use less energy than the initial renderings. Thus, the benefit of distillation is smaller on subsequent traversals of the document: ranging from negligible to 20% with an average value of 5%.

5.4.3 Background activities

I next measured the energy used to perform background activities such as auto-correction and spell-checking. Whenever a user enters text, PowerPoint may perform background processing to analyze the input and offer advice and corrections to the user. When battery levels are critical, such background processing could be disabled to extend battery lifetime.

I measured the effect of auto-correction, spell-checking, style-checking, and providing advice through the Office Assistant (paperclip). I created a Visual Basic program which enters a fixed amount of text on a blank slide. The program sends keystrokes to PowerPoint, pausing for a specified amount of time between each keystroke.

Figure 5.7(a) shows the energy used to enter text with no pause between keystrokes;

Figure 5.7(b) shows energy usage with a 100 ms. pause between keystrokes. I first measured energy usage with no background activities enabled, and then successively enabled auto-correction, spell-checking, style-checking, and the Office Assistant. Thus, the difference between any bar in Figure 5.7 and the bar to its left shows the amount of additional energy used to perform a specific background activity. For example, the difference between the first two bars in each graph shows the effect of auto-correction.

Figure 5.7 shows that auto-correction expends negligible energy when entering text—the additional energy can not be distinguished from experimental error. Spell-checking and style-checking incur a small additional cost. With no pause between entering characters, employing these options adds a 5.0% energy overhead—with a 100 ms. pause between characters, the overhead is 3.3%.

The Office Assistant incurs a more significant energy penalty. With no pause between typing characters, enabling the Assistant leads to a 9.1% increase in energy use. With a 100 ms. pause, energy use increases 4.9%. In fact, even when the user is performing no activity, enabling the Office Assistant still consumes an additional 0.3 Watts, increasing power usage 4.4% on the measured system. Adaptively disabling the Office Assistant can therefore lead to a small but significant extension in battery lifetime.

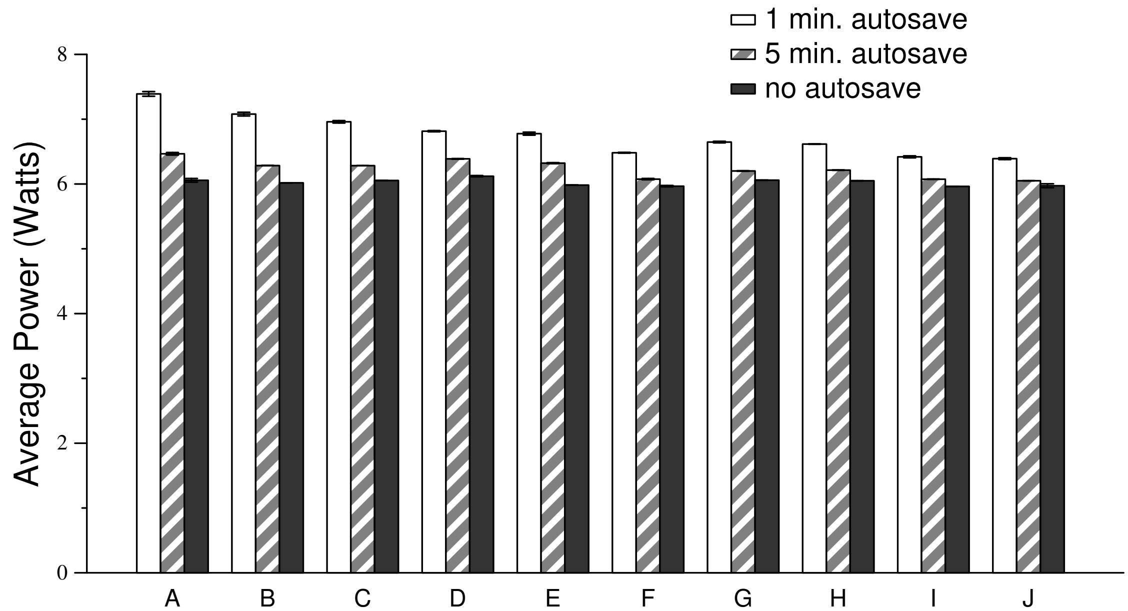
5.4.4 Autosave

Autosave frequency is another potential dimension of energy-aware adaptation. After a document is modified, PowerPoint periodically saves an AutoRecovery file to disk in order to preserve edits in the event of a system or application crash. Autosave may be optionally enabled or disabled—if it is enabled, the frequency of autosave is a configurable parameter. Since autosave is performed as a background activity, it often will have little effect upon perceived application performance. However, the energy cost is not negligible: the disk must be spun up, and, for large documents, a considerable amount of data must be written.

Since periodic autosaves over the wireless network would be prohibitively slow, I assumed that documents are stored on local disk. I created a Visual Basic program to help quantify the energy impact of autosave. The program loads a PowerPoint document, makes a small modification (adds one slide), and then performs no further activity for twenty minutes. To avoid spurious measurement of initial activity associated with loading and modifying the presentation, I waited for ten minutes after the modification was made to the document before I began measuring power usage.

Figure 5.8 shows power usage for three autosave frequencies. For each presentation, the first bar shows power usage when the full-quality version of the document is modified and a one minute autosave frequency is specified. The next bar in each data set shows the effect of a five minute autosave frequency. The final bar shows the effect of disabling autosave—this is the maximum power reduction that can be achieved by modifying autosave parameters.

As can be seen by the difference between the first two bars of each data set, changing the autosave frequency from 1 minute to 5 minutes reduces power usage 5–12%, with an average reduction of 8%. The maximum possible benefit is realized when autosave is



This figure shows how the frequency of PowerPoint autosave affects power usage. The three bars in each data set show power use with a 1 minute autosave frequency, with a 5 minute autosave frequency, and with autosave disabled. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Figure 5.8: Effect of autosave options on application power usage

disabled. As shown by the difference between the first and last bars in each data set, this reduces power usage 7–18% with an average reduction of 11%. Thus, depending upon the user's willingness to hazard data loss in the event of crashes, autosave frequency is a potentially useful dimension of energy-aware adaptation.

5.5 Summary

I began this chapter by asking several questions: Can one show significant energy reductions for office applications commonly executed on laptop computers? Is it possible to add energy awareness without access to application source code? Is this approach valid for applications executing on source-code unavailable operating systems (i.e. Windows)?

As the results of Section 5.4 show, the answer to all these questions is "yes". Using Puppeteer's component-based adaptation approach, one can add energy-awareness without application or operating system source code. Distillation reduces the energy needed to load PowerPoint presentations from a remote server by 49%. Distillation can also lead to significant energy savings when editing and saving presentations. Finally, I showed how modifications such as disabling the Office Assistant and lowering autosave frequency can lead to further energy conservation.

Chapter 6

System support for energy-aware adaptation

The previous two chapters demonstrated the feasibility of energy-aware adaptation. They showed that applications can modify their behavior to significantly reduce the energy usage of the platforms on which they execute. However, since energy-aware adaptation comes at the cost of degraded fidelity, applications should only adapt their behavior when necessary. This chapter explores how system support can guide applications to make appropriate adaptation decisions.

The next section explores goal-directed adaptation, a feedback technique which estimates the importance of reducing energy usage. By measuring energy supply and demand, the operating system is able to determine the correct balance between energy conservation and application fidelity. Section 6.2 then shows how the system can improve the agility of adaptation decisions by learning from a history of past application energy usage.

6.1 Goal-directed adaptation

Current operating systems provide little support for energy management. Typically, the system reports the expected remaining battery lifetime at the current rate of energy usage. In addition, the system usually provides a limited number of device-specific parameters that modify hardware power management policies.

Users who wish to extend their computer's battery lifetimes must manually adjust several parameters such as the screen brightness and disk spin-down timeout to select the correct level of energy conservation. In order to set these parameters correctly, they must develop an intuitive notion of how each parameter affects energy use. Further, they must periodically observe the estimate for remaining battery lifetime in order to verify that the chosen settings are correct.

Goal-directed adaptation inverts this process by making energy management the responsibility of the operating system. The user specifies only desired battery lifetime. The system assumes responsibility for making the correct tradeoffs between quality and energy conservation that ensure that the battery lasts for the specified duration. Rather than adjusting power management parameters, the system adjusts application fidelity, using feedback to determine correct settings. The system also performs the task of monitoring remaining battery lifetime to ensure that the chosen fidelity settings are correct.

6.1.1 Design considerations

The most important consideration in the design of goal-directed adaptation is ensuring that the specified time goal is met whenever feasible. Clearly, users will only trust the system to manage their battery energy if it proves that it can reliably meet the specified goals. When a user specifies an infeasible duration, one so large that the available energy is inadequate even if all applications run at lowest fidelity, the system should detect the problem and alert the user as soon as possible.

An important secondary goal is providing the best user experience possible. This translates into two requirements: first, applications should offer as high a fidelity as possible at all times; second, the user should not be jarred by frequent adaptations. Goal-directed adaptation balances these opposing concerns by striving to provide high average fidelity while using hysteresis to reduce the frequency of fidelity changes.

When the user specifies a battery lifetime, it is important to recognize that the duration represents the amount of work the user wishes to accomplish while on battery power. Adaptive strategies that extend battery lifetime but accomplish less total work should therefore be avoided. For example, consider a workload of scientific calculations performed on the Itsy v1.5. Reducing processor clock frequency decreases the average power used to perform a calculation, but increases total energy consumption. Thus, although battery lifetime is extended, the total number of calculations performed will be decreased. This scenario makes it clear that fidelity should only be reduced when applications use less total energy per unit of work at the reduced fidelity. All of the fidelity reductions studied in the previous two chapters have this property.

6.1.2 Implementation

The Odyssey platform for mobile computing, described in Section 2.3, provides the basis for implementing goal-directed adaptation. I created a simple user interface that allows users to specify goals for battery duration and receive feedback about the feasibility of the specified goals.

Further, I modified Odyssey to perform three tasks periodically. First, Odyssey determines the residual energy available in the battery. Second, it predicts future energy demand. Third, based on these two pieces of information, it decides if applications should change fidelity and notifies them accordingly.

Figure 6.1: User interface for goal-directed adaptation

User interface

Often the estimate for needed battery lifetime will be driven by external criteria—for example, the expected duration of a flight, commute, or meeting. Since the exact duration of such events is often unknown, Odyssey allows the user to respecify the time goal when necessary. Whenever this happens, Odyssey either adapts to meet the new goal or notifies the user that it is infeasible.

Figure 6.1 shows the user interface for goal-directed adaptation. This interface is designed to be as simple as possible so as to avoid unnecessary user distraction. The user specifies the goal for battery duration by adjusting the slider in the middle of the dialog. When conditions change, the user may readjust the slider to specify a new battery goal. As time passes and the battery drains, the slider moves to the left to express the change in expected remaining battery lifetime. For example, the dialog in Figure 6.1 shows that the expected remaining battery lifetime is 138 minutes. After one hour, the dialog will show 78 minutes of remaining lifetime, assuming that the user does not adjust the battery goal in the meantime. While current operating systems provide similar dialogs, they are output-only; they do not provide users with the ability to change the expected battery lifetime.

The dialog also displays the state of battery management to the user. Figure 6.1 shows the normal state; the battery lifetime is being managed by Odyssey and the specified goal appears feasible. If Odyssey determines that the goal is infeasible, the interface changes the dialog title and displays a red background. This warns the user that the battery will expire sooner than the specified time—the user may then decrease activity or specify a shorter duration. Similarly, when Odyssey determines that the battery will last longer than the specified duration even if all applications execute at their highest fidelities, the interface changes the dialog title and displays a green background. If the mobile computer is connected to wall-power, the slider is disabled since goal-directed adaptation is only meaningful when the battery is discharging.

Determining residual energy

Odyssey determines residual energy by measuring the amount of charge in the battery. Many modern mobile computers ship with a Smart Battery [79], a gas gauge chip which reports detailed information about battery state and power usage. On such platforms, Odyssey periodically queries the Smart Battery chip to determine remaining battery capacity.

The specific query interface may be machine-dependent. For example, the Itsy v2.2 contains a Dallas Semiconductor DS2437 Smart Battery chip [12]. Odyssey queries the battery through a one-wire bus connected to a general purpose input/output (GPIO) pin of the StrongArm 1100 processor. A Linux device driver for this chip was developed by Compaq Western Research Lab. Once per minute, Odyssey performs an `ioctl` on the device to query remaining capacity.

The Advanced Configuration and Power Interface (ACPI) specification [36] provides a more standard interface to Smart Battery information. The Linux ACPI driver exports battery status information through the `/proc` interface. On the IBM ThinkPad T20 laptop, Odyssey reads the remaining capacity through this interface once every ten seconds.

While the above methods are best for deployed systems, they may not be ideal in laboratory settings. Older mobile computers such as the Itsy v1.5 and IBM ThinkPad 560X lack the necessary hardware support for measuring battery capacity. Additionally, since an actual battery must be used to obtain measurements, it is difficult to control for non-ideal battery behavior. Finally, running large numbers of experiments is difficult because one must recharge the battery after each trial.

To facilitate evaluation, Odyssey can optionally use a *modulated* energy supply. At the beginning of evaluation, the initial value for the modulated supply is specified to Odyssey. As the evaluation proceeds, a digital multimeter samples the actual power usage of the mobile computer and transmits the samples to Odyssey. When it receives a new sample, Odyssey assumes constant power consumption between samples and decrements the modulated energy supply by the product of the sampled power usage and the sampling period. When the value of the modulated energy supply reaches zero, Odyssey reports that the battery has expired.

Predicting future energy demand

To predict energy demand, Odyssey assumes that future behavior will be similar to recently-observed behavior. Odyssey uses smoothed observations of present and past power usage to predict future power use. This approach is in contrast to requiring applications to explicitly declare their future energy usage—an approach that places unnecessary burden on applications and is unlikely to be accurate.

Odyssey uses methods similar to those described in the previous section to observe power usage. On the Itsy v2.2, Odyssey queries the Smart Battery to measure current and voltage levels once per second. It multiplies the two values to calculate power usage. On the ThinkPad T20, an artifact of the Linux ACPI driver prevents Odyssey from querying current and voltage levels. Instead, Odyssey estimates power usage by periodically sampling remaining battery capacity and dividing the difference in capacity by the sample period. When the battery supply is modulated, Odyssey uses the power samples collected by the external digital multimeter.

To smooth power estimates, Odyssey uses an exponential weighted average of the form:

$$\text{new} = (1 - \alpha)(\text{this sample}) + (\alpha)(\text{old}) \tag{6.1}$$

where α is the gain, a parameter that determines the relative weights of current and past power usage. Once future power usage has been estimated, it is multiplied by the time remaining until the goal to obtain future energy demand.

Odyssey varies α as energy drains, thus changing the tradeoff between agility and stability. When the goal is distant, Odyssey uses a large α. This biases adaptation toward stability by reducing the number of fidelity changes—there is ample time to make adjustments later, if necessary. As the goal nears, Odyssey decreases α so that adaptation is biased toward agility. Applications now respond more rapidly, since the margin for error is small.

Currently, Odyssey sets α so that the half-life of the decay function is approximately 10% of the time remaining until the goal. For example, if 30 minutes remain, α is chosen so that the present estimate will be weighted equally with more recent samples after approximately 3 minutes have passed. Specifically, if one sample is collected per second, and T is the time remaining to the goal:

$$\alpha = 2^{-1/(0.1 \times T)} \tag{6.2}$$

The choice of 10% is based on a sensitivity analysis, discussed in Section 6.1.4.

Triggering adaptation

When predicted demand exceeds residual energy, Odyssey advises applications to reduce their energy usage. Conversely, when residual energy significantly exceeds predicted demand, applications are advised to increase fidelity. In this chapter, fidelity reduction is the sole method of energy conservation—Chapter 7 describes the system support necessary to support remote execution as a further dimension of energy conservation.

The amount by which supply must exceed demand to trigger fidelity improvement is indicative of the level of hysteresis in Odyssey's adaptation strategy. This value is the sum of two empirically-derived components: a variable component, 5% of residual energy, and a constant component, 1% of the initial energy available. The variable component reflects the bias toward stability when energy is plentiful and toward agility when it is scarce; the constant component biases against fidelity improvements when residual energy is low. As a guard against excessive adaptation due to energy transients, Odyssey caps fidelity improvements at a maximum rate of once every 15 seconds.

To meet the specified goal for battery duration, Odyssey tries to keep power demand within the zone of hysteresis. Whenever power demand exceeds those bounds, adaptation is necessary. Odyssey then determines which applications should change fidelity, as well as the magnitude of change needed for each application. Ideally, the set of fidelity changes should modify power demand so that it once again enters the zone of hysteresis.

Often, there will be many possible sets of adaptations that yield the needed change in power demand. In these cases, Odyssey must refer to an *adaptation policy* to decide which alternative is best. Although one can implement many different adaptation policies, my initial prototype used a simple one based on user-specified priorities. Each application specifies a list of supported fidelities—lower fidelity levels are assumed to use less energy than higher fidelities. The user assigns static priorities to arbitrate between applications. Odyssey always degrades a lower-priority application before degrading a higher-priority one—upgrades occur in reverse order. Once the lowest-priority application is degraded to its lowest fidelity level, Odyssey degrades the fidelity of the next lowest-priority application.

After an application adapts, Odyssey verifies whether the change in power demand is sufficient. It resets the estimate of power demand to a value that is precisely in the middle of the zone of hysteresis—this represents an optimistic assumption that the change in fidelity will prove sufficient. As Odyssey takes new power measurements, the estimate of power demand will either stay within the zone of hysteresis, indicating that the change in fidelity was correct, or stray outside the bounds, indicating that additional adaptation is necessary. In the later case, Odyssey will continue to adapt application behavior until it reaches a value where power demand stays within the zone of hysteresis.

In the rest of this chapter, this adaptation policy will be referred to as the *incremental policy*, because it changes fidelity one step at a time. The incremental strategy is sufficient to evaluate goal-directed adaptation. However, it is clearly inadequate to model many of the policies that a user may wish to express. Section 6.2 describes the system support necessary to support more detailed policies.

6.1.3 Basic validation

Experiment design

To validate goal-directed adaptation, I executed the two energy-aware applications described in Section 4.8: a composite application involving speech recognition, map viewing and Web access, run concurrently with a background video application. To obtain a continuous workload, the composite application executes every 25 seconds. This has the effect of holding the amount of work (number of recognitions, image views, and map views) constant over time.

The video application supports four fidelity levels: full-quality, Premiere-B, Premiere-C, and the combination of Premiere-C and reduction of the display window. The speech recognition component of the composite application supports full and reduced quality recognition on the local machine. The map component has four fidelities: full-quality, minor road filtering, secondary road filtering, and the combination of secondary road filtering and cropping. The Web component supports the five fidelities shown in Figure 4.15. I prioritized these components so that speech had the lowest priority, and video, map, and Web had successively higher priority.

Client applications execute on the IBM 560X laptop, and communicate with servers using a 900 MHz 2 Mb/s Lucent WaveLAN wireless network. To isolate the performance of goal-directed adaptation, Odyssey uses a modulated energy supply. In addition, Odyssey uses the incremental adaptation policy described in the previous section.

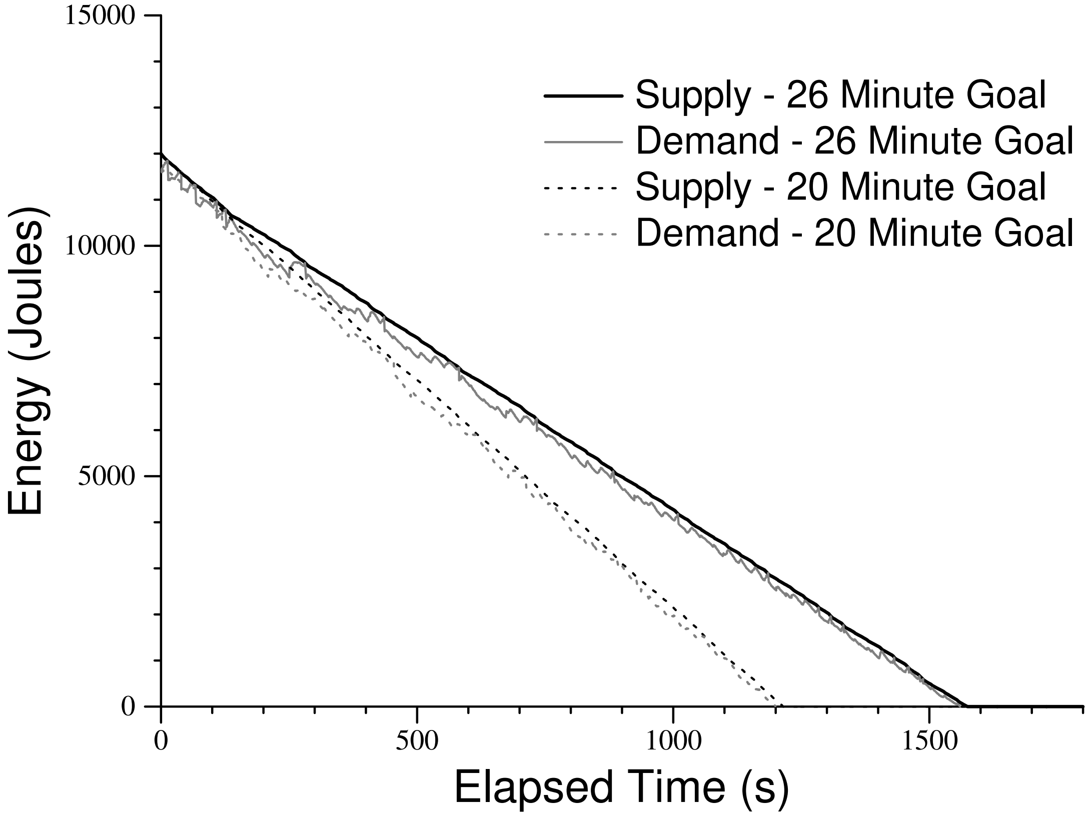
At the beginning of each experiment, I provided Odyssey with an initial energy value and a time goal. I then executed the applications, allowing them to adapt under Odyssey's direction until the time goal was reached or the residual energy dropped to zero. The former outcome represents a successful completion of the experiment, while the latter represents a failure. I noted the total number of adaptations of each application during the experiment. I also noted the residual energy at the end of the experiment—a large value suggests that Odyssey may have been too conservative in its adaptation decisions and that average fidelity could have been higher.

All experiments used a 12,000 Joule modulated energy supply. This lasts 19:27 minutes when applications operate at highest fidelity, and 27:06 minutes at lowest fidelity. The difference between the two values represents a 39.3% extension in battery life. I deliberately chose a small initial energy value so that I could perform a large number of experiments in a reasonable amount of time. The value of 12,000 Joules is only about 14% of the nominal energy in the IBM 560X battery. Extrapolating to full nominal energy, the workload would run for 2:18 hours at highest fidelity, and 3:13 hours at lowest fidelity.

Results

Figures 6.2 and 6.3 show detailed results from two typical experiments: one with a 20 minute goal, and the other with a 26 minute goal. Figure 6.2 shows how the supply of energy and Odyssey's estimate of future demand change over time during the two experiments. In both trials, Odyssey meets the specified goal for battery duration. In addition, once the time goal is reached, residual energy is quite low. The graph also confirms that estimated demand tracks supply closely. The most visible difference between the two trials is the slope of the supply and demand lines. After an initial adaptation period of approximately three minutes, Odyssey adjusts power usage in the 26 minute trial in order to extend battery lifetime.

The four graphs of Figure 6.3 show how the fidelity of each application varies during the two experiments. For the 20 minute goal, the high priority Web and map applications remain at full fidelity throughout the experiment; the video degrades slightly; and speech runs mostly at low fidelity. For the 26 minute goal, the highest priority Web application runs

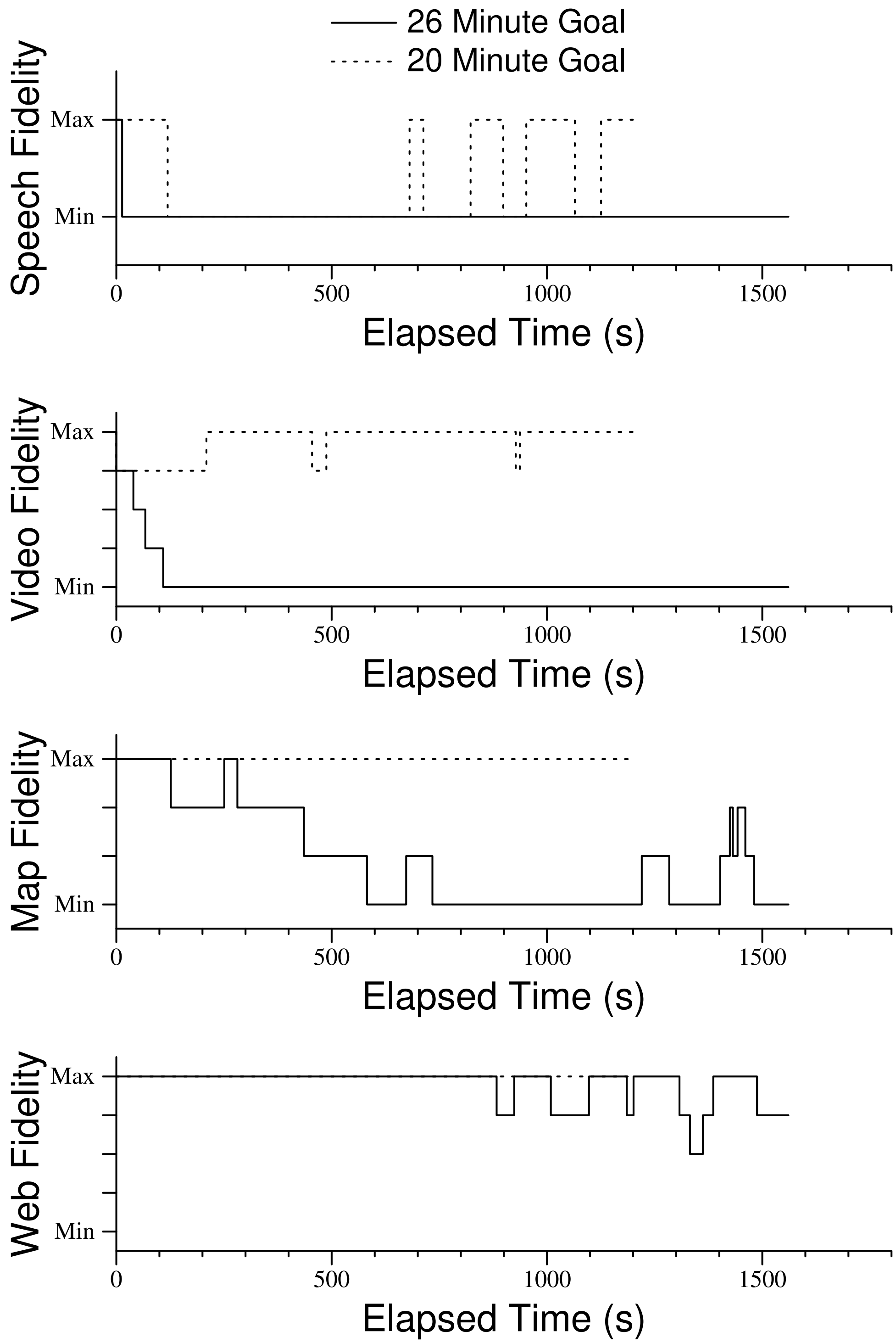


This figure shows how Odyssey meets user-specified goals for battery durations of 20 and 26 minutes when running the composite and video applications described in Section 4.8. It shows how the modulated supply of energy and the estimated energy demand change over time.

Figure 6.2: Example of goal-directed adaptation—supply and demand

mostly at the highest fidelity, while the other three applications run mostly at their lowest fidelities. The difference in fidelity levels between the two trials, shown most clearly for the video application, accounts for the difference in power demand shown in Figure 6.2. For both time goals, applications adapt less frequently at the beginning of the experiment and more frequently as the time goal nears. This shows the effect of the variable gain which makes the system stable when the goal is distant and agile when the goal is near.

Figure 6.4 summarizes the results of five trials for each time goal of 20, 22, 24, and 26 minutes. These results confirm that Odyssey is doing a good job of energy adaptation. The desired goal was met in every trial. In all cases, residual energy was very low: the largest average residue, for the 20 minute experiment, is still only 1.2% of the initial energy value. The average number of adaptations by applications is generally low, but there are some cases where it is high. However, the cases that exhibit high number of adaptations are an artifact of the small initial energy value, since the system is designed to exhibit greater agility when energy is scarce.



This figure shows how Odyssey meets user-specified goals for battery durations of 20 and 26 minutes when running the composite and video applications described in Section 4.8. It shows changes in application fidelity. The applications are prioritized with speech having the lowest priority, and video, map, and Web having successively higher priority.

Figure 6.3: Example of goal-directed adaptation—application fidelity

Specified Duration (s)	Goal Met	Residue		Number of Adaptations			
		Energy (J)	Time (s)	Speech	Video	Map	Web
1200	100%	145.2 (25.3)	15.3 (1.9)	10.8 (1.6)	11.0 (4.0)	0.4 (0.9)	0.0 (0.0)
1320	100%	107.5 (61.5)	12.9 (7.2)	2.8 (0.4)	28.2 (5.2)	1.6 (2.6)	0.0 (0.0)
1440	100%	101.2 (22.3)	13.0 (4.5)	5.0 (7.9)	22.6 (9.8)	9.6 (3.8)	1.2 (1.8)
1560	100%	60.2 (28.7)	8.7 (5.9)	1.0 (0.0)	6.0 (2.8)	15.4 (4.6)	7.6 (5.9)

This figure shows system behavior when the composite application executes concurrently with the video player. Each experiment uses a 12,000 Joule modulated energy supply. Each row shows the result of specifying a different battery-duration goal. The second column shows the percentage of trials in which the energy supply lasted for at least the specified duration. The next two columns show the average residual energy at the end of the experiment. The remaining columns show the average number of adaptations performed by each application. Each entry represents the mean of five trials with standard deviations given in parentheses.

Figure 6.4: Summary of goal-directed adaptation

6.1.4 Sensitivity to half-life

The choice of smoothing function is very important in the design of goal-directed adaptation. I therefore examined how changing the half-life parameter used to calculate the gain, α, affects system performance. I used the same experimental setup as in Section 6.1.3. On the client, I executed the video application and displayed a half-hour video clip. I used a modulated energy supply of 13,000 Joules and specified a 30 minute time goal so the supply would last for the entire video.

Figure 6.5 summarizes the results of five trials each for half-life values of 1%, 5%, 10%, and 15%. A half-life of 1% is clearly too unstable—the system produces the largest residue with this value, and the video player adapts excessively. For larger half-life values, the system is more stable. However, with a 15% half-life, the system is insufficiently agile, failing to meet the goal in one of the five trials. I have also encountered similar results in less detailed analysis of other adaptive applications. This led me to the current approach of using a 10% half-life to calculate the value of α.

6.1.5 Validation with longer duration experiments

The short duration (20–30 minutes) of the experiments in Sections 6.1.3 and 6.1.4 allowed me to explore the behavior of Odyssey for many parameter combinations. Having established the feasibility of goal-directed adaptation, I then ran a small number of longer duration experiments to confirm its benefits in more realistic scenarios.

I used the same experimental setup as in section 6.1.3. I began each experiment with an energy supply of 90,000 Joules, roughly matching a fully-charged ThinkPad 560X battery. I specified an initial time duration of 2:45 hours, but extended this goal by 30 minutes at

Half-Life	Goal Met	Residue (J)		Adaptations	
0.01	100%	204.6	(17.7)	93.6	(3.7)
0.05	100%	124.1	(38.0)	33.2	(4.0)
0.10	100%	129.2	(21.6)	14.6	(5.4)
0.15	80%	97.6	(22.2)	6.8	(2.9)

This figure shows sensitivity to the half-life value used for smoothing. In each experiment, I assume a 13,000 Joule energy supply. The second column shows the percentage of trials in which the energy supply lasted for at least the specified duration. The next column shows the residual energy at the end of the experiment. The final column shows the number of adaptations performed. Each entry shows the mean of five trials with standard deviation given in parentheses.

Figure 6.5: Sensitivity to half-life

the end of the first hour. This change reflects the possibility that a user may modify the estimate of how long the battery needs to last. Finally, I used a simple stochastic model to construct an irregular workload. During any given minute, each of four applications (video, speech, map, and Web) may independently be active or idle. An active application executes a fixed workload for one minute; for example, the video application shows a one minute video, and the map application fetches five maps with five seconds of think time after each map display. After each minute, there is a 10% chance of switching states; that is, an active application stays active, and an idle one stays idle, with probability 0.9.

Figure 6.6 presents the results of five trials of this experiment, each generated with a different random number seed. In every case, Odyssey succeeded in meeting its time goal. Four of the five trials ended with a residual energy that was less than 1% of the initial supply. Only in one trial (trial 3) was the residue noticeably higher (2.8%), implying that Odyssey was too conservative in its adaptation.

In spite of the bursty workload, Figure 6.6 shows fewer adaptations than Figure 6.4, which had a steady workload. This is a consequence of two interactions between the hysteresis strategy and the longer-duration goal. First, the zone of hysteresis is much larger, since it is proportional to total energy supply. Second, smoothing is more aggressive when the goal is distant. Combined, these two factors cause Odyssey to ignore minor fluctuations in power usage except toward the end of each trial.

6.1.6 Overhead

The power overhead imposed by goal-directed adaptation is the sum of the overhead of measuring power usage and the overhead of using these measurements to predict energy demand. The measurement overhead is quite low. Most Smart Battery solutions can provide power measurements at the frequency Odyssey requires using less than 10 mW. [12, 88]. Additionally, if a mobile computer already includes a Smart Battery (as with the IBM T20 laptop and Itsy v2.2), Odyssey imposes minimal additional measurement overhead.

Trial	Goal Met	Residual Energy (J)	Number of Adaptations			
			Speech	Video	Map	Web
1	Yes	345	1	5	5	1
2	Yes	381	1	10	7	11
3	Yes	2486	8	13	5	0
4	Yes	554	2	10	6	8
5	Yes	464	5	6	14	0

This figure shows system behavior for bursty workloads. In each trial, I assume an energy supply of 90,000 Joules. After one hour, the initial goal of 2.75 hours is extended by a half hour. Each row shows the result of a trial using a different randomly-generated workload. The second column shows whether the energy supply lasted for at least the specified duration. The next column shows the residual energy at the end of the experiment. The remaining columns show the number of adaptations performed by each application.

Figure 6.6: Longer duration goal-directed adaptation

I have measured Odyssey's prediction overhead and found it to be only 4 mW. Therefore, the total power overhead imposed by goal-directed adaptation is less than 14 mW.—only 0.25% of the background power consumption of the IBM 560X.

6.2 Use of application resource history

The previous section showed that goal-directed adaptation can be successfully implemented without needing to anticipate how changes in application fidelity affect power usage. The incremental adaptation policy allows Odyssey to eventually converge on the fidelity level that correctly balances energy use and application quality. Yet, when Odyssey can anticipate the relationship between fidelity and power use, it can be more agile in adapting to changes in power demand. In addition, anticipating this relationship allows applications specify a more natural range of fidelities and adaptation policies.

This section describes how Odyssey obtains these benefits by maintaining a history of application energy usage. The work on which this section reports was done jointly with Dushyanth Narayanan [64]. As applications execute, Odyssey monitors and logs the fidelities at which they operate and their corresponding energy consumption. From the data, Odyssey learns functions which relate fidelity and energy usage. From these functions, Odyssey predicts how power usage changes with fidelity.

Next, I examine in more detail the benefits of using application resource history to predict future behavior. Section 6.2.2 describes how Odyssey measures and logs energy usage. Section 6.2.3 describes how Odyssey uses the data to learn functions which predict application energy use; Sections 6.2.4 and 6.2.5 show how the learned functions are used to improve adaptation decisions. Section 6.2.6 then demonstrates how Odyssey uses the history of application energy usage to increase the effectiveness of goal-directed adaptation.

6.2.1 Benefits of application resource history

If Odyssey cannot determine the relationship between fidelity and energy use, it will not be able to anticipate the magnitude of change in power usage that will result when applications change fidelity. Thus, when Odyssey desires to increase or decrease power use, it must adapt application fidelity one discrete step at a time. Further, after each adaptation, it must pause to assess how much power usage has changed.

This means that application fidelities must be carefully chosen, since the number of fidelities affects the agility of the system. If a large number of fidelities are specified, Odyssey is sluggish to react to changes in power supply and demand. Before arriving at the correct fidelity, it must transition through many intermediate fidelity levels, pausing at each one to assess the impact of the adaptation on power usage. Potentially, Odyssey may even miss the specified goal for battery duration if it is not sufficiently agile in adapting when a decrease in power usage is needed. At the other extreme, if only a small number of fidelities are specified, the system may oscillate between two widely-separated fidelity levels because no intermediate value is available, leading to an excessive number of adaptations.

However, when Odyssey can predict how fidelity changes will affect energy usage, it can calculate the exact change in application fidelity that will yield a desired change in power usage. It can then adjust fidelity directly to the new value without traversing through intermediate values. With this approach, the number of fidelity levels specified by an application will not directly impact system agility.

Use of energy history also lets applications support a more natural range of fidelities and adaptation policies. Applications can specify continuous dimensions of fidelity, whereas if Odyssey uses the incremental policy, only a single, discrete dimension of fidelity can be specified (since Odyssey can only adapt in discrete steps). Applications can also specify adaptation policies which directly relate fidelity and energy use. For example, applications may wish to specify policies such as "Choose the best fidelity that uses no more than x% of the energy of the highest fidelity" or "Degrade fidelity if the decrease in energy use is greater than the corresponding decrease in fidelity". Odyssey can only evaluate such policies if it is able to predict the amount of energy that will be used at different fidelities.

6.2.2 Recording application resource history

Odyssey's fundamental unit of history is an *operation*: a discrete unit of work performed by an application at one of possibly many fidelities. For example, each application described in Chapter 4 performs one basic operation: the video player displays videos, the speech recognizer recognizes utterances, the map viewer displays maps, and the Web browser displays images.

Odyssey and applications collaborate to generate a history of application energy use. Applications initially describe the operations that they plan to execute, including possible fidelities and *input parameters*, variables that affect the complexity of the operation. They then signal Odyssey when the execution of each operation begins and ends. During execution, Odyssey measures energy usage. Later, when the operation completes, Odyssey logs

```
register_fidelity   (char* conf_file,
                     int* optype_id);
begin_fidelity_op   (const char* dataname,
                     int optype_id,
                     int num_params,
                     fid_param_val_t* params,
                     int num_fidelities,
                     fid_param_val_t* fidelities,
                     int* opidp);
int end_fidelity_op (int optype_id,
                     int opid,
                     int failed);
```

Figure 6.7: Odyssey multi-fidelity API

its energy usage, fidelity, and input parameters.

Figure 6.7 shows the relevant portion of Odyssey's multi-fidelity API, through which applications provide the information Odyssey needs to log resource usage. Initially, applications call `register_fidelity()` to describe the types of operations that they plan to execute. For each operation type, the application specifies the name of a configuration file. Odyssey returns a unique identifier for the operation type—the application uses the identifier in future calls.

Figure 6.8 shows the configuration file for the Web browser described in Section 4.7. The first line specifies the names of the application and operation type. The next line names a file to which resource history will be logged. The third line describes the single input parameter to the operation, the number of bytes in the uncompressed image. The fourth line describes the single fidelity, which is the degree of JPEG compression to be employed. The configuration file specifies that this dimension of fidelity is continuous and ranges in value between 5 and 100. Odyssey also allows applications to specify discrete parameters or fidelities which can take on one of several enumerated values. The remainder of the configuration file is not relevant to history logging, and will be described in Section 6.2.4.

Before executing an operation, applications call `begin_fidelity_op()`, which has two purposes. It asks Odyssey which fidelity should be used for the operation. It also signals Odyssey to begin measuring resource usage for the operation. The application specifies the name of the data object being manipulated (i.e. the title of a video, the name of a Web image, etc.), as well as the type of operation to be performed. The application also specifies the specific values of input parameters to the operation. For example, the Web browser obtains the number of bytes of the image to be fetched from the HTML headers and passes the information to Odyssey. Odyssey returns the fidelity at which the operation

```
description web:fetchimage
logfile /usr/odyssey/etc/web.fetchimage.log
param imagesize ordered 0-10000000
fidelity compression ordered 5-100
mode normal
hintfile /usr/odyssey/lib/web_fetchimage_hints.so
utility web_fetchimage_utility
```

Figure 6.8: Sample configuration file for a Web browser

should be performed, as well as a unique handle by which the application can refer to the operation.

After an operation completes, the application calls `end_fidelity_op`. This causes Odyssey to write an entry in the appropriate log file. Each entry includes resource usage information, such as the time and energy to execute the operation. Each entry also contains the value of each input parameter and the fidelity at which the operation was performed.

Odyssey measures energy usage as described in Section 6.1.2. If the hardware platform supports ACPI, Odyssey queries the amount of energy remaining in the battery at the start and end of each operation. The difference between the two values is the total energy used by the operation. If battery level information is not available, Odyssey samples power usage during operation execution, and calculates energy usage by multiplying average power usage by execution time. For example, when executing on the Itsy v2.2, Odyssey reads current levels from the DS2437 Smart Battery chip six times per second. These measurements are multiplied by the last voltage reading and the duration of the sample period to calculate the total energy use during the measurement period. Odyssey accumulates the energy measurements taken during operation execution, and logs the sum as the total energy usage of the operation.

Both methods for measuring operation energy use are inadequate if more than one operation executes simultaneously. With concurrent operations, it is unclear what percentage of the total energy expenditure should be attributed to each operation. One possible method for apportioning energy usage would be to allocate energy usage according to the percentage of CPU, disk, network, and other resources used by the operation. However, this method requires detailed resource accounting, as well as per-platform calibration of the energy costs of activities such as network transmissions and disk accesses. I have therefore chosen the simpler approach of excluding from the history log any operation which overlaps with another simultaneous operation. This sacrifices some data, but increases the accuracy of the logged values.

6.2.3 Learning from application resource history

From the history of application resource usage, Odyssey learns functions which relate input parameters and fidelity to energy usage. As the study of application energy usage in Chapter 4 shows, simple linear models provide a good fit for many applications. Odyssey supports this common case by providing library routines which generate linear models of energy use from log data.

Odyssey reads the previously logged data whenever an application registers a new operation type. From the data, Odyssey generate a linear model of energy usage. If the operation type has a continuous dimension of fidelity or a continuous input parameter, a model is generated using linear regression. If more than one continuous dimension exists, multiple variable linear regression is used. When the operation type has one or more discrete dimensions, Odyssey generates a different linear model for each possible combination of enumerated values. For example, the speech application discussed in Section 4.5.1 has one discrete dimension of fidelity: it can use either a full-sized or reduced vocabulary for recognition. For this application, Odyssey develops two separate linear models, one for each possible fidelity.

Later, Odyssey queries the model to determine how much energy an operation is likely to use if it is performed with different combinations of fidelity and input parameters. Sometimes, there will not be enough data to return a specific prediction, for example, if the operation has not yet been performed for one of the possible values of an enumerated type. In this case, the predictor returns a more generic prediction for energy use, one that is generated from a model that ignores the enumerated type.

In theory, for some applications, the relationship between energy use and fidelity may be quite complex. Odyssey supports such applications by allowing them to supply application-specific functions for predicting energy usage. In practice, I have found that the default linear models suffice for all applications that I have studied, including those described in Chapter 4.

6.2.4 Using application resource history to evaluate utility

Odyssey uses the history of application resource usage to determine the best fidelity at which to execute operations. For each operation, Odyssey chooses the one that maximizes the value of an application-specific *utility function* specified in the configuration file. For example, in Figure 6.8, the last two lines specify that the Web application's utility function is `web_fetchimage_utility()` in the library `web_fetchimage_hints.so`.

A utility function takes as input specific values for each input parameter and fidelity dimension, and returns a numerical result that represents the desirability of executing the operation with those values. When an application calls `begin_fidelity_op`, Odyssey searches the space of possible fidelities to select the fidelity which maximizes the value of the utility function. Input parameters are fixed to those specified by the application in the `begin_fidelity_op` call; fidelity varies as the search progresses.

```
extern fidelity desired_fidelity;

int utility (fidelity f)
{
    if (f == desired_fidelity) {
        return (1);
    } else {
        return (0);
    }
}
```

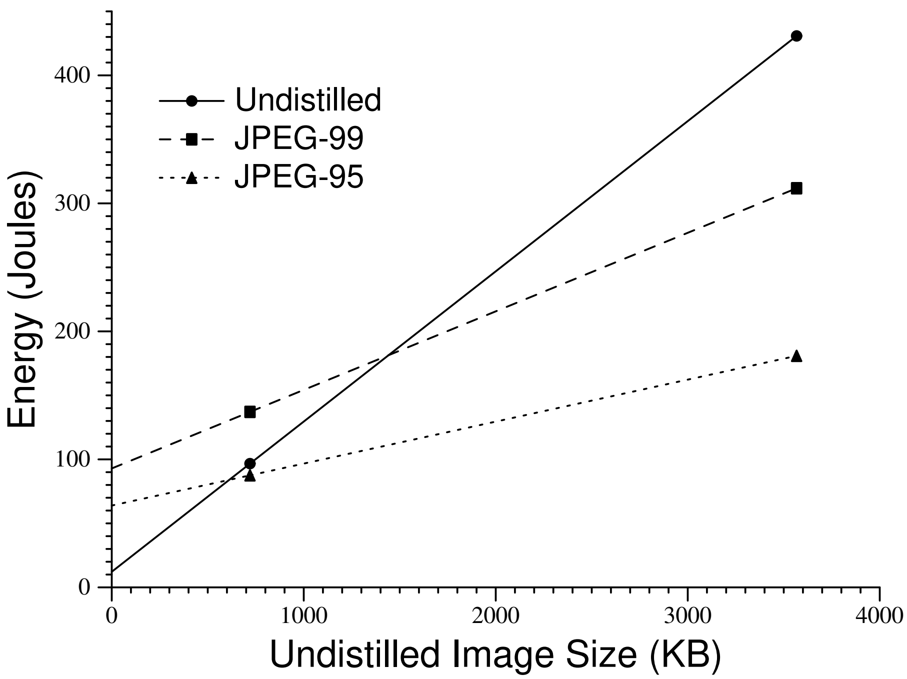
Figure 6.9: Utility function for the incremental policy

The Odyssey multi-fidelity solver, further described in [64], exhaustively searches discrete dimensions of fidelity and uses gradient-descent heuristics to search continuous dimensions. Thus, it is not guaranteed to find an optimal solution in all cases. The solver iteratively calls an operation's utility function, and selects the fidelity at which the utility function returns the highest value.

Energy-aware applications specify utility functions which balance fidelity and energy use. Goal-directed adaptation determines the current importance of energy conservation; utility functions express a specific policy for adapting to that importance.

Section 6.1.3 described the simple incremental adaptation policy: each application specifies a single, discrete dimension of fidelity, and Odyssey increases or decreases fidelity by one step whenever supply and demand diverge. One can trivially express this policy as a utility function, shown in Figure 6.9. Odyssey represents the desired fidelity for an application with a global parameter, $desired\_fidelity$. The utility function returns 1 if the fidelity passed into the function is equal to $desired\_fidelity$, and 0 otherwise. Thus, the function will be maximized only when the input fidelity is equivalent to the desired fidelity.

However, this simple policy has a clear drawback: the assumption that all operations should be performed at the same fidelity, irrespective of the values of the input parameters. Figure 6.10 illustrates why this assumption is incorrect. Each line shows how Web browser energy use at a specific fidelity varies with an input parameter, the uncompressed image size. The highest fidelity, shown by the solid line, occurs when the image is shown without distillation. The remaining two lines show energy usage when the image is distilled to JPEG quality levels of 99 and 95. Data points show specific measured energy values for two image sizes: 3.5 MB, and 720 KB. For large images, the highest fidelity uses more energy since it transmits the most data over the wireless network. However, for small images, the lower fidelities use more energy because image distillation introduces a significant latency which results in the client waiting longer for the image to arrive.



This figure shows how Web browser energy usage varies with fidelity and the size of the uncompressed image. The three lines show how energy usage changes with image size for three different fidelities; the data points represent measured energy usage for two specific image sizes.

Figure 6.10: Web energy use as a function of fidelity and image size

Now consider the following example scenario. Originally, the application can show all images at the highest fidelity and still meet the specified goal for battery lifetime. However, when the user extends the goal, the system must adapt to reduce energy usage by a small amount (5%). If the system lowers fidelity to JPEG-99, it saves energy if a large (3.5 MB) image is displayed, but uses more energy if a small (720 KB) image is displayed. The system could possibly select the JPEG-95 fidelity, which saves energy for both sizes. However, this needlessly degrades large images—JPEG-99 is sufficient to reduce energy usage 5%. Also, for images smaller than 500 KB, JPEG-95 uses more energy than the highest fidelity.

The above scenario shows that it is often impossible to select a fidelity which produces the needed change in energy usage across all possible input parameters. Thus, it is infeasible for goal-directed adaptation to adjust application fidelity directly; a fidelity choice that is correct for one set of input parameters may not be correct for another. To surmount this problem, Odyssey introduces a layer of indirection in the form of a global feedback parameter, c, that represents how critical energy conservation is at the current moment. The value of the parameter ranges from 0, when energy usage is unimportant, to 1, when it is of utmost importance. When Odyssey wishes to adjust application behavior, it changes the value of c, rather than adjust application fidelity directly.

```
extern double c;
extern fidelity min_fidelity, max_fidelity;

int utility (fidelity f)
{
    if ((energy(f) / energy(max_fidelity)) <= (1-c)) {
        return (goodness(f));
    } else if (f == min_fidelity) {
        return (0.01); /* Small value */
    { else {
        return (0);
    }
}
```

Figure 6.11: Utility function for history-based policy

The utility functions of energy-aware applications use the value of c to decide the correct tradeoff between fidelity and energy use. For example, one could ask each application to select the best possible fidelity which uses only $(1 - c)$ as much energy as the highest fidelity. When the value of c is 0.1, each application would reduce its energy usage by at least 10%. Given a specific fidelity value, the utility function for this policy, shown in Figure 6.11, uses the history of application energy usage to predict the amount of energy needed to perform the operation at the specified fidelity. It compares that value to the energy needed to perform the same operation at the highest possible fidelity. If the ratio of the two energy values is higher than desired, i.e. greater than $(1 - c)$, the function returns zero, signifying that the input fidelity should not be chosen. When the ratio between the two values is less than $(1 - c)$, Odyssey returns a numerical value corresponding to the goodness of the fidelity. Thus, the utility function selects the best fidelity supported by the application that meets the goal for energy reduction. In the event that no fidelity provides the desired energy reduction, the utility function returns a small positive value for the fidelity that conserves the most energy, ensuring that it will be selected.

Alternative policies are also possible. For example, one could select the fidelity which yields the greatest "bang-for-the-buck", i.e. the one which yields the greatest decrease in energy usage for the smallest decrease in fidelity. In all such policies, the history of application energy usage plays a vital role. Odyssey can correctly evaluate such utility functions only by anticipating how much energy an operation will consume at various fidelity levels.

An important consideration is whether Odyssey should represent the importance of energy conservation with a single, global feedback parameter or with one parameter for each currently running application. Multiple feedback parameters would allow Odyssey to optimize energy usage across applications. For example, one application may be able to conserve a large amount of energy with a small decrease in fidelity, and thereby avoid the need for other applications to adapt. However, the optimization problem is very complex—Odyssey would need to search the space of all possible fidelities for all applications simultaneously, while possibly considering other important factors such as user-specified application priorities.

I have therefore chosen to simplify the problem by using only a single, global feedback parameter. This allows Odyssey to implement "fair" adaptation policies, for example, asking each application to decrease energy usage by a fixed percentage. User-specified priorities can be easily incorporated. For instance, one could use an approach similar to lottery scheduling [95] to prioritize applications; if one application has twice as many tickets as another, it would be asked to reduce its energy usage only half as much as the other.

6.2.5 Using application resource history to improve agility

Odyssey uses the history of application energy usage to improve the agility of goal-directed adaptation. During normal operation, predicted energy demand remains close to the available supply; it stays in a zone of hysteresis with a value no greater than the energy supply and no lower than the 95% of the current supply minus 1% of the initial energy supply. When demand strays significantly from supply, Odyssey changes the value of the global feedback parameter, c. When Odyssey increases c, application behavior is biased towards energy conservation; when Odyssey decreases c, applications execute operations with higher fidelity.

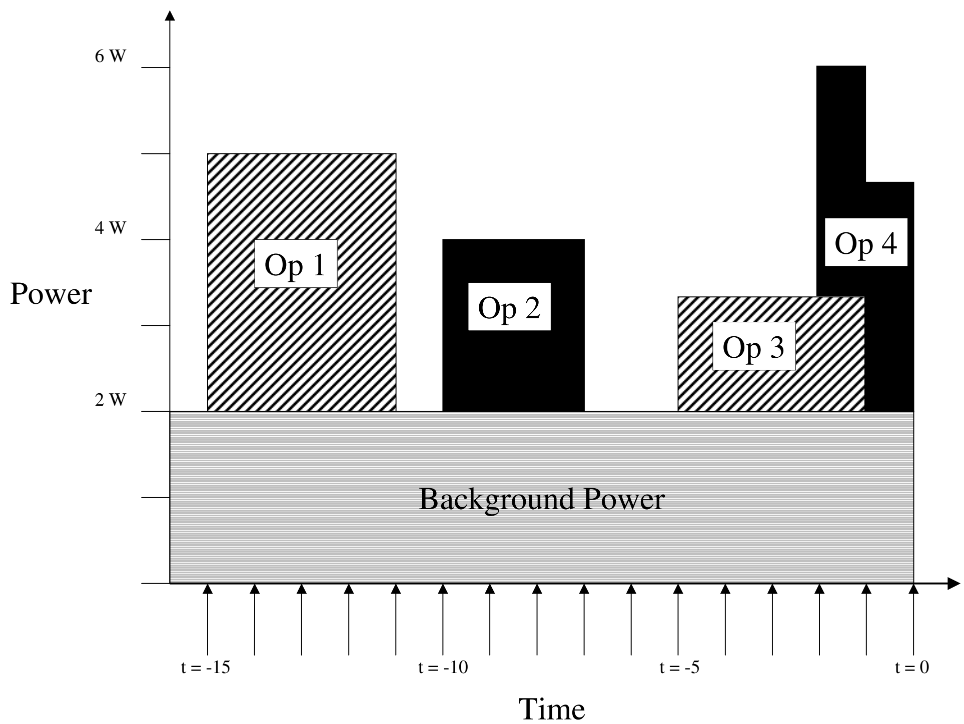
Without application energy history, it is difficult for Odyssey to determine the amount by which it should adjust c. The incremental approach, as described in Section 6.1.2, adjusts c by a small, fixed amount, then pause to assess if the resulting change in power demand is sufficient. If many small steps are needed, the system is slow to react to changes in power supply and demand.

However, when application history is available, Odyssey adapts more agilely. Odyssey first calculates the current power demand as described in Section 6.1.2. It then calculates the ideal power demand, one that falls in the middle of the zone of hysteresis. If S is the current energy supply, t the time remaining to the goal, and S_i the initial energy supply, the ideal power demand, P_{ideal} is:

$$P_{ideal} = (S + (0.95\ S - 0.01\ S_i))/2t \tag{6.3}$$

In principle, Odyssey simply predicts the ideal value of c, c_{ideal}, that will produce a future power drain of P_{ideal}. Then, Odyssey sets the value of c directly to c_{ideal} without needing to traverse through intermediate values.

Unfortunately, Odyssey cannot calculate c_{ideal} directly. Utility functions return the desirability of each fidelity given a specific value of c. There exists no inverse function which reveals what value of c would yield a specified fidelity (and, hence, a specified power drain). Because utility functions are application-specified, they are, in effect, black boxes, making the construction of the inverse function prohibitively difficult. Additionally, when diverse



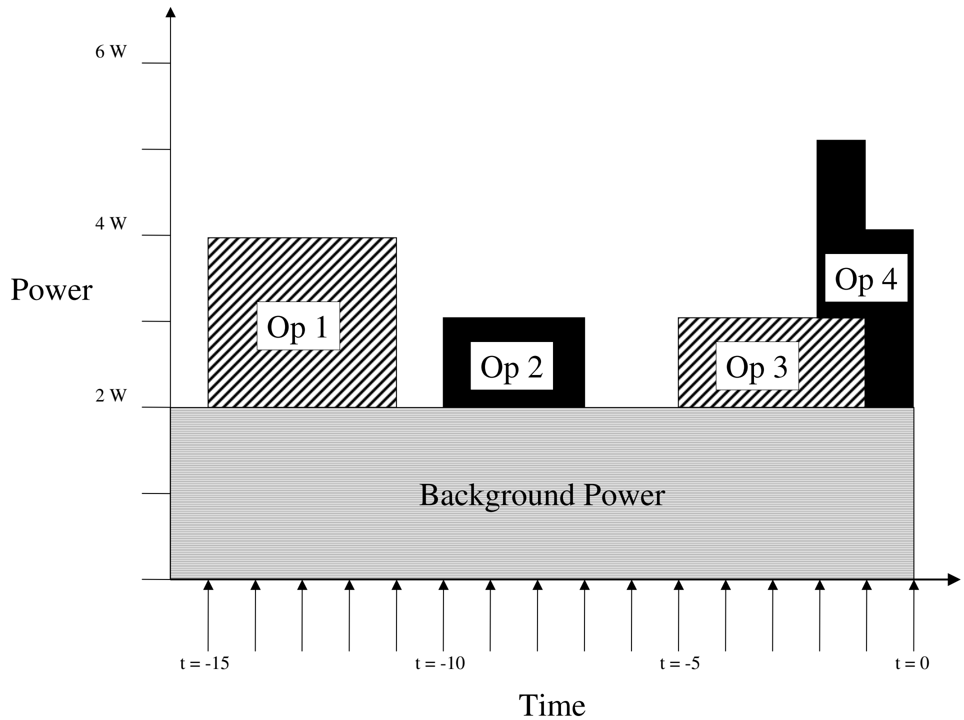
This figure shows a hypothetical example of how Odyssey calculates the impact of the choice of the feedback parameter, c, on power demand. Odyssey calculates the dynamic energy usage of each operation performed in the recent past—this is represented by the boxes labeled "Op 1" through "Op 4". Odyssey adds each operation's dynamic power usage to the background power usage of the hardware platform to generate a waveform that predicts how power usage would have varied over time, given a specific value of c (0.1 in this case).

Figure 6.12: Example of operation history replay for $c = 0.1$

operations are being performed by multiple applications, the utility functions for each operation may return different fidelities depending upon operation type and the value of the input parameters.

Odyssey takes a more indirect route to calculate c_{ideal}. It assumes that the recent past predicts the future, and asks the following question: *What (ideal) value of c would have yielded the ideal power usage in the past?*.

Odyssey maintains an in-memory list of operations that have executed recently. Each list item records the variables previously used to determine operation fidelity, including input parameters and resource availability. Odyssey can therefore revisit past fidelity decisions, and calculate what its decision would have been if conditions had been different. Specifically, Odyssey can determine what fidelity it would have chosen for various values of c.



This figure continues the hypothetical example of how Odyssey calculates the impact of the choice of the feedback parameter, c, on power demand. It shows predicted power demand for c = 0.2.

Figure 6.13: Example of operation history replay for c = 0.2

To calculate c_{ideal}, Odyssey *replays* the operation log to determine how power demand varies with c. For a given value of c, Odyssey first determines the fidelity at which each past operation would have executed. It then predicts the amount of dynamic energy each operation would have used. It uses the history of application energy usage to predict the total energy that would have been used to perform an operation. From this value, it calculates dynamic energy usage by subtracting out background power usage, i.e. the product of the background power usage of the mobile computer and the execution latency of the operation.

Odyssey then constructs a hypothetical waveform that captures its prediction of how the mobile computer's power usage would have changed over time for the specified value of c. The waveform reflects the background power usage of the machine and the predicted dynamic energy usage of each operation performed in the past. Finally, Odyssey calculates power demand by applying the exponential smoothing function shown in Equation 6.1 to the waveform.

Figure 6.12 shows a hypothetical example that helps illustrate this process. Here, Odyssey predicts power demand when c is equal to 0.1. There are four recently executed operations—consider operation 1 which executed for 4 time units starting at time -15. Odyssey first determines the fidelity at which operation 1 would have executed if c were equal to 0.1. It then predicts the operation's dynamic energy usage by consulting the history of application energy usage. In this example, Odyssey predicts that operation 1's dynamic energy usage would be 12 Watts. It also performs similar calculations for the other three operations.

Odyssey then generates the waveform that represents predicted power usage over time. Total power usage at any given time is the sum of the background power usage of the hardware platform and the dynamic power usage of each currently executing operation. As shown in Figure 6.12, Odyssey makes the simplifying assumption that operation power usage is constant over the life of an operation. In this figure, the total power usage waveform is given by the area under all shaded regions.

Odyssey determines what its estimate for power demand would have been if it had seen power usage equivalent to the generated waveform. It starts at the beginning of the waveform and calculates the exponential weighted average of power demand using Equation 6.1. In the given example, Odyssey would predict power demand to be about 3.81 Watts. This entire calculation can be thought of as solving a function, $DP(c)$ which maps c to power demand.

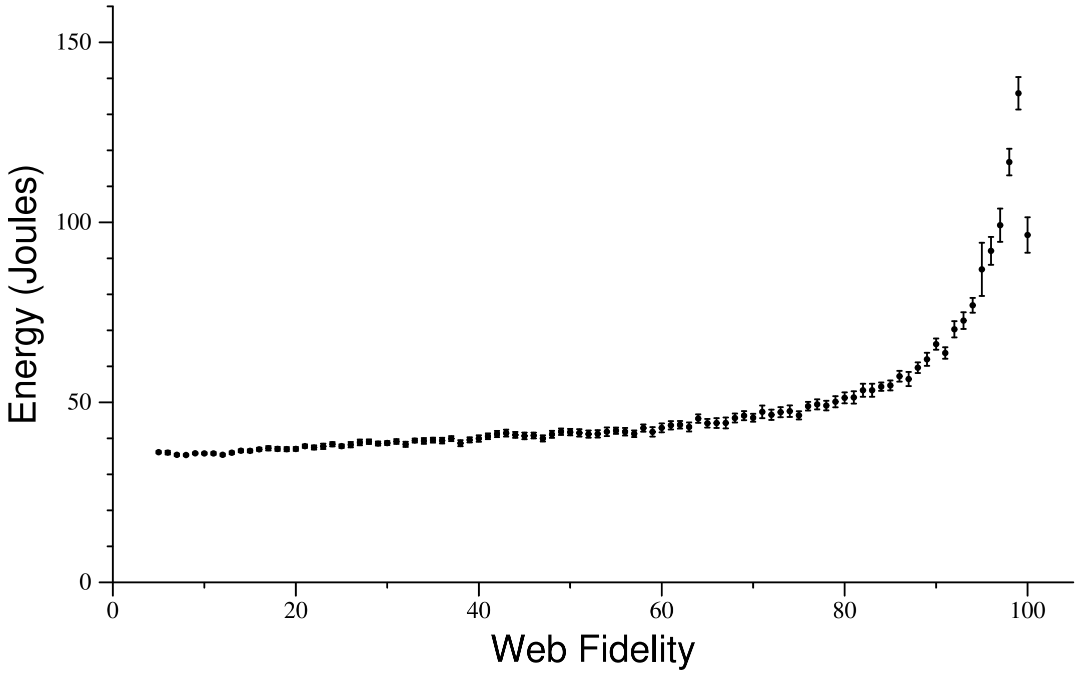
Figure 6.13 shows the same calculation for a different value of c. When c increases to 0.2, Odyssey selects lower fidelity levels for each operation. Consequently, the dynamic energy usage of each operation is lower than in Figure 6.12. For $c = 0.2$, Odyssey would predict power demand to be about 3.18 Watts.

Odyssey first determines the impact of the current value of c: $DP(c_{current})$. It then performs a binary search on c to determine c_{ideal}; i.e. the value of c_{ideal} such that $DP(c_{ideal}) - DP(c_{current})$ is closest to $P_{ideal} - P_{current}$. When this value is found, Odyssey sets the new value of c to be c_{ideal}, and changes its current estimate of power usage by $DP(c_{ideal}) - DP(c_{current})$.

In the example given by Figures 6.12 and 6.13, if $c_{current}$ were 0.1, and Odyssey needed to reduce power demand by 0.63 Watts, then c_{ideal} would be 0.2 since $DP(0.1) - DP(0.2) = 0.63$ Watts.

Why does Odyssey use each operation's dynamic power usage to calculate c_{ideal}? As shown in Section 4.8, when multiple applications execute concurrently, dynamic power usage is a much better metric than total power usage for calculating each application's impact. Thus, dynamic power usage is a better metric for assessing the impact of changing c when operations execute concurrently. Figures 6.12 and 6.13 illustrate this possibility since the execution of operations 3 and 4 overlap.

The use of dynamic power as a metric makes one important simplification: it assumes that changing fidelity will not affect when subsequent operations are started. For example, when using a speech recognizer, a user may pause a fixed amount of time between utterances. Lowering fidelity may speed recognition and allow the user to start the next recognition sooner. Predicting such effects is difficult because the system must anticipate



This figure shows how energy varies with fidelity level for the Web browser described in Section 4.7. Fidelities 5–99 show the energy used by the client to wait for a remote server to distill a 720 KB GIF image to the corresponding JPEG quality level, fetch the image from the server, and display it to the user. Fidelity 100 shows the energy used to fetch and display the image without distillation. Each data point shows the average amount of energy used; the error bars are 90% confidence intervals.

Figure 6.14: Energy use as a function of fidelity for the Web browser

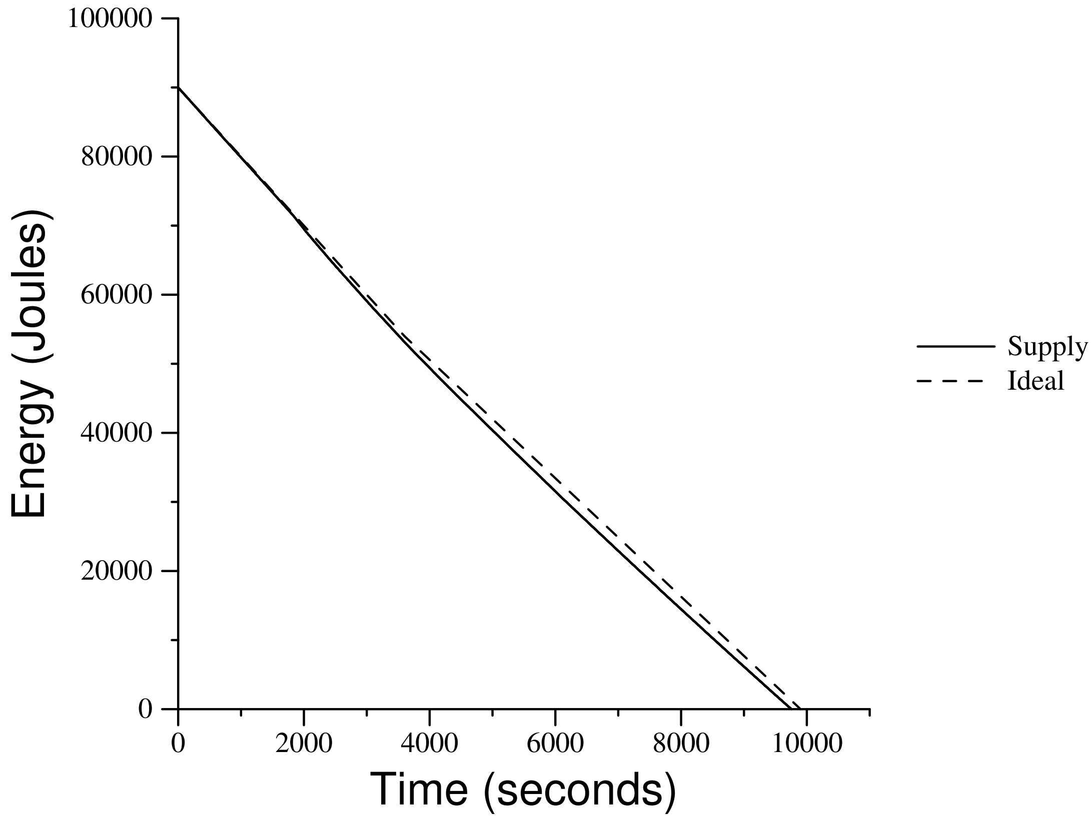
the usage pattern of the application; I have therefore chosen to ignore these effects in calculating c_{ideal}. Errors due to this simplification will eventually be detected by the feedback mechanism of goal-directed adaptation. However, they may cause Odyssey to converge more slowly on the correct value of c.

6.2.6 Validation

Experimental design

I validated the benefit of application resource history by executing two experimental scenarios: one which uses the incremental adaptation policy described in Section 6.1.2, and one which uses the history-based policy described in Section 6.2.5. Each scenario models the actions of a user browsing digital photographs from an album stored on a remote server. The user loads 5 images per minute—each image is approximately 720 KB in size.

Odyssey and the Web browser execute on the client: an IBM 560X laptop computer. The images are stored on a local server running the Odyssey distiller and an Apache Web



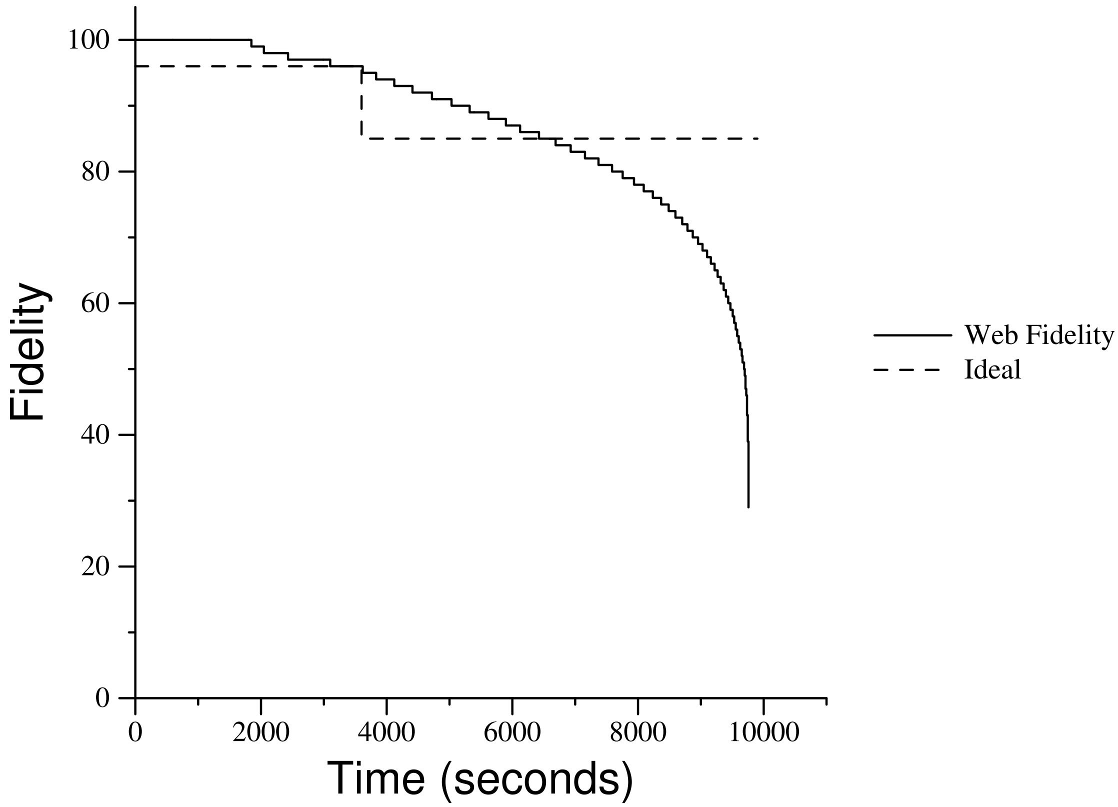
This figure shows how the supply of energy changes over time when Odyssey uses the incremental adaptation policy. The solid line shows the actual change in energy usage—the dashed line shows the hypothetical change that would be achieved by an ideal adaptive policy.

Figure 6.15: Change in energy supply for the incremental policy

server. The two machines are connected by a 900 MHz 2 Mb/s Lucent WaveLAN wireless network.

The Web application performs one type of operation: fetching an image from the server. JPEG quality is the sole dimension of fidelity for the operation. It is specified as a discrete dimension ranging from 5 to 100. There are 96 discrete fidelities, making this a challenging scenario for the incremental adaptation policy. Fidelity 100 corresponds to loading an undistilled image—the remaining fidelities correspond to distilling the image to the equivalent JPEG quality level before fetching it from the remote server. The image size is the sole input parameter for the operation. However, image size is not relevant in these experiments since all images are of roughly the same size.

I first obtained a detailed history of energy usage by performing 4000 image loads from a set of similarly-sized digital photographs. Figure 6.14 summarizes the logged data for operation energy usage. At any given fidelity, energy use is fairly predictable, with higher quality levels tending to use more energy than lower ones. There is a discontinuity in the

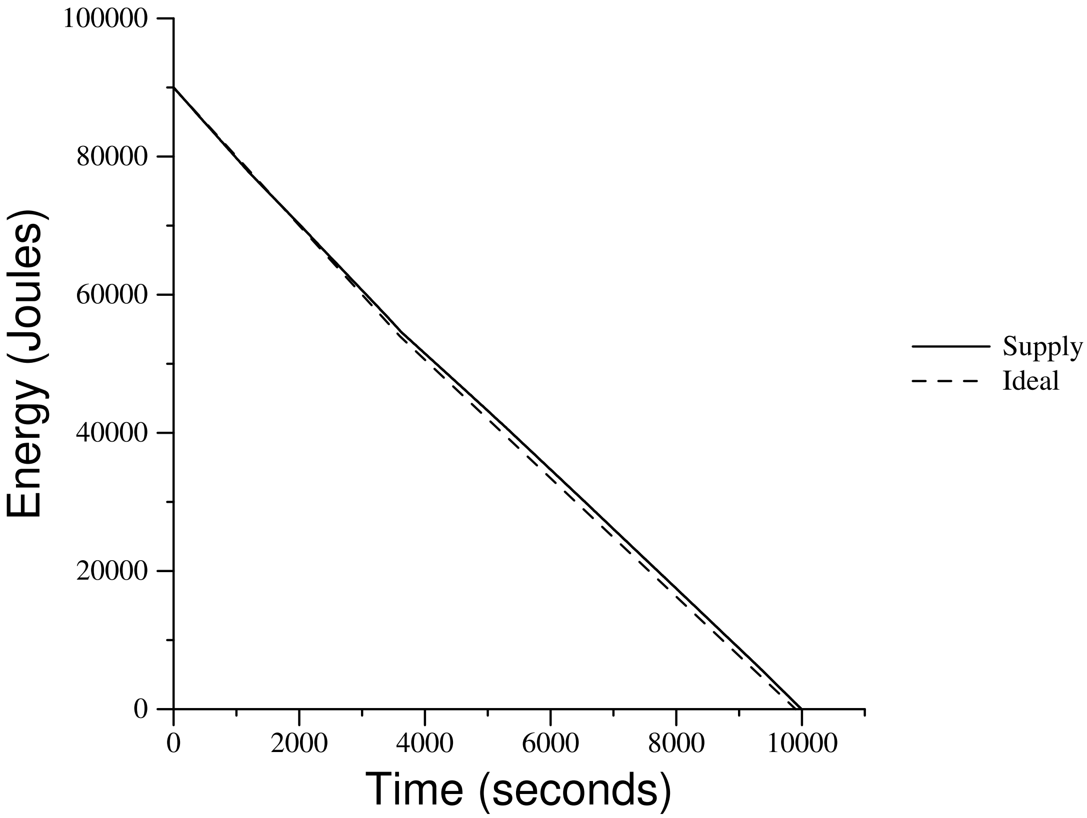


This figure shows how Web fidelity changes over time when Odyssey uses the incremental adaptation policy. The solid line shows actual Web fidelity levels—the dashed line shows the hypothetical fidelity levels that would be selected by an ideal adaptive policy.

Figure 6.16: Change in fidelity for the incremental policy

data: fidelity 100 uses less energy than fidelities 97–99. The reduction in data transmitted over the network is small at fidelities 97–99. The energy savings due to reduced network usage is less than the additional energy the client expends waiting for the server to distill the image. At this image size, fidelities 97–99 are clearly inferior to fidelity 100 because they expend more energy to fetch a lower-quality image. Therefore, a correct adaptation policy should not employ these fidelities.

After gathering resource history data, I executed five trials for each adaptation policy. I used a modulated energy supply and specified an initial energy value of 90,000 Joules, roughly equivalent to the amount of energy in the ThinkPad's battery. I specified an initial time goal of 2.5 hours. After 1 hour of execution, I modified the time goal by specifying that the battery should last an additional 15 minutes.



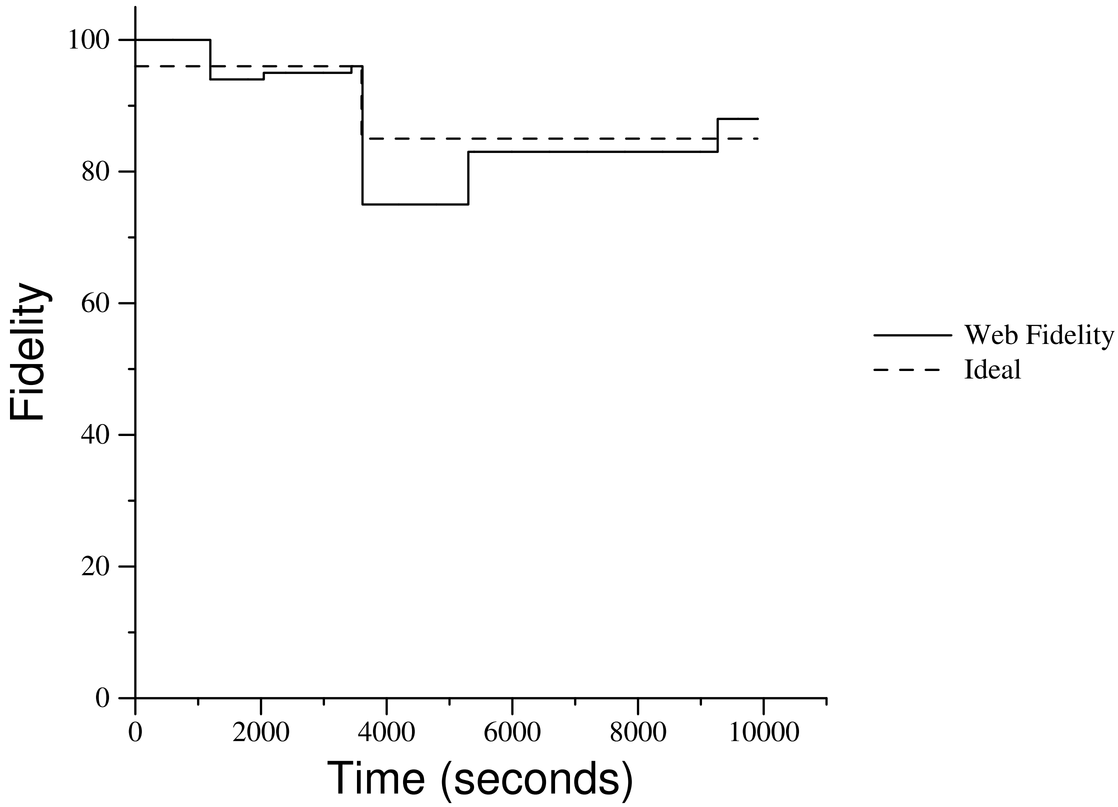
This figure shows how the supply of energy changes over time when Odyssey uses the history-based adaptation policy. The solid line shows the actual change in energy usage—the dashed line shows the hypothetical change that would be achieved by an ideal adaptive policy.

Figure 6.17: Change in energy supply for the history-based policy

Results

Figures 6.15 and 6.16 show results from one typical trial using the incremental adaptation policy. The solid line in Figure 6.15 shows how energy supply changes over time; the dashed line shows the ideal rate of drain that would be achieved by a hypothetical, perfect adaptive strategy. In the trial, the rate of drain tracks the ideal fairly closely for the first hour, although energy is used at a slightly higher rate during this time period. Once the user requests an additional 15 minutes of battery life, the two lines diverge more noticeably. Odyssey uses too much energy after the adjustment and is never able to recover fully. The energy supply expires almost 3 minutes too early.

Figure 6.16 gives more insight into why Odyssey misses the time goal. The solid line shows the fidelity levels used by the Web application throughout the trial. The dashed line shows the fidelity levels that would be used by the ideal adaptive policy. The ideal policy uses knowledge of future activity to select the highest constant fidelity which will meet the



This figure shows how Web fidelity changes over time when Odyssey uses the history-based adaptation policy. The solid line shows actual Web fidelity levels—the dashed line shows the hypothetical fidelity levels that would be selected by an ideal adaptive policy.

Figure 6.18: Change in fidelity for the history-based policy

specified goal for battery life. The ideal policy does not have future knowledge of changes in the specified goal—hence, there is a change in fidelity at the 1 hour mark.

After the user changes the specified time goal at 3600 seconds, Web fidelity decreases exponentially. Initially, changes in fidelity are relatively infrequent because the goal is distant. As the goal nears, Odyssey is more agile and fidelity changes are more frequent. Web fidelity remains higher than ideal for a significant portion of time: from 3600 to 6435 seconds, causing the laptop to expend too much energy during this period. Even though Odyssey tries to compensate by lowering fidelity for the remainder of the trial, it is unable to conserve enough energy to meet the specified goal for battery duration.

It is also interesting to note that the incremental adaptive policy chooses Web fidelities 97–99 during the time period lasting from 1845 to 3113 seconds. Without application history, it cannot know that these fidelities are clearly inferior to fidelity 100. This accounts for the slight overconsumption of energy during the first hour of the trial.

Figures 6.17 and 6.18 show results from one typical trial using the history-based adaptation policy. As can be seen in Figure 6.17, the rate of drain of the energy supply tracks

Adaptation Policy	Goal Met?	Expiration Time (s)		Number of Adaptations	
Incremental	0%	+79.7	(9.7)	56.8	(11.2)
History-Based	100%	-146.6	(11.2)	7.8	(1.3)

This figure compares the effectiveness of adaptation with and without the use of application resource history. The first row shows results without application history; the second row shows results when history is used to guide adaptation. For each scenario, the second column shows the percentage of trials that met the specified time goal. The third column shows the average expiration time for the energy supply, relative to the specified time goal. The final column lists the average number of adaptations per trial. Five trials were performed for each scenario—standard deviations are given in parentheses.

Figure 6.19: Summary of the effectiveness of application resource history

the ideal quite closely. Odyssey maintains a slight surplus throughout the trial and meets the goal for battery duration. Once the goal is reached, the amount of residual energy remaining is small: approximately 1.0% of the initial energy supply.

Figure 6.18 shows the corresponding changes in application fidelity. For the first hour, Odyssey chooses fidelities that are close to ideal. It also avoids the clearly inferior fidelities from 97 to 99. When the user respecifies the goal, Odyssey immediately decreases fidelity. The initially chosen fidelity is slightly less than ideal, reflecting Odyssey's conservative bias. However, Odyssey soon detects the discrepancy and readjusts the fidelity to be closer to ideal. In contrast to the previous scenario, Odyssey is much more agile in converging upon an acceptable fidelity level. Using application history, Odyssey requires only two adaptations to reach a good fidelity choice—without application history, Odyssey is too sluggish, and fails to reach an acceptable fidelity.

Figure 6.19 summarizes the results of the five trials. Use of application resource history provides a clear benefit. All five trials that used resource history met the time goal—the energy supply lasted an average of 79.7 seconds longer than the specified duration. When the incremental policy was used, the energy supply expired too early in all trials—on average, the supply expired 146.6 seconds before the time goal. Further, the average number of adaptations is much smaller with the history-based policy because Odyssey converges much more quickly upon acceptable fidelity levels.

The use of application history incurs some additional overhead, especially when Odyssey determines a new value for the feedback parameter c. During the five trials, Odyssey spent an average of 0.83 seconds to calculate a new value for c. However, since this calculation occurs relatively infrequently, an average of 8 times per 3.25 hour trial, the effect on overall performance is minimal.

It is worth noting that this scenario represents a difficult challenge for the incremental policy because the Web application has a large number of fidelities and the difference in energy usage between two consecutive fidelity levels is usually very small. One can potentially avoid the problems encountered in this scenario if one carefully chooses the number of potential fidelity levels for the application. For example, in the experiment reported in Section 6.1.3, I specified only five potential fidelities for the Web application. In addition, none of the five fidelities were clearly inferior to any other fidelity. Thus, with careful tuning and a bit of black magic, it is possible to make the incremental policy approach the performance of the history-based policy. Conversely, one of the main benefits of the history-based policy is that it is self-tuning—it requires no special adjustments in order to yield acceptable results.

6.3 Summary

In this chapter, I have shown how goal-directed adaptation allows Odyssey to effectively manage battery energy as a resource. The user provides hints that specify how long a mobile computer needs to operate on battery power, and Odyssey meets the specified goals by adjusting the fidelity of energy-aware applications.

Odyssey monitors energy supply and demand to determine the current importance of energy conservation. It predicts future energy demand from measurements of past usage. When there is substantial mismatch between predicted demand and available energy, Odyssey notifies applications to adapt. Using this approach, I have demonstrated that Odyssey can extend battery life to meet user-specified goals that vary by up to 30%.

I have also shown how the use of application resource history improves the effectiveness of goal-directed adaptation. When energy-aware applications execute, Odyssey monitors and logs their energy use. From the logged data, it learns functions which relate fidelity and input parameters to energy usage, allowing it to predict future application energy consumption. With these predictions, Odyssey can support a wider range of adaptation policies, including those which directly relate energy and fidelity. Further, Odyssey reacts more agilely to changes in power demand, adjusting application fidelity to appropriate levels without the need to traverse through intermediate values.

Chapter 7

Remote execution

Whenever there exists the possibility of trading some dimension of quality or performance for reduced energy usage, energy-aware adaptation is possible. While application fidelity is an important dimension of energy-aware adaptation, it is by no means the only one.

Remote execution is another such dimension. If applications execute part of their functionality on remote infrastructure, they may reduce their use of local hardware resources and conserve energy. However, energy reduction can come at a price. Remote execution can lead to decreased performance, primarily due to the need to transfer data between local and remote machines. In such cases, energy-aware adaptation is necessary to balance the competing concerns of energy use and performance.

This chapter describes Spectra, a remote execution system designed for battery-powered mobile clients used in pervasive computing environments. Spectra enables applications to combine the mobility of small devices with the greater resources of large, wall-powered compute servers. Spectra provides applications with a mechanism for executing remote operations, while ensuring data consistency between local and remote machines. It monitors both application resource usage and the availability of resources in the environment in order to provide dynamic advice to applications about how and where they can most profitably execute functionality. Spectra is tightly integrated with Odyssey and leverages Odyssey's support for energy-aware adaptation. Its advice lets applications dynamically balance the competing goals of performance, energy use, and application fidelity.

The next two sections discuss Spectra's target environment and design considerations. Section 7.3 describes Spectra's implementation, and Section 7.4 presents an evaluation of the system.

7.1 Target environment

Spectra is targeted at pervasive computing environments—such environments are saturated with computing and communication capability, yet gracefully integrated with human users. The need for graceful integration leads to smaller, less obtrusive computing devices. Since people are naturally mobile, these devices are often battery powered, and their size limits the amount of energy that can be stored. In addition, device size constrains the availability of other resources such as compute power and storage capacity. Remote execution using wireless networks to access compute servers fills a natural role in pervasive computing, allowing applications to combine the mobility of small devices with the greater resources of static, wall-powered compute servers

Future support for mobile applications will vary widely with location. Some well-conditioned environments may provide plentiful wireless bandwidth and powerful compute servers. Other locations may be resource-impoverished, with poor connectivity and little infrastructure support. Pervasive applications must therefore adapt to a changing environment. When little infrastructure exists, they must execute most functionality on the mobile client. However, when the environment is well-conditioned, they should discover and use the resources offered.

A broad range of resource-hungry applications such as speech recognition, natural language translation, and augmented reality will be important in pervasive computing environments. These applications may be executed on a wide range of mobile hardware, from powerful laptops to wristwatch-sized devices. While customizing these applications for each platform is possible, it would be more desirable to create a single implementation that adjusts its behavior to match the platform.

Variation in the environment and in client capability makes it quite difficult for developers to statically determine which components of an application should be executed remotely. Instead, location decisions should be made *dynamically* when applications execute.

Spectra simplifies the task of developing applications for pervasive computing. Applications statically specify which code components might *possibly* profit from remote execution. When applications execute, Spectra continually monitors resource supply and demand to advise them how and where they should execute the specified components. Spectra's advice lets applications adapt to changes in resource availability without requiring them to explicitly specify their resource requirements. Instead, Spectra monitors application behavior, models their resource usage, and predicts future resource needs.

Spectra targets applications which perform remote operations of one second or more in duration. Examples of such applications include speech recognition, rendering for augmented reality, and document preparation. Location decisions for these types of applications significantly impact performance and energy consumption. Thus, it is often beneficial to invest some effort to ensure that the correct decision is made. Applications which perform remote operations of shorter duration, for example, tens of milliseconds, will not benefit from Spectra since system overhead will negate the performance and energy benefits achieved by making correct location decisions.

7.2 Design considerations

Pervasive computing introduces several unique challenges for remote execution. In this section, I discuss these challenges and how they are addressed in the design of Spectra.

7.2.1 Competing goals for functionality placement

In pervasive computing environments, a remote execution system must reconcile multiple, possibly contradictory goals. Performance remains important, but is no longer the sole consideration. It is also vital to conserve energy so as to prolong battery lifetime. Quality must also be considered explicitly, since a resource-poor mobile device may only be able to provide a low fidelity version of a data object or computation, while a stationary machine may be able to generate a better version. These goals will often conflict—for example, executing a code component on a remote server might offer reduced client energy usage at the cost of increased execution time.

Spectra explicitly considers these competing goals when advising applications where to place functionality. For each alternative placement, it predicts application performance, energy use, and quality. It then balances competing goals when selecting from the alternatives using the system support for energy-aware adaptation discussed in the previous chapter. I have therefore chosen to tightly integrate Spectra and Odyssey. One can view Spectra as an extension to Odyssey that supports remote execution as a second dimension of energy-aware adaptation.

Together, Spectra and Odyssey advise applications how and where they should execute functionality. By simultaneously considering both execution location and application fidelity, they can make better choices than if they considered location and fidelity independently.

7.2.2 Variation in resource availability

Pervasive computing is characterized by tremendous variation in resource availability. Network characteristics and remote infrastructure available for hosting computation vary with location. File cache state and CPU load on local and remote machines significantly impact application performance. Application energy consumption varies depending upon the specific platform on which the application executes. Thus, variation in any resource can significantly change the optimal placement of functionality. A remote execution system must continually monitor resource availability and adapt to changes in the environment.

Spectra includes a set of *resource monitors* that measure local and remote resource availability. Each monitor measures a single resource or set of related resources—for example, the network monitor reports available bandwidth and latency to remote servers. The monitors are implemented in a modular framework, which enables developers to easily add measurement capability for new resources.

7.2.3 Self-tuning operation

To predict the time and energy needed to execute a code component, the remote execution system must match resource availability with the resource demands of the component. For example, the time needed to transfer data over a network can be roughly predicted by dividing the amount of data that the application will transfer by the available network bandwidth. Thus, a remote execution system must have a model of application resource demand in order to make correct placement decisions.

One possible approach would be to require applications to explicitly specify their resource requirements. However, this seems infeasible for most applications. Many resources are important in pervasive computing, and an application would have to specify models for each. Also, models for some resources are platform specific—application energy usage, for instance, depends upon hardware characteristics. The burden of specifying such models seems too high; few developers would be willing to invest the necessary effort.

Spectra takes an alternate, *self-tuning*, approach. It observes applications execute, measures their resource usage, and generates models of resource consumption. It then uses the models to predict future demand.

7.2.4 Modification to application source code

Systems such as Coign [32] provide remote execution services without source code modification by exploiting externally visible object interfaces. This approach is attractive because it supports closed-source applications and requires little development effort. However, a little application-specific knowledge can go a long way.

Spectra asks developers to specify possible methods of partitioning applications. Often there will be only a few useful partitions. While developers will often have an intuitive notion of which partitions are useful, it may prove difficult to automatically extract such partitions from source code. First, the best partitions may not occur along object boundaries. Second, since the number of potential partitions is exponential with the number of objects, selecting partitions for large applications may be computationally intractable. Finally, it will be impossible to support legacy applications which do not provide externally visible object interfaces.

Application-specific knowledge can also be used to improve models of resource demand. Often resource usage will depend upon a few application-specific parameters—for example, the amount of CPU cycles required to translate a sentence from Spanish to English often depends upon the length of the sentence to be translated. Spectra provides an interface that allows applications to specify the operations they perform and the important parameters that affect operation complexity. Spectra models consider the effect of these parameters, leading to more accurate predictions.

I have tried to situate Spectra in a sweet spot of the design space. In exchange for a minimal amount of source code modification, Spectra provides significantly better advice about how and where to execute functionality than could be provided by a system that requires no application modification.

The benefits provided by Spectra increase with the amount of effort that developers invest. For applications without an interactive interface such as a compiler or Latex, Spectra provides considerable benefit without source code modification. Spectra can execute such applications entirely on a remote server or on the client. Spectra learns application resource requirements and chooses the best location for execution. With some source code modification, Spectra provides additional benefits. If one identifies specific code components that may benefit from remote execution and inserts Spectra calls in the application, Spectra chooses the best partitioning of functionality between local and remote machines. If one then adds the ability to execute these components at multiple fidelities, Spectra also chooses the best fidelity for each component. Finally, one can specify input parameters for each operation and application-specific prediction algorithms that improve the accuracy of Spectra's decisions. Thus, developers can choose the right amount of source code modification for each application.

7.2.5 Granularity of remote execution

A key issue in the design of a remote execution system is the granularity of code that can be executed remotely. Fine-grained remote execution yields increased flexibility since it creates more opportunities to locate functionality on remote servers. However, coarse-grained remote execution may lead to better performance since it often reduces the volume of network transmission and amortizes the overhead of deciding where to locate functionality over a larger unit of execution.

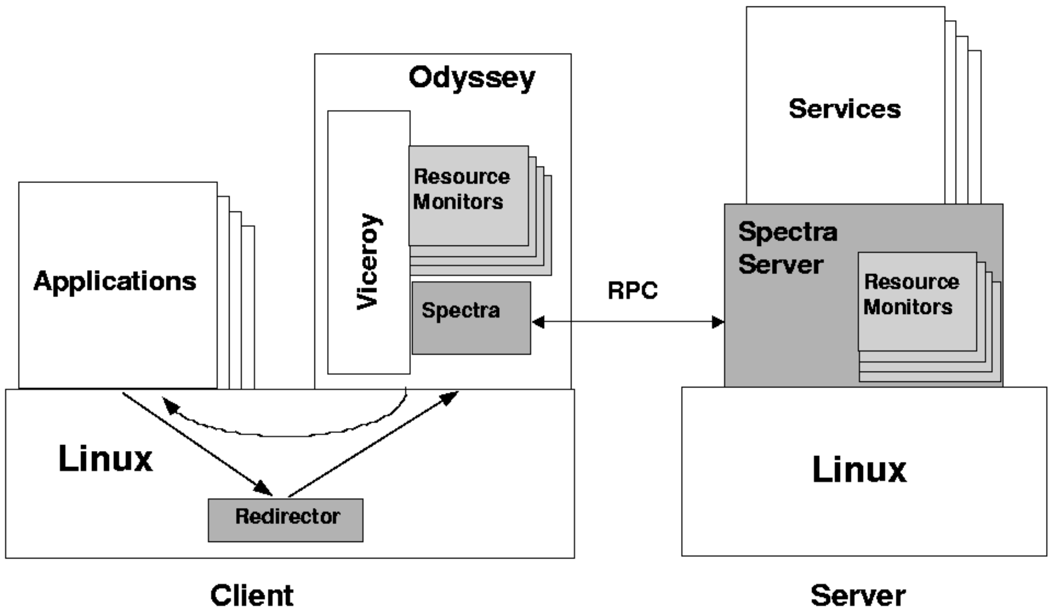
Since Spectra is designed to make intelligent placement decisions that balance competing goals and adjust to changes in resource availability, its decision overhead is non-negligible. Therefore, Spectra targets applications that perform relatively coarse-grained operations—currently one second or more in duration. Examples of such applications include speech recognition and language translation. Applications that perform shorter remote operations, i.e. a few milliseconds in duration, will not benefit from Spectra since system overhead will negate the performance and energy benefits achieved by making correct location decisions.

7.2.6 Support for remote file access

Many of Spectra's target applications access a large amount of file data. For remote execution to yield correct results, file operations performed on remote servers must produce the same results that would be produced if the operation were performed on the client. Systems such as Butler [67] solve this problem by using a distributed file system that presents a consistent file system image across all machines on which functionality may be executed.

Unfortunately, file consistency comes at significant cost in pervasive computing environments. Network connections to file servers often exhibit high latency, low bandwidth, or intermittent connectivity. When an application modifies files, it will block for long periods of time waiting for data to be reintegrated to file servers.

This performance consideration led me to adopt the Coda file system [40]. Coda provides strong consistency when network conditions are good. Under low bandwidth conditions, Coda buffers file modifications on the client to improve performance. Buffered modifications are reintegrated to file servers in the background. Until a modification is



This figure shows the typical architecture for a Spectra client. The shaded components comprise the Spectra remote execution system. The Spectra client is tightly integrated with the Odyssey platform for mobile computing—the two run as a single process. The Spectra server runs as a separate process.

Figure 7.1: Spectra architecture

reintegrated, it is not visible on other machines.

Spectra interacts with Coda to provide the requisite amount of consistency for remote execution. Before it executes functionality remotely, Spectra predicts what files will be accessed by the application. If any such files have buffered modifications on the client, Spectra triggers their reintegration to file servers before executing the remote operation. Further, Spectra considers the performance impact of data consistency when deciding where to locate operations.

7.3 Implementation

7.3.1 Overview

Figure 7.1 shows Spectra's software architecture—shaded regions are Spectra components. Spectra leverages the functionality of both Coda and Odyssey. Spectra consists of a client, which executes on the same machine as the application, and a server, which executes on machines that may perform work on behalf of clients. It is common for a single machine to run both the client and server.

The tight integration between Spectra and Odyssey is reflected in Figure 7.1. The Spectra client and Odyssey execute as a single process. In addition, Spectra and Odyssey share a common kernel module that is used to intercept and redirect calls made by applications.

The remainder of this section provides a detailed description of Spectra's implementation. Section 7.3.2 describes the extensions to the Odyssey API needed to support remote execution. Section 7.3.3 outlines the Spectra architecture. In Section 7.3.4, I describe Spectra's resource measurement mechanism, and in Section 7.3.5, I show how these measurements are used to predict future application resource needs. Section 7.3.6 shows how Spectra ensures data consistency when operations are executed remotely. Section 7.3.7 discusses how Spectra decides where functionality should be executed. Along with Sections 7.3.4 and Section 7.3.5, the work reported in this section was performed jointly with Dushyanth Narayanan. Finally, Section 7.3.8 describes the applications that have been modified to use Spectra.

7.3.2 Application interface

Spectra's application interface is an extension of the Odyssey API described in Section 6.2. A Spectra application identifies code components that may possibly benefit from remote execution. It registers these components as operations, using the `register_fidelity_op` call described in Section 6.2.4. Thus, Spectra broadens the definition of an operation to include code components that may execute at one of possibly many locations, as well as code components that may execute at one of possibly many fidelities.

When a Spectra application registers an operation, it specifies a set of possible *execution plans* that represent different methods of partitioning functionality between local and remote machines. For example, the speech recognition application described in Section 4.5 has three possible execution plans: local, remote, and hybrid. When the local execution plan is chosen, the recognition is done entirely on the client. When the remote plan is chosen, the recognition is done entirely on a remote server. When the hybrid plan is chosen, the computation is split—the first phase is done locally, and the remainder is done on a remote server.

Prior to executing an operation, the application calls `begin_fidelity_op` to determine how and where the operation should execute. Odyssey and Spectra collaborate to choose the best fidelity and location for execution. Odyssey chooses the fidelity at which the operation will be executed, and Spectra chooses which execution plan will be used. When the execution plan involves a remote server, Spectra also chooses a particular server to use. In the speech example, Spectra chooses a remote server when it uses the hybrid or remote plan, but does not need to choose a server in the local case.

Spectra applications execute operations by making remote procedure calls (RPCs) to local and remote machines. Applications invoke remote services using the `do_remote_op` system call provided by Spectra. While applications could theoretically invoke remote functionality directly, use of this call allows Spectra to precisely measure the amount of local and remote resources used by the application. Additionally, Spectra acts as a central

```
# service type method location             working dir
speech    p    exec   /coda/re/bin/janus    /coda/re/speech
latex     p    exec   /coda/re/bin/latexs   .
test      t    exec   /usr/odyssey/testsrv  .
```

Figure 7.2: Sample Spectra server configuration file

repository for information about server and network status—if one application detects that a server is unavailable, Spectra can immediately inform all other applications executing operations on that server.

Spectra provides a similar `do_local_op` system call that lets applications invoke functionality on the local machine. The syntax of this call is identical to `do_remote_op`—this simplifies application development by letting local and remote functionality be invoked the same way. However, for performance reasons, an application may choose to eschew this call and invoke local functionality with a direct procedure call to avoid crossing protection boundaries.

An operation often consists of multiple RPCs. For example, when an execution plan is split between local and remote machines, the application uses both `do_local_op` and `do_remote_op` calls to perform the operation. Applications signal the end of operation execution by calling `end_fidelity_op`. Since the start and end of execution are marked, Spectra and Odyssey can precisely measure the resources consumed by the operation.

7.3.3 Architecture

Spectra clients maintain a database of servers willing to host computation. The database indicates whether each server is currently accessible and stores snapshots of recent status (CPU load, file cache state, etc.). Currently, potential servers are statically specified in a configuration file. I have designed Spectra so that it could also use a service discovery protocol [1, 90] to dynamically locate additional servers, but this feature is not yet supported.

Application code components executed on Spectra servers are referred to as *services*. A configuration file, such as the one shown in Figure 7.2, specifies the services that a server may execute. The first column on each line specifies a unique name for the service. The second column specifies whether the service is persistent or transient. Transient services exist only to service a single RPC; persistent services may exist for multiple RPCs and may maintain state between RPCs. Thus, a service which requires substantial initialization is usually persistent. For example, because the loading and initialization of a speech model is time-consuming, speech recognition is implemented as a persistent service.

The third column specifies the method of executing each service. Currently, all services run as separate processes in order to provide protection against malicious or faulty application code. When a new service is started, the server forks and execs a child process. The

```
service_init (&argc, &argv);
while (1) {
    if (service_getop (&optype, &opid, path,
                       &indata, &inlen) < 0) {
        return (-1);
    }

    rc = run_latex (indata, inlen, &outdata, &outlen);

    service_retop (opid, NULL, outdata, outlen);
}
```

Figure 7.3: Sample service implementation

Spectra server uses two Unix pipes to communicate with the service.

The last two columns in Figure 7.2 specify the location of the service executable and the working directory in which it should be executed. Normally, service executables are placed in the Coda file system. This simplifies deployment: any machine which is a Coda client will execute the latest version of a service. As the last line shows, it is also possible to execute services located on local file systems. However, the system maintainer is then responsible for ensuring consistency.

The Spectra server is multi-threaded, allowing it to accept multiple concurrent requests. Each RPC is handled by a separate thread, which routes the request to the appropriate service. When a transient request is received, the server thread dynamically creates a new service to handle the request. If a RPC requests a persistent service, the server thread determines if the specified service is currently running. If it is not running, the service can be created on demand.

A persistent service may choose to handle multiple requests simultaneously. When a server thread receives a RPC for a service, it assigns a unique identifier to the request and writes the request to the service's input pipe. When the service replies, it specifies the identifier of the request to which it is replying.

Spectra provides an application library which simplifies service implementation. Figure 7.3 shows the main loop of a simple service. The `service_init` library function parses the command line arguments and extracts Spectra-specific information, including the input and output pipes used to communicate with the Spectra server. In the main loop, the service calls `service_getop`, which blocks until a request is received. The function returns the type of operation requested, a unique identifier associated with the request, and application-specific input data. The sample service has only one request type—services that handle more than one request type multiplex on `optype`. When processing is com-

`init`	Initializes the monitor.
`term`	Called on termination.
`predict_avail`	Predicts the amount of resources available for operation execution.
`get_measured_cnt`	Returns the number of resources measured.
`start_op`	Starts measuring resource usage for an operation.
`stop_op`	Stops measuring resource usage. Returns the amount of resource consumed.
`add_usage`	Adjusts measurements to account for resources consumed on a Spectra server.
`update_predictions`	Refreshes the cached predictions of a proxy monitor.

Figure 7.4: Resource monitor functions

plete, the service calls `service_retop`, passing in the request identifier and application-specific output data.

7.3.4 Resource monitors

Spectra measures supply and demand for many different resources in order to make correct remote location decisions. This functionality is implemented as a set of *resource monitors*, code components that measure a single resource or a group of related resources. The resource monitors are contained within a modular framework shared by Spectra clients and servers. In pervasive computing environments, it is difficult to anticipate what resources might impact application execution—Spectra's measurement framework makes it easy to add new measurement capability. Further, the modular design allows one to implement several different methods of measuring a particular resource and choose the appropriate method for each particular execution environment.

Currently, Spectra's implementation includes six types of monitors: CPU, network, battery, file cache state, remote CPU, and remote file cache state. Each monitor provides the common set of functions shown in Figure 7.4. Prior to executing an operation, Spectra generates a *resource snapshot*, which provides a consistent view of the local and remote resources available for execution. Spectra first generates a list of servers that could possibly execute some portion of the operation; this list may include the local client machine. For each server in the list, Spectra iterates through the set of resource monitors, calling `predict_avail`. Each monitor returns predicted resource availability—for example, the network monitor returns predicted network bandwidth and latency for communicating with the specified server. The predictions in the snapshot are used to decide how and where the operation will execute.

During operation execution, the resource monitors observe application resource usage. Before executing an operation, Spectra calls `start_op`, which alerts each monitor to begin observation. After the operation completes, Spectra calls `stop_op`, which terminates measurement and returns the amount of resources consumed. The `add_usage` function accounts for resource usage on Spectra servers—the monitor adds reported resource consumption to the total resource usage of the operation.

Spectra logs measured resource consumption and creates models that predict future resource demand. Thus, the more an operation is executed, the more accurately its resource usage is predicted.

The CPU monitor

The CPU monitor predicts availability using a smoothed estimate of recent load. The monitor first determines the amount of competition for the local CPU by measuring the percentage of CPU cycles recently used by other processes. It then calculates the percentage of cycles available for operation execution by assuming that background load will remain unchanged and that the operation will get a fair share of the CPU. It multiplies this value by the processor speed to predict the number of CPU cycles per second the operation will receive. Narayanan et al. [64] describe this prediction algorithm in more detail.

The monitor observes CPU usage by associating an operation with the identifier of the executing process. This distinguishes the CPU usage of concurrent operations. Before and after execution, the monitor observes CPU statistics for the executing process and its children using Linux's `/proc` file system. It returns total cycles used by the operation.

The network monitor

The network monitor predicts network bandwidth and latency using an algorithm first developed for Odyssey [68]. Predictions are based upon passive observation of communication between the Spectra client and remote servers. The RPC package logs the sizes and elapsed times of short exchanges and bulk transfers. The short, small RPCs give an approximation of round trip time, while the long, large bulk transfers give an approximation of throughput. The network monitor periodically examines the recent transmission logs, and determines the instantaneous bandwidth available to the entire machine. It then estimates how much of that bandwidth is likely to be available for communicating with each server, assuming that the first hop is the bottleneck link in the network. When the network monitor is queried, it returns the predicted latency and bandwidth available for communicating with a specified server.

Observing network usage is trivial since all client-server communication passes through Spectra. Whenever an application performs a RPC, the network monitor records the number of bytes sent and received. After the operation completes, the monitor returns total bytes transmitted, total bytes received, and the number of RPCs performed.

The battery monitor

The implementation details of the battery monitor have already been described in Chapter 6. Each method of observing battery status and energy usage (Smart Battery, ACPI, and modulation) is implemented as a separate monitor. The modular design of the resource monitor framework makes it easy to select the appropriate measurement methodology when compiling for different hardware platforms.

When asked to predict resource availability, the battery monitor returns the amount of energy remaining in the client's battery (if the computer is currently battery-powered). It also returns the global feedback parameter, c, which represents the current importance of energy conservation. During operation execution, the monitor observes energy usage. After the operation completes, it returns the total amount of energy consumed. Since it is difficult to distinguish the energy usage of concurrent operations, Spectra ignores data gathered from concurrently executed operations when modeling demand and predicting future energy needs.

The file cache state monitor

File cache state is an important resource in pervasive environments because data access consumes significant time and energy when items are unavailable locally. The file cache state monitor estimates these costs by predicting which uncached objects will be accessed by an operation.

The file cache state monitor interacts with the Coda file system to predict cache state. Coda hides server access latency by caching files on clients. When the data accessed by an operation is cached locally, the operation will take less time and consume less energy. To predict this effect, the monitor asks Coda which files are in its cache. The monitor performs an `ioctl` on the Coda pseudo-device, which causes Coda to write a list of currently cached files to a temporary file. Although it is possible that cache state will change slightly during operation execution, the changes are unlikely to be significant. The monitor also obtains an estimate of the rate at which uncached data will be fetched from file servers.

During operation execution, the monitor observes Coda file accesses by listening to a special Unix port. Upon operation completion, the monitor returns the names and sizes of accessed files. This method does not yet support concurrent operations. However, a similar approach to that used by the CPU monitor could be used if Coda were to associate a process identifier with each file access. Section 7.3.6 describes how this list of file accesses is used to predict which files will be accessed during future operations.

The remote proxy monitors

Resource monitors on Spectra servers measure remote resource state. They communicate this information to *remote proxy monitors* running on Spectra clients. Currently, Spectra servers execute local CPU and file state monitors, which communicate with remote CPU and file cache state proxy monitors on clients.

Prior to operation execution, each proxy monitor predicts resource availability on servers that could be used for execution. The proxy monitors cache availability information to improve performance. If a server has recently reported availability, the proxy returns the cached value. Otherwise, the proxy queries the server dynamically. The staleness threshold is resource-specific—I currently use a one minute timeout for both CPU and file cache data.

Spectra periodically polls servers to determine if they are active. When a server sends a positive acknowledgment, it also returns its resource snapshot, which includes the availability of all measured resources. Upon receiving a snapshot, the Spectra client calls the `update_predictions` function of each proxy monitor. This process lets Spectra update the resource status of servers that are not in active use.

When Spectra executes a remote RPC, server monitors observe resource usage and report the total resource consumption as part of the RPC response. The Spectra client passes this data to proxy monitors by calling their `add_usage` functions. The proxy monitors accumulate server resource consumption and report the total when the operation completes.

7.3.5 Predicting resource demand

Spectra uses measurements of application resource usage to generate models that predict future demand. It assumes that the resources consumed by an operation will be similar to those consumed by recent operations of similar type.

Spectra provides default *predictors* that model resource demand. For numerical data such as CPU and energy consumption, Spectra's default predictor generates simple linear models of application behavior. The default file predictor is somewhat more complicated, and is described in the next section. We believe that Spectra's default predictors will prove sufficient for most applications; in fact, we use them for all current Spectra applications. However, we also recognize that some applications may require more complex predictors to generate accurate predictions. Spectra therefore provides an interface through which application-specific predictors can be specified.

When an application calls `register_fidelity`, Spectra creates predictors for each resource type. Each predictor reads the logged resource usage data and generates a parameterized model of demand that is stored in memory. When subsequent operations are performed, Spectra updates the in-memory model in addition to logging resource usage.

Each model predicts resource demand as a function of application fidelity and operation input parameters. Fidelities and input parameters may be either discrete or continuous. The default predictor uses binning to model discrete variables: it maintains a separate prediction for each possible discrete value. The default predictor also maintains a generic prediction that is independent of any discrete variable—this prediction is used whenever a specific combination of discrete variables has not yet been encountered. The default predictor uses linear regression to model continuous variables. It adjusts for changes in application behavior over time by giving more recent samples a greater weight in its predictions.

For some applications, resource usage depends heavily upon the specific data on which an operation is performed. For example, the input document to the Latex document preparation system will significantly affect resource usage: a 100 page document consumes more CPU cycles and battery energy than a 2 page document.

Spectra's default predictor anticipates this relationship with data-specific models of resource usage. Applications such as Latex associate each operation with the name of a data object. The default predictor maintains a LRU cache of the most recent data objects. When asked to predict future demand, the predictor uses a data-specific model to predict resource usage if it finds such a model in its cache. Otherwise, it uses the more general, data-independent model. The size of the LRU cache is a parameter specified by applications.

7.3.6 Ensuring data consistency

Since Coda relaxes data consistency under poor network conditions to achieve acceptable performance, Spectra must interact with Coda to ensure that remotely-executed operations read the same data that they would have read if they had been executed locally. This is vital for applications such as compilers and document processors that read files commonly modified on clients.

Prior to executing an operation, Spectra predicts which files are likely to be accessed. Spectra provides a default file predictor that builds upon the numerical predictor described in the previous section. In essence, the file predictor maintains a numerical prediction of access likelihood for each file that may be accessed by an operation. When updating each file's model, the file predictor assigns the value of 1 to a file access, and the value of 0 when a file is not accessed. Each resulting prediction thus represents the likelihood that a given file will be accessed.

Spectra uses the file predictor to estimate the cost of servicing cache misses. It compares the list of files that may be accessed by an operation to the list of cached files. For each uncached file, it estimates the number of bytes of data that must be fetched from file servers by multiplying the file size by the predicted access likelihood. Summing this value over all files that may be accessed yields a prediction for the total number of bytes that must be fetched to perform an operation. Spectra divides this prediction by the rate at which Coda will service cache misses to estimate the total amount of time that will be spent servicing cache misses.

Spectra uses the file predictor to maintain data consistency. Before executing an operation remotely, Spectra ensures that all modifications to files with non-zero access likelihood have been reintegrated to file servers. Spectra also ensures that modifications made during the remote execution of an operation are immediately visible to the client. The file cache state monitor provides a list of files accessed during an operation. The Spectra server reintegrates all local modifications to those files and delays its reply to the RPC until the reintegration completes.

If a large amount of data must be reintegrated in poor network conditions, then data consistency significantly increases remote operation execution time. Spectra estimates the added execution time before deciding where to execute an operation. It calculates the reintegration cost by multiplying the size of the modifications by the predicted bandwidth available to the file server. If the predicted reintegration costs are too high, Spectra avoids them by executing the operation locally.

7.3.7 Selecting the best option

When applications call `begin_fidelity_op`, Spectra selects a location and fidelity at which to perform the operation. It first determines which servers could possibly execute the operation. Then, Spectra and Odyssey poll the resource monitors to obtain a snapshot of resource availability on the client and potential remote servers. Finally, the Odyssey multi-fidelity solver, described in Section 6.2.4, is used to decide how and where to execute the operation. The server selects the alternative that maximizes an input utility function. Because it uses search heuristics, it is not guaranteed to select the optimal alternative—however, as the results in Section 7.4 show, it usually selects a very good option.

Spectra provides a default utility function that has so far proven sufficient for all Spectra applications. As with Spectra predictors, applications may override the default utility function with an application-specific implementation. The default utility function evaluates execution alternatives by their impact on *user metrics*. User metrics measure performance or quality perceptible to the end-user—they are thus distinct from resources, which are not directly observable by the user (other than by their effect on metrics). Spectra currently considers three user metrics: execution time, energy usage, and application fidelity.

A utility function represents a specific characterization of the desirability to the user of each possible alternative for execution. If the utility function poorly characterizes user preferences, then the alternative selected by Spectra may not be the most desirable (even when it maximizes the utility function). In my evaluation of Spectra, I assume that the utility function represents a good characterization of user preferences. A qualitative evaluation of the results in Section 7.4 suggests that the default utility function produces a reasonable approximation of user preferences—the choices made by Spectra appear to be correct.

As the solver searches the space of possible alternatives, it calls the utility function with specific input parameters, fidelities, execution plans, and server choices. To evaluate each alternative, the default utility function first predicts a context-independent value for each metric: total execution time, total energy usage, and a vector representing application fidelity. It then weights each value by how important it currently is to the user. It returns the product of the weighted values as the utility of the alternative.

The default utility function predicts execution time to be the sum of local and remote CPU time, network transmission time, time to service cache misses, and time to ensure data consistency. This simple model reflects Spectra's current implementation, which does not allow computation and network transmission to overlap.

The utility function uses the models of operation resource usage described in Sections 7.3.5 to predict resource demand. It matches demand to the prediction of resource availability contained in the resource snapshot. It calculates local CPU time by dividing the predicted cycles needed for execution by the predicted amount of cycles per second available on the local machine. Remote CPU time is calculated similarly. Network transmission time is calculated by predicting the number of bytes to be transmitted and dividing by the available bandwidth. Spectra then adds the affect of latency by multiplying the predicted number of round trips by the round-trip time. Spectra calculates the time to service file cache misses and ensure data consistency as described in the previous section.

The importance of execution time is application-specific. Therefore, Spectra requires each application to provide a function that expresses the desirability of different latency values. Many Spectra applications use the simple expression, $1/T$ where T is the predicted execution time. This has the nice property that an operation that takes twice as long to execute is only half as desirable to the user.

The default utility function predicts energy consumption using the history of past application energy usage, as described in Section 6.2.3. Spectra weights the importance of energy conservation by using goal-directed adaptation. The weighted energy component of utility is calculated as $(1/E)^{kc}$, where E is predicted energy usage, c is the global feedback parameter, and k is a constant (currently set to 10). The exponential function allows c to act as a weighting factor—high values of c cause energy usage to be the dominant component of the utility calculation. Thus, when c is 0, energy usage does not impact utility at all, and when c is 1, energy usage has a very large impact. The value of k limits the maximum impact of energy usage on the utility calculation. Higher values of k make energy usage more dominant when energy conservation is extremely critical, but make the utility calculation very sensitive to slight changes in the value of c when energy conservation is less critical. I have set the value of k to 10 to balance these concerns.

Fidelity is a multidimensional metric of application-specific quality. Since fidelity is fixed as an input to the utility function, no prediction is necessary. However, each application must provide a function which specifies the desirability of each fidelity as a numerical value. Spectra's default utility function calls the provided function to obtain the application-specific desirability of the input fidelity.

7.3.8 Applications

Three applications have been modified to use Spectra: the Janus speech recognizer, the Latex document preparation system, and the Pangloss-lite natural language translator [24]. I modified the first two applications, while Pangloss-lite was modified by SoYoung Park.

Speech recognition

The Janus speech recognizer was previously described in Section 4.5. It has three possible execution plans: local, hybrid, and remote. It also supports two fidelities: full-quality and reduced-quality recognition. Janus assigns reduced-quality recognition the value 0.5, and full-quality recognition the value 1.0. For execution time, Janus specifies that utility is inversely proportional to the expected latency.

The speech recognizer can use a task-specific vocabulary to recognize utterances in a particular context. Since the choice of vocabulary influences resource usage, Janus specifies the vocabulary name when executing an operation–this allows Spectra to generate data-specific predictions for recently used vocabularies. Since the data files accessed by Janus are typically static, there are no data consistency issues for the application.

Document preparation

The Latex document preparation system generates a graphical DVI file from multiple input files. Latex only has a single fidelity, since documents must typically be generated at highest quality. The application has two possible execution plans: local, in which all processing is done on the client, and remote, in which all processing is done on a remote server. As with Janus, Latex specifies that utility is inversely proportional to the expected latency.

I have modified Latex to use Spectra by creating two minimal code components: a front-end and a Spectra service. The front-end calls Spectra to select an execution location. It specifies the name of the top-level input file so Spectra can parameterize its predictions by document. The service runs Latex as a child process when requested.

Data consistency is a significant consideration when executing Latex. The user will often have locally modified one or more input files. Spectra must ensure that any modifications are reintegrated before processing is done remotely,

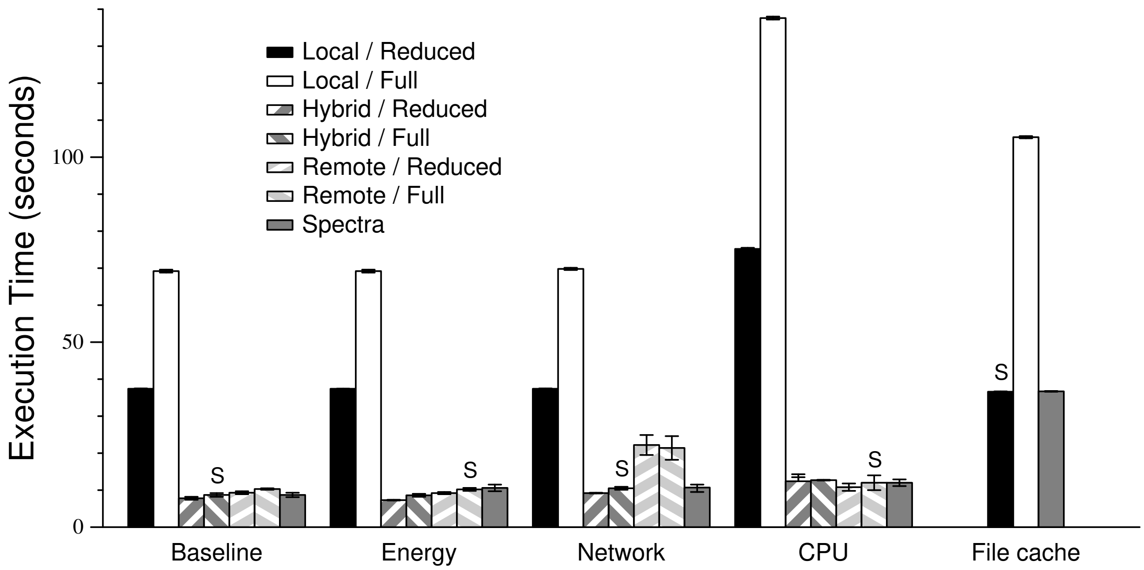
Natural language translation

Pangloss-Lite translates text from one language to another. It can use up to three translation engines: EBMT (example-based machine translation), glossary-based, and dictionary-based. Each engine returns a set of potential translations for phrases contained within the input text. A language modeler combines the output of the engines to generate the final translation.

Pangloss-Lite fidelity increases with the number of engines used for translation. Since the EBMT engine has the largest data set, it is assigned a fidelity of 0.5. The glossary-based and dictionary-based engines produce subjectively worse translations—they are assigned fidelity levels of 0.3 and 0.2, respectively. When multiple engines are used, their respective fidelities are added since the language modeler can combine their outputs to produce a better translation. For example, when the EBMT and glossary-based engines are used, a fidelity of 0.8 is assigned.

For Pangloss-Lite, any execution that takes longer than 5 seconds is undesirable—i.e., the execution component of the utility function assigns it a value of 0. Conversely, all translations that take less than half a second are equally good; they are assigned a utility value of 1. Translations that take time, *T*, between one half and five seconds are assigned utility $(T - 0.5)/(5 - 0.5)$.

All three translation engines and the language modeler may be executed remotely. While execution of each translation engine is optional, the language modeler must always be executed. Thus, there are 54 separate execution plans from which Spectra may choose. Pangloss-Lite seldom modifies its data files, so data consistency is of little concern.



This figure shows speech recognition execution time for the five resource scenarios described in Section 7.4.1. Each data set represents a different scenario. The first six bars show the execution time for each possible combination of execution plan and fidelity. In each data set, the alternative selected by Spectra is marked with a "S". The final bar in each data set shows execution time when Spectra is used to select the best alternative. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

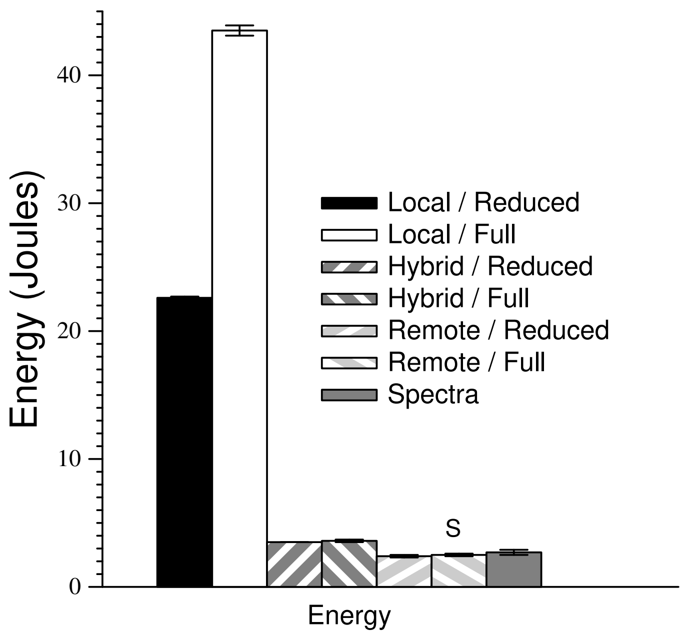
Figure 7.5: Speech recognition execution time

7.4 Validation

My validation of Spectra measured how well Spectra adapts to changes in resource availability. I executed Janus, Latex, and Pangloss-Lite under a variety of scenarios in which I varied resource availability. For each scenario, I measured application latency and energy usage for each possible combination of fidelity, execution plan, and server choice. I also asked Spectra to choose one of the possible alternatives for application execution. If Spectra chose the best possible alternative, I considered the outcome successful; otherwise, I considered it a failure. The results of this validation are shown in the next three sections. Section 7.4.4 provides a more detailed evaluation of the overhead of Spectra execution.

7.4.1 Speech recognition

For the speech recognition validation, I limited possible execution to two machines. The client machine, an Itsy v2.2 pocket computer, represents the type of small, highly-mobile devices used for pervasive computing. Spectra used the Smart Battery energy monitor to measure Itsy energy usage. An IBM T20 laptop with a 700 MHz Pentium III processor and 256 MB DRAM served as a possible remote server. Since the Itsy lacks a PCMCIA slot



This figure shows speech recognition energy usage for the energy resource scenario described in Section 7.4.1. The first six bars show energy usage for each possible combination of execution plan and fidelity. Spectra selected the Remote / Full alternative, so it is marked with a "S". The final bar shows energy usage when Spectra is used to select the best alternative. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

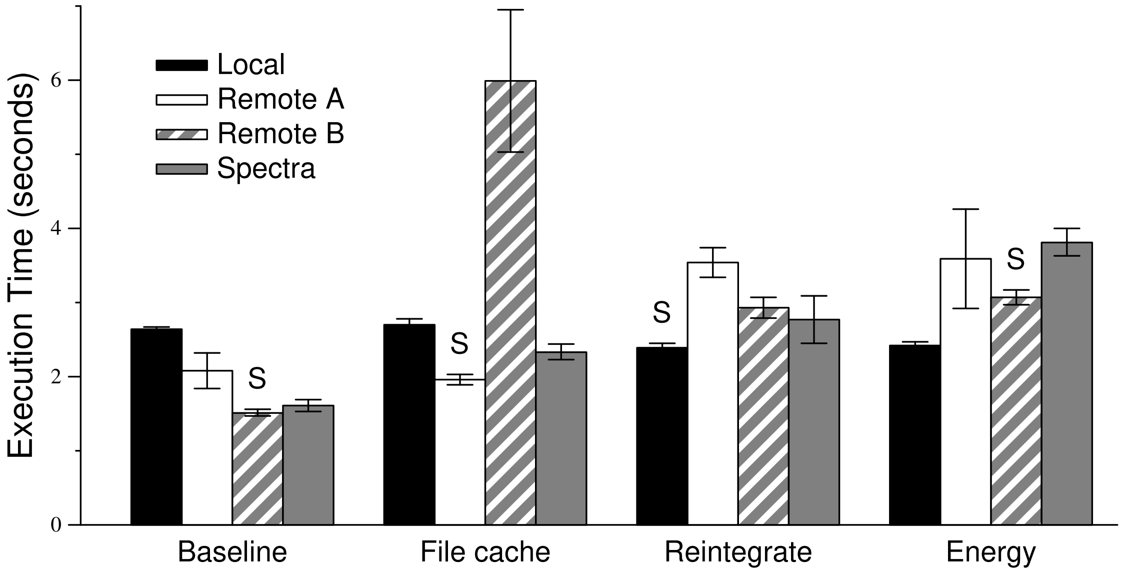
Figure 7.6: Speech recognition energy usage

(such as is available on the Compaq iPAQ handheld), the two machines were connected with a serial link.

I first recognized 15 utterances so that Spectra could learn the application's resource requirements. I then measured how well Spectra performed when recognizing a new utterance in five different resource scenarios.

Figure 7.5 shows measured execution time for each scenario. The first data set shows results for the baseline scenario, in which both computers have no significant CPU activity and are connected to wall power. The first six bars show execution time for each alternative available to Spectra. The local execution plan is clearly inferior to the hybrid and remote plans, taking 3–9 times as long to execute. The large disparity is caused by the speech recognizer's substantial use of floating-point instructions, which are emulated in software on the Itsy's StrongArm processor. However, using the hybrid execution plan and performing some computation locally takes less time than using the remote execution plan.

The "S" label in each scenario in Figure 7.5 indicates which alternative was chosen by Spectra. In the baseline scenario, Spectra correctly chooses the hybrid execution plan and full vocabulary. This combination takes less time to execute than all but one alternative. The quicker alternative executes only slightly faster, but has a fidelity only half as desirable



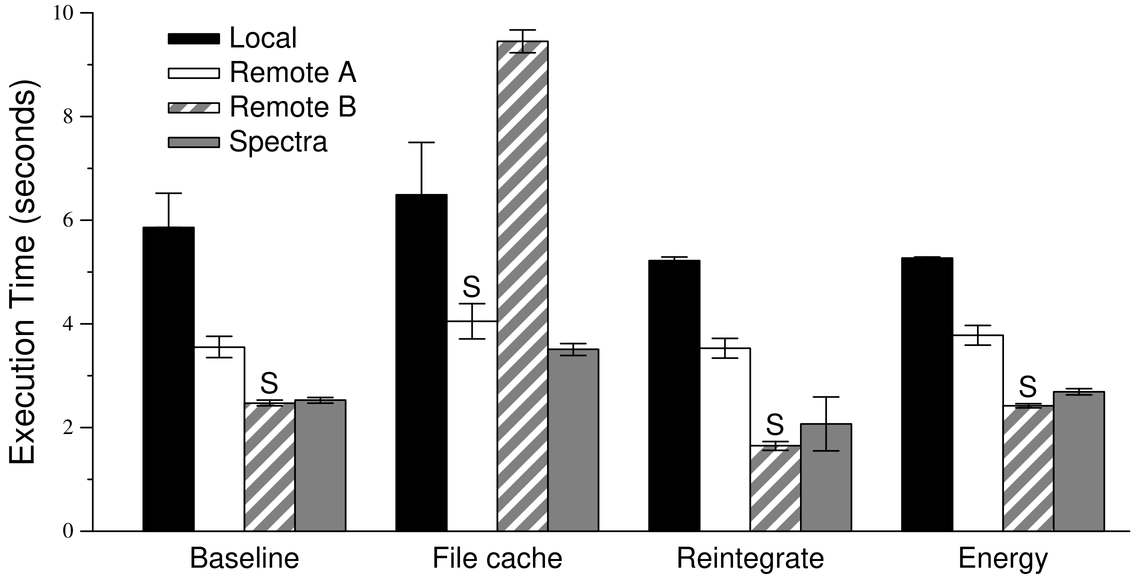
This figure shows Latex execution time when preparing a 14-page document in each of the four resource scenarios described in Section 7.4.2. Each data set represents a different scenario. The first three bars show the execution time for each possible execution location—the location selected by Spectra is marked with a "S". The final bar in each data set shows execution time when Spectra is used to select the best alternative. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Figure 7.7: Latex execution time for the small document

to the user. The last bar in each data set shows the execution time when Spectra chooses the best alternative—the difference in height between this bar and the one indicated with an "S" shows Spectra's decision overhead. In all scenarios the overhead is minimal—the difference in height is within the 90% confidence intervals.

Each remaining scenario differs from the baseline by varying the availability of a single resource. In the energy scenario, the client is battery-powered with an ambitious battery lifetime goal of 10 hours. The second data set in Figure 7.5 shows latency results for this scenario, and Figure 7.6 shows measured energy usage. Since energy conservation is critical, Spectra chooses the remote execution plan and the full vocabulary. Although hybrid execution takes less time, it consumes more energy because a portion of the computation is done on the client. As in the baseline scenario, Spectra correctly chooses to avoid the reduced vocabulary—the small energy and latency benefits do not outweigh the decrease in fidelity.

The network scenario halves the bandwidth between the client and server. This makes remote execution especially undesirable, and Spectra correctly chooses to use the hybrid execution plan and full vocabulary. The CPU scenario loads the client processor by executing a CPU-intensive background job. The cost of local computation increases, making the remote execution plan more attractive. Spectra chooses the remote plan because the



This figure shows Latex execution time when preparing a 123-page document in each of the four resource scenarios described in Section 7.4.2. Each data set represents a different scenario. The first three bars show the execution time for each possible execution location—the location selected by Spectra is marked with a "S". The final bar in each data set shows execution time when Spectra is used to select the best alternative. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Figure 7.8: Latex execution time for the large document

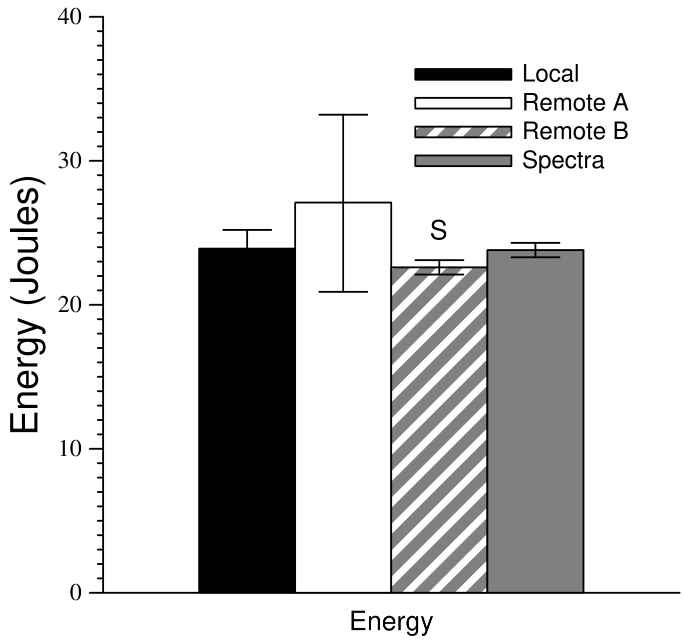
performance cost of doing the first recognition phase locally now outweighs the benefit of reduced network usage.

In the file cache scenario, the Spectra server is made unavailable by simulating a network partition. However, the Coda file servers remain accessible. Prior to execution, the 277 KB language model for the full vocabulary is flushed from the client's cache. This does not affect reduced-quality speech recognition. However, the execution time of full-quality recognition increases significantly because the language model must be refetched from a Coda file server. Spectra anticipates the cache miss, and chooses to use reduced-quality recognition. Since full-quality recognition would be approximately 3 times slower than reduced-quality recognition, the decrease in fidelity is acceptable.

7.4.2 Document preparation

I next evaluated how well Spectra supports the Latex document preparation system. Latex's resource requirements differ markedly from those of Janus. Latex performs less computation per operation, but reads and writes a larger number of files. Since some of the input files are likely to have been modified on the client, data consistency is important.

I executed the Latex front-end and the Spectra client on an IBM ThinkPad 560X laptop.



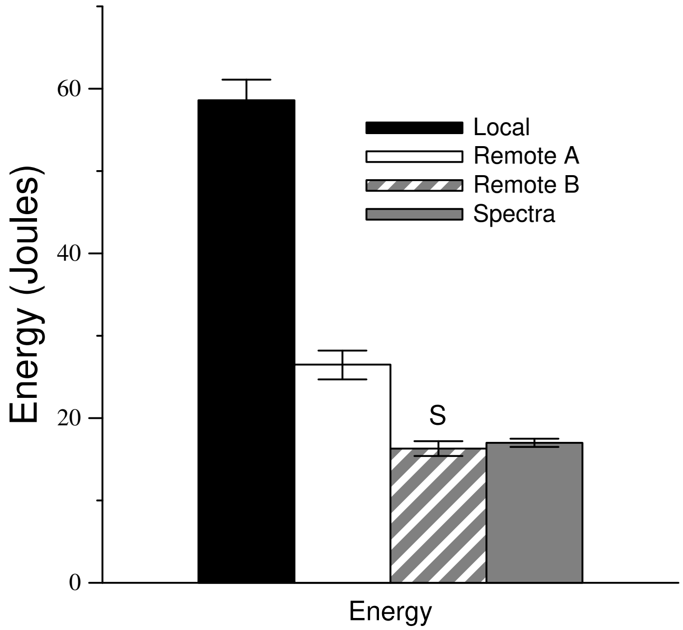
This figure shows Latex energy usage when preparing a 14-page document in the energy resource scenario described in Section 7.4.2. The first three bars show energy usage for each possible execution location. Spectra selected Remote B, so that alternative is marked with a "S". The final bar shows energy usage when Spectra is used to select the best location. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Figure 7.9: Latex energy usage for the small document

I ran Spectra servers on the laptop and on two remote servers; server A had a 400 MHz Pentium II processor, server B had a more powerful 933 MHz Pentium III processor. Thus, there are three possible alternatives: local execution on the laptop, remote execution on server A, and remote execution on server B. The laptop communicated with the servers using a 2 Mb/s 2.6 GHz wireless WaveLAN network.

I used two documents for evaluation: the smaller document was 14 pages in length, and the larger was 123 pages. I first executed Latex 10 times on each document to allow Spectra to learn application resource requirements. I then measured how well Spectra performed when preparing the documents in four different resource scenarios.

Figure 7.7 shows the measured execution time for preparation of the smaller document. The first data set shows the baseline scenario, in which all computers have no significant CPU activity and are connected to wall power. All files needed for document preparation are cached on every machine. Since little data is transmitted over the network, CPU speed is the primary consideration in this scenario. Remote execution proves faster than local execution, with server B preparing the document more quickly than server A due to its faster processor. As shown by the "S" above the third bar, Spectra chooses to use server B. The fourth bar in the data set shows that the time used by Spectra to choose the correct



This figure shows Latex energy usage when preparing a 123-page document in the energy resource scenario described in Section 7.4.2. The first three bars show energy usage for each possible execution location. Spectra selected Remote B, so that alternative is marked with a "S". The final bar shows energy usage when Spectra is used to select the best location. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

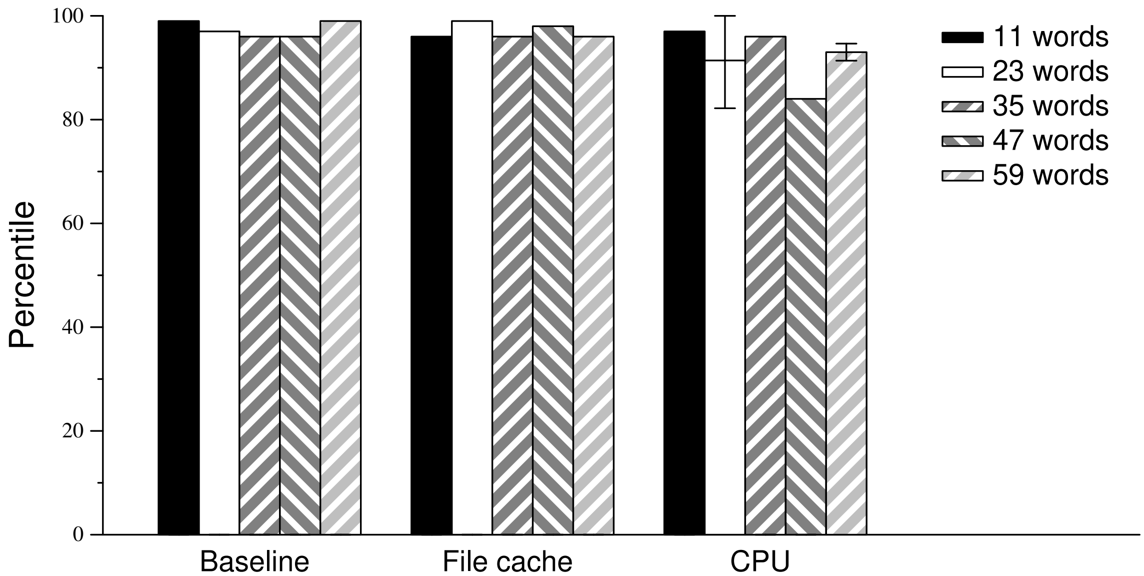
Figure 7.10: Latex energy usage for the large document

execution plan is minimal. Figure 7.8 shows similar results for the larger document.

In the file cache scenario, server B does not have any document input files cached locally. The execution of Latex is delayed while it fetches files from the server. Figures 7.7 and 7.8 show that file cache misses greatly increases the time needed to execute Latex on server B. Spectra correctly anticipates this and switches execution to server A for both documents.

In the reintegrate scenario, a 70 KB input file for the smaller document is modified on the client. Before preparing the smaller document on a remote server, Spectra must ensure that the modified input file is reintegrated to the file servers. As Figure 7.7 shows, reintegration over the wireless network significantly increases execution time for both remote execution options. Since the modification already exists on the client, the speed of local execution is unaffected. Spectra therefore chooses to use the local execution plan for the smaller document.

The modified source file is not an input for the larger document. As Figure 7.8 shows, Spectra predicts that the modified file will not be needed and does not force reintegration. Spectra therefore chooses the fastest plan: execution on server B. In this scenario, data-specific prediction allows Spectra to choose the fastest execution plan for each document.



This figure shows the accuracy of Spectra's choices for how and where to execute Pangloss-Lite operations. It shows results for five different sentences translated in three resource scenarios. In each case, the 100 possible alternatives available to Spectra are ranked by their utility—the height of each bar gives the percentile into which Spectra's choice falls. Thus, a height of 99 indicates that Spectra's choice is in the 99th percentile. Each bar represents the mean of five trials. The error bars show 90% confidence intervals.

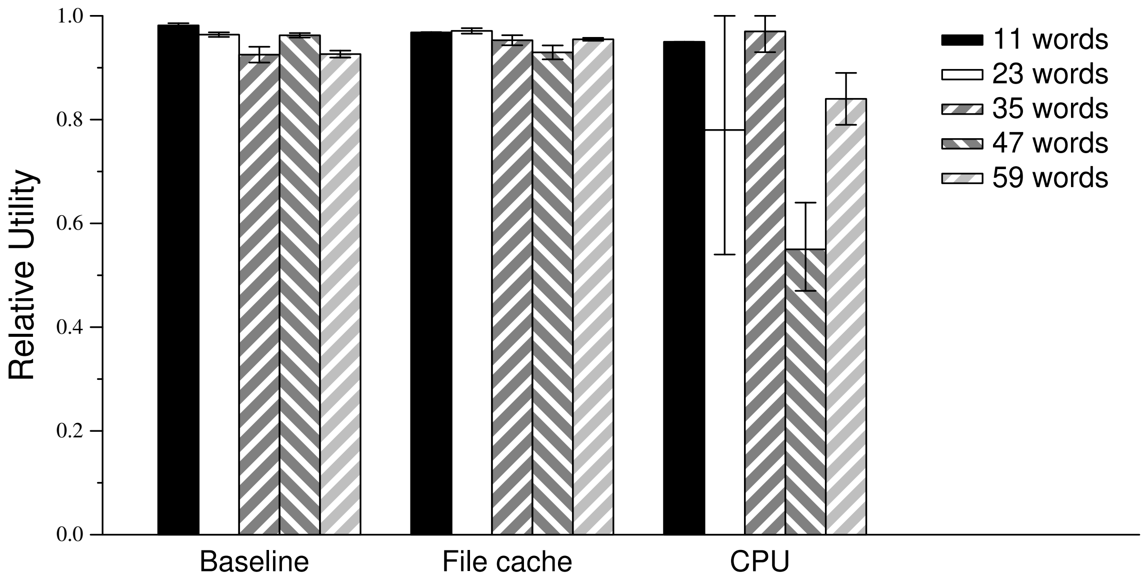
Figure 7.11: Accuracy of Spectra choices for Pangloss-Lite

The energy scenario is identical to the reintegrate scenario, except that the client is disconnected from wall power and a very aggressive goal for battery lifetime is specified. For the smaller document, Spectra chooses server B, even though this takes more time to execute. Figure 7.9 shows the reason for this choice: remote execution on server B uses slightly less energy than the other options. Because energy is of paramount concern due to the aggressive battery goal, Spectra opts for energy savings over a smaller execution time. The choice for the larger document is much clearer, since execution on server B saves both time and energy.

7.4.3 Natural language translation

I evaluated Pangloss-Lite using the same experimental setup as for Latex—this evaluation was performed with the help of SoYoung Park. I translated a set of 129 sentences from Spanish to English to allow Spectra to learn application resource requirements. I then asked Spectra to choose the best option for translating five additional sentences of different length from Spanish to English.

Pangloss-Lite has more alternatives for execution than either previous application—Spectra may choose from 100 different combinations of location and fidelity. Due to the



This figure shows the utility of using Spectra to choose how and where to execute Pangloss-Lite operations. It shows results for five different sentences translated in three resource scenarios. Each bar shows the ratio of the utility when Spectra chooses an alternative to the utility achieved when an oracle with no overhead makes the optimal choice. Thus, the ideal height of 1.0 indicates that Spectra has made the best possible choice and incurred no overhead. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Figure 7.12: Relative utility of Spectra choices for Pangloss-Lite

large number of alternatives, the results are presented in a different format. I rank the alternatives by the utility they achieved. Each bar in Figure 7.11 shows the percentile into which Spectra's chosen alternative falls. A value of 99 indicates that Spectra has made an optimal choice, while a value of 0 indicates that Spectra has made the worst possible choice. Each bar in Figure 7.12 compares the utility achieved when Spectra is used to choose an alternative to the optimal utility that would be achieved if an oracle with no overhead chose the best possible alternative. Even when Spectra chooses the optimal alternative, it usually will not achieve a ratio of 1.0 because its overhead increases overall translation time.

In the baseline scenario, all computers have no significant CPU activity, are connected to wall power, and have all data files cached. For the three smallest sentences, Spectra executes all three translation engines. For the two larger sentences, Spectra does not execute the glossary-based engine. Its choice of which engines to execute is optimal for all sentences. This scenario shows the importance of modeling operation input parameters—Spectra correctly predicts that execution time will increase with sentence size and switches to a lower fidelity to achieve acceptable performance for larger sentences.

Spectra runs the dictionary-based engine locally for the four smallest sentences and executes all other components on server B. Remote execution yields significant performance improvements for the glossary-based and EBMT engines because they have large CPU requirements. The location of the dictionary-based engine and the language modeler does not affect performance much because their processing requirements are small. Thus, although Spectra's location choice for these two components is not optimal for all sentences, the performance penalty is small (less than 0.07 seconds).

The baseline scenario illustrates an important property of Spectra. When alternatives have significant difference in utility, Spectra almost always makes the correct decision. If it does choose an alternative that is not best, the utility of its choice is usually very close to optimal. For the five sentences in the baseline scenario, the utility of Spectra's choices are all within 2% of optimal. Even adding in the overhead of picking the correct alternative, as shown in Figure 7.12, the utility of Spectra's choices are all within 7% of optimal.

In the file cache scenario, a 12 MB file needed by the EBMT engine was not in server B's cache. This made execution of the EBMT engine prohibitively slow on server B. Spectra therefore chose to use server A for remote execution in this scenario. Spectra executed all engines for the smallest sentence, but did not execute the glossary-based engine for the others. Because server A is slower than server B, the application more quickly reaches the crossover point where the the fidelity benefit of executing the glossary-based engine is less than the performance cost.

As with the baseline scenario, Spectra's choices of which engines to execute are optimal. While it chooses the correct location to execute the glossary-based and EBMT engines, it sometimes incorrectly places the dictionary-based engine and language modeler. However, the performance impact of these incorrect location decisions is less than 0.1 seconds for all sentences. With decision overhead, the utilities achieved by Spectra are within 7% of optimal.

In the CPU scenario, I varied the file cache scenario by executing two CPU-intensive processes on server A. This proved to be an interesting scenario, since the optimal choice of server and engines to execute was different for each sentence. For the 11 and 35 word sentences, Spectra's only mistake was in locating the language modeler, and it achieved utility within 5% of optimal. For the 23 word sentence, Spectra made an optimal choice in four of five trials, but chose the incorrect server in one trial, leading to the high variance shown in Figure 7.12. For the 47 word sentence, Spectra consistently chose to use server A when server B was optimal, and consequently achieved utility only 55% of optimal—this was the worst decision made by Spectra in any of our experiments. For the largest sentence, Spectra incorrectly chose to execute the glossary-based and dictionary-based engines in four trials when the combination of EBMT and dictionary-based engines was best.

In general, Spectra did an excellent job of choosing the best alternative for Pangloss-Lite execution. On average, it achieved 91% of the optimal utility.

7.4.4 Overhead

I measured Spectra's overhead by performing a null operation, one which returns immediately after being invoked. Figure 7.13 shows the amount of time needed to execute the null

Activity	No Servers	1 Server	5 Servers
register_fidelity_op	1.2 ms.	1.5 ms.	1.2 ms.
begin_fidelity_op	8.3 ms.	13.1 ms.	65.5 ms.
file cache prediction*	5.2 ms.	8.5 ms.	8.5 ms.
other resource prediction	0.8 ms.	1.6 ms.	7.2 ms.
choosing an alternative	0.4 ms.	1.0 ms.	43.4 ms.
other activity	1.9 ms.	2.0 ms.	6.4 ms.
do_local_op	5.9 ms.	4.7 ms.	5.1 ms.
operation execution	4.9 ms.	4.0 ms.	4.0 ms.
other activity	1.0 ms.	0.7 ms.	1.1 ms.
end_fidelity_op	2.1 ms.	2.1 ms.	2.2 ms.
total	18.4 ms.	21.4 ms.	74.0 ms.

This figure shows the amount of time Spectra takes to perform a null operation. The three columns show the amount of time spent on each activity when 0, 1, and 5 remote servers are available for execution. The times shown represent the mean of ten trials. This figure shows results for a relatively small local file cache (less than 100 files); for a full cache, measured time for file cache prediction is 359.6 ms.

Figure 7.13: Spectra overhead

operation when Spectra has 0, 1, and 5 remote servers on which it can choose to locate the operation.

With no remote servers are available, the null operation takes 18 ms. to execute. The majority of this time is spent in the `begin_fidelity_op` and `do_local_op` system calls. File cache prediction takes 5.2 ms. with a relatively empty cache; however, it can take as long as 359.6 ms. when the file cache is full. The excessive overhead of file cache prediction is due to the inefficient file system interface—every time a prediction is needed, Coda writes the entire cache state to a temporary file. If Spectra replaced this interface with one that sends incremental updates through a Unix socket whenever files are evicted from or added to the cache, then the overhead of file prediction would be much less. The execution of the local operation takes 4.9 ms., mostly due to the cost of inter-process communication.

Spectra's overhead increases with the number of potential servers. As Figure 7.13 shows, the primary reason for the additional overhead is the time taken to choose the best alternative. With each additional server, Spectra has more possible alternatives from which to choose. However, total Spectra overhead is only 74 ms. with five servers, which is very reasonable for the targeted set of applications—those that perform operations of a second or more in duration.

7.5 Summary

Remote execution is an important dimension of energy-aware adaptation. Applications can execute part of their functionality on remote servers to reduce their consumption of battery energy on a client machine. However, the goal of reducing energy consumption will often conflict with competing goals such as reducing execution time and maintaining high application fidelity.

Spectra provides the necessary system support to help applications arbitrate between these competing goals. Spectra is designed for pervasive environments: it predicts the availability of battery energy, file cache state, network bandwidth, and CPU cycles. It also observes application resource usage to predict future needs. Using these predictions, it evaluates possible locations and fidelities for execution, and advises applications how and where they can best execute operations.

Evaluation of Spectra using a speech recognizer, a document preparation system, and a natural language translator shows that Spectra often chooses the best location and fidelity for execution despite wide variation in resource availability. When Spectra does not make the best decision, its choice is usually very close to optimal. Further, the overhead of the system is reasonable for applications which perform remote operations of a second or more in duration.

Chapter 8

Related work

This dissertation is one of the first comprehensive efforts to address higher-level energy management. To the best of my knowledge, it includes several novel contributions:

- It has described the design and implementation of PowerScope, the first tool to use profiling to map energy consumption to program structure.
- It has shown that applications can modify their behavior to conserve significant amounts of energy when battery levels are critical. The key method of conservation is reduction of application fidelity.
- It has demonstrated that the operating system can effectively manage energy as a resource and meet user-specified goals for battery lifetime.
- It has shown how system support for remote execution can enable applications to dynamically balance the competing goals of energy conservation, performance, and high fidelity.

While there is no single research effort which spans all the subjects discussed in this dissertation, there is a large amount of work that intersects one or more areas. In this chapter, I discuss work related to each part of the dissertation. The next section describes other approaches to energy measurement. Section 8.2 describes related work in energy management, dividing this topic into efforts aimed at the higher levels of the system and efforts aimed at hardware devices. Section 8.3 summarizes the most relevant work in adaptive resource management, and Section 8.4 describes work related to Spectra.

8.1 Energy measurement

While PowerScope is the first energy profiler, its development was aided by previous work in CPU profiling. In particular, the implementation of the System Monitor is closely related to similar sampling-based profilers designed to run continuously, such as Morph [100] and DCPI [3].

Other than the profiling technique used by PowerScope, there are two main approaches to measuring application energy usage. The first approach, which I shall refer to as *energy accounting*, measures the energy consumption of specific system activities. Then, when applications execute, the energy accounting tool counts the number of times each activity occurs. By multiplying the number of occurrences by the energy usage per activity, the tool estimates total application energy usage.

PowerMeasure and StateProfiler, developed by Lorch and Smith [53, 54], provide energy measurements for Apple Macintosh PowerBook Duo laptops. PowerMeasure benchmarks the power expended by hardware components such as the disk and CPU while they operate in various power states. For example, it measures the power expended by the hard disk when it is spinning and when it is idle. During application execution, StateProfiler records transitions between power states and uses the benchmark data to estimate total energy consumption. The tools report average power usage and total energy expended by each hardware component.

The Millywatt tool, developed by Cignetti et al. [11], estimates energy usage for applications executing on PalmOS handheld computers. The tool models power usage for each hardware device state, as well as the cost of transitioning between states. When applications are executed in a modified simulator, the tool traps system calls to infer when transitions occur between device power states. From this information, the tool estimates the energy usage of the simulated application.

Neugebauer and McAuley propose to incorporate energy accounting into the Nemesis operating system [66]. They observe that Nemesis already provides accurate resource accounting of traditional resources such as CPU, network, display, and disk. Bellosa [5] performs energy accounting using CPU performance counters. He proposes counting the number of total instructions executed, number of floating-point instructions, and number of cache misses to estimate the amount of energy consumed by the CPU and memory subsystem.

Energy accounting has several drawbacks. First, correct accounting relies on the completeness of the energy model. The model developer must benchmark every hardware device that may be a significant source of power consumption. The model must also include all possible power states and transition costs for each device. This means that developing an accurate model is non-trivial. Further, a new model must be developed for each hardware platform on which energy measurements are needed. The second drawback is that accounting does not capture variance in power expenditure within a single state. For example, CPU power usage varies with the instruction mix [87] and wireless network power usage varies with the amount of channel congestion [77]. Finally, it is difficult to imagine expending this approach to capture the effects of power management performed entirely in hardware, such as the 802.11 standard for wireless networks. One significant advantage of energy accounting is that it can separate out the energy consumption of asynchronous activities such as disk spin-up which PowerScope assigns to the currently executing process. A hybrid approach combining profiling and accounting may therefore be advantageous.

The second main approach to energy measurement is *power analysis*, which uses a model of the energy cost of individual instructions to compute the total energy consumed by the execution of a program. This approach is further sub-divided into instruction-level and architecture-level power analysis. Instruction-level power analysis is exemplified by Tiwari et al. [86, 87, 49], who construct per-instruction energy models for several processors by measuring the steady-state energy consumption of each instruction as it executes within a tight loop. They augment their model by measuring the additional energy consumed by inter-instruction switching costs when two different instructions execute sequentially, and by measuring the effect of various stalls and cache misses. Klass et al. [42] develop a model that closely approximates inter-instruction energy effects, but requires much fewer measurements to construct.

Architecture-level power analysis uses a detailed hardware model to estimate power and energy consumed by program execution. Two examples of this approach are SimplePower [91] and Wattch [6], both of which extend the popular SimpleScalar framework to provide power estimates. A similar approach is used by Simunic et al. [82] to develop a cycle-accurate simulator specialized for Hewlett Packard's SmartBadge platform.

Instruction-level and architecture-level power analysis is targeted at exploring alternative hardware architectures and compiler optimizations. Unlike PowerScope, power analysis does not require a physical hardware implementation, allowing hardware designers to explore many possible architectures. However, because of the detailed nature of the models, power analysis is most successful when applied to tight kernels of code such as those found in popular benchmark suites. It is not clear how power analysis can scale to analyze the behavior of large, dynamic programs such as Web browsers. Power analysis also does not model the energy effects of hardware components such as the network, disk, and display—such components constitute a large portion of the total energy consumption of mobile computers.

Several researchers have used more ad-hoc approaches to perform useful characterizations of the energy and power usage of mobile computers. Warren [96] examines the relationship between average power consumption and active (non-idle) power consumption, developing a methodology for predicting the active power consumption for applications which are composed of idle and non-idle phases. Marsh and Zetel [60] measure the steady state component power consumption for several laptop models, as well as the effect of several simple power-saving strategies. Ellis [18] measures the power consumption of a PalmPilot Professional while it operates in several different states, and Ikeda [34] reports on how the power usage of components of the IBM ThinkPad laptop has changed over time.

8.2 Energy management

Most previous work in energy management has concentrated on the lower-levels of the system. To date, very few efforts have focused on the operating system and applications. Next, I describe those efforts that focus on the higher-level of the system. Then, I discuss previous research in hardware energy management, addressing work in processor, network, and disk power management. I conclude by detailing comprehensive energy management approaches which span multiple layers.

8.2.1 Higher-level energy management

Ellis' Milly Watt project [18, 89] is exploring the development of a power-based API that allows a partnership between applications and the operating system in setting energy use policy. They have demonstrated that a power-aware page allocation policy coupled with dynamic hardware policies can dramatically improve memory energy-efficiency [47]. This work opens a further dimension for energy-aware applications—by reducing their memory footprint, applications can allow the operating system to transition unneeded memory banks to low-power states.

Chase's Muse architecture [8] reduces the energy needs of a hosting center by managing server resources. Muse employs an economic model in which customers bid for service. When the marginal benefit of increased revenue from customers exceeds the dynamic cost of energy usage, Muse powers up additional servers. While Muse's goal, maximizing service center profit, is very different from Odyssey's goal of meeting desired battery lifetimes, both systems share the general approach of trading reduction of service for energy conservation.

Several projects have explored the use of compiler optimizations for reducing application energy usage. Tiwari et al. [87, 86] examine the energy benefit of instruction-level optimizations such as instruction reordering and energy-driven code generation. They achieve limited gains, mostly by exploiting processor-specific features. Simunic et al. [82] examine energy-efficient optimizations for a MPEG application executing on a StrongArm 1100 processor. They note that compiler optimizations produce only a 1% energy benefit for the application, while hand-crafted source code optimizations produce a 35% energy benefit. However, since their hand-coded optimizations also produce a 32% reduction in execution time, it is not clear how much of the energy reduction is simply due to the faster execution achieved by improved software implementation. If compiler optimizations prove successful in reducing application energy usage, it is likely that they will be complementary to energy-aware adaptation.

8.2.2 Processor energy management

Most processor energy management techniques exploit the energy benefits of reducing the CPU clock frequency. Burd and Broderson [7] use an analytic model to show that reducing CPU clock frequency, by itself, does not reduce processor energy usage. However, when combined with dynamic voltage scaling, reducing the clock frequency can noticeably reduce processor energy usage. This approach is used in many modern mobile processors, including the Transmeta Crusoe chip [63].

Given a processor which can operate at multiple clock frequencies and voltages, the operating system must balance performance and energy conservation concerns to select the best processor speed for operation. Scheduling for variable-speed processors is therefore another dimension of energy-aware adaptation.

There have been a great number of scheduling algorithms proposed for variable-speed processors. They can be most easily classified along the axis of application transparency. At one end are scheduling algorithms which require no application modification. These transparent algorithms infer application performance needs by observing their behavior. At the other end of the axis are algorithms which require applications to specify their performance requirements.

The first transparent scheduling algorithm was proposed by by Weiser et al. [97]. They observe that for any given amount of computation to be completed within a specified time period, the minimum energy is expended when the clock frequency is set so that the computation takes the entire time period. Their algorithm uses observations of recent CPU load to predict future CPU needs. It adjusts the clock frequency to eliminate projected idle time. The clock frequency is recomputed at fixed time intervals, limiting the maximum delay penalty that can be incurred by running at a reduced frequency. Both Govil et al. [27] and Pering et al. [71] propose several alternatives to this algorithm and simulate their performance. They find that complex prediction schemes are often ineffective, but that simple prediction schemes can outperform Weiser's algorithm. Martin [62] extends Weiser's algorithm to account for the effects of non-ideal batteries and memory hierarchies.

Grunwald et al. [29] measure the effectiveness of several transparent algorithms on an Itsy v1.5 modified to support voltage scaling. They find that predictions based upon CPU load observations are insufficient to provide energy savings while still meeting the performance goals for real-time multimedia and interactive applications.

Another class of algorithms attempts to infer application processing requirements with more detailed monitoring techniques. Flautner et al. [20] monitor inter-process communication to classify tasks into interactive, periodic, producer, and consumer categories. For each process, they also monitor processing requirements to predict future CPU needs. They adjust processor frequency so that each application's predicted computations will finish within the inferred deadlines, capping the maximum amount of performance degradation possible. The PACE system [55], proposed by Lorch and Smith, monitors CPU activity and I/O events to associate processing with specific user interface events. They calculate a statistical model of the processing requirements for each event, and adjust CPU clock frequency to minimize energy usage while ensuring with high probability that processing for interactive events completes within a 50 ms. deadline. Flautner notes a serious drawback of this class of algorithms: when an application performs many different activities, each of which has different processing requirements, it is very difficult to discriminate between activities and maintain separate predictors for each activity. For such applications, generic predictions may simply prove too inaccurate.

A final class of algorithms requires applications to specify their processing requirements to the operating system. Pering [72] uses a real-time approach, in which applications specify either a minimum rate at which they wish to execute, or a deadline by which a particular unit of work needs to complete. The scheduler attempts to minimize energy consumption by running at the lowest possible clock frequency and voltage that meets application constraints. Pillai and Shin [73] propose several similar algorithms which target periodic applications executing on an embedded processor. Shin et al [81] describe an alternative approach, in which the compiler automatically determines application processing requirements and inserts explicit voltage scaling system calls into the application.

Odyssey's design falls somewhere between the last two classes of algorithms. Like the techniques proposed by Flautner and Lorch, Odyssey monitors application activity in order to determine future processing needs. However, Odyssey requires that applications register operations and signal their execution. This allows Odyssey to overcome the difficulty noted by Flautner—Odyssey can determine which one of many possible activities is occurring and can accurately predict its processing requirements. At the same time, Odyssey requires significantly less modification to application source code than the real-time class of algorithms. In particular, Odyssey does not require applications to specify their future processing requirements. Finally, Odyssey is unique in providing the ability to account for shifting priorities between energy use and performance. While some voltage scheduling algorithms provide tunable parameters that could be used to balance these concerns, they provide no guidance as to how these parameters should be set.

8.2.3 Storage power management

Most efforts to reduce storage energy consumption have concentrated on saving power by spinning down disk drives when they are not in use. Wilkes [98] first presented the idea of adaptive disk power management, which uses a history of past disk activity to predict when the disk can be productively put in a low-power mode. He also hypothesized that a similar algorithm might be useful for determining when disks should be restored to full-power mode.

Greenawalt [28] uses a stochastic model to investigate threshold policies for spinning down the disk. The model assumes that disk requests follow a Poisson arrival pattern, which may not be particularly realistic. Douglis, Krishnan, and Marsh [17] compare fixed-threshold and predictive spin-down strategies with an optimal policy. They simulate these strategies for two traces of disk activity, and conclude that fixed-threshold strategies with short timeouts are best, and that the performance of predictive strategies is hindered by the randomizing effect of the buffer cache. Li et al. [51, 50] study several disk power management parameters. Their results show that spinning down the disk after two seconds of inactivity saves more energy than any other fixed-timeout strategy, while producing a small number of spin-up delays.

Douglis, Krishnan, and Bershad [16] explore adaptive strategies for spinning down the disk. They define a new metric, *bumps*, to be the number of disk spin-ups that occur when the disk has been spun down for less than a parameterized multiple of the spin-up delay. They present simulation results that show that their adaptive strategies can achieve a better balance between bumps and energy consumption than a fixed spin-down threshold. However, it is not clear that the bumps metric adequately reflects the adverse effect of spin-ups on the user. Golding et al. [26] present a taxonomy of algorithms for predicting idle time, and apply these algorithms to the task of deciding when to power down the disk. They describe an implicit power-performance tradeoff where algorithms that are successful in reducing energy consumption create more spin-up delays, and algorithms that create few spin-up delays consume more energy. Lu et al. [57, 58, 56] also simulate several adaptive disk power management algorithms and evaluate their operation under Windows NT.

Like the CPU policies described above, none of these disk algorithms account for differing priorities between energy use and performance. In addition, the evaluations of these algorithms all assume that patterns of disk activities are fixed. They do not analyze the possible beneficial effect of changing application or operating system behavior.

Other work in storage power management has focused on replacing hard-disk drives in mobile computers with other, low-power media. Douglis et al. [15] study the possibility of using flash memory as a replacement for hard drives. They report that flash offers low energy consumption, good read performance, and acceptable write performance. Marsh et al. [59] examine the possibility of using flash as a second-level buffer cache to reduce power consumption by allowing the hard disk to be idle more often. Schlosser et al. [80] note that MEMS-based storage may have significantly less power requirements than traditional storage devices. If mobile computers contain multiple storage media, Odyssey could possibly make power-performance tradeoffs by adaptively deciding which media applications should use.

8.2.4 Network power management

Stemm and Katz [85] measure the power consumption of several wireless network devices and show that the majority of energy is consumed while these interfaces are idle. They propose a protocol that shuts down the interface when not in use, creating an implicit tradeoff between energy consumption and the average delay for incoming packets. Kravets and Krishnan [43] also present a communication protocol for disabling wireless network interfaces when not in use. They discuss the tradeoff between energy consumption and the average delay for incoming packets, and show that an adaptive strategy performs better than a strategy with a fixed-length timeout. However, they do not provide any guidance for how one should set the parameters of such an adaptive strategy.

Kravets, Schwan, and Calvert [44] study how communication protocol parameters, such as the rate of packet acknowledgment, affect the energy consumption of a mobile host. They are in the process of developing a framework that monitors network conditions and adapts communication protocol parameters to reduce energy consumption. This framework is similar in spirit to Odyssey, but concentrates on the network layer rather than the application level.

In the realm of ad-hoc networking, Xu et al. [99] and Chen et al. [10] both propose routing algorithms that disable some nodes in the network in order to conserve energy. Li et al. [52] advocate using a routing algorithm that maximizes the total lifetime of the network, defined to be the amount of time until which a message cannot be sent due to insufficient battery energy. Ad-hoc routing allows yet another dimension of energy-aware adaptation, in which the routing algorithm trades routing performance for decreased energy usage.

8.2.5 Comprehensive power management strategies

The Advanced Power Management (APM) specification [35] provides a standard interface for power management. It allows the operating system to query the power state of devices such as the hard drive and place these devices in low-power modes. More recently, the Advanced Configuration and Power Interface (ACPI) specification [36] has expanded the APM interface by allowing detailed power management of individual hardware components. These initiatives define mechanisms for power management, but do not dictate policy. They are therefore complementary to the work described in this dissertation.

Lu, Simunic, and De Micheli [58] describe a software architecture in which user-level *power managers* collect utilization information from hardware devices and control the power state of the devices. Simunic et al. [83] extend this idea by proposing a specific model of hardware device usage patterns, based upon time-independent semi-Markov decision processes. They show that their approach can achieve better results than alternative policies when managing processor, hard disk, and wireless network power states. While their power managers will play a role similar to Odyssey in determining which energy-performance tradeoffs to make, their work is at an earlier stage of maturity.

8.3 Adaptive resource management

My thesis has two important characteristics which differentiate it from the large body of previous work in adaptive resource management. First, energy is a largely unexplored resource in this area. It differs from more commonly studied resources such as network and CPU in that present resource allocation decisions have a great impact on future resource availability. Second, Odyssey does not assume that applications have a-priori knowledge of their resource (energy) requirements—instead Odyssey observes the behavior of applications to infer their needs.

In addressing the challenge of managing energy as a resource, Odyssey builds upon several previous systems which have provided adaptive resource management. Chen's work in the Amaranth project [75, 76, 48] addresses resource management in the presence of multiple QoS dimensions and multiple resources. This work targets environments in which applications specify their resource requirements and users specify their preferences with utility curves. Chen restricts the types of utility functions that users can specify, allowing her to calculate the optimal fidelity at which applications should execute. Odyssey takes a different approach, allowing application writers to specify code procedures which encapsulate arbitrary utility functions, but not guaranteeing that applications will execute at optimal fidelities.

Like Amaranth, the ERDoS project [9] also targets multidimensional QoS data, requiring users to specify a benefit function detailing their preferences and applications to describe their resource requirements. Odyssey differs from both ERDoS and Amaranth in that it does not require applications to explicitly specify their resource requirements. This has considerable benefit when managing energy as a resource, since it is often difficult for applications to know how much energy they will consume on different platforms and with different power management settings.

The Rialto operating system [38] provides modular, user-centric resource management that attempts to dynamically maximize the user's perceived utility of the entire system. Although Rialto provides a flexible framework for managing multiple resources, it is not clear whether battery could fit cleanly into the framework. Rialto makes the simplifying assumption that resources can be allocated independently; however, an application's energy use clearly depends upon its usage of CPU, network, and disk resources. Also, Rialto requires that applications be aware of their own resource requirements.

Goal-directed adaptation utilizes feedback control to perform resource management. Feedback-based approaches have been previously used for such resource management problems as network congestion control [37] and CPU scheduling [84]. The SWiFT toolkit [25] takes a more general approach by providing a framework for building feedback-based highly adaptive systems using modular composition of simple building blocks. Odyssey's use of feedback is unique in two ways. First, Odyssey extends the use of feedback to a novel domain, that of energy management. Second, Odyssey adjusts the statistical gain specifically to reflect dynamically changing relative priorities for stability and agility in the system.

8.4 Remote execution

Remote execution is a well-established field in systems research. While most remote execution systems target only performance benefits, a few recent systems have explored how remote execution can reduce application energy use.

Rudenko et al. [77] demonstrate the potential energy benefits of remote execution by comparing the energy cost of executing several tasks both locally and remotely. Their RPF framework [78] assists in migrating tasks and adaptively decides whether a given task should be executed locally or remotely based upon a history of past power consumption. Kunz's toolkit [46] uses similar considerations to locate mobile code.

Although both systems monitor application execution time and RPF also monitors battery use, neither monitors individual resources such as network and cache state, limiting their ability to cope with resource variation. The prediction models used in these systems also seem overly simplistic. Neither accounts for the performance impact of concurrent processes running on the client or server. In addition, neither exploits the full potential of the energy-performance tradeoff—they migrate jobs only in cases when both energy usage and performance are not adversely affected.

Kremer et al. [45] propose using compiler techniques to select tasks that might be executed remotely to save energy. At present, this analysis is static, and thus can not adapt to changing resource conditions. Such compiler techniques are complementary to Spectra, in that they could be used to automatically select Spectra operations and insert Spectra calls into executables.

Vahdat et al. [89] note issues considered in the design of Spectra: the need for application-specific knowledge and the difficulty of monitoring remote resources.

Butler [67] uses the AFS distributed file system to provide consistency between local and remote machines. Spectra's use of Coda reflects a difference in target environments; AFS is an appropriate choice in fixed environments, but Coda's support for disconnected and weakly-connected operation is vital in mobile and pervasive environments.

Several remote execution systems designed for fixed environments analyze application behavior to decide how to locate functionality. Coign [32] statically partitions objects in a distributed system by logging and predicting communication and execution costs. Abacus [2] monitors network and CPU usage to migrate functionality in a storage-area network, and Condor monitors goodput [4] to migrate processes in a computing cluster. Because these systems are not designed for pervasive environments, they do not monitor the range of resources considered by Spectra. In addition, they do not support adaptive applications which modify their fidelity as well as their execution location.

Chapter 9

Conclusion

Energy management has been a critical problem since the earliest days of mobile computing. There is continuing evidence that it will remain a challenge as the field evolves. A large research investment in low-power circuit design and hardware power management has led to more energy-efficient systems. Yet, there is a growing realization that more is needed—the higher levels of the system, the operating system and applications, must also contribute to energy conservation.

This dissertation is one of the first detailed explorations of higher-level energy management. It has put forth the thesis that energy-aware adaptation, the dynamic balancing of application quality and energy conservation, is an essential part of a comprehensive energy management strategy. It has validated the thesis with two specific examples. The first shows that applications can significantly reduce the energy usage of the platforms on which they execute by degrading their fidelity. The second shows that applications can adapt to variation in resource availability by changing the location where functionality is executed. Finally, this dissertation has shown that the operating system can effectively manage battery energy as a resource. Using goal-directed adaptation, the operating system monitors energy supply and demand and uses feedback to meet user-specified goals for battery duration.

In the next section, I review the specific contributions of this dissertation in more detail. Then, in Section 9.2, I discuss some of the possible directions for future research generated by this work. Section 9.3 concludes by reviewing the major lessons that should be taken from this research.

9.1 Contributions

This dissertation makes contributions in three major areas. The first area is conceptual—this consists of the novel ideas generated by this work. The second area is a set of artifacts: PowerScope, Odyssey, Spectra, and the other software systems that I have constructed to validate my thesis. The final area of contribution is the energy measurements and experimental results which validate the ideas presented in the dissertation.

9.1.1 Conceptual contributions

Energy-aware adaptation is the primary concept studied in this dissertation. In order to achieve reduced energy usage, one must often sacrifice some dimension of performance or application quality. While energy-aware adaptation has many possible dimensions, I have studied two specific ones to validate the concept: application fidelity and remote execution. This dissertation has shown that by embracing the dynamic tradeoffs available through energy-aware adaptation, mobile systems can significantly extend their battery lifetimes.

Goal-directed adaptation is the second key concept discussed in this dissertation. I have shown that the operating system can effectively manage battery energy as a resource by using feedback to meet user-specified goals for battery lifetime. By monitoring energy supply and demand, the operating system can balance the competing concerns of application quality and energy conservation. Thus, while energy-aware adaptation provides the mechanism for achieving significant energy savings, goal-directed adaptation provides the guidance in deciding when energy conservation is appropriate.

The final important concept contained in this dissertation is the use of application resource history to guide adaptation decisions. I have shown that an adaptive system can respond more agilely and effectively to changes in energy demand by maintaining and consulting a history of previous application energy usage. In addition, the use of history allows applications to naturally express preferences which directly compare fidelity, performance goals, and projected energy usage.

9.1.2 Artifacts

In the course of this dissertation, I have developed three major artifacts: the PowerScope energy profiler, the energy extensions to Odyssey, and the Spectra remote execution system.

PowerScope is the first tool that uses profiling to map energy consumption to application structure. It enables developers to optimize their code for energy-efficiency by using statistical profiling to identify those components responsible for the bulk of energy consumption. As improvements are made, PowerScope quantifies their benefits and helps expose the next target for optimization. Through successive refinement, developers can improve a system to the point where energy consumption meets design goals.

I have made several improvements to the Odyssey platform for mobile computing. Primary among these are the extensions for goal-directed adaptation that enable Odyssey to monitor energy supply and demand and meet user-specified goals for battery lifetime. I have also added a modular resource measurement library to Odyssey—this library provides a clean framework for monitoring all resources of concern to a client operating in mobile and pervasive environments. In addition to refitting Odyssey's existing network measurement into this framework, I have created measurers for energy and file cache state.

Spectra is the first remote execution system to balance the competing concerns of performance, application quality, and energy conservation. It is designed specifically for clients operating in pervasive environments—it monitors supply and demand of CPU, network, file cache, and energy resources. In addition, it contains mechanisms for ensuring the consistency of applications which execute part of their functionality remotely.

9.1.3 Evaluation results

The evaluation results in this dissertation contain several important insights for the development of higher-level energy management. First, I have shown that applications can significantly reduce their energy usage by modifying the quality of data presented to the user. Further, these reductions are predictable and complementary to existing hardware power management techniques. These potential savings exist for many types of applications: open-source and closed-source; CPU-intensive and network-intensive; and multimedia and office applications.

The evaluation results also show that remote execution can be a significant source of energy savings for applications such as speech recognition, document processing, and language translation. However, they also reveal that caution must be taken when deciding where to locate functionality: the energy cost of additional network activity may often be greater than the savings achieved by reduced CPU usage. Often, the availability of network, CPU, and file cache resources significantly impacts the optimal placement of functionality.

Finally, the results contained in this dissertation show that goal-directed adaptation can meet user-specified goals for battery lifetime that vary by as much as 30%. Further, the use of application resource history improves both the agility and effectiveness of goal-directed adaptation.

9.2 Future work

Energy management remains a relatively new topic in systems research. Consequently, each of the systems on which I have worked has opportunities for further improvement. In the rest of this section, I discuss some of the interesting paths for future research which I have not yet had the time to explore. Next, I describe possible improvement to the PowerScope energy profiler. In Sections 9.2.2 and 9.2.3, I discuss the possibility of bringing energy-aware adaptation into the realms of hardware power management and closed-source operating systems. In Sections 9.2.4 and 9.2.5, I speculate on possible extensions to the Spectra remote execution system.

9.2.1 Hybrid energy measurement

One of the drawbacks of PowerScope's profiling approach to energy measurement is that the tool does a poor job of capturing the effects of asynchronous activity such as disk accesses. The energy cost of an I/O request that triggers disk activity is erroneously charged to the processes that execute while the request is being serviced, rather than to the process that originally triggered the disk activity.

PowerScope's accuracy could therefore be improved by incorporating some degree of energy accounting, as discussed in Section 8.1. The first step in this process is identifying which asynchronous activities are significant sources of energy usage—disk accesses and the receipt of network packets are obvious candidates. Then, one could develop a benchmark suite that measures the energy cost of these activities on each hardware platform. Finally, kernel modifications would be necessary to identify when each activity occurs.

Whenever PowerScope's System Monitor takes a sample of system activity, it could identify which activities were occurring during sample collection. It could also record additional information that identifies the amount of each activity generated by a process—for example, the number of disk accesses and network packets received. The Energy Analyzer would then subtract the power and energy cost of asynchronous activities from each sample, so that the energy assigned to a given process reflects only the cost of its own execution. It would assign energy impact of asynchronous activities to the process that initiated them.

The hybrid approach promises to incorporate the best features of accounting-based and profile-based energy measurement. Energy accounting provides the ability to correctly assign the energy costs of asynchronous activities, while profiling provides accurate energy measurements without the need for complex models of energy consumption. Profiling also provides the ability to differentiate subtle differences in power measurement within a single hardware state; for example, the energy cost of executing different instructions. A combination of the two techniques should therefore prove more accurate than when either is used in isolation.

9.2.2 Application-aware power management

This dissertation has explored only two dimensions of energy-aware adaptation: application fidelity and remote execution. Many other possible dimensions exist—of these, the most obvious is hardware power management.

Current approaches to hardware power management make implicit tradeoffs between performance and energy conservation. Modern processors reduce their clock frequency to save energy. Hard disks spin down after periods of inactivity, saving energy but increasing the time needed to service the next request. Wireless networks interfaces disable receivers for short periods of time, saving energy at the cost of reduced throughput.

Yet, current approaches to hardware power management typically use only a single policy for balancing energy and performance, typically hoping to exploit a knee in the energy-performance curve. For example, the voltage scheduling algorithms discussed in Section 8.2.2 all attempt to maximally reduce energy consumption without significantly affecting the performance observed by the user. They do not adjust for the user's dynamically changing priorities for performance and energy conservation.

Using a single policy seems unnecessarily restrictive. A user who plans to operate on battery power for only a short time wants to maximize performance. In contrast, a user who wishes to accomplish the maximum possible quantity of work while operating on battery power will be willing to sacrifice some performance for decreased energy usage. In addition, the optimum tradeoff between energy usage and performance may vary depending upon application workload. For example, when the user is running a remote shell application, the performance impact of disabling the network interface during idle periods may have a large impact on application quality, but the perceived impact of lowering the client CPU speed may be negligible. A coordinated policy for trading energy usage and performance should therefore consider the possible benefits in terms of reduced energy usage and the possible costs in terms of reduced application performance before making power management decisions for individual hardware components.

Since these considerations are essentially identical to those considered in this dissertation for application fidelity and remote execution, it seems likely that Odyssey could be successfully extended to manage hardware power states. Odyssey's knowledge of desired battery lifetime and ability to monitor energy supply and demand enable it to make appropriate tradeoffs between energy consumption and application performance. For example, when battery levels are critical, it could use a more aggressive policy for setting clock frequency, spinning down the disk, and managing network connectivity.

While extending Odyssey to include hardware power management certainly seems feasible, the interaction between data fidelity and hardware power management may be complex. For instance, when the demand for energy exceeds supply, it may be hard to decide whether to lower application fidelity or use more aggressive power management strategies. Answering this question precisely will require the ability to estimate the potential energy benefits of various hardware power management strategies, as well as the ability to estimate their effect on application performance.

9.2.3 Support for adaptation in closed-source environments

Chapter 5 explored the feasibility of trading application fidelity for energy conservation in closed-source environments. It showed that for at least one popular application, i.e. PowerPoint, one can significantly decrease energy usage by lowering fidelity. The next logical step in this work is to implement system support for energy-aware adaptation in a closed-source environment, specifically the Windows family of operating systems.

Odyssey currently has two features that would help in the task of porting system support to Windows. First, the Viceroy is already implemented as a user-level process. One could replace the system call interface with a library-based implementation that uses IPC to communicate with the Viceroy process. Second, Odyssey does not currently enforce its resource decisions—the cooperative model of resource management is much easier to support in a closed-source environment.

The most challenging part of porting system support for energy-aware adaptation to a closed-source environment would be the implementation of the resource measurers. Since these components perform detailed measurements of per-process CPU, network, and power usage, they are most logically placed in the kernel. However, a user-space implementation is not infeasible. For example, Odyssey could use ACPI to measure power usage and check battery status.

9.2.4 Extensions to Spectra

As Spectra is a relatively young system, there are a number of areas in which it could be improved. My experience with the system suggests that making predictions often involves tradeoffs between speed and accuracy. For example, when estimating remote CPU availability, Spectra can either use a slightly stale cached value, or it can query the server to obtain more accurate information. If the difference between possible alternatives is slight, as for example with short-running operations, Spectra would do better to make a "quick and dirty" decision using the stale, cached data. However, when possible alternatives differ significantly, Spectra should invest more effort to choose the optimal alternative. This suggests that Spectra itself should be adaptive—it should balance the amount of effort used to decide between alternatives against the possible benefit of choosing the best alternative.

Another interesting area of future research is server provisioning. When a set of remote servers is servicing multiple clients, load-balancing may improve performance. Additionally, Spectra might make more accurate predictions if it were to anticipate and adjust for queueing delays on a server when multiple clients are requesting the same service.

One could also improve Spectra's execution model by supporting parallel execution of functionality on multiple machines. This could provide considerable benefit for applications like Pangloss-lite, which have multiple code components which are not sequentially dependent. If Spectra were to execute these components in parallel, it could provide results superior to those achieved in Section 7.3.8.

Since resource logs can grow quite large for complex operations, Spectra would benefit from methods that compress log data without sacrificing significant semantic content. Finally, there are a number of additional applications which could benefit from Spectra's remote execution services, including rendering for virtual reality and compilation.

9.2.5 Proactive service management

Spectra attempts to select the optimal location and fidelity for application execution given the current availability of resources in its environment. One can imagine a better system, one which is *proactive*. Such a system would not take the resource environment as a given—instead, it would anticipate how it could change the environment to make it more favorable for application execution.

For example, consider a situation where Spectra must repeatedly decide to perform an operation on one of two servers: the first has a faster processor and a cold file cache; the second has is slower but has necessary files cached locally. Currently, Spectra would always select the server with the slower processor if cache effects enable it to achieve faster response times. However, if Spectra were proactive, it would simultaneously execute an operation on the slower server and also warm the file cache of the faster server. The next time the operation is executed, Spectra would produce an even faster response than with either of the options possible before.

The key to such behavior is proactive service management. Traditional service discovery protocols generates a list of candidate servers for a given operation. Given such a list, Spectra currently selects the best server to perform the operation. However, Spectra could be modified to identify possible corrective actions which would improve the desirability of each server. If a set of corrective actions would make a server more desirable than the one selected, Spectra would initiate those actions. When Spectra next executes the operation, the application will benefit from the improved resource environment. Possible corrective actions include warming the file cache, making CPU and network reservations, and switching to a different network interface.

9.3 Closing remarks

It appears likely that battery energy will continue to be a significant constraint in the design of mobile systems for the foreseeable future. Since size and weight considerations limit battery capacity, reducing system energy usage will continue to be a primary concern.

Energy management should be addressed at every layer of the system. Low-power circuit design and hardware power management provide only a partial solution. The higher levels of the system can also make considerable contributions towards energy conservation. The results in this dissertation show that the use of fidelity reduction and remote execution can significantly extend the battery lifetimes of mobile systems. Further, the results show that energy conservation efforts at different levels of the system are complementary—for example, fidelity reduction improves the effectiveness of hardware power management by increasing the opportunity to put devices in low-power states.

Inevitably, energy management involves implicit tradeoffs between energy conservation, performance, and application quality. Mobile systems should provide maximum flexibility by embracing these tradeoffs. When battery lifetime is critical, they should optimize their behavior for energy conservation. When battery lifetime is unimportant, they should maximize performance and quality. Energy-aware adaptation provides the ability to realize both of these goals. It therefore will be a vital component of a complete energy management solution in future mobile systems.

Bibliography

[1] Adjie-Winoto, W., Schartz, E., Balakrishnan, H., and Lilley, J. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–216, Kiawah Island, SC, December 1999.

[2] Amiri, K., Petrou, D., Ganger, G., and Gibson, G. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[3] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, Saint-Malo, France, October 1997.

[4] Basney, J. and Livny, M. Improving goodput by co-scheduling CPU and network capacity. *International Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[5] Bellosa, F. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[6] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 83–94, Vancouver, Canada, June 2000.

[7] Burd, T. D. and Broderson, R. W. Processor design for portable systems. *Journal of VLSI Signal Processing*, 13(2/3), August/September 1996.

[8] Chase, J. S., Anderson, D. C., Thakar, P. N., Vahdat, A. M., and Doyle, R. P. Manging energy and server resources in hosting clusters. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, pages 103–116, Banff, Canada, October 2001.

[9] Chatterjee, S., Sydir, J., Sabata, B., and Lawrence, T. Modeling applications for adaptive QoS-based resource management. In *Proceedings of the 2nd IEEE High-Assurance System Engineering Workshop (HASE97)*, August 1997.

[10] Chen, B., Jamieson, K., Balakrishnan, H., and Morris, R. Span: An energy-efficient coordination algorithm for topology maintenance in Ad Hoc wireless networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MOBICOM '01)*, pages 85–96, Rome, Italy, July 2001.

[11] Cignetti, T. L., Komarov, K., and Ellis, C. S. Energy Estimation Tools for the Palm. In *Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Boston, MA, August 2000.

[12] Dallas Semiconductor Corp., 4401 South Beltwood Parkway, Dallas, TX. *DS2437 Smart Battery Monitor*, 1999.

[13] de Lara, E., Wallach, D. S., and Zwaenepoel, W. Opportunities for bandwidth adaptation in Microsoft Office documents. In *Procceedings of the 4th USENIX Windows Systems Symposium*, Seattle, WA, August 2000.

[14] de Lara, E., Wallach, D. S., and Zwaenepoel, W. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, March 2001.

[15] Douglis, F., Cáceres, R., Kaashoek, F., Li, K., Marsh, B., and Tauber, J. Storage alternatives for mobile computers. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 25–37, Monterey, CA, November 1994.

[16] Douglis, F., Krishnan, P., and Bershad, B. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pages 121–137, Ann Arbor, MI, April 1995.

[17] Douglis, F., Krishnan, P., and Marsh, B. Thwarting the power-hungry disk. In *Proceedings of 1994 Winter USENIX Conference*, pages 293–307, San Francisco, CA, January 1994.

[18] Ellis, C. S. The case for higher-level power management. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 162–167, Rio Rico, AZ, March 1999.

[19] Farkas, K. I., Flinn, J., Back, G., Grunwald, D., and Anderson, J. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, Santa Clara, CA, June 2000.

[20] Flautner, K., Reinhardt, S., and Mudge, T. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MOBICOM '01)*, pages 260–271, Rome, Italy, July 2001.

[21] Flinn, J., de Lara, E., Satyanarayanan, M., Wallach, D. S., and Zwaenepoel, W. Reducing the energy usage of office applications. In *Proceedings of the FIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[22] Flinn, J., Farkas, K. I., and Anderson, J. *Power and Energy Characterization of the Itsy Pocket Computer (Version 1.5)*. Compaq Western Research Laboratory, February 2000. Technical Note TN-56.

[23] Fox, A., Gribble, S. D., Brewer, E. A., and Amir, E. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 160–170, Cambridge, MA, October 1996.

[24] Frederking, R. and Brown, R. D. The Pangloss-Lite machine translation system. In *Expanding MT Horizons: Proceedings of the Second Conference of the Association for Machine Translation in the Americas*, pages 268–272, Montreal, Canada, 1996.

[25] Goel, A., Steere, D., Pu, C., and Walpole, J. Adaptive resource management via modular feedback control. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[26] Golding, R., Bosch, P., Staelin, C., Sullivan, T., and Wilkes, J. Idleness is not sloth. In *Proceedings of the Winter USENIX Conference*, pages 201–212, New Orleans, LA, January 1995.

[27] Govil, K., Chan, E., and Wasserman, H. Comparing algorithms for dynamic speed setting of a low-power CPU. In *Proceedings of the 1st ACM International Conference on Mobile Computing and Networking (MOBICOM '95)*, pages 13–25, Berkeley, CA, November 1995.

[28] Greenawalt, P. Modeling power management for hard disks. In *Proceedings of the Symposium on Modeling and Simulation of Computer Telecommunication Systems*, pages 62–66, Durham, NC, January 1994.

[29] Grunwald, D., Levis, P., III, C. B. M., Neufeld, M., and Farkas, K. I. Policies for dynamic clock scheduling. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, October 2000.

[30] Haarsten, J. C. The Bluetooth radio system. *IEEE Personal Communications*, 7(1):28–36, February 2000.

[31] Hamburgen, W. R., Wallach, D. A., Viredaz, M. A., Brakmo, L. S., Waldspurger, C. A., Bartlett, J. F., Mann, T., and Farkas, K. I. Itsy: Stretching the bounds of mobile computing. *IEEE Computer*, 13(3):28–35, April 2001.

[32] Hunt, G. C. and Scott, M. L. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating System Design and Implemetation (OSDI)*, pages 187–200, New Orleans, LA, February 1999.

[33] IBM Corporation, http://www.ibm.com/. *IBM ThinkPad A Series Notebooks*, May 2001.

[34] Ikeda, T. ThinkPad low-power evolution. In *IEEE International Symposium on Low-Power Electronics*, pages 6–7, October 1995.

[35] Intel Corporation and Microsoft Corporation. *Advanced Power Management (APM) BIOS Interface Specification*, February 1996.

[36] Intel, Microsoft, and Toshiba. *Advanced Configuration and Power Interface Specification*, February 1998. http://www.teleport.com/˜acpi/.

[37] Jacobson, V. Congestion avoidance and control. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 314–329, Stanford, CA, August 1988.

[38] Jones, M. B., Leach, P. J., Draves, R. P., and Barrera, J. S. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 12–17, Orcas Island, WA, May 1995.

[39] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., Gifford, D. K., and Kaashoek, M. F. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain, CO, December 1995.

[40] Kistler, J. J. and Satyanarayanan, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[41] Klaiber, A. The technology behind Crusoe processors. Technical report, Transmeta Corporation, January 2000.

[42] Klass, B., Thomas, D. E., Schmit, H., and Nagle, D. F. Modeling inter-instruction energy effects in a digital signal processor. In *Proceedings of the Power-Driven Microarchitecture Workshop*, Barcelona, Spain, June 1998.

[43] Kravets, R. and Krishnan, P. Power management techniques for mobile communication. In *Proceedings of The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM '98)*, pages 157–168, Dallas, TX, October 1998.

[44] Kravets, R., Schwan, K., and Calvert, K. Power-aware communication for mobile computers. In *Proceedings of The 6th International Workshop on Mobile Multimedia Communication*, San Diego, CA, November 1999.

[45] Kremer, U., Hicks, J., and Rehg, J. M. Compiler-directed remote task execution for power management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, PA, October 2000.

[46] Kunz, T. and Omar, S. A mobile code toolkit for adaptive mobile applications. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications*, pages 51–59, Monterey, CA, December 2000.

[47] Lebeck, A. R., Fan, X., Zeng, H., and Ellis, C. S. Power aware page allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 115–126, Cambridge, MA, November 2000.

[48] Lee, C., Lehoczky, J., Rajkumar, R., and Siewiorek, D. On quality of service optimization with discrete QoS options. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1999.

[49] Lee, M. T.-C., Tiwari, V., Malik, S., and Fujita, M. Power analysis and low-power scheduling techniques for embedded DSP software. *IEEE Transactions on VLSI Systems*, 5(1):123–135, March 1997.

[50] Li, K. Towards a low power file system. Master's thesis, Computer Science Division, University of California at Berkeley, May 1995.

[51] Li, K., Kumpf, R., Horton, P., and Anderson, T. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the 1994 Winter USENIX Conference*, pages 279–291, San Francisco, CA, January 1994.

[52] Li, Q., Aslam, J., and Rus, D. Online power-aware routing in wireless ad hoc networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MOBICOM '01)*, pages 97–107, Rome, Italy, July 2001.

[53] Lorch, J. R. A complete picture of the energy consumption of a portable computer. Master's thesis, Department of Computer Science, University of California at Berkeley, 1995.

[54] Lorch, J. R. and Smith, A. J. Apple Macintosh's energy consumption. *IEEE Micro*, 18(6):54–63, November/December 1998.

[55] Lorch, J. R. and Smith, A. J. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, Cambridge, MA, June 2001.

[56] Lu, Y.-H., Chung, E.-Y., Simunic, T., Benini, L., and De Micheli, G. Quantitative comparison of power management algorithms. In *Proceedings of Design, Automation, and Test in Europe*, pages 157–161, Paris, France, March 2000.

[57] Lu, Y.-H. and De Micheli, G. Adaptive hard disk power management on personal computers. In *Proceedings of the 9th Great Lakes Symposium on VLSI*, pages 50–53, Ypsilanti, MI, March 1999.

[58] Lu, Y.-H., Simunic, T., and De Micheli, G. Software controlled power management. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pages 157–161, Rome, Italy, May 1999.

[59] Marsh, B., Douglis, F., and Krishnan, P. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Sciences*, pages 451–60, Maui, HI, 1994.

[60] Marsh, B. and Zenel, B. Power measurements of typical notebook computers. Technical Report MITL-TR-110-94, Matsushita Information Technology Laboratory, May 1994.

[61] Martin, T. and Siewiorek, D. A power metric for mobile systems. In *Proceedings of the 1996 International Symposium on Lower Power Electronics and Design*, pages 37–42, Monterey, CA, August 1996.

[62] Martin, T. L. *Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computing*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.

[63] McKenna, D. A methodology for evaluating mobile computing devices. Technical report, Transmeta Corporation, February 2000.

[64] Narayanan, D., Flinn, J., and Satyanarayanan, M. Using history to improve mobile application adaptation. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 30–41, Monterey, CA, August 2000.

[65] Narayanaswami, C. and Raghunath, M. T. Application design for a smart watch with a high resolution display. In *Procceedings of the Fourth International Symposium on Wearable Computers (ISWC'00)*, Atlanta, Georgia, October 2000.

[66] Neugebauer, R. and McAuley, D. Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS. In *Procceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.

[67] Nichols, D. Using idle workstations in a shared computing environment. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 5–12, Austin, TX, November 1987.

[68] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 276–287, Saint-Malo, France, October 1997.

[69] Othman, M. and Hailes, S. Power conservation strategy for mobile computers using load sharing. *Mobile Computing and Communications Review*, 2(1), January 1998.

[70] Palm Corporation, http://www.palm.com/. *Palm m505 Handheld*.

[71] Pering, T., Burd, T., and Broderson, R. Dynamic voltage scaling and the design of a low-power microprocessor system. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Monterey, CA, August 1998.

[72] Pering, T., Burd, T., and Broderson, R. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Rapallo, Italy, July 2000.

[73] Pillai, P. and Shin, K. G. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, pages 89–102, Banff, Canada, October 2001.

[74] Qualcomm Inc., San Diego, CA. *Eudora Macintosh User Manual*, 1993.

[75] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. A resource allocation model for QoS management. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

[76] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. Practical solutions for QoS-based resource allocation problems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1998.

[77] Rudenko, A., Reiher, P., Popek, G. J., and Kuenning, G. H. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1):19–26, January 1998.

[78] Rudenko, A., Reiher, P., Popek, G. J., and Kuenning, G. H. The Remote Processing Framework for portable computer power saving. In *Proceedings of the ACM Symposium on Applied Computing*, San Antonio, TX, February 1999.

[79] SBS Implementers Forum, http://www.sbs-forum.org/. *Smart Battery Data Specification, Revision 1.1*, December 1998.

[80] Schlosser, S., Griffin, J., Nagle, D., and Ganger, G. Designing computer systems with MEMS-based storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 1–12, Cambridge, MA, November 2000.

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

[82] Simunic, T., Benini, L., and De Micheli, G. Energy-efficient design of battery-powered embedded systems. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 212–217, November 1999.

[83] Simunic, T., Benini, L., Glynn, P., and Micheli, G. D. Dynamic power management for portable systems. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM '00)*, pages 11–19, Boston, MA, August 2000.

[84] Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J. A feedback-driven proportional allocator for real-rate scheduling. In *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 145–158, New Orleans, LA, February 1999.

[85] Stemm, M. and Katz, R. H. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Science, Special Issue on Mobile Computing*, 80(8):1125–1131, August 1997.

[86] Tiwari, V., Malik, S., and Wolfe, A. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4), December 1994.

[87] Tiwari, V., Malik, S., Wolfe, A., and Tien-Chien, L. M. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2):1–18, August 1996.

[88] USAR Systems, Inc., 568 Broadway, Suite 405, New York, NY. *USAR ACPITroller II - Zero-Power ACPI KBC with Built-in Smart Battery System Manager*, 1999.

[89] Vahdat, A., Lebeck, A. R., and Ellis, C. S. Every Joule is precious: A case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[90] Viezades, J., Guttman, E., Perkins, C., and Kaplan, S. *Service Location Protocol*. IETF RFC 2165, June 1997.

[91] Vijaykrishnan, N., Kandemir, M., Irwin, M. J., Kim, H. S., and Ye, W. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 95–106, Vancouver, Canada, June 2000.

[92] Viredaz, M. A. *The Itsy Pocket Computer Version 1.5 User's Manual*. Compaq Western Research Laboratory, July 1998. WRL Technical Note TN-54.

[93] Viredaz, M. A. and Wallach, D. A. *Power Evaluation of a Handheld Computer: A Case Study*. Compaq Western Research Laboratory, May 2001. WRL Technical Note 2001-1.

[94] Waibel, A. Interactive translation of conversational speech. *IEEE Computer*, 29(7):41–48, July 1996.

[95] Waldspurger, C. A. and Weihl, W. E. Lottery scheduling: Flexible proportional-share resource mangement. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, Monterey, CA, November 1994.

[96] Warren, J. Interaction between architecture performance and power consumption in mobile systems. Master's thesis, Carnegie Mellon University, March 2000.

[97] Weiser, M., Welch, B., Demers, A., and Shenker, S. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 13–23, Monterey, CA, November 1994.

[98] Wilkes, J. Predictive power conservation. Technical Report HPL-CSP-92-5, Hewlett-Packard Laboratories, February 1992.

[99] Xu, Y., Heidemann, J., and Estrin, D. Geography-informed energy conservation for ad hoc routing. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MOBICOM '01)*, pages 70–84, Rome, Italy, July 2001.

[100] Zhang, X., Wang, Z., Gloy, N., Chen, J. B., and Smith, M. D. System support for automated profiling and optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.