Reducing the Energy Usage of Office Applications

Jason Flinn[1], Eyal de Lara[2], M. Satyanarayanan[1], Dan S. Wallach[3], and Willy Zwaenepoel[3]

[1] School of Computer Science, Carnegie Mellon University
[2] Department of Electrical and Computer Engineering, Rice University
[3] Department of Computer Science, Rice University

Abstract. In this paper, we demonstrate how component-based middleware can reduce the energy usage of closed-source applications. We first describe how the Puppeteer system exploits well-defined interfaces exported by applications to modify their behavior. We then present a detailed study of the energy usage of Microsoft's PowerPoint application and show that adaptive policies can reduce energy expenditure by 49% in some instances. In addition, we use the results of the study to provide general advice to developers of applications and middleware that will enable them to create more energy-efficient software.

1 Introduction

Battery energy is one of the most critical resources for mobile computers. Despite considerable research effort, no silver bullet for reducing energy usage has yet been found. Instead, a comprehensive effort is needed—one that addresses all layers of the system: hardware, operating system, middleware, and applications.

One promising piece of a comprehensive solution is *energy-aware adaptation*, in which applications modify their behavior to reduce their energy usage when battery levels are critical. The potential benefits of energy-aware adaptation were first explored in the context of Odyssey [6]. That work showed that energy-aware applications can often significantly extend the battery lifetimes of the laptop computers on which they operate by trading fidelity, an application-specific metric of quality, for reduced energy usage. However, since only multimedia, open-source applications running on the Linux operating system were studied, it was not clear that this technique would be relevant to the Windows office applications that users commonly run on laptop computers.

Several important questions follow: Can one show significant energy reductions for the type of office applications that users most commonly execute on laptop computers? Is it possible to add energy-awareness to applications for which source code is unavailable? Is this approach valid for applications executing on closed-source operating systems (i.e. Windows)?

In this paper, we describe how Puppeteer [2], a component-based middleware system, allows us to add energy-awareness to applications. Puppeteer takes

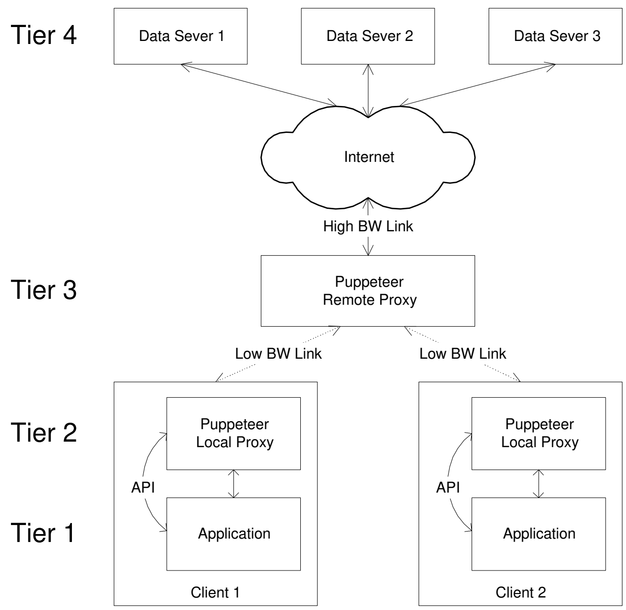


Fig. 1. Puppeteer architecture

advantage of well-defined interfaces exported by applications to modify their behavior without source code modification. We demonstrate the feasibility of this approach by studying energy saving opportunities for Microsoft's popular PowerPoint application. By using Puppeteer to distill multimedia content from presentations stored on remote servers, we reduce the energy needed to load presentations by 49%. Further, the benefits of distillation extend to reducing energy use while the document is being edited and saved.

In addition, we identify several instances where PowerPoint can be made more energy-efficient. From these specific instances, we present general advice for developers of applications and middleware that will enable them to create more energy-efficient software in the future.

In the next section, we provide an overview of Puppeteer. In Section 3, we describe our energy measurement methodology. Section 4 shows how we can modify PowerPoint behavior to reduce energy usage when battery levels are critical. Section 5 discusses opportunities for making PowerPoint and similar applications more energy-efficient. In Section 6, we speculate on the applicability of component-based adaptation to applications other than PowerPoint, and in the remainder of the paper, we discuss related work and conclude.

2 Puppeteer

Puppeteer is a system for extending component-based applications, such as Microsoft Office or Internet Explorer, to support adaptation in mobile environ-

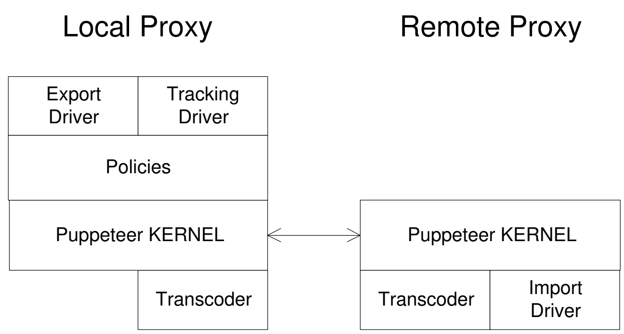


Fig. 2. Puppeteer local and remote proxy architectures

ments. It uses the exported APIs of applications and the structured nature of the documents they manipulate to implement adaptation without changing source code. It supports subsetting adaptation (where only a portion of the elements of a document are provided to the application; for instance, the first page), and versioning adaptation (where a different version of some of the elements is provided to the application; for instance, a low-resolution version of an image). It uses the document structure to extract subsets and versions of the document. Furthermore, Puppeteer uses the exported APIs of the applications to incrementally increase the subset of the document or improve the version of the elements available to the application. For instance, it uses the APIs to insert additional pages or higher-resolution images into the application.

Figure 1 shows the four-tier Puppeteer system architecture. It consists of applications to be adapted, Puppeteer local proxies, the Puppeteer remote proxy, and the data servers. Applications and data servers are completely unmodified. Data servers can be arbitrary repositories of data such as Web servers, file servers or databases. All communication between applications and data servers goes through the Puppeteer local and remote proxies, which work together to perform the adaptation. The Puppeteer local proxy manipulates the running application through a subset of the application's external programming interface.

Figure 2 shows the architecture of the Puppeteer local and remote proxies. It consists of application-specific policies, component-specific drivers, type-specific transcoders and an application-independent kernel. The Puppeteer local proxy is in charge of executing adaptation policies. The Puppeteer remote proxy is responsible for parsing documents, exposing their structure, and transcoding components as requested by the local proxy.

The adaptation process in Puppeteer is divided roughly into three stages: parsing the document to uncover the structure of the data, fetching selected components at specific fidelity levels, and updating the application with the newly fetched data.

When the user opens a document, the Puppeteer remote proxy instantiates an *import driver* for the appropriate document type. The import driver parses the document, and extracts its component structure (the *skeleton*, a tree structure) and the data associated with the nodes in the tree. Puppeteer then transfers the document's skeleton to the Puppeteer local proxy. The policies running on the local proxy fetch an initial set of elements from within the skeleton at a specified fidelity. These policies may be static, or may depend on *tracking drivers* that detect the occurrence of certain events, such as moving the mouse over an image, causing the image to be loaded.

Puppeteer uses the *export driver* for the particular document type to supply this set of components to the application as though it had the full document at its highest level of fidelity. The application, believing that it has finished loading the document, returns control to the user. Meanwhile, Puppeteer knows that only a fraction of the document has been loaded and will use the application's external programming interface to incrementally fetch remaining components or upgrade their fidelity.

3 Measurement methodology

All measurements were collected in a client-server environment. Microsoft PowerPoint 2000 executes on the client: a 233 MHz Pentium IBM 560X laptop with 96 MB of memory. 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 Lucent WaveLan wireless 802.11 network.

We measured client energy usage with a HP3458a digital multimeter. When collecting data, we attached the multimeter's probes in series with the external power input of the client laptop and removed the client's battery to eliminate the effects of charging. We also connected an output pin of the client's parallel port to the external trigger input of the multimeter—this allowed the multimeter and client to coordinate during the taking of measurements.

We created a dynamic library that allows a calling process to precisely indicate the start and end of measurements. The process 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 1357.5 times per second. When the measured event completes, the application calls `stop_measuring`, which returns the elapsed time since `start_measuring` was called.

To calculate total energy usage, we 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 (which varies by less than 0.25% in the course of an experiment) yields the average power usage. This is multiplied by the elapsed time to calculate total energy usage.

We assume aggressive power management policies. All measurements were taken using a disk-spindown threshold of 30 seconds (the minimum allowed by Windows NT). Unless otherwise noted, the wireless network uses standard 802.11 power management. Audio input and output is disabled. However, the display is

Presentation	Full-Quality Size (MB)	Distilled Size (MB)	Ratio
A	15.02	1.67	0.11
B	11.42	0.47	0.04
C	7.26	0.83	0.11
D	3.11	3.11	1.00
E	2.23	1.32	0.59
F	1.72	0.11	0.07
G	1.07	0.36	0.34
H	0.87	0.75	0.86
I	0.20	0.20	1.00
J	0.08	0.08	1.00

Fig. 3. Sizes of sample presentations

not disabled during measurements since PowerPoint is an interactive application; instead it is reduced to minimum brightness.

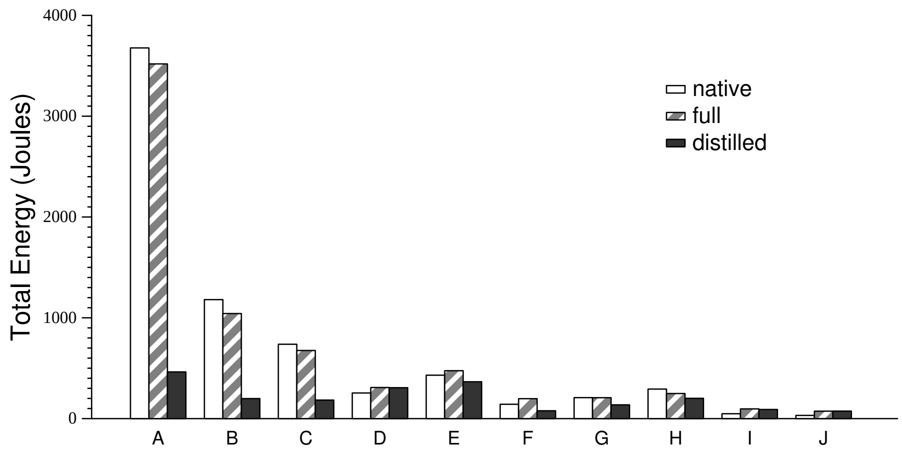
An ideal battery can be modeled as a finite store of energy. If an application expends some amount of energy to perform an activity, the energy supply available for other activities is reduced by that amount. In reality, batteries never behave ideally. As power draw increases, the total energy that can be extracted from the battery decreases. In addition, recovery effects may apply—a reduction in load for a period of time may result in increased battery capacity [11].

In this paper, we assume the ideal model for battery behavior, but note that most of the techniques proposed decrease average power usage. Thus, the gains reported here will be slightly understated.

4 Benefits of adaptation

In this section, we examine whether component-based adaptation can be used to add energy-awareness to Microsoft PowerPoint. As with many office applications, PowerPoint enables users to incorporate increasing amounts of rich multimedia content into 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.

Puppeteer allows us to load a distilled version of a document when battery levels are critical. The distilled version initially omits all multimedia content not on the first or master slide. Placeholders are inserted into the document to



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 remote Puppeteer proxy, with full mode loading the entire presentation and distilled mode loading a lower-quality version. On average, distilled mode uses 60% of the energy of native mode. Each bar represents the mean of five trials—90% confidence intervals are sufficiently small so that they would not be visible on the graph.

Fig. 4. Energy used to load presentations

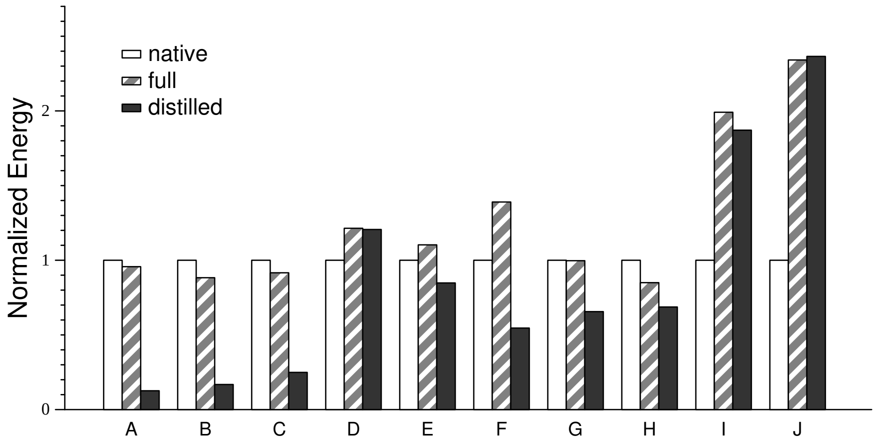
represent omitted objects. Later, if the user wishes to edit or view a component, she may click on the placeholder, and Puppeteer will load the component and dynamically insert it into the document. Thus, when users edit only a subset of a document, the potential for significant energy savings exists.

In Sections 4.1 and 4.2, we measure the impact of distillation on several activities commonly performed in PowerPoint. Then, in Section 4.3 we examine the impact of background tasks such as spell-checking and the Office Assistant. Finally, we quantify the energy benefit of modifying autosave frequency in Section 4.4.

4.1 Loading presentations

We first examined the potential benefits of loading distilled PowerPoint presentations. We assume that presentations are stored on a remote server. There are many reasons to store documents in a remote, centralized location. Multiple users may wish to collaborate in the production of the document. Also, storage space on mobile clients may be limited. Finally, remote storage offers protection against data loss in the event that a mobile computer is damaged or stolen.

We measured the energy used to fetch presentations from the server and render them on the client. We chose a sample set of documents from a database of 1900 presentations gathered from the Web as described by de Lara et al. [1]. From the database, we selected ten documents relatively evenly distributed in



This figure shows the relative energy used to load ten PowerPoint presentations from a remote server, as described in Figure 4. For each data set, results are normalized to the amount of energy used to load the document in native mode.

Fig. 5. Normalized energy used to load presentations

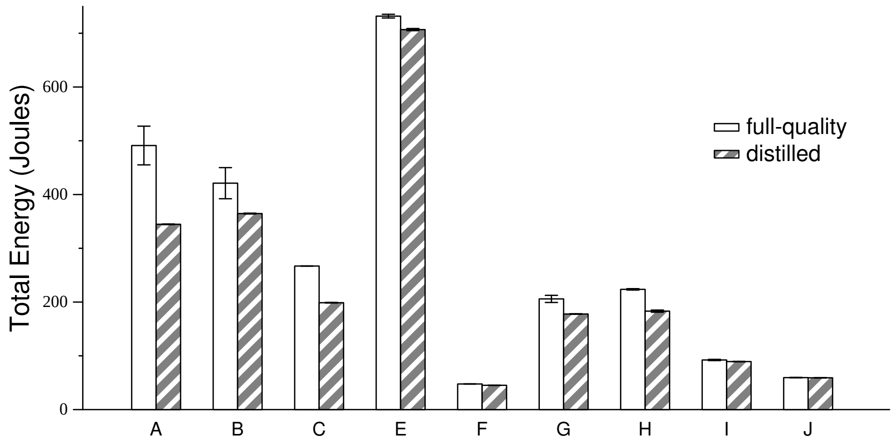
size. Figure 3 shows the sizes of these documents, as well as the reductions achieved by distillation. As might be expected, larger documents tend to have more multimedia content, although there is considerable variation in the data. For three documents (D, I and J), distillation does not reduce document size.

For each document, we first measured the energy used by PowerPoint to load the presentation from a remote Apache Web server (we will refer to this as "native mode"). We also investigated the cost of loading documents from a remote NT file system—these results are not shown since the latency and energy expenditure is significantly greater than when using the Web server.

We then measured the energy used to load each document from a remote Puppeteer proxy running on the same remote machine. We measured two modes of operation: "full", in which the entire document is loaded, and "distilled", in which a degraded version of the document is loaded.

Figure 4 shows the total energy used to fetch the documents using native, full, and distilled modes. In Figure 5 we 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 of native mode.

Interestingly, full mode sometimes uses 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



This figure shows the amount of energy needed to page through a presentation. For each data set, the left bar shows energy use when a full-quality presentation is loaded using native mode, and the right bar shows energy use for a reduced-quality version of the same presentation. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Fig. 6. Energy used to page through presentations

application 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—this issue will be discussed in Section 5.1.

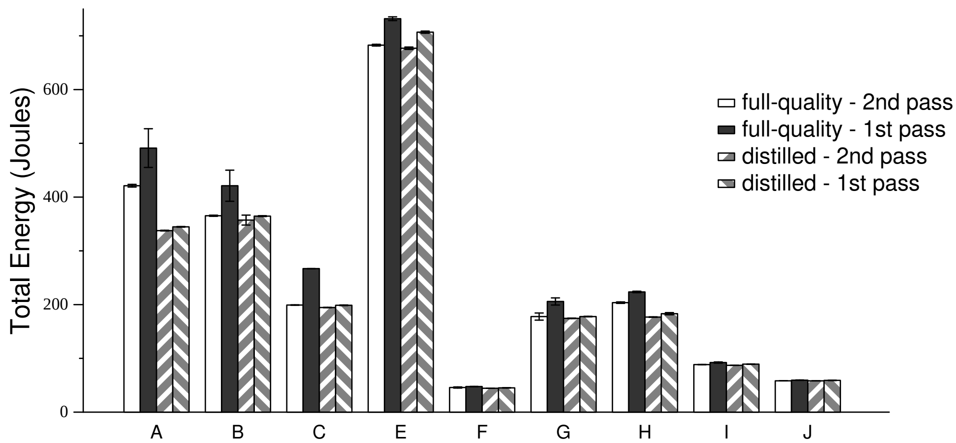
The results in Figures 4 and 5 show that while most documents benefit from distillation, some suffer an energy penalty. If Puppeteer could predict which documents will benefit from distillation, it could distill only those documents and fetch the remaining documents using native mode.

One possible prediction method is to distill only presentations that are larger than a fixed threshold, since 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 does not distill documents I and J, and consequently, uses 52% of the energy of native mode to load the ten documents.

Alternatively, Puppeteer could distill only documents that have a percentage of multimedia content greater than a threshold. As shown in Figure 3, distillation does not reduce the size of three documents. If Puppeteer does not distill these documents, it uses only 51% of the energy of native mode.

4.2 Editing presentations

We next measured how distillation affects the energy needed to edit a presentation. While it is intuitive that loading a smaller, distilled version of a document requires less energy, it is less clear whether distillation also reduces energy use



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 loaded using native mode, and the right two bars show energy use for a reduced-quality version. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Fig. 7. Energy used to re-page through presentations

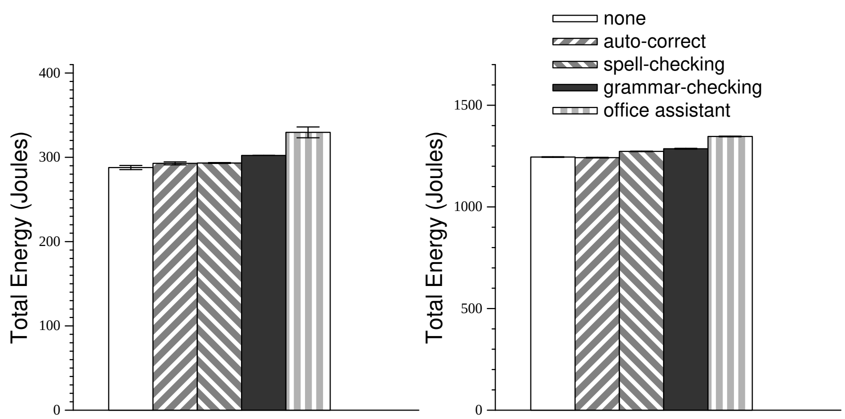
while a document is edited. Although Puppeteer does not yet support reintegration of changes made to a distilled copy of a document, we are currently adding this functionality to the system.

Naturally, energy use depends upon the specific activities that a user performs. While a definitive measurement of potential energy savings would require a detailed analysis of user behavior, we can reasonably estimate such savings by looking at the energy needed to perform common activities.

One very common activity is paging through the slides in a presentation. We created a Visual Basic program which simulates this activity by loading the first slide of a presentation, then sending `PageDown` keystrokes to PowerPoint until all remaining slides are displayed. After each keystroke, the program waits for the new slide to render, then pauses for a second to simulate user think-time.

We measured the energy used to page through both the full-quality and the distilled version of each document. Figure 6 presents these results for nine of the ten presentations in our 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 6, 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 that allows it to quickly re-render the slide, thereby reducing the energy needed for redisplay. This effect is shown in Figure 7, which displays the energy used to page through



(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. The leftmost bar in each graph was measured with no background activities enabled, and the rightmost bar in each graph was measured with all background activities enabled. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Fig. 8. Energy used by background activities during text entry

each document a second time. For comparison, Figure 7 also shows the energy needed to page through each document initially. Comparing the heights of corresponding bars shows that subsequent slide renderings use less energy than initial renderings. Thus, the benefits of distillation are smaller on subsequent traversals: ranging from negligible to 20% with an average value of 5%.

4.3 Background activities

We 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. When battery levels are critical, Puppeteer could disable background processing to extend battery lifetime.

We measured the effect of auto-correction, spell-checking, style-checking, and the Office Assistant (Paperclip). We 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 one.

Figure 8(a) shows the energy used to enter text with no pause; Figure 8(b) shows energy usage with a 100 ms. pause between keystrokes. We first measured energy usage with no background activities, and then successively enabled auto-correction, spell-checking, style-checking, and the Office Assistant. Thus, the difference between any bar in Figure 8 and the bar to its left shows the additional energy used by a specific background activity. For example, the difference between the first two bars in each chart shows the effect of auto-correction.

Auto-correction expends negligible energy when entering text—the additional energy cannot be distinguished from experimental error. Spell-checking and style-checking incur a small additional cost. With no pause between entering characters, these options add a 5.0% energy overhead, but with a 100 ms. pause between characters, the overhead is only 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 300 mW., 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.

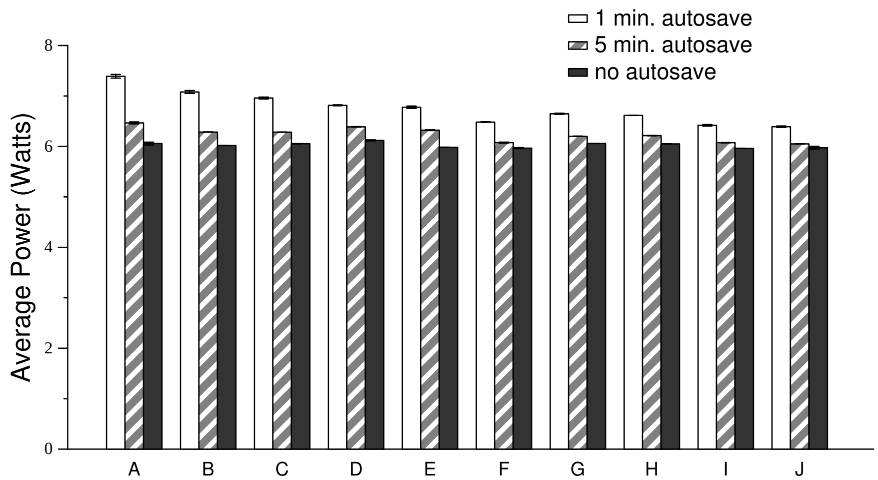
4.4 Autosave

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 configurable. 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, we assume that documents are stored on local disk. We created a Visual Basic program which loads a PowerPoint document, makes a small modification (adds one slide), and then observes power usage for several minutes, during which no further modifications are made. To avoid spurious measurement of initial activity associated with loading and modifying the presentation, the program waits ten minutes after making the modification before measuring power use.

Figure 9 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 shows the effect of specifying a five minute autosave frequency. The final bar shows the effect of disabling autosave—this represents the minimum power drain that can be achieved by modifying the 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 from 5% to 12%, with an average reduction of 8%. The maximum possible benefit is realized when autosave is disabled. As shown by the difference between the first and last bars in each data set, this reduces power usage from 7% to 18%



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.

Fig. 9. Effect of autosave options on application power usage

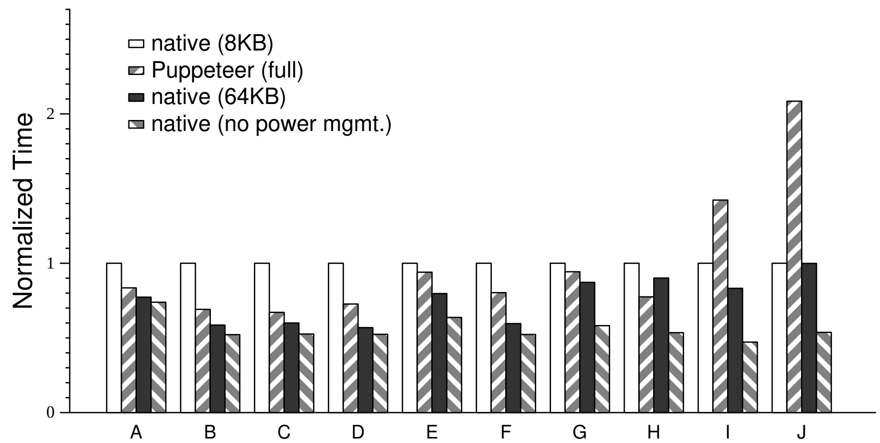
with an average reduction of 11%. Thus, depending upon the user's willingness to hazard data loss in the event of crashes, autosave frequency is a potentially useful dimension of energy-aware adaptation.

5 Energy-efficiency

As shown in the previous section, there are many opportunities to conserve energy usage by modifying PowerPoint behavior. However, these opportunities require one to sacrifice some dimension of application-specific quality in order to extend battery lifetime. Thus, before employing adaptive strategies, one should ensure that the application is as energy-efficient as possible, i.e. that it consumes the minimum amount of energy needed to perform its function.

Increasing software energy-efficiency extends battery lifetime without degrading application quality. Further, it increases the effectiveness of energy-aware adaptation, since the relative savings achieved by adaptive strategies will be greater if the fixed cost of executing the application is lower.

During our study, we discovered several areas in which PowerPoint could be more energy-efficient. In this section, we show that with component-based adaptation, Puppeteer can often increase PowerPoint's energy-efficiency without source code modification. Further, while we observed these areas in the context of a single application, we believe that the principles behind them are general enough so that they can be applied by most software developers. From these



This figure shows how network parameters influence the time needed to load applications from a remote server. The first bar in each data set shows the time needed when a document is loaded from Apache using default network settings. The second bar shows the time needed when the document is loaded from a remote Puppeteer proxy. The third bar shows time to load the document from Apache with 64 KB socket buffer and TCP receive window sizes. The final bar shows time to load the document from Apache with network power management disabled. Each bar represents the mean of five trials—90% confidence intervals are sufficiently small that they would not be visible on the graph.

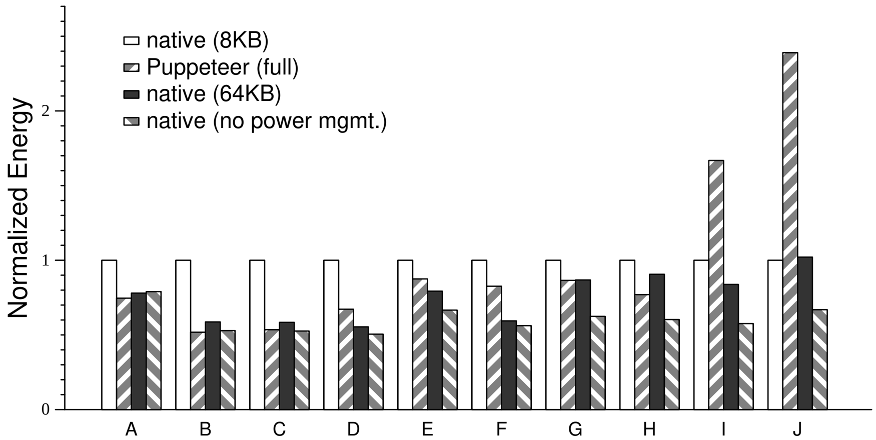
Fig. 10. Effect of power management on time needed to load presentations

specific observations, we formulate some general principles that can be used to make applications and middleware more energy-efficient.

5.1 Transparent power management

Layering of functionality is a well-known software engineering principle. It reduces software complexity by allowing developers to implement and modify each layer without needing to know details about other layers. However, layered implementations sometimes exhibit suboptimal performance because opaque layering precludes optimizations that span multiple layers.

Hardware power management is often implemented using a layered approach. Each device (network, disk, CPU, etc.) individually implements power management algorithms without considering application or other system activity. Typically, such algorithms use fixed timeouts—for example, hard disks enter power saving states after periods of inactivity, and wireless network clients periodically disable their receivers. On the other hand, applications normally do not consider power management when scheduling their activities. In our study of PowerPoint behavior, we have found two instances where this layered approach leads to unnecessary energy expenditure and suboptimal performance.



This figure shows how network parameters influence the energy needed to load applications from a remote server. The scenarios used to generate the bars in each data set are identical to those described in Figure 10. Each bar represents the mean of five trials—90% confidence intervals are sufficiently small that they would not be visible on the graph.

Fig. 11. Effect of power management on energy needed to load presentations

Transparency in network power management Figures 10 and 11 show the normalized time and energy needed to load PowerPoint documents from a remote server. The first bar of each data set shows results for the most naive method—loading the presentation from Apache employing the default network settings, including those for power management. For comparison, the second bar in each data set, shows results when Puppeteer loads the presentation.

Initially, we were greatly surprised that Puppeteer often takes significantly less time and energy to fetch a presentation than when the document is fetched directly from Apache. Puppeteer uses up to 33% less time and 48% less energy than native mode. Since Puppeteer parses the presentation on the server and reconstructs it on the client, the time and energy needed to fetch a document with Puppeteer should be greater. Although this discrepancy puzzled us initially, we now believe that the answer lies in the interaction of network power management and the communication patterns used by the two methods.

To save energy, the wireless network client disables its receiver when no incoming packets are waiting at the network base station. While the receiver is disabled, incoming packets are queued. Every 100 ms., the client restarts its receiver and checks if new packets have arrived. Since the wireless network used in this study has a 2 Mb/s nominal bandwidth, up to 25 KB of data may be queued at the base station while the receiver is disabled. This has the effect of giving the wireless network a high $bandwidth * delay$ product.

Native mode uses a single HTTP/1.1 connection to fetch data from the server, while Puppeteer fetches data using four simultaneous connections. Since the default Windows and Apache settings specify 8 KB socket buffer and TCP window sizes, a single TCP connection cannot fully utilize the wireless network. However, Puppeteer uses four simultaneous connections—this artifact of its implementation allows it to achieve far better network utilization.

The third bar in each data set shows the effect of increasing the socket buffer and TCP window sizes to 64 KB. This compensates for the high *bandwidth∗delay* product of the wireless network—native mode now fetches presentations, on average, 26% faster and uses 26% less energy than with the 8 KB defaults. Given these results, we chose to use the 64 KB sizes as the default for measurements presented in this paper (i.e., for native mode in Section 4.1).

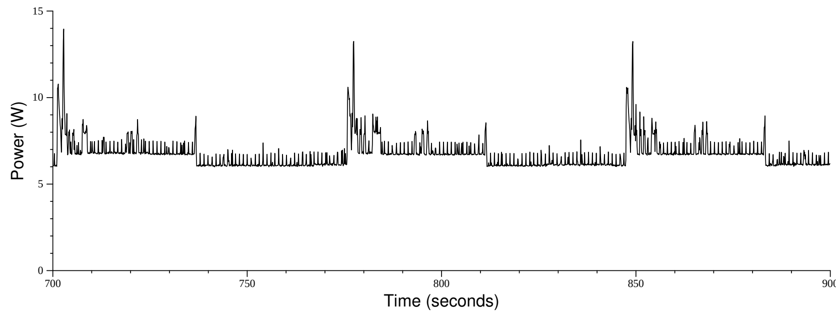
However, adjusting these parameters does not fully compensate for the effect of power management. The last bar in each data set shows results when network power management is disabled. Loading a document without power management uses 22% less time and 18% less energy than when using power management and 64 KB buffers. The remaining performance and energy penalty may be caused by a variety of factors, including TCP ack compression.

These results show the potential benefit of transparent power management. The most naive approach to loading documents incurs a significant time and energy penalty. This penalty can be reduced if the power management layer exposes details about power management strategies. Puppeteer, acting as a proxy for PowerPoint, could then take corrective action by increasing socket buffer and TCP window sizes, or by opening multiple network connections for data transfer. An even more promising approach is for applications and middleware to expose details about their behavior to the power management layer. If Puppeteer were to indicate large data transfers, network power management could be disabled for the duration of the transfer. The benefit of such an approach is shown by the difference between the first and last bars in each data set in Figures 10 and 11.

Transparency in disk power management Transparency can also improve disk power management. To illustrate this, we specified an autosave frequency of one minute and modified a presentation as described in Section 4.4. Figure 12 shows resulting power drain over time. For clarity, we show only a portion of our measurements, ranging from 700 to 900 seconds after the modification.

The variation in power drain can be attributed to the interaction of PowerPoint autosave and disk power management. At approximately 700 seconds, a large power spike is caused by PowerPoint's writing of the AutoRecovery file and the resulting spin-up of the hard drive. After the write completes, the disk spins for 30 seconds—power usage remains relatively steady since the application is idle. Windows then spins down the hard drive, and power usage remains steady at a lower rate for 30 seconds. At approximately 775 seconds, the pattern repeats, since one minute has passed since PowerPoint last wrote the AutoRecovery file.

At first glance, disk power management appears effective since the energy saved by spinning down the disk for 30 seconds is greater than the excess energy caused by transitions. However, with additional knowledge about application behavior, the power management layer could do even better.



This figure shows how PowerPoint power usage varies over time with a specified autosave frequency of one minute. To generate the measurements, a document was loaded into PowerPoint and modified. The x-axis gives the amount of time that has passed since the modification was performed.

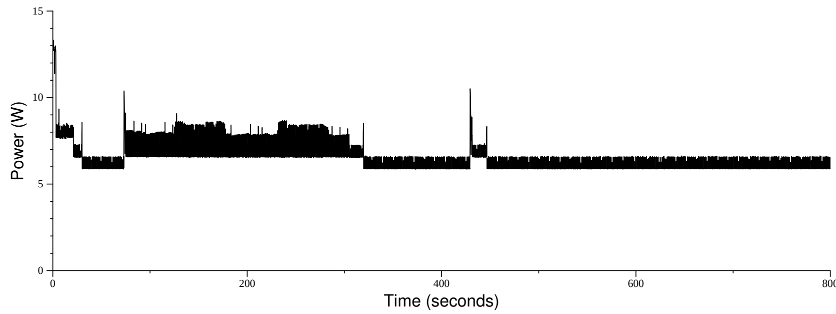
Fig. 12. PowerPoint power usage with 1 minute autosave

The OS saves energy by spinning down the hard drive if the disk remains in a low-power state long enough to counterbalance the energy cost of power-state transitions. Timeout-based power management policies [3, 4, 9, 15] are based on the observation that disk accesses are often closely correlated together in time. Immediately after an access is observed, it is likely that another access will be seen soon. Spinning down the disk is undesirable, since the next access will likely occur before the break-even point for energy savings. However, as time passes without an access, the likelihood of another access happening soon decreases. Eventually, the estimated energy impact of spinning down the disk becomes favorable, and the OS transitions the hard drive to a low-power state.

In Figure 12, the disk power management layer could benefit if Puppeteer were to specify that autosave occurs at regular intervals and is uncorrelated with other activity. If Puppeteer indicates when autosave is occurring, the power management algorithm could omit these disk accesses from its prediction algorithm (since they are uncorrelated with other accesses). In Figure 12, the disk would spin down immediately after autosave completes, saving 19.2 Joules per autosave. This represents a significant 4% relative reduction in energy use. Potentially, the power management layer could also use such knowledge to anticipate future behavior—for example, it might forego spinning down the disk when it anticipates that an autosave will occur in the near future.

5.2 Minimizing low-priority activities

Complex applications such as PowerPoint often perform background activities using low-priority threads. Examples include animation effects that lead to more pleasing user interfaces and data reorganizations that optimize future performance. When a computer operates on wall-power, such activities are scheduled



This figure shows how PowerPoint power usage varies after loading a document and performing no other activity. The x-axis gives the amount of time that has passed since the document was loaded.

Fig. 13. PowerPoint power usage after loading a presentation

whenever system resources such as the CPU are not needed by higher priority tasks—use of these resources is essentially free if they are uncontended.

However, when a computer operates on battery power, background computation is no longer free because it expends the finite supply of energy. Thus, developers should reevaluate these activities and only perform them if the potential benefit is greater than the cost of reduced battery lifetime.

Our study of PowerPoint revealed two instances in which low-priority tasks account for significant energy usage. The first is animation of the Office Assistant as discussed in Section 4.3. Even when the user is performing no activity, enabling the Assistant increases energy usage 4.4%, most likely due to animation of the Assistant's graphic image. When Puppeteer detects that the client is operating on battery power, it could adaptively disable the Office Assistant to save power.

Such animation effects are not restricted to PowerPoint. Web browsers, for instance, usually have a number of graphic devices to indicate activity while Web data is being fetched. While such animation can create more pleasing user experiences, application developers should carefully consider energy costs when designing interfaces. It may be more advisable to curtail unnecessary animation if a computer is operating on battery power.

The second instance of background energy consumption is more complex. After PowerPoint loads a document, we observe a variable amount of background energy consumption. Because this activity is extremely correlated with loading a presentation, we attribute it to application activity or OS activity performed on behalf of the application. Without source code, we cannot confirm the precise nature of the activity, but we believe it is caused by PowerPoint creating a backup of the presentation on disk. However, the duration of the activity is longer than one might expect (about 4 minutes to create a 15 MB file)—this may be due to inefficiency or the desire to prevent interference with foreground activity.

This activity has minimal impact on performance, probably because it is as a low-priority task. However, as shown in Figure 13, the energy costs can be substantial. We generated this data by loading a presentation and then performing no further activities. The impact of the initial background activity is shown by increased power usage during the first 300 seconds. For the document shown in Figure 13, the activity increases PowerPoint energy usage by 383 Joules. Average power use during the first five minutes increases by over 20%.

Since the energy costs are substantial, it is unlikely that they can be amortized across future activity if PowerPoint does not execute for a long period of time. When on battery power, it may therefore be advisable to forego these activities. Alternatively, one could perform them in a more efficient manner, perhaps by assigning them a higher priority so they would complete faster. This would save energy by allowing the disk to spend more time in low-power states.

There is a strong correlation between presentation size and the magnitude of the initial energy cost. Thus, when Puppeteer distills a document before loading it, as discussed in Section 4, it reduces the impact of initial background activity. For large documents, loading a distilled version reduces initial energy costs 63%.

5.3 Event-based programming

Periodic activities are common in application and system software. Such activities include polling for event completion and updates of disk files to preserve modifications in the event of a future crash. The performance impact of periodic activities is often minimal. However, the energy impact can be considerably larger. Each time an activity is performed, it consumes energy. In addition, hardware components such as the CPU, network, and disk must be transitioned from low-power states to perform the activity, wasting further energy.

Where possible, a better alternative is to replace periodic activities with event-based implementations. For example, instead of using a polling loop to check for event completion, one can have the event execution trigger a callback function. Such changes, while quite simple, can dramatically reduce energy use.

Our examination of PowerPoint shows that periodic autosave is a significant energy expenditure. Consider Figure 12 as a motivating example. Three large power spikes at 700, 775, and 850 seconds represent energy used to save the AutoRecovery file. Unfortunately, this activity is performed periodically, even though the presentation is not modified during this time period. A more energy-efficient implementation could use an event-driven model where disk updates are performed only after a fixed amount of data has been modified. Although this implementation artifact is best addressed in the application, Puppeteer can fix this problem without modifying application source. When Puppeteer detects that no activity has occurred for a period greater than the current autosave frequency, it can disable autosave until it detects further activity. In Figure 12, this would completely eliminate the power spikes without sacrificing data consistency.

6 Discussion

The previous results show that we can significantly reduce PowerPoint energy usage by modifying application behavior. The key to realizing these benefits is Puppeteer's component-based adaptation strategy, which modifies behavior without access to source code. However, we will be able to generalize these results to other, closed-source applications only if component-based adaptation proves to be widely applicable. Thus, it is useful to examine what characteristics of PowerPoint make it a good candidate for component-based adaptation, and to see if these characteristics exist in other applications.

PowerPoint has two main characteristics that allow Puppeteer to modify its behavior. It has a well-defined data format that lets Puppeteer parse the content of presentations. It also has an external API that allows Puppeteer to trigger application events, for example, redisplay of a slide. The API also notifies Puppeteer of external events such as change of focus. On the other hand, PowerPoint has no explicit interfaces for power management or degrading document quality. This is encouraging since it indicates that an application need not explicitly support energy-awareness to realize its benefits. If its interface is sufficiently rich, energy-awareness may be implemented entirely by proxy.

Based on these observations, what other applications are likely candidates for component-based adaptation? The remaining applications in Microsoft's Office suite are obvious candidates, as they have data formats and APIs similar to PowerPoint's. Web browsers such as Netscape Navigator are also good candidates. HTML is a well-defined data format, and Netscape's remote interface allows external applications to manipulate its behavior [8]. Database applications may also benefit, since they have standard interfaces for manipulating data such as ODBC and SQL. Thus, for many common applications, a component-based approach is likely to prove useful in implementing energy-aware adaptation.

When rich external APIs are unavailable, other approaches may suffice. For example the Visual Proxy [13] takes advantage of the structured nature of interactions between applications and window managers to extend the functionality of a Web browser and a word processor. Such approaches are less convenient than component-based adaptation, but, with some sweat, can extend the range of applicability of proxy-based solutions.

7 Related work

We believe this study is the first to explore how one can reduce the energy usage of office applications. We extend the concept of energy-aware adaptation first introduced in Odyssey [6] to support applications and operating systems where source code is unavailable.

The concept of using distillation to reduce mobile client resource use has been previously explored by Fox et al. [7]. They show that dynamic distillation of data can allow mobile computers to adapt to poor-bandwidth environments.

Several recent research efforts have advocated cooperation between applications and the operating system in managing energy resources. The MilliWatt project [5, 14] is developing a power-based API that allows a partnership between applications and the system in setting energy use policy. Neugebauer and McAuley [12] show how the Nemesis operating system can be extended to provide applications with detailed information about energy usage. Lu et al. [10] propose the development of user-level power managers which could potentially incorporate feedback from applications into power management decisions. Our study of PowerPoint behavior provides further motivation for these projects by demonstrating that significant energy savings could be achieved with transparent power management.

8 Conclusion

We began this paper by asking several questions: Can one show significant energy reductions for the type of office applications that users most commonly execute on laptop computers? Is it possible to add energy-awareness to applications for which source code is unavailable? Is this approach valid for applications executing on closed-source operating systems (i.e. Windows)?

As the results of Section 4 show, the answer to all these questions is "yes". Puppeteer's component-based adaptation approach allows us to modify PowerPoint behavior without access to application or operating system source code. By distilling presentations and excluding multimedia data, we can reduce the energy needed to load presentations from a remote server by up to 49%. Distillation also leads to significant energy savings when editing and saving presentations. Finally, we showed how modifications such as disabling the Office Assistant and lowering autosave frequency can lead to further reductions in energy use.

Our study also revealed several instances where PowerPoint energy-efficiency could be improved. From these specific instances, we extracted general advice for developers who wish to make their applications and middleware more energy-efficient. We showed that transparent power management can lead to reduced energy usage by allowing applications and power management systems to incorporate knowledge of each other's activities. We then showed that applications can significantly reduce energy usage by minimizing low-priority activities and by replacing periodic models of application behavior with event-driven models.

This study has found several powerful tools that allow developers to decrease the energy usage of PowerPoint and similar applications. We believe the next logical step for this work is the development of system support for energy-aware adaptation and transparent power management in closed-source environments such as Windows. Such support will make it easier for software developers to achieve the energy reductions we have discussed. We then hope to expand system support beyond the single application we have studied to handle multiple, concurrently executing applications.

Acknowledgements

This research was supported by the National Science Foundation (NSF) under contract CCR-9901696, the Defense Advanced Research Projects Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918, IBM Corporation, Nokia Corporation and Intel Corporation. The views and conclusions contained in this document are

those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, DARPA, USN, IBM, Nokia, Intel, or the U.S. government.

References

1. 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.
2. 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, California, March 2001.
3. 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.
4. 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.
5. Ellis, C. S. The case for higher-level power management. In *The 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 162–167, Rio Rico, AZ, March 1999.
6. Flinn, J. and Satyanarayanan, M. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems and Principles*, pages 48–63, Kiawah Island, SC, December 1999.
7. 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*, pages 160–170, Cambridge, MA, October 1996.
8. James, P. *Official Netscape Navigator 3.0 Book*. Netscape Press, 1996.
9. 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.
10. 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.
11. 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.
12. 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-VII)*, Schloss Elmau, Germany, May 2001.
13. Satyanarayanan, M., Flinn, J., and Walker, K. R. Visual proxy: exploiting OS customizations without application source code. *Operating Systems Review*, 33(3):14–8, July 1999.
14. 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.
15. Wilkes, J. Predictive power conservation. Technical Report HPL-CSP-92-5, Hewlett-Packard Laboratories, February 1992.