ECOSystem: Managing Energy as a First Class Operating System Resource*

Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahdat
Department of Computer Science
Duke University
{*zengh,xiaobo,carla,alvy,vahdat*}@*cs.duke.edu*

Technical Report CS-2001-01

March 25, 2001; revised August 8, 2001

Abstract

The energy consumption of computers has recently been widely recognized to be a major challenge of systems design. Our focus in this paper is to investigate what role the operating system can play in improving energy usage without depending on application software being rewritten to become energy-aware. Energy, with its global impact on the system, is a compelling reason for unifying resource management. Thus we propose the Currentcy Model that unifies energy accounting over diverse hardware components and enables fair allocation of available energy among applications. Our particular goal is to extend battery lifetime for mobile devices. We have implemented ECOSystem, a modified Linux, that incorporates our currentcy model and demonstrates the feasibility of explicit control of the battery resource. Experimental results show that ECOSystem can hit a target battery lifetime, and for reasonable targets, can do so with acceptable performance.

1 Introduction

One of the emerging challenges of computer system design is the management and conservation of energy. This goal manifests itself in a number of ways. The goal may be to extend the lifetime of the batteries in a mobile/wireless device. The processing power, memory, and network bandwidth of such devices are increasing rapidly, often resulting in an increase in demand for power, while battery capacity is improving at only a modest pace. Other goals may be to limit the cooling requirements of a machine or to reduce the economic costs of powering a large computing facility. Each scenario has slightly different implications. In this work, we focus on battery lifetime which allows us to exploit certain characteristics of battery technology.

Ideally, the problem of managing the energy consumed by computing devices should be addressed at all levels of system design - from low-power circuits to applications capable of adapting to the available energy source. Many research and industrial efforts are currently focusing on developing low-power hardware. We have previously advocated the value of including energy-aware application software as a significant layer in the design of energy efficient computing systems [8]. It is now a widely held view in the community that application involvement is important; however, the *necessity* of application involvement for achieving energy savings via software has not yet been shown. Thus, an important question to ask is what the operating system can do within its own resource management functions to improve energy usage without assuming any explicit cooperation from applications. Our scientific objective in this paper is to explore the degree to which energy-related goals can be achieved at the OS-level, exploiting existing, state-of-the-art hardware features, but requiring no application-specific knowledge or the ability of applications to adapt to energy constraints. This point of view also has practical implications since we can not depend on many current applications being rewritten to become energy-aware, at least until it is demonstrated that the effort needed will produce dramatically better results than systems-based approaches or until a suitable infrastructure is available to facilitate and support such redesign.

One of the major contributions of our work is the introduction of an energy accounting model, called the *currentcy model*, that unifies resource management for different components of the system and allows energy itself to be explicitly managed. Unifying resource management has often been mentioned as a desirable goal, but a focus on energy provides

*This work is supported in part by the National Science Foundation (EIA-99772879,ITR-0082914), Intel, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328). Additional information on this work is available at `http://www.cs.duke.edu/ari/millywatt/`.

a compelling motivation to seriously pursue this idea. Energy has a global impact on all of the components of the entire system. In our model, applications can spend their share of energy on processing, on disk I/O, or on network communication - with such expenditures on different hardware components represented by a common model. A unified model makes energy use tradeoffs among hardware components more explicit.

In general, there are two problems to consider at the OS-level for attacking most of the specific energy-related goals described above. The first is to develop resource management policies that eliminate waste or overhead and make using the device as energy efficient as possible. An example is a disk spindown policy that uses the minimal energy whenever the disk is idle. This has been the traditional approach and has typically been employed in a piecemeal, per-device fashion. We believe our currentcy model will provide a framework to view such algorithms from a more systemwide perspective. The second approach is to change the offered workload to reduce the amount of work to be done. This is the underlying strategy in application adaption where the amount of work is reduced, often by changing the fidelity of objects accessed, presumably in an undetectable or acceptably degraded manner for the user of the application. Unfortunately, without the benefit of application-based knowledge, other ways of reducing workload demands must be found. Our currentcy model provides a framework in which to formulate policies intended to selectively degrade the level of service to preserve energy capacity for more important work. Assuming that previous work on per-device power management policies provides an adequate base upon which to design experiments, we concentrate first on the less-explored second problem – formulating strategies for adjusting the quality of service delivered to applications. Later, we will return to the first issue and revisit such policies, expressed in terms of our model.

Observing that the lifetime of a battery can be controlled by limiting the discharge rate [20, 26], the energy objective we consider in this work for our energy management policies is to control the discharge rate to meet a specified battery lifetime goal. The first level allocation decision is to determine how much currentcy can be allocated to all the active tasks in the next time interval so as to throttle to some target discharge rate. Essentially, this first-level allocation determines the ratio of active work that can be accomplished to enforced idleness that offers opportunities to power down components. Then, the second level decision is to proportionally share this allocation among competing tasks.

We have implemented an OS prototype incorporating these energy allocation and accounting policies. Experiments quantify the battery lifetime / performance tradeoffs of this approach. We demonstrate that the system can hit a target battery lifetime. In certain regions of the design space, the battery lifetime goal can be achieved with acceptable performance degradation. Our proportional sharing serves to distribute the performance impact among competing activities in an effective way.

The paper is organized as follows. In the next section, we outline the underlying assumptions of this work, including the power budget, the characteristics of batteries, and the nature of the expected workload of applications. In Section 3, we present the currentcy model and the design of the currentcy allocator. In Section 4, we describe the prototype implementation and in Section 5, we present the results of experiments to assess the benefits of this approach. We discuss related work in the next section and then conclude.

In future work, after exploring the possibilities for OS-centric energy management, we can begin to identify complementary ways in which applications can interact with OS policies to enhance their effectiveness. We plan to consider how charging policies for the use of different devices will suggest appropriate interactions with applications that can be included in an effective API that is consistent with our model.

2 Background and Motivation

For the users of mobile computing, battery lifetime is an important performance measure. Typically, users and system designers face a tradeoff between maximizing lifetime and traditional performance measures such as throughput and response time. We assume a workload consisting of a mix of interactive productivity applications and multimedia processing. Depending on the applications of the device, the actual goal might be to have the battery last just long enough to accomplish a specified task (e.g., finish the scheduled presentation on the way to the meeting) or a fixed amount of work (e.g., viewing a DVD movie). Thus, metrics have been proposed that try to capture the tradeoff between the work completed and battery lifetime [21]. Alternatively, the work might not be fixed, but an acceptable quality of service is desired for as long as possible (e.g., for an MP3 player). Extending battery lifetime must usually be balanced against some loss in performance. Thus, our job is twofold: to achieve the battery lifetime goal and to find a fair way to distribute the performance impact among applications. Certain battery characteristics must be considered when trying to control battery lifetime, as described in Section 2.1.

OS-level energy management can be split across two dimensions. Along one dimension, there are a wide variety of devices in the system (e.g., the CPU, disks, network interfaces, display) that can draw power concurrently and are amenable to very different management techniques. This motivates a unified model that can be used to characterize the power/energy consumption of all of these components. In the other dimension, these devices are shared by multiple applications. The power usage of two simultaneously active

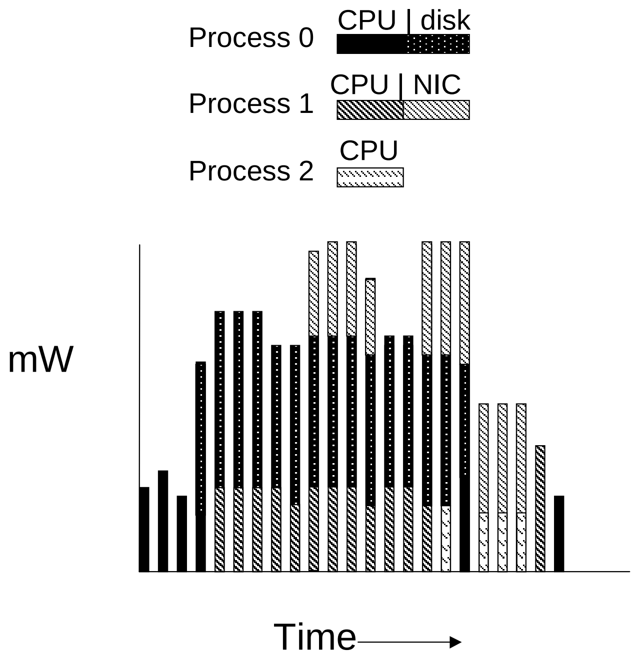


Figure 1: Accounting challenges of multiple devices and processes

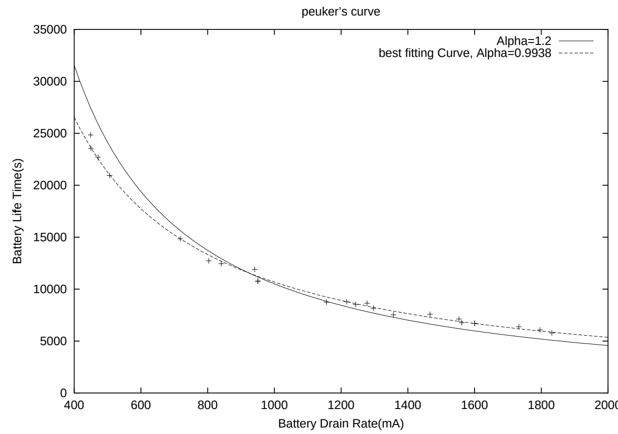


Figure 2: Battery Lifetime vs. Drain Rate

hardware components may be caused by two different applications. For example, the disk may be active because of an I/O operation being performed by a "blocked" process while another process occupies the CPU. This presents additional accounting challenges. Consider the scenario portrayed in Figure 1 involving three different processes and three different resources (CPU, disk, and wireless card). During the highest levels of power consumption, process 0's disk activity, process 1's network usage, and CPU processing by any one of the three processes all contribute. Using a program counter sampling technique, as in PowerScope [10], would inaccurately attribute power costs to the wrong processes.

Solving the accounting problem is a prerequisite to managing the battery resource. This involves (1) understanding the nature and determining the level of resource consumption, (2) appropriately charging for use of the various devices in the system, and (3) attributing these charges to the responsible entity. We introduce the currentcy model to coherently charge for the energy consumption of many asynchronously active devices and we adapt *resource containers* [1] to serve as the abstraction to which energy expenditures are charged. We elaborate upon the accounting challenges in Section 2.2.

2.1 Battery Characteristics

Understanding the nature of battery technology is key to its management. In particular, the effective energy capacity that can be delivered by a charged battery varies based on the discharge rate. Peukert's equation can be used to approximate the lifetime of a given battery, t, under various discharge rates:

$$t = \frac{C}{I^{\alpha}} \tag{1}$$

where C is the *rated* capacity of a battery, I is the current, and α is a constant for a given type of battery. This equation models the non-linear behavior inherent to current battery technologies, such that increasing I by a factor of x reduces t by a factor of $x^{-\alpha}$. Each battery has its specific α value. They are different even for batteries of the same technology. But for a particular battery, the α is a constant.

Figure 2 shows life time vs. drain rate curves for two batteries. These two batteries have the same capacity of $C = 3.6Ah$ at the drain rate of 1A. One of the curves has the α value of 1.2, which can be found on many currently used batteries. The other curve has a α value of 0.9938. This curve is the best fitting curve to the measured points of our IBM Thinkpad T20 battery. To obtain these points in the validation tests, microbenchmarks with power consumption values measured with a Fluke multimeter were used. We timed how long it took for a fully charged battery to run down completely while running the microbenchmark that achieved the particular drain rate under test (shown as measured points in the figure). This graph shows that with the same drain rate and rated capacity, battery lifetime can be quite different for batteries of different characteristics. Our results indicate that by achieving a particular average power consumption we can directly determine the battery lifetime, but the characteristics of the battery must be accurately modeled.

Peukert's equation assumes constant current. Our current approach only achieves an average power consumption as an abstraction of a constant discharge rate. However, using Peukert's simple model appears to be adequate for our purposes of setting a target value for power consumption.

Whereas large peaks may cause overly optimistic errors in the calculation of battery life based on an average drain rate [22], other work [3, 20] suggests that bursty discharge patterns can lengthen time until the cut-off voltage is reached by allowing recovery in the chemical and physical behavior of the battery. Clearly, there is a need for continuing research on modelling battery behavior. When a more suitable model becomes available, we can easily adopt it for use in our system.

2.2 Energy Resource Accounting Challenges

In trying to achieve a given discharge rate, the first challenge is to accurately determine the level of resource consumption on a per-device basis as a function of time. One recent development in the OS-directed management of the battery resource is the *Smart Battery* interface in the ACPI specifications [14] and compatible battery devices that support it. This interface allows the system to query the status of the battery, including the present remaining capacity, present drain rate, and voltage. The Smart Battery seems to be a potentially powerful tool in support of energy management. However, our investigations of the current capabilities of the Smart Battery have revealed limitations for our purposes. The operation of querying the interface is too slow to be useful for gathering power consumption data at every scheduling timeslice without introducing unacceptable overhead. In addition, the averaging of power consumption data returned by the query makes attributing an accurate power consumption value to a particular process problematic, even with only the CPU involved. Experiments with two synthetic benchmarks that individually produce a distinct, stable drain rate show that when they are scheduled together, the reported power consumption values can not be differentiated between the two competing processes.

Even if more fine-grained, overall power consumption data were available, the issue of multiple components argues for a different approach to tracking and attributing power/energy usage. As illustrated in Figure 1, observing an instantaneous power consumption value and attributing it to the currently running process does not accurately account for the contribution of other devices to the overall drain rate. As that example shows, one process may be responsible for the activity of the disk and another process may be responsible for power consumption in the network interface while the currently running process computes within the CPU.

Our solution is to embed a model within the operating system to track the parallelism of hardware components and their energy use. When the system uses a given component, we consult our model to determine how much energy will be used by the operation and which entity is responsible. The following section elaborates on our unified energy model.

3 The Currentcy Model

The key feature of our model is the use of a common unit—*currentcy*—for energy accounting and allocation across a variety of hardware components and tasks. Currentcy becomes the basis for characterizing the power requirements and gaining access to any of the managed hardware resources. It is the mechanism for establishing a particular level of power consumption and for sharing the available energy among competing tasks.

One unit of currentcy represents the right to consume a certain amount of energy within a fixed amount of time. The subtle difference between a unit of currentcy and a guarantee for an equivalent x Joules of energy is a time limit on use of the currentcy which has the desired effect of pacing consumption.

Incorporating a generalized energy accounting model within the operating system provides the flexibility necessary to uniformly support a range of devices. The model can be parameterized according to the specific power characteristics of the host platform. With existing hardware support, there is no alternative that can provide the information about the power consumption of individual components needed for accounting. A side-effect of embedding this model in the system is that it also makes it possible to vary assumptions about the system's power budget to emulate alternative device characteristics. Thus, while our target environment uses energy in a certain fashion, we can also design experiments based on the profile of a PDA where the CPU and display costs are significantly reduced and the hard drive may be eliminated altogether, or replaced by a device with very different characteristics e.g., compact flash memory.

The remainder of this section describes the overall structure of our energy model and how currentcy can be credited to tasks and debited upon resource use in such a way as to achieve a given battery lifetime.

3.1 System Energy Model

The system power costs are characterized in two parts of our model: The first part is the *base* power consumption that includes the low power states of the explicitly energy-managed devices as well as the default state of the devices not yet being considered. The larger the proportion of the system that gets included in the base category, the less opportunity there will be to affect improvements on top of it. As we shall see, while our experimental prototype with 3 managed devices (i.e., the CPU, disk, and network) is adequate to demonstrate our ability to unify multiple components under the currentcy model, the base remains a large, static factor in the range of drain rates we are able to produce on the laptop. Thus, we are interested in investigating how changing this aspect of the power budget may affect the behavior of the energy allocation strategies we propose.

The second part of the system model is the specification of the costs of the more active states for each of the explicitly managed devices. Thus, the halted state of the CPU and the spun-down state of the disk fall into the base while CPU activity and spinning the disk are explicitly modeled. Each of these higher power states is represented by a charge policy that specifies how currentcy is to be deducted to pay for that use of the component.

The level of detail in this part of the model depends on the information that is available to the OS and the management choices available to it. The status of the device must be visible to the OS - either in terms of state information or as observable transition events that cause higher power use - to allow tracking of the state. Our current prototype is very coarse-grained (e.g., CPU halted or active) but the model can support finer-grain information such as that available using event counters to track processor behavior as suggested by Bellosa [2]. This would allow more accurate accounting that could charge differently for various types of instructions or the frequency of cache misses.

3.2 Currentcy Allocation

Our overall goal is to achieve a user specified battery lifetime by limiting the discharge rate.

There are two facets to the allocation strategy. The first level allocation determines how much currentcy can be made available collectively to all tasks in the system. We divide time into energy-epochs. At the start of each epoch, the system allocates a specific total amount of currentcy. The amount is determined by the drain rate necessary to achieve the target battery lifetime. By distributing less than 100% of the currentcy required to drive a fully active system during the epoch, components are idled or throttled. There are constraints on the accumulation of unspent currentcy so that epochs of low demand do not amass a wealth of currentcy that could result in very high future drain rates.

The second aspect of currentcy allocation is the distribution among competing tasks. When the available currentcy is limited, it is divided among the competing tasks according to user-specified proportions. During each epoch, an allowance is granted to each task according to its specified proportional share of currentcy. There is a cap on the maximum amount of currentcy any application can save.

Our model utilizes resource containers [1] to capture the activity of an application or task as it consumes energy throughout the system. Resource containers are the abstraction to which currentcy allocations are granted and the entities to be charged for energy consumption. They are also the basis for proportional sharing of available energy. Resource containers deal with variations in program structure that typically complicate accounting. For example, an application constructed of multiple processes can be represented by a single resource container for the purposes of energy accounting.

3.3 Currentcy Payback

The actual resource management that achieves the throttling of drain rate is based on a pay-as-you-go policy whereby a resource container gains access to a managed device. Consider the CPU – the process scheduler will allow ready processes to run as long as their associated resource containers have currentcy to pay for the time slice. When there are no processes whose resource containers have any remaining currentcy left, even though they may be otherwise ready to run, the processor is halted until the next allocation. Similarly, I/O operations that cause disk activity result in currentcy being deducted from the associated resource container. In this way, energy tradeoffs become explicit. Currentcy spent on I/O is no longer available to pay for CPU cycles.

Each managed device has its own charging policy that reflects the costs of the device. For example, the disk policy may try to spread out the payments for spinup or for spinning during the timeout period prior to spindown. The base costs are not explicitly charged to resource containers, but obviously factor into the overall drain rate target. As we continue to develop the system, elements will migrate from the base into the category of explicitly managed and modelled devices.

Investigating the design space of policies that can be formulated in this currentcy model is beyond the scope of this paper and is a topic on on-going research. However, our implementation of one set of initial policies is described in Section 4. This allows us to demonstrate the feasibility of the idea, to gain experience with the system, and to identify problems that motivate future research.

4 Prototype

We have implemented our currentcy model in the Linux operating system running on an IBM ThinkPad T20 laptop. This section describes our prototype implementation which we have named **ECOSystem** for the Energy-Centric Operating System. First, we provide a discussion of the specific power consumption values that are used to parameterize our model for the various hardware components in the T20. In the next section we examine the effects of changing these values to represent alternative platforms (e.g., PDA).

4.1 Platform Power Characteristics

We measure the power characteristics of our Thinkpad hardware and use the resulting values as parameters to the model coded within the ECOSystem kernel. Within ECOSystem, we currently model three primary devices: CPU, Disk, and

	Cost	Time Out (Sec)
Access	1.65mJ	
Idle 1	1600mw	0.5
Idle 2	650mw	2
Idle 3	400mw	27.5
Standby (disk down)	0mw	N/A
Spin up	6000mJ	
Spin down	6000mJ	

Table 1: Hard disk power state and time-out values

Network Interface. All other devices contribute to the base or background power consumption of 13W for the platform.

CPU

The CPU of our laptop is a 650MHz PIII. In our CPU energy accounting model, we assume that the CPU draws a fixed amount of power (we currently use 15.55 W) for computation. This is a coarse-grained abstraction. Ideally, one would like to charge differently for different processor behavior (e.g., various types of instructions or the frequency of cache misses, etc.) and this would be compatible with our modelling approach (e.g., by using Bellosa's event counters [2]). However, in this paper, we use a single CPU power consumption value established by measuring the power while running a loop of integer operations.

Disk

Most of today's hard disks support the ATA interface which uses a timeout based power management scheme. The ATA standard defines a set a power states and the interface to control the timeout value for each state. Unfortunately, the hard disk in our laptop is an IBM Travelstar 12GN that has more power states than the ATA standard. This complicates hard disk energy accounting since it prevents the OS from reading the true power state of the disk. Furthermore, the Travelstar power state transitions are managed by an unknown internal algorithm and can not be set through the ATA interface, making it difficult to develop a timeout based model. Therefore, we approximate our disk's power consumption using a timeout based model derived from typical hard disks. Table 1 shows the values used in our model. The disk model is set to spin down after 30 seconds. To achieve comparable effects on timing, we also set the Travelstar to spin down after 30 seconds.

Wireless Network Interface

The network interface used in our system is an Orinoco Silver wireless PC card that supports the IEEE 802.11b standard. This card can be in one of three power modes: Doze (0.045W), Recieve (0.925W), and Transmit (1.425W). IEEE 802.11b supports two power-utilization modes: Continuous Aware Mode and Power Save Polling Mode. In the former, the receiver is always on and drawing power, whereas in the latter, the wireless card can be in the doze mode with the access point queuing any data for it. The wireless card will wake up periodically and get data from the base station. In the Power Save Polling Mode, the wireless card consumes a small fraction of the energy compared to the Continuous Aware Mode and most of the power is consumed by sending or receving data for the user application. In the ECOSystem prototype, we always use the Power Save Polling Mode and the maximum sleep time is set to 100 milliseconds (the default sleep time).

According to 802.11b, data retransmission may occur at the MAC layer as the result of data corruption. Data retransmission can consume additional energy invisible to the OS and can affect the accuracy of our energy accounting. The 802.11 standard specifies an optional Request-to-Send/Clear-to-Send (RTS/CTS) protocol at the MAC layer. When this feature is enabled, the sending station can transmit and receive a packet without any chance of collision. RTS/CTS adds additional bandwidth overhead to the network by temporarily reserving the medium, but it can prevent the data retransmission due to the "hidden node" problem and may save energy on the wireless card. In our tests, we enable RTS/CTS for transmissions larger than 1024 bytes. The MTU is 1500 bytes in our system.

4.2 The ECOSystem Implementation

We modified RedHat Linux version 2.4.0-test9 to incorporate energy as a first-class resource according to the model described in Section 3. Our changes include a rudimentary implementation of resource containers to support the two dimensions of our model: energy allocation and energy accounting. Below we elaborate on the kernel modifications associated with each of these dimensions.

4.2.1 Currentcy Allocation

ECOSystem supports a simple interface to manually set the target battery lifetime and to prioritize among competing tasks[1] These values are translated into appropriate units for use with our currentcy model (one unit of currentcy is valued at 0.01mJ). The target battery lifetime is used to determine how much total currentcy can be allocated in each energy epoch. The task shares are used to distribute this available currentcy to the various tasks.

Each resource container has two fields: tickets and available-currentcy. The tickets determine the proportion of

[1] We use the terms "task" and "resource container" interchangably. One or more processes may comprise a task.

the currentcy allocated in each epoch to a particular container. Currentcy is allocated to and deducted from the available-currentcy value of the container. To perform the per-epoch currentcy allocation, we introduce a new kernel thread *kenrgd* that wakes up periodically and distributes currentcy appropriately. We empirically determine that a one second period for the energy epoch is sufficient to achieve smooth energy allocation. If a task does not use all its currentcy in an epoch, it can accumulate currentcy up to a maximum level (which is also proportional to a task's share), beyond which any extra currentcy is discarded.

In our current implementation, a process is scheduled for execution only if its corresponding resource container has currentcy available. We modified the Linux scheduler to examine the appropriate resource container before scheduling a process for execution. Under this policy, all tasks expend their currentcy as quickly as possible during a given energy epoch. This approach may produce bursty power consumption and irregular response times for some applications. Work is underway on providing a proportional scheduler based on [23] that will more smoothly spread the currentcy expenditure throughout the entire energy epoch.

4.2.2 Currentcy Accounting

Tasks expend currentcy by executing on the CPU, performing disk accesses or sending/receiving messages through the network interface. The cost of these operations is deducted from the appropriate container. When the container is in debt (available-currentcy $\leq$ zero) none of the associated processes are scheduled or otherwise serviced. The remainder of this section explains how we perform energy accounting for the CPU, disk, and network card.

CPU

Our CPU accounting and charging policy is based on a hybrid of sampling and standard task switch accounting. It is easily integrated with Linux's support for time-slice scheduling. Accounting at a task switch provides accurate accounting of processor time used. However, to prevent long-running processes from continuing to run with insufficient currentcy, we deduct small charges as the task executes. Then, if it runs out of currentcy during its time-slice, it can be preempted early. Thus, we modify the timer interrupt handler to charge the interrupted task for the cost of executing one tick of the timer. In our system, a timer interrupt occurs every 10ms and we assume a constant power consumption of 15.5W for CPU execution. Therefore, at each timer interrupt, the appropriate resource container's currentcy will be reduced by 15,540 (155.4mJ).

Hard Disk

Energy accounting for hard disk activity is very complex. The interaction of multiple tasks using the disk in an asynchronous manner makes correctly tracking the responsible party difficult. Further complexities are introduced by the relatively high cost of spinning up the disk and the large energy consumption incurred while the disk is spinning. We have implemented what we believe to be a reasonable policy, however futher research is clearly necessary on this complex topic.

To track disk energy consumption, we instrument file related system calls to pass the appropriate resource container down to the buffer cache. The container ID is stored in the buffer cache entry. This enables accurate accounting for disk activity that occurs well after the task initiated the operation. For example, *write* operations can occur asynchronously, with the actual disk operation performed by the I/O daemon. When the buffer cache entry is actually written to disk, we deduct an amount of currentcy from the appropriate resource container. Energy accounting for *read* operations is performed similarly.

We can break disk cost into four categories: spinup, access, spinning, and spindown. The cost of an access is easily computed by
$\frac{active-state-power-cost(W)}{disk-accesss-bandwidth(KB/s)} * buffersize(KB)$.
The energy consumed to access one buffer on disk is 1.65mJ. Since a dirty buffer cache entry may not be flushed to disk for some time, multiple tasks may write to the same entry. Our current policy simply charges the cost of the disk access to the last writer of that buffer. While this may not be fair in the short-term, we believe the long-term behavior should average out to be fair. The remaining disk activities present more difficult energy accounting challenges.

The cost of spinning up and down the disk is shared by all tasks using the disk during this session between spinup and spindown. It is charged at the end of the session and is divided up on the basis of the number of buffers accessed by each task. We assume that the spin up or spin down takes 2 seconds and that the average power is 3,000mW, leading to a total energy cost of 6,000mJ.

The cost for the duration of time that the disk remains spinning waiting for the timeout period to expire (30 second minium) is shared by those tasks that have recently accessed the disk (in essence, those that can be seen as responsible for preventing an earlier spindown). This is done by incrementally charging the tasks that have performed accesses within the last 30 second window in 10ms intervals (timer interrupt intervals). On each timer interrupt, if the disk is spinning, the energy consumed during this interval, as determined by the disk power state and length of the interval (10ms), is shared among those tasks active in the last 30 seconds.

Our present implementation does not handle all disk activity. In particular, inode and swap operations are not addressed. The swap file system has its own interface and does not follow the vnode to file system to file cache to block-device hierarchy.

Network Interface

Energy accounting for the network interface is implemented in ECOSystem by monitoring the number of bytes transmitted and received. These values are then used to compute the overall energy consumption according to the following equations:
$E_{send} = (sent\_bytes * transmit\_power)/bit\_rate$
$E_{recv} = (received\_bytes * receive\_power)/bit\_rate.$

The energy consumption is calculated at the device driver according to the full length of the packet including the hearder. We have instrumented the socket structure and the TCP/IP implementation in order to track the task responsible. When a socket is created for communication, the creator's container ID is stored in the socket. For outgoing packet, the source socket hence the source task is identified at the device driver. For incoming packet, the energy consumption for receiving the packet is computed and initially stored with the packet when it is received. The destination socket of this packet will be available after it is processed by the IP layer. Currentcy is deducted from the destination task at this moment. If packets are reassembled in the IP layer, the energy cost of the reassembled packet is the sum of all fragmented packets.

We believe that our approach with TCP/IP connections can also be applied to other types of protocols such as UDP/IP and IPV6. In IPV6, the destination socket may be available before being processed by the IP layer which can ease our job of task tracking.

5 Experiments and Results

This section presents experimental results using our prototype implementation. We begin by presenting our methodology for evaluating the effectiveness of our energy management policies. The main thrust of our evaluation begins as we present results on our system's ability to throttle application execution to achieve specified battery lifetimes by maintaining a particular average drain rate. We examine throttling effects on application performance.

5.1 Experimental Methodology

We use a combination of microbenchmarks and real applications to evaluate our energy management policies. The microbenchmarks enable targeted evaluation of various system components. The real applications we use are netscape, x11amp, and the mcf benchmark from the SPEC2000 suite. The primary metrics are battery lifetime and application performance. By specifying the battery lifetime, and hence target drain rate, we can measure application performance for the given lifetime.

For each of our applications, we define an evaluation metric that we believe correlates with user-perceived performance. Our first application, netscape, is representative of an entire class of interactive applications where a user is accessing information. The performance metric we use for netscape is the time required to complete the display of a web page. We assume the page must be read from the network and that the netscape file cache is updated, so all three of our managed devices are included (CPU rendering, disk activity, and a network exchange). We obtain values for the performance metric by inserting a few lines of javascript code into the web pages to be loaded. For netscape, we model different user think times between page requests that has the effect of allowing some amount of currentcy to accumulate between events.

Our next application, x11amp, is an MP3 player. This is representative of a popular battery-powered application with user-perceived quality constraints. Since each song has a specific play time, we evaluate this application's performance by measuring any slowdown in playback. This is done by comparing the actual time to complete the song against the length of the song. Any slowdown in playback manifests itself as disruption (e.g., silence) in the song.

The final application is a computationally intensive optimization code which could be viewed as representative of the kind of computations that might be done at sensor nodes. The performance metric is execution time.

Our first set of results are obtained using the power consumption values for the T20 configuration described in Section 4 as parameters to our energy model. Those values were obtained using microbenchmarks and measuring actual power consumption with a Fluke multimeter.

5.2 Targeting Battery Lifetime

Achieving a target battery lifetime is an essential design objective of ECOSystem. We performed a variety of experiments using our CPU integer-intensive microbenchmark. As shown in Figure 3, we found that these tests achieved the target battery lifetime with little residual battery capacity.

While these tests are encouraging, we are aware that there exist several potential sources of error in the energy accounting that could cause our behavior under the model to deviate from the target battery lifetime. For example, variations in cache behavior that are not captured by our flat CPU charge or the existence of high peak loads that violate assumptions of constant current would introduce error in our lifetime estimate. One remedial approach that we have investigated involves making periodic corrections. By regularly obtaining the remaining battery capacity via the smart battery inter-

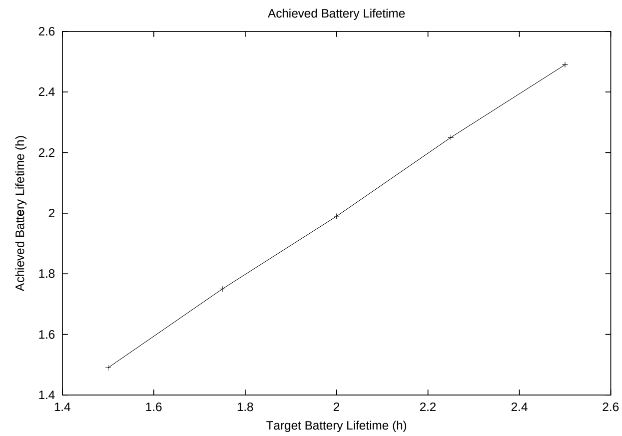


Figure 3: Achieving Target Battery Lifetime

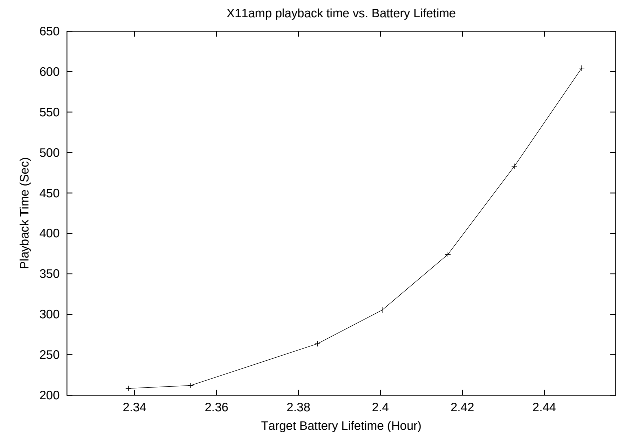


Figure 5: x11amp Performance vs. Battery Lifetime

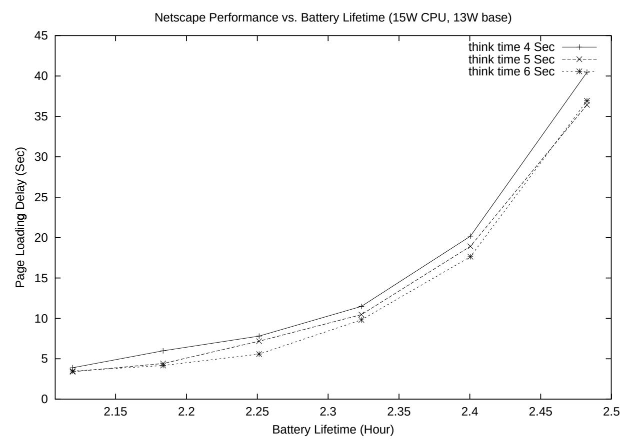


Figure 4: Netscape Response Time vs. Battery Lifetime

face, our system can take corrective action by changing the amount of currentcy allocated in each energy epoch. If we appear to be under charging for the CPU, then currentcy can be reduced. If it appears that we are over charging, then currentcy can be increased.

To investigate the impact of energy accounting inaccuracies, we use our CPU intensive microbenchmark, but deliberately introduce accounting error for the CPU power consumption (14W instead of the measured 15.5W). To make corrections ECOSystem polls the battery every 30 seconds to obtain a new value for remaining capacity. With the periodic correction, ECOSystem achieves near perfect battery lifetime.

The experiments based on the parameters of the T20 model are such that the potential increase in battery life that may result from adjustments in drain rate will not be able to overcome a loss in performance. The major effectiveness of ECOSystem lies in allocating the limited energy among multiple applications. However, we first show the battery lifetime vs. performance tradeoff for cases of our applications running alone in the system.

Figures 4 and 5 show performance vs. battery lifetime for netscape and x11amp, respectively. For netscape, we consider 3 different values of think time (4, 5, and 6 seconds) between operations. The data points in the lower left of each graph corresponds to the response time with no throttling. From these results we see that as the target battery lifetime increases, response time generally increases. Response time stays reasonably acceptable while extending the lifetime from 2 to 2.35 hours; beyond that, it rises more sharply. Response time is inversely proportional to think time as a result of the task accumulating currentcy during think time. The longer the think time, the more currentcy accumulates and the less likely it is that the task will be throttled.

X11amp is not an interactive application and thus does not have a think time to vary. Figure 5 shows the playback times for x11amp playing a 201 second song (unthrottled). The shape of the performance degradation is similar to that of netscape.

5.3 Sharing Limited Capacity

We assume that for machines such as the Thinkpad T20, users often want to run multiple tasks and allocate different amounts of energy to the various applications. To explore this scenario, we run two of our benchmarks concurrently. We measure the average power consumption of each task for various allocations and the appropriate performance metric for that application. These experiments test the proportional sharing that is invoked when currentcy allocations become limited. For our energy management to be successful in this scenario, the energy should go to the more vital task according to its specified share. Currentcy allocation is the mechanism for selectively curtailing activity in an appropriate way so as to preserve battery life for the more important tasks.

To evaluate ECOSystem's ability to proportionally allocate energy according to user specified shares we first run two instances of the CPU intensive SPEC benchmark mcf. The metrics we use for evaluation are the average power consumed and the CPU utilization for each instance of mcf. We show the shares in terms of both the specified proportion of overall available allocation and how that translates into percentage of CPU energy. In Table 2, the available power to be allocated (12.4W) is less than that required for full utilization of the CPU (15.5W). The power used and the CPU utilizations match the allocations perfectly in this case.

Table 3 shows our results for two overall allocations of 3W and 5W and three different proportions (50% : 50%, 40% : 60%, and 30% : 70%) for two instances of Netscape running concurrently. Netscape is an interactive application involving multiple devices. The cost of spinning the disk can be shared by the two instances. The page loading delay is subject to variations in the condition of the network, but averages 3.5 seconds. Without throttling, one instance of Netscape viewing the CNN home page with a 5 second think time requires an average of 3.6 W. Running two instances requires 6.7W with disk cost sharing. These results of sharing a limited allocation show that the power used by each instance does reflect the proportion given, but not precisely. The performance however is not proportional to the energy shares. With a sufficient allocation of 3.5W, Netscape 2 with 70% of the 5W allocation achieves performance close to the network latency. Note the page loading latency of the first and last lines in which Netscape 1 gets 1.5W of 3W and 5W respectively. The better performance in the last line is explained by Netscape 2 sharing a larger proportion of the disk cost with its greater allocation.

5.4 Work Accomplished

The increase in battery lifetime is often accompanied by the degradation in performance. In some systems the potential slowdown in application performance can be largely offset by the increase in battery lifetime allowing the same amount of overall work to be accomplished. However, for our experimental platform this is not the case.

Figure 6 shows the number of pages a user can view with netscape for a given battery lifetime and think times of 4, 5, and 6 seconds. These results show that reducing average power consumption to increase battery lifetime causes less work to be done. This is not surprising as we have seen the super-linear increases in page loading delay and playback time in Figure 4 and Figure 5. This is due to the large disk and base power consumption.

The base power consumption can affect the system in two ways. First, it can limit the range over which we can control drain rate along the battery lifetime curve. This is the case for the T20 configuration, meaning that there is little opportunity for extending the battery life. The second effect is that if the base power consumption is high relative to the range of possible drain rates, it limits how much difference our management efforts can make in the overall drain rate.

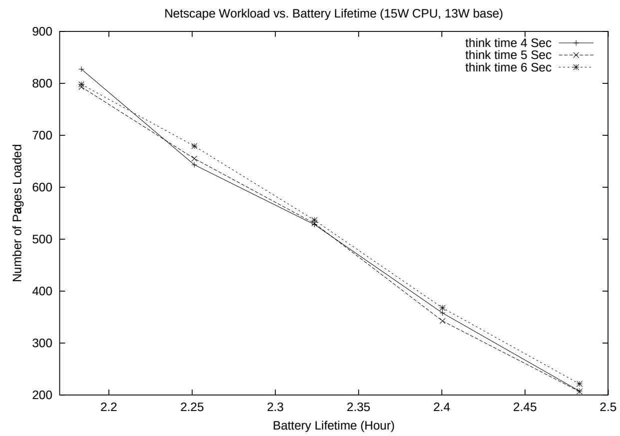


Figure 6: Netscape Page Views vs. Battery Lifetime

Disk power consumption can also influence the overall amount of work done. The high power costs of the disk along with constrained allocations to the tasks can disturb request patterns, affect disk spinup/down and result in relatively more overhead in using the disk.

We see that the large base power consumption has negative impact on the amount of work that can be accomplished. To evaluate this we utilize the flexibility of our model to emulate an entirely different platform that is representative of a future device with a 2W processor, 0.05W base power, and MEMS-based storage. Our MEMS storage power characteristics are based on those presented by Schlosser et al [28]. An access costs 0.112mJ, transitioning to active mode costs 5mJ and is charged only to the task that causes the transition. We assume the energy to remain active is 100mW, and the timeout to standby mode is 50ms. While the device is active, we use the same incremental accounting method as before; on each 10ms timer interrupt we charge the last task to access the device for the entire 10ms interval. For these experiments we use a 3.7Wh battery.

Figure 7 shows response time and number of page views versus battery lifetime for our PDA/MEMs platform. These results show that with the low power disk and low base power consumption, the page loading time now increases almost linearly with the extended battery lifetime. We can also see that battery lifetime can increase significantly (from 8 to 16 hours), but the number of page views still decreases steadily.

Figure 8 explores the impact of base power consumption on the number of page views for our PDA/MEMs platform.

Energy Share (CPU %)	MCF 1			MCF 2		
	Power Alloc (W)	Ave. Power Used (W)	CPU util (%)	Power Alloc (W)	Ave. Power Used (W)	CPU util (%)
50% : 50% (40% : 40%)	6.2	6.2	40%	6.2	6.2	40%
37.5% : 62.5% (30% : 50%)	4.65	4.65	30%	7.75	7.75	50%
25% : 75% (20% : 60%)	3.1	3.1	20%	9.3	9.3	60%

Table 2: Proportional Energy Allocation: 12.4W (80% of CPU's 15.5W); Two instances of MCF.

Energy Share (Total Alloc)	Netscape 1			Netscape 2		
	Power Alloc (W)	Ave. Power Used (W)	Page Load Latency (sec)	Power Alloc (W)	Ave. Power Used (W)	Page Load Latency (sec)
50% : 50% (3W)	1.5	1.47	22.48	1.5	1.49	19.33
40% : 60% (3W)	1.2	1.16	29.91	1.8	1.78	15.42
30% : 70% (3W)	.9	.92	53.74	2.1	2.06	12.47
50% : 50% (5W)	2.5	2.45	7.79	2.5	2.41	7.11
40% : 60% (5W)	2.0	1.93	11.09	3.0	2.97	5.10
30% : 70% (5W)	1.5	1.50	16.88	3.5	3.27	3.57

Table 3: Proportional Energy Allocation: Two instances of Netscape, 5 sec. think time.

The x-axis is normalized battery lifetime (lower values correspond to less battery life) and the y-axis is the number of page views. From these results we see that a base power consumption less than 0.02W will not have a significant negative impact on the number of page views.

Our results argue for the additional benefits of reducing baseline power consumption from the hardware perspective. Not only will the baseline battery lifetime increase, but additional optimizations (such as throttling the maximum drain rate) can lead to even more significant relative improvements in battery lifetime.

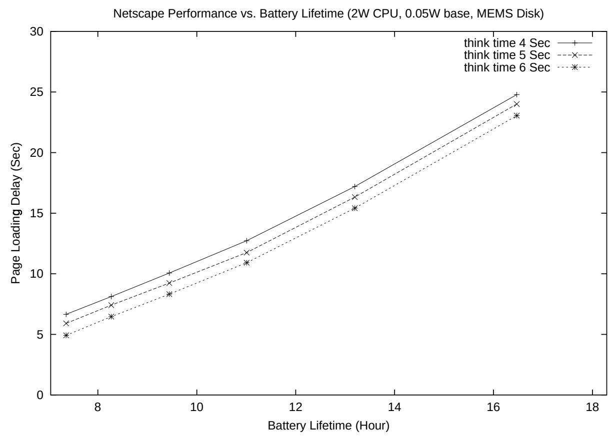
6 Related Work

Attention to the issues of energy and power management is gaining momentum within Operating Systems research. Recent work has made the case for recognizing energy as a first-class resource to be explicitly managed by the Operating System [31, 8].

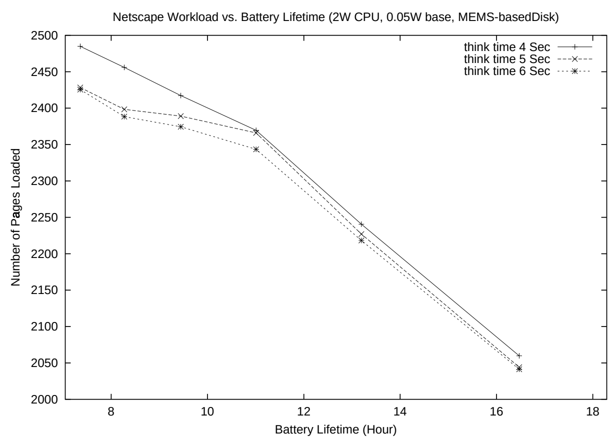
Work by Flinn and Satyanarayanan on energy-aware adaptation using Odyssey [9] is closely related to our effort in several ways. Their fundamental technique differs in that it relies on the cooperation of applications to change the fidelity of data objects accessed in response to changes in resource availability. The goal of one of their experiments is to demonstrate that by monitoring energy supply and demand to trigger such adaptations, their system can meet specified battery lifetime goals. They do not consider the nonlinear characteristics of batteries in their study, but they do test whether they can reach the designated lifetime goal before depleting a fixed energy capacity and to do so without having too much residual capacity leftover at the end of the desired time (which would indicate an overly conservative strategy). They achieve a 39% extension in lifetime with less than 1.2% of initial capacity remaining. For their approach, the performance tradeoff takes the form of degraded quality of data objects.

There has been previous work on limiting CPU activity levels, in particular for the purpose of controlling processor temperature, via the process management policies in the operating system. In [27], the operating system monitors processor temperature and when it reaches a threshold, the scheduling policy responds to limit activity of the "hot" processes. A process is identified as "hot" if it uses the CPU extensively over a period of time. As long as the CPU temperature remains too high, these hot processes are not allowed to consume as much processor time as they would normally be entitled to use. This work only considers the power consumption of the CPU as opposed to our total system view. This strategy was implemented in Linux and results show that power constraints or temperature control can be successfully enforced. The performance impact is selectively felt by the hot processes which are likely not to be the foreground interactive ones.

The idea of performing energy-aware scheduling using a throttling thread that would compete with the rest of the active threads has been proposed by Bellosa [2]. The goal is to lower the average power consumption to facilitate passive cooling. Based upon his method of employing event counters for monitoring energy use, a throttling thread would get



a) Response Time



b) Page Views

Figure 7: Netscape Performance for PDA with MEMs-disk

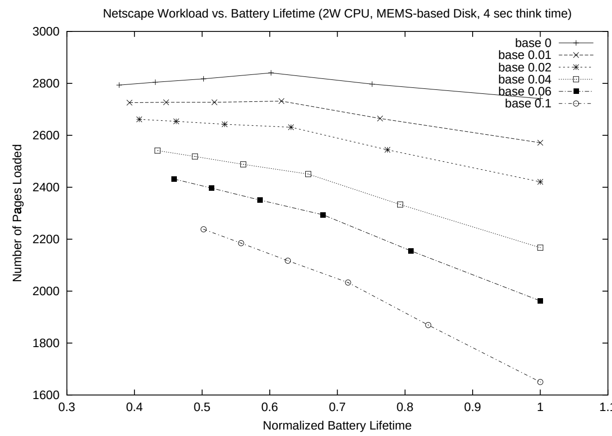


Figure 8: Base Power Consumption and Available Work

activated whenever CPU activity exceeded some threshold. When the throttling thread gets scheduled to run, it would halt the CPU for an interval.

The term “throttling” (which we have used in a very general sense) is most often associated with the growing literature on voltage/clock scheduling [24, 25, 12, 33, 11, 34, 15] for processors that support dynamic voltage scaling. Here, the “scheduling decision” for the OS is to determine the appropriate clock frequency / voltage and when changes should be performed. Interval-based scheduling policies track recent load on the system and scale up or down accordingly. Task-based algorithms associate clock/voltage settings with the characteristics (e.g. deadlines, periodic behavior) of each task.

The body of literature on power/energy management has been dominated by consideration of individual components, in isolation, rather than taking a system-wide approach. Thus, in addition to the CPU-based studies mentioned above, there have been contributions addressing disk spindown policies [19, 6, 5, 17, 13], memory page allocation [18, 4], and wireless networking protocols [16, 29]. The emphasis is most of this work has been on dynamically managing the range of power states offered by the devices. This work is complementary to our currentcy model and will impact the charging policies for such devices under our system.

Beyond the energy management domain, there has been a desire within the OS research community to unify various resources that are traditionally managed separately. For example, both lottery scheduling [32, 30] and resource containers [1] claim to have the potential to be extendible to multiple resources.

7 Summary and Conclusions

In this paper, we have tackled one of the most important problems haunting the design of battery-powered mobile and wireless computing systems - the management of battery lifetime and the energy/power resource, in general. We have taken the position that the operating system has an important role to play in managing energy as a first-class resource. We do not wish to depend on all applications being rewritten to become energy-aware. Our job is twofold: (1) trying to manage the battery to achieve the target battery lifetime with minimal residual energy. (2) when it becomes necessary to curtail work in order to deal with the limited energy resource, to selectively allocate the energy among multiple applications based on proportional sharing.

We offer the following contributions to this emerging research field:

- We propose a Currentcy Model that unifies diverse hardware resources under a single management framework. Because of the global effect that energy/power has on the operation of all that hardware components, the need to address energy management coherently has been a compelling impetus to developing a unifying model covering multiple resources.

- We have implemented a prototype energy-centric operating system, ECOSystem, that incorporates our model and demonstrates techniques for explicit energy management with a total system point of view. We have applied this system toward the specific problem of extending battery lifetime by limiting discharge rate. This system provides a testbed for formulating various other resource management policies in terms of currentcy.

- We have gained insights into the complex interactions of energy conservation and performance by running experiments with real and synthetic benchmarks on our prototype. We demonstrate successful results on achieving a target battery lifetime and on proportionally sharing energy among multiple tasks. We have identified the factors that affect the effectiveness of our approach.

This is a promising first step and the creation of a powerful infrastructure to pursue additional opportunities to manage energy by the operating system. By making the tradeoffs explicit among the many devices and tasks that consume energy in a system, the Currentcy Model can serve as a powerful "language" in which to formulate the complex relationships involved in a unified view of resource management.

One interesting avenue of future research suggested by our results is to consider the interaction between ECOSystem and the applications on the one hand and hardware devices on the other. While an explicit goal of our work has been to understand the potential benefits of OS energy management with unmodified applications and current hardware, we believe that ultimately maximum benefits can only be achieved through a rich interaction between applications, the operating system, and the hardware. In fact, one immediate observation of our work is that reducing the baseline power consumption in future hardware platforms, will result in significant additional operating system opportunities to manage energy

8 References

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.

[2] F. Bellosa. The benefits of event-driven accounting in power-sensitive systems. In *Proceedings of the SIGOPS European Workshop*, September 2000.

[3] C. F. Chiasserini and R. Rao. Pulsed battery discharge in communication devices. In *Proceedings of MOBICOM*, August 1999.

[4] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramiam, and M. Irwin. DRAM energy management using software and hardware directed power mode control. In *Proceedings of 7th Int'l Symposium on High Performance Computer Architecture*, January 2001.

[5] F. Douglis, P. Krishnan, and B. Bershad. Adaptive Disk Spindown Policies for Mobile Computers. In *2nd USENIX Symposium on Mobile and Location-Independent Computing*, April 1995. Monterey CA.

[6] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the Power Hungry Disk. In *Proceedings of the 1994 Winter USENIX Conference*, pages 293–306, January 1994.

[7] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Operating Systems Design and Implementation*, October 1996.

[8] C. S. Ellis. The Case for Higher-Level Power Management. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 1999.

[9] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.

[10] J. Flinn and M. Satyanarayanan. PowerScope: A tool for profiling the energy usage of mobile applications. In *Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 2–10, February 1999.

[11] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Proceedings of first annual international conference on Mobile computing and networking*, November 1995.

[12] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[13] D. Helmbold, D. Long, and B. Sherrod. A Dynamic Disk Spin-Down Technique for Mobile Computing. In *Proc. of the 2nd ACM International Conf. on Mobile Computing (MOBICOM96)*, pages 130–142, November 1996.

[14] Intel Corporation, Microsoft Corporation, and Toshiba Corporation. Advanced configuration and power interface specification. http://www.teleport.com/ acpi, December 1996.

[15] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of 1998 international symposium on Low power electronics and design*, pages 197–202, August 1998.

[16] R. Kravets and P. Krishnan. Power Management Techniques for Mobile Communication. In *Proc. of the 4th International Conf. on Mobile Computing and Networking (MOBICOM98)*, pages 157–168, October 1998.

[17] P. Krishnan, P. Long, and J. Vitter. Adaptive Disk Spin-Down via Optimal Rent-to-Buy in Probabilistic Environments. In *Proceedings of the 12th International Conference on Machine Learning*, pages 322–330, July 1995.

[18] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS IX)*, November 2000.

[19] K. Li, R. Kumpf, P. Horton, and T. Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *USENIX Association Winter Technical Conference Proceedings*, pages 279–291, 1994.

[20] D. Linden. *Handbook of Batteries*. McGraw Hill, 2nd edition, 1995.

[21] T. Martin and D. Siewiorek. A power metric for mobile systems. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, August 1996.

[22] T. Martin and D. Siewiorek. Non-ideal battery properties and low power operation in wearable computing. In *Proceedings of the 3rd International Symposium on Wearable Computers*, October 1999.

[23] J. Nieh, C. Vaill, and H. Zhong. Virtual-time round-robin: an O(1) proportional share scheduler. In *Proceedings of the USENIX Technical Conference*, June 2001.

[24] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of International Symposium on Low Power Electronics and Design*, 2000.

[25] T. Pering, T. D. Burd, and R. W. Brodersen. The Simulation and Evaluation of Dynamic Scaling Algorithms. In *Proceedings of the International Symposium on Low Power Electronics and Design*, August 1998.

[26] R. Powers. Batteries for low power electronics. *Proc. of the IEEE*, 83(4):687–693, April 1995.

[27] E. Rohou and M. Smith. Dynamically managing processor temperature and power. In *Proceedings of 2nd Workshop on Feedback Directed Optimization*, November 1999.

[28] S. Schlosser, J. L. Griffin, D. Nagle, and G. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of Ninth Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[29] M. Stemm and R. Katz. Measuring and Reducing Energy Consumption of Network Interfaces in Hand-Held Devices. In *Proceedings of 3rd International Workshop on Mobile Multimedia Communications (MoMuC-3)*, September 1996.

[30] D. Sullivan, R. Haas, and M. Seltzer. Tickets and currencies revisited: Extensions to multi-resource lottery scheduling. In *Proceedings of 7th Workshop on Hot Topics in Operating Systems (HOTOS-7)*, March 1999.

[31] A. Vahdat, C. Ellis, and A. Lebeck. Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[32] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional share resource management. In *Proceedings of Symposium on Operating Systems Design and Implementation*, November 1994.

[33] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proceedings of First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[34] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of IEEE Symp. on Foundations of Computer Science*, October 1995.