Currentcy: Unifying Policies for Resource Management

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

Abstract

The global system nature of energy creates challenges in developing operating system policies to effectively manage energy consumption. Our proposed *currentcy model* creates the framework for the operating system to manage energy as a first-class resource. Furthermore, currentcy provides a powerful mechanism to formulate unified energy management policies across diverse competing applications and spanning devices with very different power characteristics. We claim that unifying energy management enables more coherent and efficient energy consumption.

This paper presents an initial exploration of the policy space enabled by the currentcy model as implemented in ECOSystem, our Linux-based prototype. We use ECOSystem to attack four specific energy-related goals: 1) currentcy conserving scheduling algorithms that reduce residual battery capacity, 2) proportional energy sharing, 3) response time variation, and 4) energy efficient disk management. Our results show that the currentcy model is a powerful framework for expressing better energy management policies than those that come from the traditional per-device approach.

1 Introduction

Energy is an increasingly important system resource. This is most evident in battery-powered mobile computing platforms, from desktop-equivalent laptops to tiny embedded sensor nodes, although its significance is becoming recognized in other computing environments as well. While a number of efforts have explored minimizing the power consumption of specific system resources (e.g., dynamic voltage scaling algorithms for the CPU, disk spindown policies), we have advocated that the OS should explicitly manage the system-wide role that energy plays [16, 5] and view it as an opportunity and challenge for resource management.

Our earlier work [19] proposes *currentcy*[1] as a unifying abstraction for the management of a broad variety of system devices that consume energy. We have designed a basic framework for explicitly allocating energy resources and accounting for energy use in the system in terms of currentcy.

In this paper, we demonstrate how currentcy can be used as a general mechanism for expressing a broad range of energy management policies that cut across both competing applications and different system devices.

Specifically, this paper makes the following contributions:

- We develop a currentcy conserving energy allocation policy to reclaim residual battery capacity.
- We develop a currentcy-based scheduling policy that proportionally shares energy according to user specified preferences.
- We develop a currentcy-based scheduling policy that achieves smooth currentcy consumption, eliminating jitter in many applications.
- We show how currentcy provides a common mechanism to expose and effectively manage the

[*]This work is supported in part by the National Science Foundation (EIA-99772879,ITR-0082914,CCR-0204367), 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/`.

[1]This is a coined term, combining the concepts of current (i.e., amps) and currency (i.e., $).

subtle interactions of applications using different resources.

- We demonstrate how to shape disk access patterns to amortize the energy costs of spinup/spindown across multiple requests.

- Our experimental results show significant savings in residual energy, lower response time variation, and reductions in average power costs for disk accesses.

The rest of this paper is organized as follows. Section 2 describes our currentcy model and its implementation in ECOSystem, our Linux-based prototype. This is followed in Section 3 by a discussion of the policy space created by the currentcy model and a presentation of several specific policies in Section 4. We evaluate our proposed policies in Section 5. Section 6 presents related work and Section 7 concludes.

2 Background

The ECOSystem approach is based upon a unifying 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 is the basis for characterizing the application 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, required to pace consumption.

There are two facets to the allocation strategy. The first level allocation determines the amount of currentcy to collectively allocate among all system tasks. 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 a 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 power consumption peaks.

The second aspect of currentcy allocation is 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.

We use 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 address 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. We use the terms "task" and "resource container" interchangeably.

We have implemented our currentcy model in the ECOSystem prototype [19], a modified RedHat Linux version 2.4.0-test9 running on an IBM ThinkPad T20 laptop. This platform has a 655MHz PIII processor and we assume an active power consumption of 15.55W. The disk is an IBM travelstar which we model in ECOSystem as costing 1.65mJ per block access and 6000mJ for both spinup and spindown. The wireless network is an Orinoco Silver PC card supporting IEEE 802.11b, it has three power modes: Doze (0.045W), Receive (0.925W) and Transmit (1.425W). All other devices contribute to the base power consumption, measured to be 13W for the platform.

ECOSystem supports a simple interface to manually set the target battery lifetime and to prioritize among competing tasks. 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. To perform the per-epoch currentcy allocation, we introduce a new kernel thread *kenrgd* that wakes up periodically and distributes currentcy appropriately.

3 Policy Building Blocks

In our model, currentcy represents available global system resources. Currentcy allocation and accounting express and enforce policies to achieve energy-related goals. In this section, we outline the various ways in which currentcy can be manipulated to implement a particular policy. The design space is rich, making an exhaustive exploration infeasible. However, Section 4 illustrates how the various parameters can be used to attain specified objectives.

3.1 Overall Currentcy Allocation

The first decision point is the overall allocation of currentcy that determines how fast or how much energy can be consumed by the system as a whole. Choices include:

Per-epoch allocation level. We must determine the per-epoch currentcy availability based on the primary energy goal. Existing work focuses on achieving a target battery lifetime. Existing models of battery lifetime assume a constant power consumption, thus we impose a limit that translates directly into the currentcy allotment.

Epoch length. This determines the rate and granularity of currentcy allocation. Long epochs provide larger allocations and the ability to spend them in a more bursty fashion. Shorter epochs may smooth the consumption rate but pose problems accumulating enough for expensive operations. This option is addressed in Section 4.3.

Dynamic adjustment. This concerns whether (and how) to allow dynamic adjustment of per-epoch allocation levels. One example is performing adjustments in allocation based on remaining capacity information from a Smart Battery to correct for under-utilization of the resource (i.e., effectively a form of global redistribution of unused currency) or errors in the cost model.

3.2 Per-task Currentcy Allocations

Given the overall allocation, the next decision is how to allocate currentcy among competing tasks.

Determination of per-task *share*. This may reflect an external priority or criticality of the task, the energy demand of the task, or some combination. The share is based on a user specification, scaled to a percentage based on all tasks in the system.

Handling of unused currentcy. When a task finishes an epoch without using its allocation, what happens to the residual currentcy? Choices include forfeiting the remaining allocation at the end of the epoch, saving it all, saving up to a dynamic or static *cap*, or distributing it among other tasks. Our initial approach [19] saved currentcy up to a fixed cap proportional to the task's share. Techniques to redistribute unused currentcy are considered in Section 4.1.

Debt limits. Do we allow a task to perform deficit spending and what are the rules on paying it back?

Subaccounts. Earmarking portions of a task's allowance for use with a particular device or by a particular thread within the resource container may require richer API support (a topic of future research).

3.3 Currentcy Accounting

On the device side, various schemes may be appropriate for charging tasks for access to devices. This may reflect actual energy costs or there may be rate structures designed to accomplish some energy objective. The strategies fall into the following categories:

Debiting. The straightforward policy is pay-as-you-go using the actual energy cost of the devices until currentcy is spent. In another scenario, charges levied against a task may dynamically vary to accomplish a subgoal (e.g., an extra "tax" to discourage use or a "sale price" to encourage use).

Bidding. The task may offer a price it is willing to support for access to an energy consuming resource. The bid does not necessarily imply that the task will be charged that amount for an activity.

Pricing. The price of a resource, which may be dynamically changing over time, is a way to encode thresholds in terms of currentcy and may interact with bids (e.g., in a negotiation protocol). Pricing may be decoupled from debiting to enforce threshold levels without skewing accurate accounting for the resource. Pricing may also encode the power state of a device (e.g., the price of a disk access is discounted when the disk is already spinning and no spinup is required).

Examples of creative combinations of charging, pricing, and bidding policies arise with the disk management policies in Section 4.4. We believe that expressing policies in terms of allocation and accounting operations on currentcy is a powerful way to unify resource management.

4 Currentcy-based Policies

In this section, we illustrate the construction of policies to address a set of subgoals, given the overall goal of achieving a target battery lifetime:

Currentcy conserving. A currentcy conserving policy provides service in response to demand for energy as long as unspent currentcy is available in an epoch. This is important to minimize residual energy when the target battery lifetime has been reached. Too much residual energy indicates an overly conservative management of the resource and lost opportunities for improved performance.

Proportional energy use. Ideally, the energy consumption of each task will match its assigned *share*. The energy consumption can be lower if the requirements of the task are low enough to be fully satisfied by the level of energy available to it.

Response time variation. The allocation of energy in epochs has the potential to cause large variations in response time and bursty behavior. Thus, one goal is to reduce the variation in response times.

Energy efficiency. Encouraging the most efficient use of a device's power saving modes allows performance to be achieved at lower energy costs.

Traditional resource management policies tend to concentrate on a single component of the system. For example, CPU scheduling algorithms are typically concerned only with tasks on the ready-to-run queue. Blocked processes have always posed subtle complications, especially for proportional and real-time schedulers as they are treated as leaving and later rejoining the set of ready-to-run processes. With the focus on energy, the complications are much more explicit. Blocked tasks may be consuming energy in devices other than the CPU. A process waiting for completion of a disk request is responsible for the energy consumption of the disk access. Ready-to-run processes may also be consuming energy in other devices (e.g., due to asynchronous I/O) while competing for the CPU. On the other hand, a process that is blocked on user input (e.g., "think time") may be considered truly idle[2]. In typical management of the wireless NIC, energy has already been spent for receiving and initial processing of incoming network packets before it is known which task should be charged (and whether that task has enough currentcy to pay). The challenge of global energy management is to address these kinds of interactions that are often hidden in per-device management.

4.1 Goal: Currentcy Conserving Scheduling

The epoch-based currentcy allocation scheme is motivated by the overall goal of achieving a target battery lifetime by approximating a constant power consumption. To prevent large power peaks, our allocation policy caps the amount of unspent currentcy a task can save from epoch to epoch. Unspent currentcy that exceeds the cap is essentially thrown away, even if there is unmet demand by other tasks with insufficient currentcy. If there are enough instances of tasks that underspend their allocation during an epoch there can be a gradual accumulation of residual energy capacity because of the forfeited currentcy.

There are a couple of ways to deal with the residual energy problem. One is to adjust the overall allocation level when the system detects that the battery is not being drained at the expected rate. If there is a consistent pattern of underspending by some tasks, the total allocation will grow, slowly at first, and be proportionally distributed to all tasks. Thus, needy tasks will benefit

[2]If we were doing accounting for the display, the display energy during such think time might be attributed to the interactive application.

from receiving their share of a larger overall allocation.

Another approach is to explicitly redistribute excess currentcy to other tasks with insufficient currentcy for their energy demands. Specifically, we propose a two-step policy that first dynamically adjusts the per-task cap to reflect each task's energy needs (captured as the level of currentcy spent in previous epochs) as well as its specified share and second redistributes currentcy amounts that overflow some task's cap to other tasks whose limits have not been reached. There are a number of solutions to the subproblem of calculating the new cap, given the history of past consumption. We have found that a formula that increases the cap quickly and decreases it slowly produces desirable behavior.

The overall consumption should more closely match the overall allocation with this redistribution (at the cost of upsetting proportionality, considered below in Section 4.2), thus reducing the residual energy. We refer to this algorithm as the Currentcy Conserving (CC) allocation policy. This allocation attempts to make supplemental currentcy allocations to tasks that have shown they can use the currentcy effectively during an epoch.

4.2 Goal: Proportional Energy Use

Even when allocations are strictly proportional, the ability to spend proportionally depends on policies that control access to resources, such as the schedulers for the CPU, network bandwidth, and disk. In addition, there are interactions between scheduling and allocation since the ability to actually spend currentcy affects future caps in our allocation algorithms. In this section, we explore the role of various schedulers (i.e., CPU, network bandwidth) in delivering the opportunity for proportional energy consumption and the role of currentcy in unifying these decisions. We begin with a consideration of CPU scheduling.

The base case for our explorations is the default Linux process scheduler, amended with the condition that the resource container of a process to be dispatched must contain available currentcy; otherwise, it is not considered ready to run again until the next epoch (when it receives a new infusion of currentcy). The amount of available currentcy (the task's *energy budget*) is not a factor that influences the scheduling decision in any more substantial way than the ability to pay or not.

One might expect that by adapting a proportional scheduler to tasks' shares, better proportional energy use can be achieved. We consider stride scheduling [17] as representative of a local (CPU-only) scheduler using each task's (static) share to determine its stride value.

Finally, we propose an energy-centric scheduler (EC scheduler) that accounts for the task's energy consumption (globally – regardless of where in the system the currentcy is spent). The next process to be selected is one whose resource container has the lowest amount of currentcy spent relative to its specified share. This can be viewed as a bidding algorithm with the lowest bidder winning. As in traditional stride scheduling, an adjustment is made to "catch up" with the current pass when a process temporarily leaves the ready queue (e.g., blocked on synchronization or a synchronous I/O operation) and then rejoins.

To ensure that a process that is intermittently ready and blocked has sufficient opportunities to spend its currentcy, we can weigh the basis against which the energy consumed this epoch is compared by a factor defined to be the task's share divided by the amount of currentcy actually spent in the last epoch. This factor produces a *dynamic share* used to replace the fixed share value in the calculation of the task's stride. This biases allocation in favor of interactive tasks and helps them consume more of their share of currentcy whenever they are actively competing for the processor. Our EC scheduler also incorporates one final feature called *self-pacing* that will be described in the next section (Section 4.3) with the goal of smoothing response times. Thus, there are three aspects of the EC scheduler that can be mixed in various combinations: the consumption-based stride, with or without dynamic shares, and with or without self-pacing.

Currentcy is a global abstraction and proportional energy use extends to all other devices on a platform[3]. This is fairly straightforward for operations initiated on this platform via system calls by running tasks. One particularly interesting challenge to achieving proportional energy use is managing the wireless network bandwidth, especially for incoming packets.

[3]Currently, we only explicitly manage the CPU, NIC, and disk subsystem.

The tricky issue for incoming traffic is that by the time a packet has been received and management actions can be applied, the energy to receive it has already been consumed in the wireless card. This makes it difficult to selectively receive packets destined for tasks with available currentcy as opposed to tasks without currentcy.

Our solution leverages the TCP protocol's congestion control that responds to a dropped packet (one that does not receive an acknowledgment) by reducing transmission rate. Thus, we modified the Linux network packet processing code to implement a work conserving proportional bandwidth allocation policy. Our scheme identifies flows whose associated tasks have consumed bandwidth beyond their currentcy-determined share. The kernel begins to drop incoming packets on such flows, triggering the TCP sender's back off response (e.g., typically reducing their transmission rate by one half). Assuming other tasks can consume released bandwidth, packet dropping will continue until all connections consume bandwidth in proportion to their task's energy share. This approach effectively moves the selective throttling of incoming traffic to the *outside* of the network card, thus saving our battery.

4.3 Goal: Low Variance in Response Time

The epoch-based allocation has the potential to produce bursty behavior as tasks consume currentcy as quickly as they can at the beginning of an epoch and then go idle after consuming their budget. One approach to smoothing consumption rates (and as a side-effect, response times), is to shorten the epoch.

Another approach to managing the rate of consumption is self-pacing in our EC scheduler. The idea is to delay a task if its consumption of currentcy is ahead of schedule during an epoch. Progress is defined as the amount of currentcy spent thus far in the current epoch divided by the task's budget for the epoch. If this progress is greater than the ratio of elapsed time in this epoch over epoch length, then the task is delayed and the processor may go idle for a short interval of time[4].

[4]This approach to stretching execution is appropriate for a non-Dynamic Voltage Scaled processor. If available, DVS would be a preferred alternative to consider.

4.4 Goal: Energy Efficiency

Encouraging more energy efficient use of devices is an important function of an energy centric operating system. Currentcy provides a means for passing along the savings to tasks that cooperate through their usage patterns. The disk presents unique challenges and opportunities for currentcy-based policies since it has non-uniform power consumption. The cost of spinning up the disk is much greater than keeping it spinning for a short duration. In this section, we consider techniques for more efficient disk access, focusing on the spinup/spindown power costs. This introduces opportunities for debiting, bidding and pricing in the context of our currentcy model. The policy space for these approaches is very large, and many solutions may require an API for application involvement. Therefore, in this paper we have limited our studies to techniques for managing disk access using pricing and bidding that can be implemented solely within the operating system.

Intuitively, we want to amortize spinups across multiple disk operations, with the goal of encouraging more bursty behavior. Our original disk charging policy [19] was already non-trivial in sharing the costs of spinning up and waiting for spindown among multiple tasks if they jointly participate in a spinup-to-spindown session. The key to more effectively manipulating the spinup/spindown behavior is shaping the disk access patterns to take advantage of this cost-sharing benefit within the charging policy.

Pricing disk accesses can be used to reward a task for performing disk accesses in bursts. One approach we investigate sets the *entry price* of a disk access that requires a spinup cost much higher than the actual cost. When the access is actually permitted, we then debit the actual cost. This forces the task to accumulate enough currentcy to ensure that it can execute for a reasonable amount of time following the first access in hopes of generating more disk accesses while the disk is spinning.

We augment this pricing policy with the ability of tasks to bid on disk accesses.[5] Tasks can indicate they are willing to contribute certain amounts toward the price of spinning up the disk. One goal of this tech-

[5]This is a natural place for API extensions. However, the OS can apply this technique transparently by checking the task's budget for sufficient surplus, analogous to a credit check.

nique is to enable multiple tasks to pool their currentcy and cooperatively use the disk in an energy-efficient manner.

Traditional techniques of skewing access patterns are amenable to currentcy-based variations. These include exploiting block caching and delaying writes while the disk is not spinning, piggybacking prefetching upon requests that spin up the disk on demand, and managing the buffer allocation. Thus, we explore a prefetch buffer allocation policy tied to the average disk access cost. Subject to limitations on the number of buffers systemwide, this policy attempts to reduce the costs (via effective prefetching) and make them uniform across tasks (which can tend to synchronize tasks into producing batches of disk activity).

We also trigger prefetching operations that cause spinups using a bidding function based on the fraction of consumed prefetch buffers. Investigating the range of potentially useful bidding functions is clearly beyond the scope of this paper. We provide results for one bidding function that sets a bid offer to zero if less than 80% of the prefetch buffers are consumed otherwise to a weighted linear value ($bid = entry\_price * (percent\_buffers\_consumed - 80)/(100-80)$. This corresponds to a function where interest is greatly increased as the task nears a demand fetch.

The operating system can also employ the pricing schemes to manage its internal operations. Specifically, we modify the disk flush daemon to wait a threshold amount of time before initiating disk writes that require spinups, with the entry price of a disk request monotonically falling throughout this period – interacting nicely with the bidding schemes previously described. The daemon performs a large number of writes once it starts flushing pages to a spinning disk. The default Linux page flush policy is to check every 5 seconds for dirty pages that have not been accessed for 30 seconds and write those to disk. Our new policy includes two parameters, FlushStart and FlushEnd, that are used to control when dirty pages are written to disk. The flush daemon checks for dirty pages that have been idle for FlushStart seconds before starting to write to disk. At this point it writes all dirty pages that have been idle for more than FlushEnd seconds. As with the spinup pricing, the goal is to create bursts of activity.

5 Evaluation

This section evaluates several of the policies described above. Our goal is not to provide the optimal policy, but to show that policies formulated within the unified currentcy model are superior to more traditional (per-device) policies and to demonstrate the advantages of the currentcy model.

5.1 Applications and Metrics

We use several applications (described in Table 1) to create typical workload scenarios for a battery-constrained laptop user. We can easily envision situations in which the user may want to have multiple tasks running concurrently (e.g., doing background jpeg encoding of a set of stored images while viewing the already encoded jpegs in slide show mode or listening to an MP3 while running through the slides of a powerpoint presentation). For each experiment we use different combinations of these applications to emphasize specific aspects of the policy space. Each application presents a different set of demands for CPU, network bandwidth, disk I/O, or interactive "think time".

Within ECOSystem, we monitor the currentcy available for allocation each epoch and the currentcy consumed by each application during each epoch. It is also possible to track consumption by device. We then present our results in terms of average power (mW) derived from currentcy consumed or allocated (mJ) per epoch (sec). We also present appropriate application-specific performance metrics.

5.2 Low Residual Energy

Our first set of experiments explores our policy for reducing residual energy through currentcy conservation. As a qualitative argument, we note that without an abstraction similar to currentcy, it is difficult to articulate precisely what residual energy means or how one might enforce a target battery lifetime. Monitoring the state of the battery as a separate device-specific resource offers little in the way of control over the resource. Thus, we compare against the original currentcy allocation [19] to show that, even with the problem defined in terms of currentcy, a battery-level feedback mechanism is less effective in reclaiming residual energy than explicit currentcy conservation. In our

Application	Description
gqview	Image viewer, available at http://gqview.sourceforge.net
ijpeg	SPEC2000 benchmark, image encoding
RealPlayer	Video player
Netscape	Web browser
x11amp	MP3 player
StarOffice	PPT presentation

Table 1: Applications

original policy residual energy accumulates if a task does not spend all of its currentcy and has exceeded its currentcy cap.

To evaluate the benefits of currentcy conservation we use a workload consisting of the gqview image viewer and ijpeg. Gqview is set to autobrowse mode where it continuously loads each of 12 images in a directory with a 10 second pause between each image. The images are copies of a high fidelity 0.5MB jpeg file, differing only in that each image has a unique number. Ijpeg is run in a loop to continuously execute the SPEC command line, encoding an image from the reference data set.

For this experiment, we set the target battery lifetime at 90 minutes, and set a desired energy share of 66.6% for gqview and 33.3% for ijpeg. These settings correspond to an overall average power of 12000mW with 8000mW and 4000mW for gqview and ijpeg, respectively. For presentation, we plot average power consumption for each epoch versus time.

Figure 1 shows our results. From this data we make several important observations. First, for the original allocation policy (Figure 1a) we see that the total power available for allocation increases dramatically near the end of the target battery lifetime. There is approximately 6.7% of the original battery capacity remaining. The simple redistribution approach that returns the unused currentcy (beyond the task's cap) to the overall energy resource initially spreads the excess over a large number of epochs, but as the target battery lifetime approaches there is less time over which to spread the excess. Intuitively, each epoch consumes only a fraction of the total excess and thus available energy continues to grow.

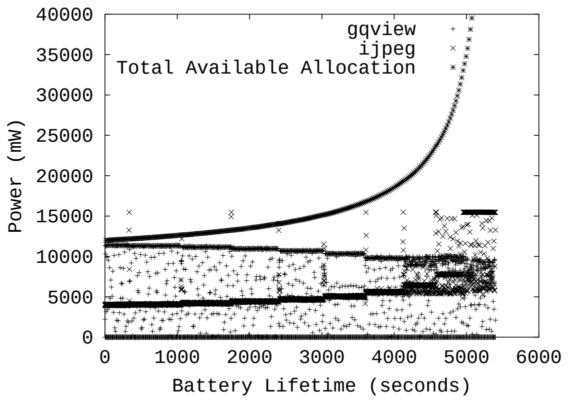
The second observation we make based on Figure 1a is that as time progresses, gqview's average power consumption decreases despite the increase in total availability. This is because the increase in available energy enables ijpeg to consume more and more CPU time, undermining gqview's ability to execute and thus consume power. Finally, we note that gqview exhibits variation in its power consumption due to variations in its execution behavior.

Figure 1b and Figure 1c show the average power consumption of gqview and ijpeg when using the currentcy conserving allocation. We use two separate graphs for clarity. From these figures we see that there is no significant change in the available energy as we near the end of the target lifetime. Little residual energy capacity remains (less than 1%). By exploiting information in tasks' currentcy budgets, the currentcy conserving allocation policy successfully utilizes the available energy as compared to the approach that reacts to observed excess battery capacity.

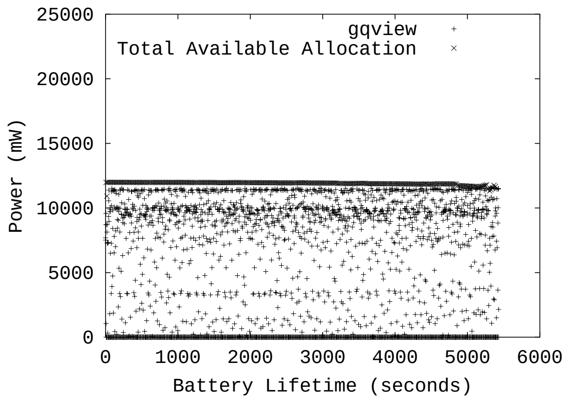
From these figures we also see the variations in both applications' power consumption due to gqview's execution variations. We observe that the power consumption of both gqview and ijpeg can exceed their allocation share because of currentcy accumulating up to their caps. To provide a better understanding of the simultaneous execution of ijpeg and gqview, Figure 1d presents average power consumption over a 100 second time interval.

5.3 Proportional Energy Use

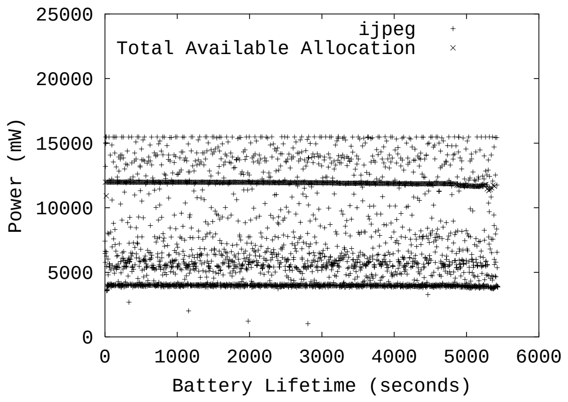
Given a particular amount of energy per epoch, we now investigate proportionally sharing this fixed energy allotment among competing tasks. The policies we implemented are described in Section 4.2. We begin by analyzing the effects of CPU scheduling using our minimally modified Linux scheduler, the static energy-based stride scheduler, and our energy-centric scheduler with dynamic shares. This is followed by



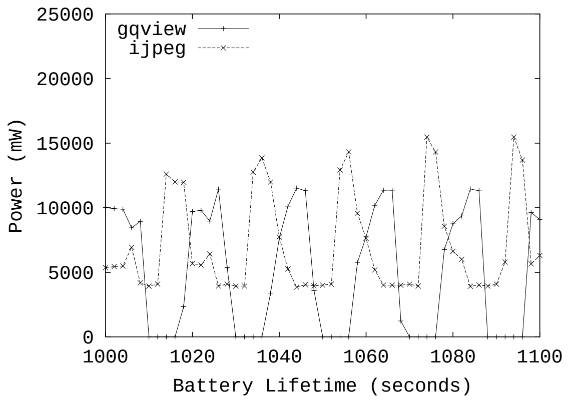
a) Original Allocation Policy



b) Currentcy Conserving Policy: gqview



c) Currentcy Conserving Policy: ijpeg



d) Time Varying Behavior

Figure 1: Average Power Consumptions and Total Allocations for the Original and the Currencty Conserving Allocation Schemes

an evaluation of the ability to proportionally share the network bandwidth.

5.3.1 Proportional CPU Scheduling

Table 2 shows our results when simultaneously running gqview and ijpeg with various total currentcy allotment and energy share values. The first column shows the total allocation and the shares for each application, mapped into power. The subsequent columns show the average power consumed and a performance metric for each application. The performance metric for gqview is the average delay to completely display the image. When running alone and unthrottled, gqview consumes an average of 5856mW. Since ijpeg is CPU bound (i.e., capable of consuming 15.55W), its performance metric is CPU utilization.

The results for the Linux scheduler show that for highly constrained scenarios (6000mW total) each application receives its appropriate share. However, as the total energy increases, gqview's average power consumption initially increases, then decreases, and it never achieves its allocated share. This is because ijpeg is always competing with gqview for the CPU, and the round-robin Linux scheduler only gives gqview 50% of the CPU when it is active.

The static energy-based stride scheduler experiences similar problems when gqview and ijpeg are competing for the CPU. Gqview is unnecessarily penalized for voluntarily reducing its energy consumption during idle periods.

Our energy-centric scheduler extends the stride scheduler in two important ways. First, it selects the next task having the lowest amount of currentcy spent relative to its share, and second it dynamically computes a task's stride by including information about

Allocation (mW) total (gqview:ijpeg)	**gqview** Ave Power Used(mW)	Average Delay (sec)	**ijpeg** Ave Power Used(mW)	CPU Util (%)
Linux Scheduler				
6000 (4000:2000)	4027	13.68	2003	12.88%
9000 (6000:3000)	5164	7.969	3781	24.32%
12000 (8000:4000)	4918	9.629	7143	45.94%
15000 (10000:5000)	4749	10.715	10423	67.03%
Stride Scheduler				
6000 (4000:2000)	4011	13.467	2005	12.89%
9000 (6000:3000)	5423	7.207	3336	21.45%
12000 (8000:4000)	5048	8.771	7073	45.48%
15000 (10000:5000)	4893	9.259	10182	65.48%
Energy-Centric Dynamic Share Scheduler				
6000 (4000:2000)	3991	13.86	2040	13.12%
9000 (6000:3000)	5508	6.712	3308	21.27%
12000 (8000:4000)	5567	6.751	6448	41.46%
15000 (10000:5000)	5521	6.804	9648	62.05%

Table 2: The Effect of CPU Scheduling Policy on Proportional Sharing of Energy

past energy consumption. We omit the energy-centric scheduler without dynamic shares since its primary benefit is to compensate for energy consumption of other devices rather than for periods of complete inactivity as in gqview. For this case, it degenerates to the above stride scheduler. From Table 2 we see that with the energy-centric scheduler, gqview always receives its appropriate energy share up until the point where it receives approximately enough, producing response times only slightly longer than the 6.14 seconds required when it executes alone.

5.3.2 Proportional Network Scheduling

Next, we consider proportional energy consumption for the combined use of the network and the CPU. If the network scheduler is not currentcy-aware, then tasks with very little currentcy allocation can indirectly throttle applications that should have a larger proportion of energy.

To create a stressful condition where the network is the bottleneck, we set the wireless ethernet card to 1Mbps. We execute RealPlayer and netscape as equal priority tasks and ijpeg as a background low priority CPU intensive application. Realplayer's input consists of video clips from Chinese television, while netscape continuously reloads a web page with 6 GIF images and a banner with a macromedia Flash advertisement. When executing without energy constraints, RealPlayer consumes about 4210mW and 50,000 B/s of network bandwidth to execute without pauses in video playback, while netscape consumes 9612mW and 84,273 B/s of bandwidth.

In all of our network experiments we use the currentcy conserving energy allocation policy and constrain the total power consumption to 12000mW. Table 3 presents results for three of the scheduler design points: 1) Our energy-centric CPU scheduler and an energy oblivious network scheduler, 2) the default Linux CPU scheduler with an energy-centric network scheduler, and 3) our combined energy-centric CPU and network schedulers. We omit the case where neither the CPU or network scheduler are energy-centric, since our results show that not having an energy-centric network scheduler fails to provide proportional energy allocation. This is because netscape is allowed to consume an unfair portion of network bandwidth, reducing RealPlayer's ability to execute, and producing an excess in currentcy that is reallocated to the other tasks. This results in netscape and ijpeg getting more of the CPU and consuming a much larger energy share than the intended allocation. The visible effect on RealPlayer is the introduction of significant pauses in the video playback.

Application	Allocation (mW)	CPU Power (mW)	Network Power (mW)	Disk Power (mW)	Total Power (mW)	Network Bandwidth (B/s)
Energy-Centric CPU Scheduler, Energy Oblivious Network						
RealPlayer	5714	2240	367	289	2897	39544
Netscape	5714	6899	548	463	7911	59698
ijpeg	572	1082	0	0	1082	0
Default Linux CPU Scheduler, Energy-Centric Network						
RealPlayer	5714	2631	441	117	3190	43455
Netscape	5714	6689	437	875	8002	43352
ijpeg	572	801	0	0	801	0
Energy-Centric CPU Scheduler, Energy-Centric Network						
RealPlayer	5714	2719	542	370	3631	55026
Netscape	5714	5558	336	274	6169	32020
ijpeg	572	2020	0	0	2020	0

Table 3: Proportional Sharing: CPU and Network

When we use an energy-centric network scheduler, but the default Linux CPU scheduler, we see that bandwidth (and network power) is allocated according to the specified proportions (1:1). However, RealPlayer still does not achieve its necessary level of power to prevent pauses in playback.

The best results are only obtained by using energy-centric schedulers for both the CPU and network. Specifically, RealPlayer has enough currentcy to execute without pausing. Delivering this level of performance is not achieved by either scheduler in isolation.

5.4 Low Variance Response Time

Given a case in which power consumption must be constrained, we explore the effects on response time of our two approaches for reducing bursty behavior: 1) reducing the epoch length and 2) extending the energy-centric dynamic share scheduler with self-pacing to enable applications to spread their execution over the entire energy epoch. This approach exploits the ability of currentcy to reflect an application's rate of progress.

One potential drawback of shorter energy epochs is that the overhead of energy allocation could become a performance bottleneck. However, experiments show the overhead for energy allocation is very small. Even if we perform energy allocation every 10ms (a timer interrupt occurs every 10ms, while the CPU scheduling quantum is 60 ms in our system), the overhead is only 206μs for 18 resource containers and 40μs for 3 containers.

To explore the effect of scheduling on application performance we run netscape and continuously load the Milly Watt web page (http://www.cs.duke.edu/ari/millywatt). This page contains a banner image and some simple text. The autoload is implemented using a javascript and this also allows us to measure the page load latency. This latency is composed of several http requests, displaying the content and updating the disk cache. We set the think time between successive page loads at 2 seconds. Executing without any throttling requires about 2197mW.

We evaluate both an epoch-based approach that uses 0.01 second epochs, and the self-pacing approach described in Section 4.3 with 10 second epochs. We allocate 1200mW to netscape and measure 54 consecutive page loads for the self-paced test and 41 page loads for the epoch based approach. Differences are apparent in Table 4 when we examine the delay for a page load. Although the average delay is similar for the two policies, the self-paced scheduler has much lower variation in the delay. This translates into a user perceived difference in performance as the self-pacing policy provides a visibly smoother display of the web page. We note that similar visible differences occur when executing other applications, such as RealPlayer, Acrobat, and StarOffice. In each case the self-pacing policy visibly reduces jitter (e.g., long pauses in RealPlayer's presentation of a video clip).

Scheduling	Power Consumption (mW)				Delay (seconds)			
	CPU	Disk	Network	Total	Min	Max	Average	Std. Dev
Unthrottled	1047	1013	136	2197	0.27	0.68	0.43	0.11
Epoch	351	812	48	1212	1.0	33.8	3. 8	5.8
Self-Pacing	313	842	43	1199	3.3	5.6	4.0	0.6

Table 4: Response Time Variation

TCP/IP protocol processing can be sensitive to the timing (smoothness) as well. Packets may be dropped and the TCP window size reduced if the packets are not processed in time. This can further exacerbate the jitter incurred by the epoch-based approach. Our self-pacing scheduler smooths out execution and reduces jitter problems.

5.5 Disk Management

The global aspect of currentcy also enables the development of policies for increasing the energy efficiency of a given device. In this section we consider disk access scheduling. In particular, we show how the currentcy model enables policies based on pricing and bidding. First, we explore techniques to coschedule disk accesses for two applications with the goal of reducing overall disk power consumption. We then show how this same approach can be used to reduce the cost of delayed disk writes by the I/O daemon. Finally, we present preliminary results on prefetch buffer management. We note that the policy space related to pricing and bidding in disk management is much larger than we can cover in this paper, and we leave further investigations as future work.

Reads In our first experiment we execute ijpeg and gqview concurrently, and each application demand fetches data from the disk. Ijpeg performs image compression on a set of ppm format image files. Each file is a copy of the same SPEC input (vigo.ppm) that is 2,359,355 bytes. If running unconstrained, ijpeg requires about 2.452 seconds to process each file and start to read the next. Gqview displays the same set of image files using autobrowse mode with a 50 second think time. We set the total power allocation to 1500mW and the two tasks each get an equal share of 750mW. These severe constraints are used to accentuate the disk's impact on performance. The entry price for initiating a disk access is set to 24000mJ (twice the combined cost of spinup and spindown). We use an immediate disk spindown.

Without pricing/bidding, the total average disk power consumption is 911mW, with 403mW and 508mW for ijpeg and gqview, respectively. Our currentcy-based policy formulated in terms of pricing/bidding reduces this value to 655mW (313mW ijpeg and 342 gqview) by engineering more task cooperation in disk spinup sessions. Furthermore, the performance of each application improves, particularly ijpeg which requires only 57 seconds to process the file compared to 74 without pricing/bidding.

The next experiment is designed to show the benefits of bidding for energy efficient disk prefetching. We set the total allocation at 1500mW and execute ijpeg (same input as above) with 300mW concurrently with x11amp, which receives 1200mW allocation. X11amp reads a 3MB file, and is amenable to prefetching because of its sequential access pattern. We use the combined pricing/bidding approach where there is a high entry price for a disk spinup and x11amp contributes by bidding based on its prefetch buffer consumption. The average total disk power consumption is 357mW compared to 565mW without pricing/bidding. Ijpeg's average disk power consumption reduces from 365mW to 229mW and its performance improves from 90 seconds per file to 66 seconds. X11amp's disk power consumption reduces from 200mW to 128mW, and it does not incur any pauses in either policy.

Writes The next experiment is designed to evaluate how the currentcy model enables formation of better deferred writeback policies. We use a microbenchmark that writes 4KB every 5 seconds. With the default Linux policy our microbenchmark incurs an average disk power consumption of 624mW with the default 30 second spindown timeout. Our new disk flushing policy uses the 24000mJ entry price and monotonically decreases this over a 60 second interval (Flushstart = 60 seconds). We set Flushend to 5 seconds and use a spindown timeout of 1 second. This policy reduces the average disk power consumption to 239mW. The benefits of scheduling disk writes could also be coupled with read scheduling to further amortize the spinup cost across a larger number of accesses.

Prefetch Buffer Allocation The role of currentcy in our prefetching policies is to balance the buffer allocation among prefetching-friendly tasks to facilitate more globally synchronized disk activity. To show the potential benefit of such cooperation, we compare local and global prefetching behavior for two applications (x11amp and StarOffice) that exhibit sequential access patterns, since the unified buffer cache in Linux can easily detect these sequences and initiate prefetching. The spindown timeout is set to 1 second. X11amp reads a 3MB file, while StarOffice reads an 11MB presentation with 14 slides and executes in auto-transition mode with approximately 20 seconds between slides.

When the prefetching is performed locally using the default Linux buffer allocation of 32 buffers for each task, the spinup and spindown costs are incurred for each task. The disk power consumption for x11amp is 186mW and StarOffice consumes 219mW for a combined 405mW.

In contrast, a global prefetching policy synchronizes the prefetching operations of the two tasks by allocating prefetch buffers according to a task's average disk access cost, which is determined by the task's buffer consumption rate. In this experiment, x11amp requires 256 prefetch buffers and StarOffice uses 1000. This significantly reduces the total disk power consumption to 280mW, with 65mW for X11amp and 215mW for StarOffice. StarOffice receives very little benefit since it is dominated by the cost to actually read the data, whereas X11amp leverages StarOffice's relatively large number of disk accesses.

6 Related Work

Attention to the issues of energy and power management is gaining momentum within operating systems research.

Work by Flinn and Satyanarayanan on energy-aware adaptation using Odyssey [6] 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. In contrast, our work focuses on managing global system resources in a unified manner. Unmodified applications and those that are not necessarily able to change "fidelity" benefit from our approach. Overall, we view our efforts as complementary: the operating system should manage global system devices in response to application requirements and the application should adapt its behavior when appropriate to reduce energy consumption.

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, there have been contributions addressing CPU frequency/voltage scheduling [18, 13, 7, 14], disk spindown policies [12, 4, 3, 10, 8], memory page allocation [11, 2], and wireless networking protocols [9, 15]. The emphasis in 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, as illustrated by our incorporation of spindown policies, and will impact the charging policies for such devices in our framework.

7 Conclusion

Energy management is an increasingly important aspect of system design. Our previously proposed currentcy model provides the framework for the operating system to manage energy as a first-class resource. This paper demonstrates that that currentcy model can be used to specify energy management policies that span multiple devices and diverse applications.

Using our ECOSystem prototype operating system, we implement several currentcy-based policies, including: currentcy conserving scheduling algorithms that reduce residual battery capacity, proportional energy sharing, self-pacing to smooth response time variation, and energy efficient disk management. Our results show that the currentcy model is a powerful framework for expressing energy management policies and that our currentcy-based policies, by being able to capture aspects of global energy use, provide more coherency to system-wide energy management.

References

[1] Gaurav Banga, Peter Druschel, and Jeffrey 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] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramiam, and M.J. 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.

[3] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *2nd USENIX Symposium on Mobile and Location-Independent Computing*, April 1995. Monterey CA.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla S. Ellis. Power aware page allocation. In *Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS IX)*, pages 105–116, November 2000.

[12] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *USENIX Association Winter Technical Conference Proceedings*, pages 279–291, 1994.

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

[14] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating syste ms. In *Proceedings of the 18th symposium on operating systems principles*, pages 89 – 102, October 2001.

[15] Mark Stemm and Randy 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.

[16] Amin Vahdat, Carla Ellis, and Alvin 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.

[17] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, 1995.

[18] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *USENIX Association, Proceedings of First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994. Monterey CA.

[19] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002. To appear.