

Currentcy: A Unifying Abstraction for Expressing Energy Management Policies

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 nature of energy creates challenges and opportunities for developing operating system policies to effectively manage energy consumption in battery-powered mobile/wireless devices. The 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 energy goals and to unify resource management policies across diverse competing applications and spanning device components with very different power characteristics.

This paper explores the ability of the *currentcy* model to capture more complex interactions and to express more mature energy goals than previously considered. We carry out this exploration in ECOSystem, an “energy-centric” Linux-based operating system. We extend ECOSystem to address four new goals: 1) reducing residual battery capacity at the end of the targeted battery lifetime when it is no longer required (e.g., recharging is available), 2) dynamic tracking of the energy needs of competing applications for more effective energy sharing, 3) reducing response time variation caused by limited energy availability, and 4) energy efficient disk management. Our results show that the *currentcy* model can express complex energy-related goals and behaviors, leading to more effective, unified management policies than those that develop from per-device approaches.

1 Introduction

Energy is an increasingly important system resource. This is most evident in battery-powered mobile computing platforms, from laptops to tiny embedded sen-

sor 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, protocols using wireless power modes), recent work advocates that the operating system should explicitly manage the system-wide role that energy plays [17, 5] and view it as an opportunity and challenge for unifying resource management.

Our recent development of a framework for an energy centric operating system [22] proposes *currentcy* as a unifying abstraction for the management of a broad variety of system devices that consume energy. This work demonstrates the use of the *currentcy* model for expressing our overall battery lifetime goal and capturing the impact of individual system devices on battery lifetime. However, just as there is no single performance metric for all workloads, there is no single energy goal that satisfies all mobile/wireless scenarios. Thus, for this work, we set out to determine whether *currentcy* is general enough to express additional complex system behavior. Specifically, this paper makes the following contributions:

1. For some applications, it is important not just to achieve a target battery lifetime but to perform more work during that lifetime. Since characterizing “work” in general-purpose workloads is difficult, we look at fully utilizing the available energy within the specified time. Consider a sensor node running on rechargeable solar cells. The goal here might be to minimize residual energy remaining at sunrise after operating through the night (when it becomes possible to recharge – assuming reliable weather forecasts) to deliver maximum system utility. Any residual capacity at the end of the designated lifetime suggests overly conservative management and lost opportunities. Experience shows that this situation can result from a mismatch of the user specifications and actual demand. Thus, we translate this goal into the *currentcy* model and develop a *currentcy* conserving

*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/>.

energy allocation policy to reclaim unspent energy by adapting to observed energy consumption patterns.

2. Experience with the ECOSystem prototype also indicates that there can be subtle interactions between energy allocation and CPU scheduling policy. Scheduling that is oblivious to energy consumption may not provide adequate opportunity to spend currentcy allocations. Returning to our sensor example, suppose there is an important task that consumes most of its energy in the wireless network interface, communicating sensor readings, rather than on processing within the CPU. Such a task may experience a form of priority inversion. Thus, we develop a currentcy-based scheduling policy that recognizes the global relevance of energy consumption anywhere in the system on the scheduling decision. The result is more robust proportional sharing of energy regardless of which resources are favored by tasks.

3. Response time variability is disruptive in many applications. Whenever energy availability is constrained – which in ECOSystem means currentcy allocations are limited – it becomes important to have a steady rate of consumption. Thus, we develop a currentcy-based scheduling policy that achieves well-paced energy consumption, reducing response time variation.

4. For devices that have nontrivial transition costs between power states, such as a disk with spindown capability, there is potential for increased energy efficiency. We demonstrate how to shape disk access patterns to amortize the energy costs of spinup/spindown across multiple requests and thereby reduce the average energy used per request. We further show the energy and performance benefits of aggressive prefetching while the disk is spinning.

In summary, our experimental results show that the currentcy framework is successful in achieving more mature energy goals than previously pursued. These include: reducing residual energy, dynamically balancing per-task energy supply and demand, lowering response time variation, correcting energy-related scheduling inversions for improved energy sharing, and increased efficiency for disk accesses.

The rest of this paper is organized as follows. Section 2 describes the currentcy model and its implementation in ECOSystem, a Linux-based prototype, followed by a discussion of related work. Next, in Section 3, we outline several methods for manipulating currentcy to implement energy-related goals. Sections 4 through 8 describe the formulation of several energy goals, beyond simple battery lifetime, built upon the currentcy management framework. We propose and evaluate solutions for

each of these problems. Section 9 concludes this paper.

2 Background and Related Work

2.1 The Currentcy Model

The ECOSystem approach is based upon a unifying currentcy model. The key feature of this model is the use of a common unit—*currentcy*—for energy accounting and allocation across a variety of hardware components and tasks. Currentcy is an abstraction for explicitly representing energy as a resource, precisely specifying energy-related goals, and capturing the interactions among energy consumers in the system. It is the basis for characterizing the application power requirements and gaining access to any of the managed hardware resources. Currentcy is the mechanism for establishing a particular level of power consumption and for sharing the available energy among competing tasks.

Originally, the primary goal of ECOSystem was to achieve a target battery lifetime. Not only is this a well-defined metric to adopt as a starting point for explicitly managing energy, but there are also interesting application scenarios for which this is an appropriate objective. Exploiting battery properties that relate lifetime and discharge rate, ECOSystem expresses this goal in terms of the currentcy model and allocation strategies.

There are two levels to the energy allocation strategy. The first level allocation determines the amount of currentcy to collectively allocate among all tasks system-wide. We divide time into energy-epochs. At the start of each epoch, ECOSystem allocates a specific total amount of currentcy. For our original purpose, the overall currentcy allocation is determined by the discharge rate necessary to achieve the target battery lifetime. By distributing less than 100% of the currentcy required to drive a fully active system during an epoch, components are idled or throttled.

The second level 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 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 that would violate our battery assumptions. Consequently, there is a cap on the maximum amount of currentcy any individual application can save. Thus, the per-task allocation represents income in each epoch, whereas the cap represents a limit on the balance accumulated within the task’s account.

ECOSystem uses a reimplementaion of the Resource Containers [1] abstraction 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 debited 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.

The energy accounting challenge of tracking energy use and attributing it to the responsible task is addressed through a power states model maintained within the framework. This model allows us to track interactions among tasks through their use of energy in device access. For example, tasks may be consuming energy in devices even when they are inactive in the CPU (or blocked). 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.

The ECOSystem prototype [22] is 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 that we model in ECOSystem with costs of 1.65mJ per block access and 6000mJ for both spinup and spindown, and with progressive costs/timeouts for levels of idle power states. 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. 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.

Initial experience and experiments with the prototype show that it can successfully deliver its goal of achieving a target battery lifetime and proportionally sharing available energy among competing applications using different devices in the system. It has also identified some

drawbacks including a disproportionate impact on performance. This work refines energy goals beyond the simple battery lifetime metric. The purpose of this paper is to explore the power of the currentcy model to express more subtle and sophisticated desired behaviors. This effort represents a move from developing the framework and mechanisms toward exploring the policy space.

2.2 Related Work

Attention to the issues of energy and power management is gaining momentum within both industry and academics.

Work by Flinn and Satyanarayanan on energy-aware adaptation using Odyssey [7] 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.

Explicit energy management has also been designed for the Nemesis operating system [13]. This proposal describes how to extend the Nemesis resource accounting mechanisms, based on a calibration of device power consumption, to account for energy use by applications. Resource management is similar to Odyssey in that it is based on collaboration with applications. In Nemesis, this takes the form of an economic model for providing feedback to processes that allow them to adapt to shortages in energy availability.

Most of the 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 [6, 8, 14, 15, 20], disk spindown policies [3, 9, 11], memory page allocation [2, 12], and wireless networking protocols [10, 16]. The emphasis in much of this work has been on dynamically managing the range of power states offered by the devices. A recent paper [21] describes techniques involving buffer management policies and an API allowing application cooperation for shaping the disk request pattern to increase the effectiveness of disk spindown. This body of work is complementary to our currentcy model, as illustrated by our incorporation of spindown policies, and will impact the debiting policies for such devices in our framework.

ECOSystem incorporates several ideas from previous work. The idea of currentcy borrows from the tickets abstraction of lottery scheduling [18, 19] with the value of a currentcy unit tied to energy. The key insight that distinguishes our work is that energy normalizes resource management across the diverse set of devices that consume it. We adopt the resource container abstraction [1] as our accounting entity in order to allow more complete accounting of activity and lazy receiver processing [4] in accounting for packet processing overhead.

3 Overview of Currentcy-based Policies

In our model, currentcy represents available global system resources. Currentcy allocation and accounting express and enforce policies to achieve energy-related goals. Next, 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 that fully utilizes all the mechanisms infeasible. However, Section 3.2 discusses several goals we want to achieve within the policy space. Section 3.3 introduces the applications and metrics used to evaluate our ability to achieve our goals.

3.1 Policy Building Blocks

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. Commonly used 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 issue is addressed in Section 7.

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 currentcy) or errors in the cost model.

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. In our prior work, 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. Techniques to redistribute unused currentcy are considered in Section 4.

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. Currentcy Accounting On the device side, various schemes may be appropriate for debiting 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, prices 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 debited 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 debiting, pricing, and bidding policies arise with the disk management policies in Section 8. We believe that expressing policies in terms of allocation and accounting operations on currentcy is a powerful way to unify resource management.

3.2 Currentcy-based Policies

The previous section has given an overview of the currentcy framework and the policy space that can be explored. In the introduction, we have articulated several energy-related goals that capture desirable behavior with the goal of achieving a target battery lifetime. In this section, we translate those goals more precisely in terms of our currentcy framework.

1. Reducing residual energy capacity. We have argued that, for certain applications, it is important to minimize residual energy capacity left 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. We translate this into an allocation that is *currentcy conserving*. A currentcy conserving policy provides service in response to demand for energy as long as unspent currentcy is available in an epoch.

2. 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 available level of energy. Even when currentcy allocations are appropriately adjusted to reflect demand, schedulers that gate access to devices may not offer opportunities to spend in proportion to allocations and may interfere with adaptations determining future allocations. We translate this goal of proportional energy use into device scheduling that is *aware of currentcy consumption/demand* throughout the system.

3. Coordination of multiple devices. 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 and allocation of CPU cycles. Processes blocked for device use have always posed subtle complications on CPU scheduling. With the focus on energy, the complications become more explicit since blocked processes can still be actively consuming energy. Tracking the consumption of currentcy captures these interactions and allows the information to

be incorporated into the scheduling policies of various devices in a coherent way.

4. Response time variation. The allocation of energy in epochs has the potential to cause large variations in response time and bursty behavior. One of our goals is to reduce the variation in response times. This translates into *carefully-paced* consumption of currentcy.

5. Energy efficiency. Encouraging the most efficient use of a device’s power saving modes allows performance to be achieved at lower energy costs. This goal translates here into reducing the average currentcy cost per disk request by encouraging coalitions of tasks to share the overheads involved. Creative pricing strategies can reward such inter-task cooperation.

The challenge of unified global energy management is to explicitly address the kinds of interactions that are often hidden in per-device management.

3.3 Applications and Metrics for Evaluation

We intend to show that our currentcy model can be used to formulate policies to address the above goals. To evaluate our policies, we use several applications (described in Table 1) to create typical workload scenarios for a battery-constrained laptop user. We 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 the amount of currentcy consumed or allocated per epoch. We also present appropriate application-specific performance metrics.

In the following five sections (Section 4 through Section 8), we illustrate the construction of currentcy-based policies to address each of the reformulated energy goals. Our goal in this paper is not to provide an optimal policy, but to show that policies formulated within the unified currentcy model offer desirable properties compared to more traditional (per-device) policies.

Application	Description	Demands
gqview	Image viewer	CPU, disk read, think time
ijpeg	SPEC2000 image encoding	CPU, image from disk or memory
RealPlayer	Video player	CPU, wireless, disk write
Netscape	Web browser	CPU, wireless, disk write, think time
x11lamp	MP3 player	CPU, disk read
StarOffice	PPT presentation	CPU, disk read, think time

Table 1: Applications

4 Low Residual Energy Through Currentcy Conserving Allocation

Our first goal is to reduce residual energy. The details of our epoch-based currentcy allocation scheme are 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.

4.1 Currentcy Conserving Allocation

There are a number 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 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. As a result of this approach, a task with a small energy share, determining its per-epoch allocation, may receive a large amount of excess currentcy. In this case, the task should have a large cap on its account balance to hold the extra currentcy. Similarly, for the tasks that consistently spend only a fraction of their energy share, the cap can be decreased to free the currentcy to the needy tasks. 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. The new cap is based on an exponential weighted average of currentcy expenditures in previous epochs. If the level of currentcy spent in the

most recent epoch is low relative to its history, then a lower weight factor (α_l) is used than otherwise (α_h). We have found that assigning weights that increase the cap quickly ($\alpha_h = 0.7$) and decrease it slowly ($\alpha_l = 0.1$) produces desirable behavior.

Second, the system redistributes currentcy amounts that overflow some task’s cap to other tasks whose limits have not been reached. Specifically, our allocation algorithm behaves as follows: a) Every epoch, all containers receive currentcy proportionally unless their caps are reached. b) No currentcy is thrown away unless all containers reach their caps. c) Any currentcy overflow from a container is redistributed to other containers with unfilled capacity. For instance, for a total of n containers, if k (where $k < n$) containers reach their caps, the currentcy not allocated to the k containers is redistributed to the other $n-k$ containers, proportional to their shares. The first step of our allocation algorithm is to sort containers so that for any i (for $1 < i < n$), $container_i$ will not reach its cap unless $container_{i-1}$ reaches its cap first. The rest of the algorithm simply walks through the container list and does the following for each $container_i$, such that $0 < i < n$:

1. Calculate the entitled currentcy according to its energy share: $CurEntitled_i = \frac{CurAvail * CurShare_i}{\sum_{j=i}^{n-1} CurShare_j}$

where $CurAvail$ was initialized to be the total overall currentcy available in this epoch.

2. Calculate the allocated currentcy as the smaller of its entitled currentcy and the amount required to reach its cap: $CurAllocated_i = \min(CurCap_i - CurUnused_i, CurEntitled_i)$.

$CurUnused_i$ is the leftover currentcy in the container at the beginning of the epoch and $CurCap_i - CurUnused_i$ is the maximum amount of new currentcy can be added to the container.

3. Calculate the available currentcy for the rest of containers, gathering excess currentcy from the containers at the top of the sorted list: $CurAvail = CurAvail - CurAllocated_i$.

The overall consumption should more closely match the overall allocation with this redistribution (at the risk of upsetting proportionality, considered in Section 5), thus reducing the residual energy. We refer to this algorithm as the Currentcy Conserving (CC) allocation policy.

4.2 Evaluation

As a qualitative argument, we note that without an explicit energy-related abstraction similar to currentcy, it is difficult to articulate precisely what residual energy means or identify means to 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, there is no “traditional” baseline policy with which it makes sense to compare. We compare against the original currentcy allocation [22] with its battery-level feedback mechanism that adaptively adjusts overall allocation levels. In that original policy, residual energy accumulates if a task does not spend all of its currentcy and has exceeded its currentcy cap causing that unspent currentcy to be lost. We show that the original approach is less effective in reclaiming residual energy than explicit currentcy conservation.

To evaluate the benefits of currentcy conservation we use a workload consisting of the gqview image viewer and jpeg. 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. The computationally intensive jpeg is run in a loop to continuously execute the SPEC command line, encoding and decoding an image from the reference data set residing in memory (SPEC command line options: `-GO.findoptcomp vigo.ppm`).

For this experiment, we set the target battery lifetime at 90 minutes, and set desired shares of 66.6% for gqview and 33.3% for jpeg. These allocation settings correspond to an overall average power of 12000mW with 8000mW and 4000mW for gqview and jpeg, respectively. This represents an overly generous allocation to gqview which needs less than an average of 7000mW (c.f., Figure 3a). Jpeg, on the other hand, can easily consume up to 15.55W in the absence of other constraints.

Figure 1 shows our results. These plots show how the total allocation (presented in mW) changes over the lifetime of the battery. They also show the average power consumption of our two applications for the three managed devices. The per-epoch measurements have been smoothed using a centered moving average over a window of twenty-one data points.

From these data, we make several observations. First,

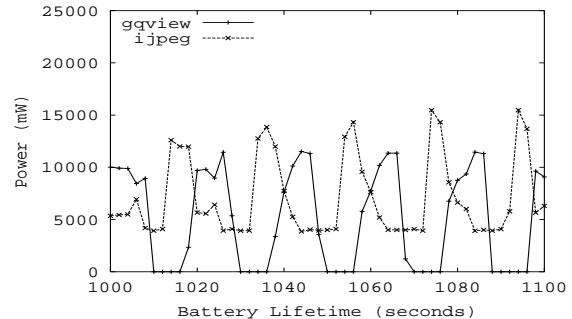


Figure 2: Time Varying Behavior with the Currentcy Conserving Allocation

for the original allocation policy (Figure 1a) we see that the total power available for allocation (the top curve) increases dramatically near the end of the target battery lifetime. There is approximately 6.7% of the original battery capacity remaining at the end. The simple redistribution approach that returns gqview’s 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. In addition, gqview still receives its share of the increasing overall allocation that it does not need.

The second observation we make based from Figure 1a is that as time progresses, gqview’s average power consumption (the middle line exhibiting some degree of scatter) decreases over time despite the increase in total availability. This is because the increase in available currentcy enables jpeg (the bottom solid line that steps up toward the end of the lifetime) to consume more and more CPU time with the baseline CPU scheduling, undermining gqview’s ability to execute when it needs to in order to consume its currentcy.

Figure 1b shows the average power consumption of gqview and jpeg when using the currentcy conserving allocation. We see that there is no significant change in the available allocation 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. Formal analysis of these policies, using control systems theory, reveals that the original policy is unstable while the currentcy-conserving policy is stable.

There are variations in the power consumption of both applications due to gqview’s execution variation that are

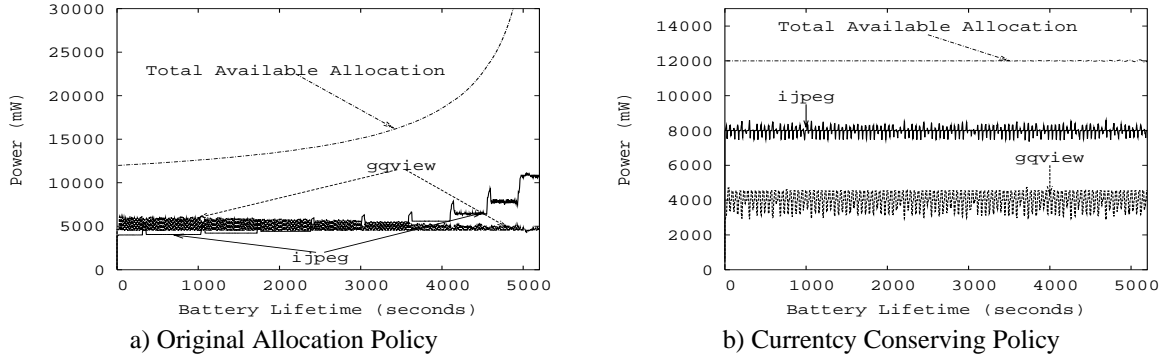


Figure 1: Average Power Consumptions and Total Allocations for the Original and the Currency Conserving Allocation Schemes

not captured in the smoothed plots. To provide a better understanding of the simultaneous execution of *ijpeg* and *gqview*, Figure 2 presents the power consumption in each epoch over a 100 second time interval. This figure shows that when *gqview* is idle (i.e., during “think” time with zero power consumption) *ijpeg* can consume maximum CPU power. However, when *gqview* is active, *ijpeg* is limited to its 4000mW allocation. There is a brief delay in this transition that occurs while *ijpeg*’s currentcy cap is adjusted. We observe that the power consumption of both *gqview* and *ijpeg* can exceed their allocation share because of currentcy accumulating up to their caps.

5 Proportional Energy Use in CPU Scheduling

Even when allocations are appropriately proportional and consistent with actual demand, 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 the CPU scheduler in delivering the opportunity for proportional energy consumption and the role of currentcy in unifying these decisions. The goal is to achieve energy use proportional to the specified currentcy share of each task, unless the task’s needs are satisfied at a lower energy consumption.

5.1 Energy Centric Scheduler

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 in 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 [19] 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 has been 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. This approach resembles compensation tickets from lottery scheduling and fractional quanta from stride scheduling [19], both of which give an advantage toward earlier scheduling of the next

quantum to a task that voluntarily relinquished part of its last quantum. The dynamic share is an adaptation that differs in two respects: it is based on a task’s system-wide energy consumption and it applies over a longer period (spanning multiple quanta occurring during the current and previous epochs).

Our EC scheduler also incorporates one final feature called *self-pacing* (described in Section 7) 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. In this section, we consider the energy-centric scheduling with dynamic shares and without self-pacing.

5.2 Evaluation

Given a particular amount of currentcy per epoch, we investigate proportionally sharing this fixed allotment among competing tasks when some of the currentcy must be spent outside of the CPU. We analyze the effects of CPU scheduling using the default round-robin scheduler minimally modified to check for currentcy, the static energy-based stride scheduler, and our energy-centric scheduler with dynamic shares. We want to show that the energy-centric scheduler can achieve energy use that is proportional to the specified currentcy share of each task, allowing a lower consumption when it is enough to satisfy a task’s performance needs. Thus, the first step is to determine whether there is a level of power consumption such that using more power does not produce significantly improved performance.

For the experiment presented, we simultaneously run *gqview* and *jpeg* with equal shares of a varying total allocation. First, we execute each application alone across the range of total allocation levels to see how the performance metric associated with that benchmark behaves. For *gqview*, configured with a think time of 10 seconds, the delay to completely display the given image decreases with increasing allocations of currentcy (mapped into average power for presentation) until around 6500mW where it levels out at approximately 6.3s. We pit *gqview* against *jpeg*, our CPU-bound benchmark that is always ready to run and whose performance metric, the delay to compress an image file, continues to decline until the maximum power consumption of the processor is reached (e.g., 15.55W). By setting the shares to be equal for the two competing applications, we are giving some benefit to the round robin and stride schedulers. In addition, the power needed by *gqview* for the disk (i.e., approximately 700mW) represents a relatively small level of consumption diverted to another device, making it more challenging for our energy-centric

scheduler to distinguish itself.

Figure 3 shows our results. Figures 3a and 3c give the power consumed by *gqview* and *jpeg* as the allocation increases for each of the three scheduling policies. There are two additional lines on the plot for *gqview* showing the proportional allocation and the maximum power based on performance. Note that the bars representing the energy-centric scheduler show that *gqview* receives its appropriate energy share up until the point where it approaches its maximum power requirement. The Linux default scheduler and the energy-based stride scheduler both favor *jpeg* at the expense of *gqview*. In the case of the default Linux scheduler, this is because *jpeg* is always competing with *gqview* for the CPU and its round-robin algorithm gives each 50% and, for *gqview*, that is only when it is active (not during its think time or disk access). The static energy-based stride scheduler experiences similar problems when *gqview* and *jpeg* are competing for the CPU (with equal share values). *Gqview* is unnecessarily penalized for voluntarily reducing its energy consumption during idle periods.

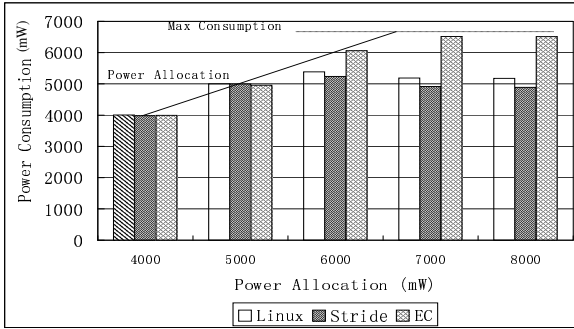
Our energy-centric scheduler extends the stride scheduler in two important ways. First, it selects the next task having the lowest amount of currentcy spent relative to its share, and second it dynamically computes a task’s stride by including information about past consumption. This allows it to compensate for currentcy consumption of the other device as well as for periods of complete inactivity as in *gqview*’s think time.

From Figures 3b and 3d, we see that with the energy-centric scheduler, *gqview*’s delay approaches its performance of 6.3s when running without competition once it is given enough power. Neither the Linux scheduler or the stride scheduler deliver *gqview* that level of performance. Meanwhile, the performance level of *jpeg* is appropriate to its allocation level. It benefits from redistributed currentcy once *gqview*’s consumption levels out, exceeding its expected performance (of running alone at that allocation level) for each scheduling choice.

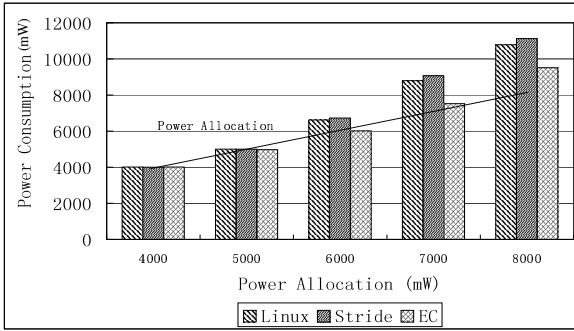
6 Coordinated Scheduling of Multiple Devices: Network Bandwidth and CPU

Currentcy is a unifying abstraction and proportional energy use extends to all other devices on a platform. Currently, ECOSystem only explicitly manages the CPU, NIC, and disk subsystem. The resource managers of various devices must cooperate toward a common goal such as proportional energy use. Otherwise, a bottleneck device with some other policy objective can disrupt currentcy flow in general.

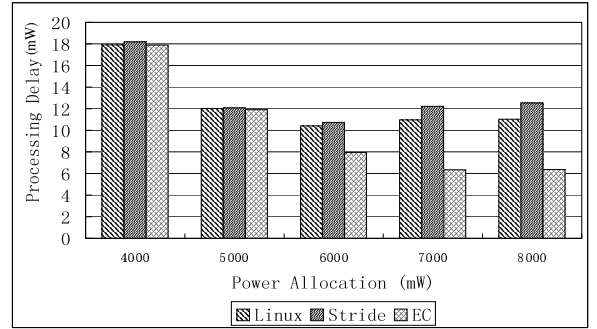
Given ECOSystem’s currentcy model, tracking per-device consumption is straightforward for operations ini-



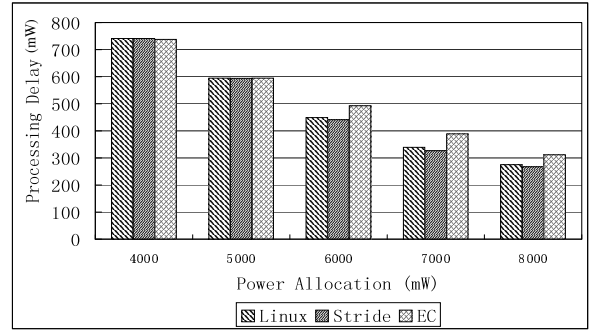
a) Gqview Power Consumption



c) Ijpeg Power Consumption



b) Gqview Delay



d) Ijpeg Delay

Figure 3: CPU Scheduling and Proportional Sharing of Energy

tiated via system calls. One particularly interesting challenge to achieving proportional energy use is managing the wireless network bandwidth, especially for incoming packets. 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 currency as opposed to tasks without currency.

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 currency-determined share and reduces their allocated bandwidth. Assuming other tasks can consume released bandwidth, this bandwidth reduction continues until all connections consume bandwidth in proportion to their task's energy share. This is accomplished by explicitly reducing the advertised window to reduce a task's available bandwidth.

To create a stressful condition for evaluation where the network is the bottleneck, we set the wireless ethernet card to 1Mbps. We execute RealPlayer, Netscape, and jpeg with shares of 9000:3000:3000. Ijpeg serves as a CPU intensive application that does not compete for bandwidth but is capable of consuming 100% of the

CPU. Realplayer plays a video clip rated at 550Kbs. Netscape continuously reloads a web page with five images with zero think time. When executing without energy constraints, RealPlayer consumes about 10643mW to execute without pauses in video playback, while Netscape consumes 3115mW, running unconstrained. Both RealPlayer and Netscape can consume all of the network bandwidth available.

In all of our network experiments we use the currency conserving energy allocation policy and constrain the total power consumption to 15000mW. Since network bandwidth is the bottleneck resource for RealPlayer and Netscape and none of the applications' energy needs are satisfied, the goal is to achieve proportional overall energy use after satisfying the constraints of proportional network bandwidth and network energy consumption. Table 2 presents results for three of the scheduler design points: 1) Our energy-centric CPU scheduler with the default TCP implementation, 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-aware.

Our results show that the conventional network scheduler fails to provide either proportional network bandwidth or energy consumption. This is because Netscape is allowed to consume an unfair portion of network band-

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	9000	2875	219	259	3354	3066
Netscape	3000	956	569	615	2142	7294
ijpeg	3000	8960	23	0	8983	0
Default Linux CPU Scheduler, Energy-Centric Network						
RealPlayer	9000	5902	700	680	7282	8032
Netscape	3000	841	113	229	1182	2701
ijpeg	3000	6611	18	0	6629	0
Energy-Centric CPU Scheduler, Energy-Centric Network						
RealPlayer	9000	8695	621	704	10020	8353
Netscape	3000	789	155	226	1170	2680
ijpeg	3000	3778	10	0	3788	0

Table 2: Proportional Sharing: CPU and Network

width. Netscape is able to take more than 50% of the bandwidth because it can open multiple connections. This reduces RealPlayer’s ability to execute and produces an excess in currentcy that is reallocated to ijpeg. This results in ijpeg getting more of the CPU and consuming a much larger energy share than its intended allocation while the needs of the other applications are not satisfied (currentcy redistributed to ijpeg would be considered acceptable if the other applications had their needs appropriately met).

When we use an energy-centric network scheduler, but the default Linux CPU scheduler, we see that bandwidth and network power are consumed by RealPlayer and Netscape closer to the specified ratio of 3 to 1. However, RealPlayer still suffers from competition for the CPU with ijpeg which results in ijpeg significantly exceeding its energy share.

The most satisfying results are obtained by using energy-centric schedulers for both the CPU and network. Both network bandwidth and network energy are consumed proportionally by RealPlayer and Netscape. RealPlayer’s share of the CPU is also protected from ijpeg by the energy-centric CPU scheduler. Netscape is throttled after receiving its share of network bandwidth (its bottleneck device) and can not consume the rest of its currentcy allocation. In this case, RealPlayer gets enough currentcy to meet its needs and execute without pausing.

7 Low Variance in Response Time Through Pacing Currentcy Expenditures

Given a case in which power consumption must be constrained, our epoch-based allocation has the potential to produce bursty behavior if 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. This approach exploits the ability of currentcy to reflect an application’s rate of progress. This approach to stretching execution is appropriate for a non-Dynamic Voltage Scaled processor. If available, DVS would be a preferred alternative to consider.

To compare these two approaches, we first look at the overhead of the first approach because it increases with the shortened epoch length and could become a performance bottleneck. However, experiments show the overhead for currentcy allocation is very small. Even if we perform allocation every 10ms (a timer interrupt occurs every 10ms, while the CPU scheduling quantum is 60ms in our system), the overhead is only 206 μ s for 18 resource containers and 40 μ s for 3 containers.

To explore the effects on response time of our two

approaches for reducing bursty performance, we run Netscape and continuously load our department's web page. This page contains a banner image and some simple text. The autload 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 with 10 second epochs. We allocate currency equivalent to an average of 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 3 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 can translate into a user perceived difference in performance as the self-pacing policy can provide 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.

8 Energy Efficient Disk I/O Through Cost-sharing

Encouraging more energy efficient use of devices is an important function of an energy centric operating system. Currency provides a means for passing along the savings to tasks that cooperate through their usage patterns. The disk presents unique challenges and opportunities for currency-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 sharing the spinup/spindown power costs. This introduces opportunities to work with debiting, bidding and pricing in the context of our currency model. The policy space for these approaches is very large, and many solutions may require an API for application involvement. For example, recent work [21] describes cooperative disk I/O operations that applications can use to facilitate such behavior. 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 without application involvement.

8.1 Shaping Access Patterns by Pricing and Bidding

Intuitively, we want to amortize spinups across multiple disk operations, which benefits from encouraging more bursty behavior. 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 debiting 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 currency 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. Tasks can indicate they are willing to contribute certain amounts toward the price of spinning up the disk. 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. One goal of this technique is to enable multiple tasks to pool their currency and cooperatively use the disk in an energy-efficient manner.

Traditional techniques of skewing access patterns are amenable to currency-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 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, delayed writes) and make them uniform across tasks (which can tend to synchronize tasks into producing batches of disk activity).

We trigger prefetching operations and flushing of delayed writes that cause spinups using a bidding function based on the fraction of consumed 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 value is greatly increased as the task nears a demand fetch. The disk flush daemon performs a large number of writes once it starts flushing pages to a spinning disk, writing back all dirty pages that have been idle for a more than 5 seconds. By

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 3: Response Time Variation

contrast, 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.

8.2 Experiments on Disk Access Scheduling and Buffer Allocation

In this section, we show how the currency 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 present preliminary results on prefetching in our coordinated buffer management system.

Accesses 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 (command line options: `-GO.compress vigo.ppm`) that is 2,359,355 bytes. When 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 currency-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

the MP3 player, `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.

Buffer Allocation It is also effective to balance the buffer allocation among prefetching-friendly tasks to facilitate more globally synchronized disk activity. To show the benefit of 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 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.

9 Conclusion

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

Using our ECOSystem prototype, we implement several currency-based policies, including: currency 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 currency model is a powerful framework for expressing energy management policies and that our currency-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 Douglass, 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] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Operating Systems Design and Implementation*, October 1996.
- [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] Krisztian Flautner and Trevor Mudge. Vertigo: Automatic performance-setting for linux. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [7] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP)*, pages 48–63, December 1999.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] Rolf Neugebauer and Derek McAuley. Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.
- [14] 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.
- [15] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th symposium on operating systems principles*, pages 89 – 102, October 2001.
- [16] 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.
- [17] 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.
- [18] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional share resource management. In *Proceedings of Symposium on Operating Systems Design and Implementation*, November 1994.
- [19] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, MIT, 1995.
- [20] 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.

- [21] Andreas Weissel, Bjorn Beutel, and Frank Bellosa. Co-operative I/O – a novel I/O semantics for energy-aware applications. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [22] 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)*, page 123, October 2002.