Experiences in Managing Energy with ECOSystem

Energy consumption is a major systems-design challenge. ECOSystem incorporates the "currentcy model," which lets the operating system manage energy as a first-class resource. It can also express complex energy-related goals and behaviors, leading to more effective, unified management policies.

Heng Zeng, Carla S. Ellis, and Alvin R. Lebeck
Duke University

Mobile devices are becoming increasingly popular, from laptops, PDAs, and cell phones to emerging platforms such as wireless sensor networks. Available battery energy has become a critical mobile-system resource. A mobile device's usefulness is often limited not by its hardware's raw speed but by its battery's energy.

Ideally, designers should address the energy problem at all system levels: hardware, operating system, and application. At the hardware level, low-power circuit design can reduce energy consumption. In addition, hardware devices can provide device-specific power management features that you can exploit with power-management policies at the system's higher levels. At the OS level, we can observe the applications' and devices' combined resource demands. Existing OS-level energy management tends to focus on individual devices. For example, scheduling for processors with dynamic voltage scaling (DVS) allocates CPU time to tasks and manipulates the CPU power states.[1–4] Similarly, disk and network policies concentrate only on the request patterns to their specific devices and on managing their available power states.[5–8] At the application level, you can save energy by making applications energy aware. An energy-aware application can decrease its power consumption by reducing its activity, and it can give hints to the device manager or change its device-access pattern to create energy-saving opportunities for hardware.[9,10]

We designed our ECOSystem (Energy-Centric Operating System) prototype to manage energy consumption at the OS level, complementing existing power-management techniques, such as DVS and application adaptation. It's based on the ideas that energy management should be a system-wide effort, that we should explicitly recognize energy as a resource, and that we should unify energy management across the system. Even managing one hardware device might require coordination with other system components. Without unified management, application-level energy-saving efforts might not result in reduced energy consumption.

ECOSystem's background

Two choices initially framed our project. First, we chose to explore energy management for general-purpose applications that weren't modified to be energy aware. This seemed the most realistic scenario to ensure wide applicability of OS-level energy management. However, it was also limiting because we'd have less information about the workload to exploit (such as deadlines, if we

Published by the IEEE CS and IEEE ComSoc ■ 1536-1268/05/$20.00 © 2005 IEEE

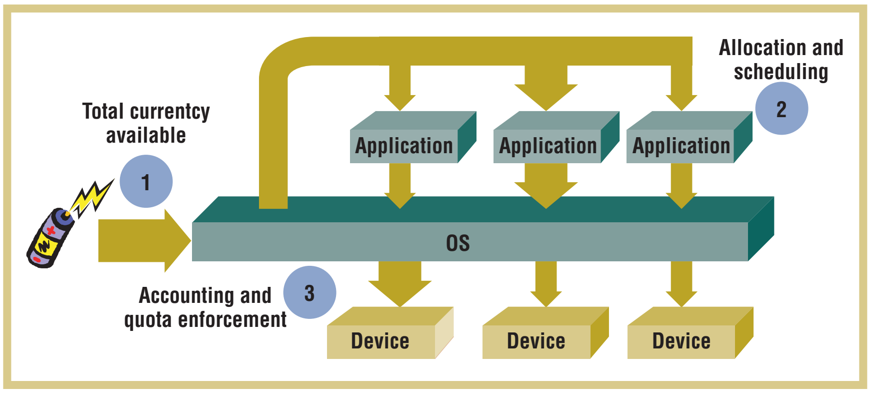

Allocation and scheduling
2
Total currentcy available
1
Application
Application
Application
OS
Accounting and quota enforcement
3
Device
Device
Device

Figure 1. Our energy-management framework.

restricted our attention to multimedia applications).

Second, we chose the energy goal of achieving a specified target battery lifetime. Imagine using a laptop during a long flight, with the battery as the only power source. You might run several applications concurrently, such as a word processor and an MP3 player. You might specify that the battery has to last for the flight's expected duration, and you might also want the word processor to receive enough energy to run responsively—at the expense of the MP3 player, if necessary. Our experience has revealed that we can achieve this goal with minimal user input, such as target battery lifetime and relative application priorities. (Our previous research publications give details on experimental evaluations.[11,12]) Other desirable energy goals (such as maximizing the number of operations per Joule) would typically require more knowledge about the applications.

The currentcy model

Energy has unique characteristics that pose challenges to its management, such as its global impact on system components; every application needs energy to access any hardware device. Our *currentcy* model is the key to meeting these challenges.

Energy-management challenges

OS-level energy management splits across multiple dimensions. First, the system must manage energy along the time dimension throughout the target battery lifetime. Also, various devices in the system (such as the CPU, disks, network interfaces, and display) can draw power concurrently and are amenable to different management techniques. Finally, multiple applications share these devices.

These multiple dimensions present additional accounting challenges. For example, you could access a hard disk well after the `write` operation is initiated, while the process that issued the operation isn't active. Furthermore, a program counter sampling technique that attributes power costs to the process running in the CPU will account inaccurately for these asynchronous activities. Instead, we must track energy consumption throughout the system. OS-level energy management creates a need for accounting that captures these multiple dimensions accurately enough for meaningful policy actions.

An energy-management framework

We propose a common unit, currentcy, as the basis for explicit energy management. We created the word *currentcy*—a combination of current and currency—to express a unified abstraction for the energy a system can spend on various devices, just like money is the unified abstraction of buying power for commodities. Currentcy quantifies the energy resource and represents it concretely within the system. One unit of currentcy represents the right to consume a certain amount of energy within a fixed amount of time. Our currentcy-based framework unifies energy-management efforts across the system (see Figure 1).

Energy management has three orthogonal dimensions: time, tasks, and devices. The first dimension involves specifying a target lifetime and dividing it into fixed-length epochs. At the beginning of each epoch, the allocation module generates a certain amount of currentcy on the basis of information such as the battery's remaining energy, the target lifetime, and platform power characteristics. This total amount of currentcy for all tasks during the epoch is also the amount of currentcy available for the system to spend on all devices.

The framework's second dimension involves sharing the limited currentcy among competing tasks. The allocation and scheduling modules cooperate to accomplish this. The allocation module allocates each epoch's total available currentcy to tasks according to criteria that reflect the user's preferences. While this allocated currentcy enables a task to access devices, the currentcy-consumption scheduling module provides opportunities necessary for the task to spend the currentcy.

In the last dimension, as devices consume energy, the accounting module charges the corresponding amount of currentcy to the tasks. If a task runs out of currentcy, the system should allow no more energy consumption or hardware access. A task that's blocked because of currentcy depletion will be reactivated when it has currentcy again; the allocation module usually gives this currentcy at the next epoch's start.

If you use one model for energy consumption across all devices, energy tradeoffs among devices become explicit; for example, currentcy spent on disk activities is no longer available for CPU cycles. The common currentcy provides a mechanism for unified management of diverse devices that recognizes these tradeoffs; for example, currentcy spent on the disk can influence coordinated CPU scheduling.

ECOSystem prototype

Our ECOSystem prototype implements the management framework and all necessary management functions (including battery management, energy accounting, currentcy allocation, and currentcy-consumption scheduling), which it implements as separate modules distributed across the system. We implemented our prototype on an IBM T20 laptop based on the Linux kernel. ECOSystem explicitly manages three of the laptop's hardware devices: the CPU, disk, and wireless card.

We learned several lessons from implementing our prototype on this platform. ECOSystem can only affect energy management in a limited range above the base power consumption of approximately 13 watts, representing unmanaged devices (such as the display) and managed devices' low power states (such as the wireless card's doze mode). We can address this by shifting more devices from the base into the managed category to widen the range.

One implication of the high base power consumption is that we can't effectively exploit any of the battery's nonlinear properties. For some platforms with very low base power consumption, we might be able to exploit the nonlinear behavior of battery lifetime versus drain rate to increase the amount of work accomplished with a longer target battery lifetime. Because of the IBM laptop's high base power consumption, we can't guarantee that more work is accomplished in a longer lifetime. Also, the processor consumes about 15.5 W when fully active, which dominates other managed devices' power consumption. Unfortunately, this reduces the overall impact of managing the other devices. For other platforms with more balanced energy budgets or by including techniques to reduce processor power (such as DVS), the interplay among multiple devices will be more important.

ECOSystem reimplements a subset of the resource container abstraction as the currentcy-holding entity. This abstraction enables sharing a container among cooperating processes (collectively called a task), tracking dependencies among tasks (such as locks), and task-to-task currentcy exchanges to handle priority inversion (such as processes that deplete their container but hold locks).

Energy accounting

Energy accounting involves measuring each device's power consumption in all power modes. Existing hardware support didn't provide the online information about each component's power consumption that we needed for accounting. So, we chose an embedded-energy model for each device, calibrated in advance.

The OS needs a mechanism to track energy consumption and attribute it correctly to each task. ECOSystem energy accounting has two other components: the task-tracking infrastructure, which identifies activities from processes, and the device-specific payback policy, which attributes the devices' energy consumption to processes.

Power model

We assume that our CPU draws a fixed amount of power for computation. This model is very simple (CPU halted or active), but the embedded-energy model approach can support finer-grain information such as using event counters. We can divide a hard disk's energy consumption into four categories: spinup, disk access, spinning, and spindown. Spinup, disk access, and spindown are one-time energy costs associated with an event, and the disk draws power continuously when it's idle but spinning. The IEEE 802.11b card supports a power-saving feature that puts the device in sleep mode most of the time, waking it up periodically for packet processing.

Activity tracking and payback

ECOSystem charges CPU power consumption to the current task for its computation's duration. Tracking disk activity is more complicated because of asynchronous operations (such as buffer caches that a daemon writes to the disk). ECOSystem exploits resource containers to correctly charge the appropriate task for these asynchronous operations and a portion of shared costs (such as spinup and spindown). Similarly, for the 802.11b card, we charge the appropriate task for outgoing packets and charge for received packets when we can identify the destination task.

Achieving a target battery lifetime

With energy accounting in place, we're ready to consider the framework's ability to achieve global energy-related goals—specifically, a target battery lifetime. Certainly, this might not be every usage scenario's goal. However, given our desire to work with unmodified applications, battery lifetime is a quantifiable metric. Unlike other types of workloads, our unmodified, general-purpose workload lacks a well-defined unit of work, such as deadlines or transactions.

To achieve a specific battery lifetime, we must limit the rate at which the devices drain energy from the battery. When the demand is too high, we can limit power consumption by either reducing the work performed (reducing the level of service or having the application adapt) or increasing efficiency (shaping disk request patterns to reduce spinup and spindown's overhead).

We can acquire the information about remaining battery energy through a standard battery-OS interface. Although we can periodically obtain battery-capacity information, the interfaces to the batteries are too slow to be useful for fine-grain energy measurements. Until we can reduce these interfaces' latency, explicit

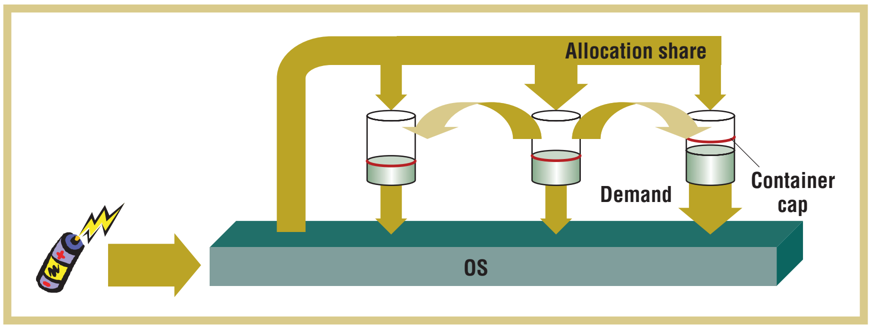


Figure 2. Currentcy-conserving allocation.

energy-management frameworks require power models as we described earlier.

With the specified battery lifetime and the remaining energy information, an energy-distribution policy manages energy availability over the target lifetime. ECOSystem uses a simple policy that divides available energy evenly along the lifetime so the system has a constant power supply. Ideally, the system would generate an amount of currentcy per epoch according to the target power consumption. However, as in all systems, errors can occur that cause the system to deviate from achieving a specific lifetime (such as an error in CPU power modeling). To make the system more robust, ECOSystem uses an online feedback module that keeps the system on track with the goal. The module uses the actual energy consumption, which it obtains periodically through the battery interface, to dynamically adjust the currentcy generated at the beginning of each epoch. We successfully demonstrated that we can achieve a target lifetime, even in the presence of an erroneous power model.

Currentcy-conserving allocation

Because currentcy is limited in each epoch, it becomes a critical system resource. The allocation and scheduling modules share its management. The allocation module we discuss in this section distributes currentcy to applications in proportion to their importance and reclaims surplus currentcy to fully use the battery for the specified lifetime.

Differentiated allocation

With multiple tasks, the system needs an abstraction to specify each task's relative importance and how to share the resource accordingly. ECOSystem takes the proportional sharing approach that many general-purpose systems use.[13,14] The relative importance you assign to each application translates into a *share* value for each task; the system allocates currentcy to the tasks proportional to their shares.

$$CurEntitled_i = Currentcy_{total} \times \frac{share_i}{\sum_{k=1}^{n} share_k}$$

We refer to the currentcy allocation for task i, $CurEntitled_i$ in the equation, as the *entitled* currentcy because it's based on the task's assigned share value without considering its currentcy demand. The entitled currentcy might not equal the demand; rather, it's the currentcy allocation guaranteed to the task. If a task's entitled currentcy allocation exceeds the demand, the difference between the two is surplus currentcy. If a task doesn't have enough allocated currentcy, it can exploit other tasks' surplus.

In ECOSystem, tasks can carry surplus currentcy from one epoch to the next. This lets them accumulate currentcy to pay for a temporary surge in demand, such as disk spinup. However, we impose a cap on the amount of currentcy the tasks can accumulate to prevent them from hoarding it unproductively. The system will forfeit any currentcy that exceeds the cap.

Currentcy-conserving allocation

To manage currentcy efficiently, the system shouldn't allocate more to a task than it demands, and it should reallocate the surplus to other tasks that can use it. So, we developed a *currentcy-conserving* allocation scheme with two properties. The *proportional* property states that entitled currentcy's allocation and surplus currentcy's reallocation should be proportional to the share value. The *conserving* property states that the system should allocate an epoch's overall currentcy to tasks as much as possible, until all tasks reach their demand. We also set the container cap based on the currentcy-consumption history; thus, the system will let a task with high past consumption accumulate more currentcy for future consumption.

Figure 2 illustrates the allocation process. Although the middle container has a large resource share (indicated by the wide incoming arrow), it has a low demand for currentcy (narrow outgoing arrow); the small container cap reflects this behavior. During allocation, the system will reallocate its surplus currentcy to the other two containers to satisfy their demands. To allocate the entitled and surplus currentcy proportionally, we've implemented an algorithm with the worst-case computational complexity of $O(n \log n)$, where n is the number of containers in the system.

To evaluate our allocation policies, we used a workload comprising two applications: *gqview* and *ijpeg*. Gqview, an image viewer, autobrowses a set of images with a 10-second think time between images and consumes 7 W. Ijpeg is a CPU-bound image-compression application that, if unconstrained, could consume 15.5 W. When we allocate 8 W to gqview and 4 W to ijpeg, we observe 8.3 percent of the battery capacity remaining without using currentcy reclamation, but only one percent with currentcy-conserving allocation.

Currentcy-centric scheduling

The system can only provide differen-

TABLE 1
Proportional sharing: CPU and network.

CPU scheduler and network properties	Application (mW)	Allocation (mW)	CPU power (mW)	Wireless network card (mW)	Hard disk power (mW)	Total power bandwidth	Network (Bytes/second)
Energy-centric CPU scheduler, energy-oblivious network	RealPlayer	9,000	2,875	219	259	3,354	3,066
	Netscape	3,000	956	569	615	2,142	7,294
	ijpeg	3,000	8,960	23	0	8,983	0
Default Linux CPU scheduler, energy-centric network	RealPlayer	9,000	5,902	700	680	7,282	8,032
	Netscape	3,000	841	113	229	1,182	2,701
	ijpeg	3,000	6,611	18	0	6,629	0
Energy-centric CPU scheduler, energy-centric network	RealPlayer	9,000	8,695	621	704	10,020	8,353
	Netscape	3,000	789	155	226	1,170	2,680
	ijpeg	3,000	3,778	10	0	3,788	0

tiated energy management with the cooperation of the allocation and scheduling modules. Without scheduling opportunities, a task can't consume any of its allocated currentcy. Its accumulated resource eventually exceeds the container cap, so the system gives it to other tasks as surplus.

Currentcy-consumption scheduling is challenging because energy isn't an independent resource. Rather, its consumption relies on other hardware devices' scheduling. We examine three scheduling policies based on different energy resource roles. The pay-as-you-go (PAYG) policy uses the original, unmodified device schedulers; currentcy's only role is to enable a task's scheduling. In *static-priority* scheduling, each hardware device's scheduling priority is its share value, so the system allocates currentcy and provides scheduling opportunities to each task proportionally to its share. Finally, *currentcy-centric* scheduling dynamically adjusts a task's scheduling priority for all devices on the basis of the ratio of its consumed currentcy to its entitled currentcy. If this ratio is low, the system increases the task's priority. We've implemented currentcy-centric schedulers for CPU time and network bandwidth.

To evaluate these scheduling policies, we simultaneously ran gqview and ijpeg with equal shares of a varying total allocation. These two applications don't access the wireless network interface, so only the CPU scheduler was involved. Gqview, configured with a 10-second think time, requires a minimum of 6,500 mW to achieve its lowest display latency of 6.3 s.

Our results show that for low total allocation (4,000 to 5,000 mW), all schedulers perform the same. However, as the total allocation increases, only the currentcy-centric scheduler enables gqview to consume its allocated share of energy and achieve its minimum delay. The other schedulers actually decrease gqview's ability to execute, thus increasing its delay; with increased total allocation, ijpeg's ability to compete with gqview for the CPU increases. The PAYG's default Linux scheduler's round-robin algorithm gives each program 50 percent of the CPU; for gqview, that's only when it's active (not during its think time or disk access). The static-priority scheduler experiences similar problems when gqview and ijpeg are competing for the CPU (with equal share values). In general, without the currentcy-centric scheduler, applications are unnecessarily penalized for voluntarily reducing their energy consumption during idle periods such as think time.

Next, we considered coordinated network and CPU scheduling. In this experiment, we executed three applications—RealPlayer, Netscape, and ijpeg—with share ratios of 9:3:3. RealPlayer plays a video stream over the network. We gave it more than half the overall share, suggesting that it's important. It was subject to network-bandwidth competition from Netscape and CPU competition from Netscape and ijpeg.

We examined the experimental results at three scheduler design points: our energy-centric CPU scheduler with the default TCP implementation for bandwidth sharing, the default Linux CPU scheduler with an energy-centric network scheduler, and our combined energy-centric CPU and network schedulers. We omitted the case where neither the CPU nor the network scheduler was energy-aware.

Our results show that at the first design point, the conventional network scheduler fails to provide either proportional network bandwidth or energy consumption (see Table 1). Netscape can take more than 50 percent of the bandwidth because it can open multiple connections. This reduces RealPlayer's ability to execute and produces excess currentcy, which the system reallocates to ijpeg. As a result, ijpeg gets more of the CPU and consumes much more energy than it's allocated, while other applications' needs are unsatisfied (redistributing currentcy to ijpeg would be acceptable if the other applications' needs were met). At the sec-

Cooperating Applications: How Currentcy Can Help

Energy-management techniques typically fall along two dimensions: 1) metrics, such as battery lifetime, and 2) workload, such as user or application involvement. Nemesis[15] and the Grace Project[1] whole-system frameworks address explicit energy management and, in particular, the battery lifetime energy goal. Nemesis depends on the applications' cooperation to meet the target battery lifetime. When the power consumption gets too high, the system charges applications for overuse, which serves as feedback that those applications can use to adapt their behavior. Grace is targeted at a workload of soft real-time multimedia applications for mobile systems. The Grace framework takes a quality-of-service optimization approach, attempting to maximize overall quality while minimizing power consumption, subject to battery lifetime and CPU-utilization constraints. Adaptive tasks can adjust their output quality; the system reacts by scaling the processor's voltage and frequency.

These systems could use the currentcy model to help manage multiple devices and enable more sophisticated scheduling policies. Currentcy-based scheduling is compatible with DVS approaches where minimal computation demands are either predicted or specified from the application, as in multimedia or embedded applications.[2–4] Using the currentcy model, an application could adapt to reduce its demands if the system can't meet them.[9] However, as our experiences with ECOSystem show, reducing one application's energy demands might not be sufficient to meet deadlines if another energy-hungry application is executing.

ond design point, RealPlayer still competes with ijpeg for the CPU, which results in ijpeg significantly exceeding its energy share. In either case, RealPlayer can't play back smoothly even though it has the highest share.

We obtained the best results by using energy-centric schedulers for both the CPU and the network. RealPlayer and Netscape proportionally consumed network bandwidth and energy, and the energy-centric CPU scheduler protected RealPlayer's CPU share from ijpeg. In this case, RealPlayer got enough currentcy to meet its needs, so it executed without pausing.

Our work thus far is only the first step toward unified energy management. Without changing the initial assumptions, we believe we could naturally extend our framework to incorporate additional hardware resources and smarter management policies. For example, our current policy to throttle a task's power consumption is to put the processor into a deep sleep state when it exhausts its currentcy. We could use the finer-grained DVS approach in this scenario to improve the task's energy efficiency, with the new perspective that the available energy resource (rather than the applications' computational demand) would determine the voltage setting.

Also, as discussed earlier, our choice of an unmodified, general-purpose workload limited the energy goals that we could meaningfully pursue. Although most mobile applications are concerned with battery lifetime, it might not be the best metric for all applications or usage scenarios. For example, our policy to achieve a target lifetime stretches applications' execution throughout the battery lifetime, but applications such as the MP3 player need at least a threshold level of power allocation to perform properly. One potential solution is to extend our framework with a layer of admission control. The MP3 player could specify its minimal resource requirement to the OS. If the system couldn't meet this requirement, it could deny the application admission and use its allocated resource more effectively elsewhere. Another approach is to extend the framework to let the application negotiate with the OS. For instance, the MP3 player could request increased power allocation (at the cost of shortened execution time) to maintain the same energy allocation; as a result, it wouldn't last throughout the lifetime.

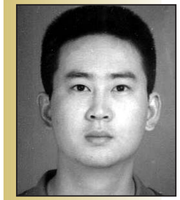
Having more information about the application facilitates energy goals other than battery lifetime. Identifying an application's unit of work (a media frame for the application to process by a deadline, for example) lets us formulate efficiency goals. We've been able to study energy efficiency in the file system, implicitly using the disk request as the work unit. We were able to formulate policies in the currentcy framework that increased energy efficiency in disk access, reducing energy costs by as much as 36.8 percent. Our ongoing work introduces application assistance by annotating reads and writes with a currentcy valuation that cooperative economic policies can use.

By explicitly managing the energy resource globally, our currentcy-based OS-level framework becomes an energy-centric system's centerpiece. However, to achieve the framework's full potential, it's essential to have an API for an application to specify its requirements. Also necessary are an infrastructure that lets the OS inform the application about environment changes and help it adapt and an intuitive interface for the user to specify resource-allocation preferences. We plan to explore these topics in future research.

ACKNOWLEDGMENTS

The National Science Foundation (0204367, EIA-9986024, and ITR-0082914) supported this work in part. Amin Vahdat was also involved in this research's early stages.

the AUTHORS



Heng Zeng is a private consultant. His research interests include operating systems and energy-efficient computing. He received his PhD in computer science from Duke University. Contact him at 600-6 S. Lasalle St., Durham, NC 27705; zengh@cs.duke.edu.

Carla S. Ellis is a professor of computer science at Duke University. Her research interests are operating systems and mobile computing. She received her PhD from the University of Washington, Seattle. She is cochair of the CRA Committee on the Status of Women in Computing Research and editor in chief of *ACM Transactions on Computing Research*. She is a member of the IEEE Computer Society and the ACM. Contact her at the Dept. of Computer Science, D324 Levine Science Research Center, Duke Univ., Durham, NC 27708-0129; carla@cs.duke.edu.

Alvin R. Lebeck is an associate professor of computer science and electrical and computer engineering at Duke University. His research interests include architectures for emerging nanotechnologies, high-performance microarchitectures, hardware and software techniques for improved memory hierarchy performance, multiprocessor systems, and energy-efficient computing. He received his PhD in computer science at the University of Wisconsin, Madison. He is a senior member of the IEEE and a member of the ACM. Contact him at the Dept. of Computer Science, D308 Levine Science Research Center, Duke Univ., Durham, NC 27708-0129; alvy@cs.duke.edu.

REFERENCES

1. W. Yuan and K. Nahrstedt, "Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems," *Proc. 19th ACM Symp. Operating Systems Principles* (SOSP 03), ACM Press, 2003, pp. 149–163.
2. P. Pillai and K.G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," *Proc. 18th ACM Symp. Operating Systems Principles* (SOSP 01), ACM Press, 2001, pp. 89–102.
3. K. Flautner and T. Mudge, "Vertigo: Automatic Performance-Setting for Linux," *Proc. 5th Symp. Operating Systems Design and Implementation* (OSDI 02), *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, ACM Press, 2002, pp. 105–116.
4. J. Lorch and A.J. Smith, "Operating System Modifications for Task-Based Speed and Voltage Scheduling," *Proc. 1st Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 03), USENIX, 2003, pp. 215–230.
5. M. Anand, E.B. Nightingale, and J. Flinn, "Self-Tuning Wireless Network Power Management," *Proc. 9th Ann. Int'l Conf. Mobile Computing and Networking* (MobiCom 03), ACM Press, 2003, pp. 176–189.
6. F. Douglis, P. Krishnan, and B. Bershad, "Adaptive Disk Spin-Down Policies for Mobile Computers," *2nd USENIX Symp. Mobile and Location-Independent Computing* (MLICS 95), USENIX, 1995, pp. 121–137.
7. D. Helmbold, D. Long, and B. Sherrod, "A Dynamic Disk Spin-Down Technique for Mobile Computing," *Proc. 2nd ACM Int'l Conf. Mobile Computing and Networking* (MobiCom 96), ACM Press, 1996, pp. 130–142.
8. A. Papthanasiou and M. Scott, "Energy Efficient Prefetching and Caching," *Proc. USENIX Ann. Technical Conf.*, USENIX, 2004, pp. 255–268; www.usenix.org/publications/library/proceedings/usenix04/tech/general/papathanasiou/papathanasiou_html/index.html.
9. J. Flinn and M. Satyanarayanan, "Energy-Aware Adaptation for Mobile Applications," *Proc. 17th ACM Symp. Operating Systems Principles* (SOSP 99), ACM Press, 1999, pp. 48–63.
10. A. Weissel, B. Beutel, and F. Bellosa, "Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications," *Proc. 5th Symp. Operating System Design and Implementation* (OSDI 2002), USENIX, 2002, pp. 117–130; www.usenix.org/publications/library/proceedings/usenix04/tech/general/papathanasiou/papathanasiou_html/index.html.
11. H. Zeng et al., "Ecosystem: Managing Energy as a First Class Operating System Resource," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS X), ACM Press, 2002, pp. 123–132.
12. H. Zeng et al., "Currentcy: A Unifying Abstraction for Expressing Energy," *Proc. USENIX Ann. Technical Conf.*, USENIX, 2003, pp. 43–56.
13. C.A. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, tech. report MIT/LCS/TR-667, Laboratory for Computer Science, MIT, 1995.
14. L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *ACM Trans. Computer Systems*, vol. 9, no. 2, 1991, pp. 101–124.
15. R. Neugebauer and D. McAuley, "Energy Is Just Another Resource: Energy Accounting and Energy Pricing in the Nemesis OS," *Proc. 8th Workshop Hot Topics in Operating Systems* (HotOS VIII), IEEE CS Press, 2001, pp. 67–74.