Apprehending Joule Thieves with Cinder

Apprehending Joule Thieves with Cinder

Stephen M. Rumble
Stanford University
353 Serra Mall
Stanford, California 94305

Ryan Stutsman
Stanford University
353 Serra Mall
Stanford, California 94305

Philip Levis
Stanford University
353 Serra Mall
Stanford, California 94305

David Mazières
Stanford University
353 Serra Mall
Stanford, California 94305

Nickolai Zeldovich
MIT
32 Vassar Street
Cambridge,
Massachusetts 02139

Abstract

Energy is the critical limiting resource to mobile computing devices. Correspondingly, an operating system must track, provision, and ration how applications consume energy. The emergence of third-party application stores and marketplaces makes this concern even more pressing. A third-party application must not deny service through excessive, unforeseen energy expenditure, whether accidental or malicious. Previous research has shown promise in tracking energy usage and rationing it to meet device lifetime goals, but such mechanisms and policies are still nascent, especially regarding user interaction.

We argue for a new operating system, called Cinder, which builds on top of the HiStar OS. Cinder's energy awareness is based on hierarchical *capacitors* and *task profiles*. We introduce and explore these abstractions, paying particular attention to the ways in which policies could be generated and enforced in a dynamic system.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design

Keywords

capacitor, energy, hierarchical

1. INTRODUCTION

Energy is a resource. Just like memory, mass storage, and CPU cycles, it should be allocated, accounted for, and scheduled in ways that are meaningful and beneficial to the system and its users. Since it is exhaustible and cannot be reclaimed once spent, the system should be especially diligent in its accounting and robust in apportioning it within a complex, dynamic, multiprogrammed environment. Previous systems have shown promise in tracking energy usage in a program's structure (Powerscope [4], Odyssey [3]) and performing accurate global system accounting using either static hardware profiles (ECOSystem [9]) or by dynamically inferring usage based on energy draw history across varying device power states (Quanto [5]). Several other works showed it is possible to use this data to ration access to other resources, i.e. the energy consumers (CPU, disk, network, etc.), in order to meet a battery lifetime goal [10, 3]. Existing systems, however, either require explicit application interaction [3] or do not address a dynamic system, in which application sets change [9]. For the modern mobile device, intended to frequently download and execute new code from the network, static partitioning of resources is insufficient.

Mobile devices today run a variety of full UNIX-like kernels with user-downloaded applications sitting atop complex software stacks. Apple's App Store and Google's Android Market both provide immediate access to a growing suite of thousands of third-party applications [1]. This explosion of mobile software complexity and application availability makes it difficult to reason about the energy requirements and expenditures of mobile systems. Additionally, the sheer bulk of available software implies that many applications are buggy, malicious, or inefficiently implemented.

Unfortunately, users currently have no quantitative means to determine how much energy an application is consuming, nor do they have much control over it, beyond choosing whether or not to run the application at all. Current systems are not in command of their limited energy budgets and, consequently, users have no means of reserving or prioritizing energy for important applications while curtailing others. Although platforms can be extended to track energy for tasks [9, 10, 5], previous research has not adequately addressed dynamic systems, application control of energy budgets (i.e. suballocation), and policy generation via system feedback.

We argue for a new operating system called Cinder. Cinder is designed to answer not only which applications are energy-expensive, but also what behavior we can expect and how they may be controlled by both the user and the applications in order to achieve their desired energy goals. Cinder empowers users and applications - the former via useful system feedback coupled with policy generation, and the latter

MobiHeld'09, August 17, 2009, Barcelona, Spain.

through delegation of energy resources and the enablement of application-level resource control.

The HiStar [8] OS has a data-centric security architecture, consisting of a minimal, trusted kernel upon which user-level library operating systems (e.g. a POSIX layer) are built. Cinder is based on HiStar and inherits many of its features. Most importantly, Cinder is built atop a minimal set of seven first-class kernel objects. Object containers, one such example, are particularly useful for resource tracking and revocation, enabling fine-grained resource accounting using a fundamental, pervasive system primitive. Finally, a fast, simple, and non-reentrant kernel greatly simplifies accounting across the protection boundary. HiStar is simple, yet sophisticated; the above are only the salient aspects involved in resource handling.

Cinder extends HiStar by adding a new fundamental kernel type for energy: *capacitors*, an abstraction for flexible, dynamic, and hierarchical energy management. Cinder also differentiates itself by maintaining *task profiles* to aid meaningful policy generation. In Cinder, a capacitor is modeled after its real-world electrical counterpart, but with a special distinction: it contains rights to request consumption of joules, rather than reservations for actual joules. A capacitor is then a storage pool for *potential energy*, with additive increase and multiplicative decrease in residual potential energy. Capacitors can naturally place a ceiling on this quantity, which would ensure that it does not grow without bound.

Cinder will also maintain *task profiles*, running statistics of application energy consumption. Since it is unreasonable to expect a user to know or care how many joules their mail reader, for example, consumes over a certain time interval, it is nonsensical to have users express energy policies in units of energy or power, or fractions thereof. Instead, we envision users expressing their desire for the system to ensure that their mail reader will run for a given number of minutes or hours based on previous application behavior. It is then the responsibility of the system to keep the users well-informed, first aiding policy generation and later enforcing them.

2. STATUS

We have implemented the abstractions described in this paper as part of the Cinder kernel running on i386, amd64, and sparc platforms. Additionally, we have ported the exokernel, POSIX libOS, and minimal userspace to the ARM architecture with a limited set of devices. We are currently bringing the system up on the HTC Dream (T-Mobile G1 with Google) platform. We have already augmented the system to schedule CPU share along its container hierarchy using a hierarchical stride scheduler [7]. We suspect, with minimal additional work, the system can be made to treat CPU resources similarly to energy, perhaps even placing them in a variant of capacitors.

Our next step will be to implement an energy model for some set of hardware, refining and validating our work both in the lab and on real devices.

3. DESIGN

3.1 Capacitors

In Cinder, a capacitor is the arbiter of energy consumption and a means for tracking such usage. Threads are associated with one or more capacitors, and derive their ability to consume energy from these relationships. Capacitors may be connected to other capacitors to realize complex energy policies and enable applications to delegate their own energy resources. In isolation, however, a capacitor in Cinder can be regarded as a 4-tuple consisting of the following:

potential energy An integer representing the right to request some amount of energy from a parent capacitor. It is crucial to note that this only represents a request - it does not represent the joules that could be expended in exercising the request, nor is it a reservation (the actual joules may or may not be available at any given time).

expenditure The total quantity of actual joules consumed from the capacitor, for accounting purposes.

input rate The wattage of the capacitor; how much potential energy is added to it per second. This serves to bound any actual energy consumption.

decay factor A multiplicative factor between 0 and 1, inclusive, which affects the quantity of potential energy in the capacitor. This may be used to forbid hoarding, preventing applications from starving others, while providing a smooth feedback signal to the system when potential energy exceeds demand.

Capacitors fill with potential energy at the specified input rate and leak based on the decay factor. They are updated on demand, similar to Quanto [5], but unlike ECOSystem [9], avoiding the complexity of choosing a single frequency appropriate to a broad spectrum of device usage. We anticipate short-term energy expenditure to be approximated based on global system knowledge (e.g. device power states), and accurate accounting and billing to be performed on various events, such as explicit power state changes and timers that estimate potential energy exhaustion. Such events will also trigger surplus decay and potential energy addition. For example, if our input rate is a constant, i, the decay factor is d, and no energy is being consumed, after t seconds the capacitor's energy allocation is modeled as:

$$E = i\left(\frac{(1-d)^t - 1}{-d}\right)$$

Though the utility of capacitors does not depend on it, modeling them with multiplicative decay has three important benefits. First, it affords applications frequent, small deviations in energy consumption, yet it still facilitates wider fluctuations while discouraging hoarding and mitigating starvation. In other words, it is expected that applications have some variability in energy consumption in the course of normal operation. Applications should not be penalized for this; however, applications that wish to make use of large amounts of energy in short periods are a threat to the integrity of the system and should be taxed in proportion to the threat they represent. With decay, it is typically the case that an application can afford to do *something*, but it is rarely the case that it can afford do something expensive. Second, using decay provides a more even and continuous signal of the system-wide decay of all the capacitors. Using a cutoff value to prevent hoarding, as other systems have [9, 10], would cause a capacitor's decayed energy to instantaneously jump from zero to a positive value when it reaches

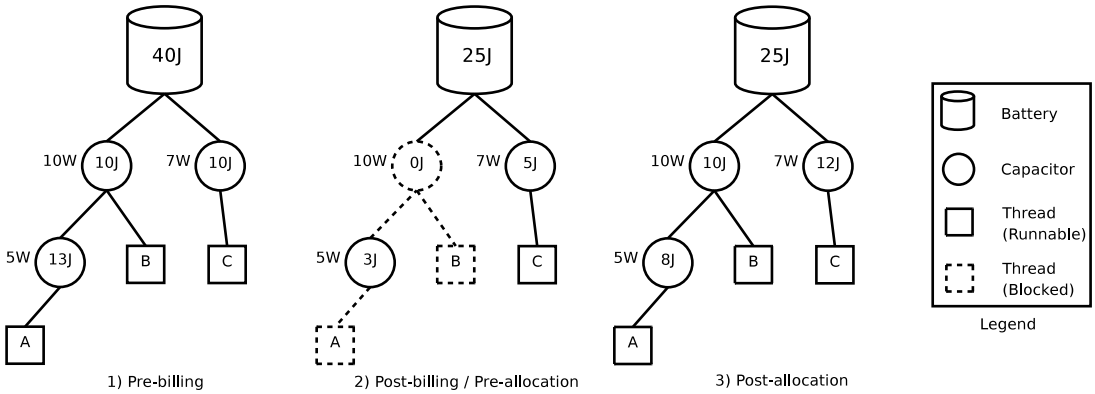

Figure 1: Example capacitor hierarchy. Lines depict the parent-child relationship between threads, capacitors, and the battery. Capacitors have a designated *input rate* (watts) and *potential energy* (joules). 1) Before any energy use is billed. 2) The hierarchy immediately after billing - note that a capacitor reaches 0 *potential energy* and descendant threads become unrunnable. 3) Infusion of *potential energy* after 2), but before any additional energy use is billed.

the cutoff. This sudden signal provides much less feedback to the system and application than a multiplicative decay and is less stable. As toolkits, power managers, and other utilities emerge, we believe that providing a continuous, gradual, and stable signal will greatly simplify their design and implementation. Finally, using a decay factor inherently binds capacitance to the input rate. Increasing a capacitor's input will also increase its maximum capacitance. Rather than tuning two separate parameters, decay makes it feasible to adapt the system by tuning only one.

3.2 Capacitor Hierarchies

Since mobile devices are becoming complex, multiprogrammed systems, they are host to dynamic workflows, comprised of several applications with varying degrees of importance and different requirements. In order to facilitate flexible energy allocation schemes, we envision individual capacitors as part of a hierarchy.

To realize this, capacitors form a tree: each capacitor has a parent capacitor. The root capacitor, representing the actual battery, has a 0 watt input rate (unless the battery is charging), a decay rate of 0, and a number of joules, likely queried from the actual battery. As the ultimate ancestor, it is special and unique in that it does not contain potential energy, but rather a reservation of actual joules. We refer to this capacitor as the "battery". A thread is attached to any number of capacitors from which it can choose to draw energy. When some energy is consumed (as a result of work done or requests made on behalf of the thread), its selected capacitor is debited, reducing its potential energy. This debit is propagated upward along the capacitor hierarchy, i.e. the potential energy of each capacitor is reduced by the same constant amount (Figure 1A,B). If *any* capacitor's potential energy along the path from the thread to the root of the capacitor hierarchy energy falls to (or below) zero the thread cannot draw more energy via that capacitor and is blocked (Figure 1B). In this way, capacitors represent a delegation of the right to request consumption of some amount, either absolutely (via the potential energy) or as a rate (via the input rate). In essence, any delegations for energy can be constructed, yet it is only when the root capacitor is finally debited that those delegations become manifest as actual joules. This ensures that all delegations, leading from the thread back to the root capacitor, agree and are satisfied that this consumption should be allowed.

Capacitors in Cinder are first-order objects that can be allocated and stored in the container hierarchy. As with all objects in Cinder, capacitors must reside in some container and, when all references to the capacitor are deleted, the capacitor must be garbage collected as well. If, in the course of operation, some capacitor that is an intermediate node gets garbage collected, a chain of capacitors (potentially linking some thread to the root capacitor) is broken. Since energy can only be extracted from a capacitor that has a path to the root capacitor (that is, the battery), the thread can no longer draw energy via this capacitor. This works naturally, since deleting a capacitor is, effectively, a revocation of the resources it was granted.

Importantly, however, a process need not always consist of threads drawing resources from a single capacitor. That is, although in the traditional UNIX sense we can imagine a process as a single container having resources associated with it and one or more threads within, Cinder is egalitarian by design - a process may create subcapacitors with different input rates, decay factors, and initial potential energy and place threads within them. This provides a simple and flexible means of suballocation; a process is at liberty to choose how its own energy allotment is to be spent.

Similarly, a thread need not be the child of a single capacitor. It is frequently the case that multiple principals may benefit from a specific task. Having a stake in the results of the task, a principal may wish to donate resources to it by allowing a thread to use potential energy from the principal's capacitor. Initially, only the thread creating the capacitor "owns" it, granting it the right to change its input rate, decay factor, and energy storage. It can delegate this privilege to other threads as it chooses. Likewise, any thread that owns the capacitor can grant other threads the privilege to consume energy from it (the "write" privilege) or it can delegate the privilege to grant that right as well.

3.3 Task Profiles

Capacitors are but a possible mechanism for complex, dynamic energy policy enforcement. They serve little purpose without a means of crafting policies that express users' actual intentions. Previous systems [3, 10] proposed that users specify a desired global system lifetime. It was then the job of either the operating system or the application to ensure that average power usage remained close to the rate necessary to meet this time goal. We consider it more useful, however, for a user to specifically express their intentions with respect to application use rather than to try for a global system lifetime. The system should be responsible for attempting to maintain the user's goals and to keep her apprised of its ability to do so in an uncertain and changing environment. To this end, additional information is useful.

For instance, a user may wish to keep their mail application running all day, while being able to watch an hour's worth of video. Interspersed could be web accesses, application downloads, and other activities of lesser importance. A global system lifetime does not represent the intention that the device reserve resources to handle a day's worth of email and watch the video clip, while permitting surplus energy to fund miscellaneous usage. In order to accomodate such policies, the system must have some *a priori* idea of the energy required and a means of adjusting and informing the user of its ability to meet the requirements.

We consider it essential, therefore, for the system to learn the energy consumption of various applications and to make this data accessible and useful in generating energy policies. Previous work [9, 4, 5] has pioneered several effective strategies for task-based energy consumption accounting. As the problem of understanding where energy is going and who is responsible is better understood, we endorse coupling this information with a policy generation engine. We suspect that even a simple system with worst, best, and average case estimations of energy consumption per application over a reasonable time interval would often suffice. Using such task profiles, the system could respond to a user's desire to run an arbitrary profiled application for a specific duration. Furthermore, such profiles could provide useful developer feedback, enabling them to determine inefficiencies across diverse hardware platforms and to amalgamate statistics to provide better energy consumption expectations to other mobile users.

Since energy consumption will vary significantly with many classes of applications (e.g. network access is expensive and often unpredictable), we cannot expect the system to be perfect. However, since much energy is consumed due to interactive, user-initiated events such as starting downloads or playing a media file, it is reasonable to engage the user in the enforcement strategy. For instance, the system may have a simple widget that indicates the system's perceived ability to meet the user's constraints given the recent or past energy consumption. That way, a user who does more work than expected can either reduce their burden on the system, or revise their expectations, perhaps by accepting the fact that e-mailing a series of large attachments will not be compatible with their video viewing.

4. APPLICATIONS

A typical mobile device consists of a wide variety of applications from various sources. In sophisticated, multiprogrammed systems, such as those present in modern mobile computing platforms, workloads are diverse and dynamic. Users may frequently use a web browser, a calendar, map software, a mail client, location sharing, and a media player. Today a user may be finding the location of the nearest restaurants and listening to audio while the device checks for email and nearby friends and activities. Though each of these services has varying importance to the user, he has no means to specify this to the system and each application is fully trusted with respect to resources. Therefore, resource delegation and isolation, particularly with respect to energy, are key in allowing users to exercise control over applications running on their behalf.

Previous systems, such as ECOSystem [9, 10], have considered a flat, proportional model of resource distribution among tasks, yet none has addressed modern realistic applications that have fine-grained resource concerns. Fortunately, it is these intricate applications themselves that are best equipped to specify such resource policies, and Cinder is designed precisely to take advantage of that. Any non-trivial application, such as mapping software, requires a number of subtasks (fetching map tiles, probing sensors, and rendering, for example). Current flat models cause parent processes to compete against their children with little control over them. Allowing energy accounting across a hierarchical structure like capacitors naturally facilitates such applications and restores control to the resource owners.

Cinder's capacitor mechanism makes it just as easy for an application to subdivide its energy share among its constituent subtasks as it does for a user. To make this practical, the concerns of users and their applications are intersected via the capacitor hierarchy, applying the policies of the superuser, the users, and the applications in a naturally composable way.

4.1 Fine-grained Control

Web browser plugins and extensions have become ubiquitous and important in the way users interact with the web on desktop platforms, and this trend is certain to spill over onto mobile devices as their browsers become more mature. With desktop computers generally being well overprovisioned, fine-grained resource control has taken a back seat to functionality. However, on mobile devices, where typically *all* resources are more precious than those on desktop machines, users cannot afford the luxury of such frivolous trust.

For example, a user may wish to install a browser extension to block advertisements by preventing their download from servers, knowing that the parsing of web pages will cause some additional computational overhead. Imagine the user has determined that he will want up to two hours of web browsing on the current battery charge, and he has instructed Cinder to assist him in achieving that goal. Cinder, in response to the user's desire, may determine that, based on prior behavior, the browser task should limit itself to two watts on average, compensating for burstiness via the potential energy of its capacitor. The system would create a capacitor with a two watt input and add a reference to the browser process.

The browser, in concert with task profiling information, may suggest and request a rate (perhaps, 0.25 watts) from the user to protect itself from the extension. It then employs Cinder's capacitors to install the necessary constraints, a

policy that naturally maps onto our framework. It creates a capacitor as a child of the first, giving it an input of 0.25 watts, and adds a reference to the extension's thread. The browser runs the extension when pages download and aborts the extension's threads if they take too long to respond (say, 100 ms), opting instead to display the version with the ads. Furthermore, the extension is free to protect itself in a similar fashion from inadvertently exhausting its own resources on large or complicated pages by parsing pages in threads that have the right to a small portion of the extension's 0.25 watts.

4.2 Buggy and Malicious Code

This same isolation extends to any code that cannot be fully trusted with energy. Applications in mobile platforms increasingly rely on poorly-vetted, potentially inefficient, and possibly malicious libraries. Despite the tight integration between the browser and its extension, all energy is carefully tracked and accounted for, ensuring that even if the extension were malicious it could not exceed its 0.25 watt quota. Cinder also inherits HiStar's fine-grained mandatory access control (explained in detail in [8]). Leveraging this system, the browser can protect the integrity and secrecy of its important data structures, which could otherwise cause the browser's own threads to consume additional energy (or worse). Given HiStar's focus on information flow control it is worth noting that capacitors can allow (potentially secret) information to flow between threads and that low integrity threads may affect high integrity threads using them. We view these channels as being explicitly authorized by the users of capacitors, and, depending on the construction of the capacitor hierarchy, they can be mitigated. We plan to formalize this more fully in future work.

4.3 Cooperative Resource Sharing

Treating capacitors as first-class objects in the operating system gives applications flexibility in how they use and share energy. Global Positioning System (GPS) sensors on mobile devices provide data that can be useful to a number of applications, but are energy hungry. Any single application alone may not be willing to bear the cost of making use of the sensors; however, capacitors give applications a way to cooperatively amortize the expensive sensor reads across interested threads.

For example, imagine a user is actively using a mapping application (`maps`) as she tracks her route on a drive. At the same time her mobile phone is taking advantage of a location sharing service (`loc`) that informs friends of her location. Both applications are making calls to the GPS daemon (`gpsd`). Inherited from HiStar, Cinder's "gate" kernel object type makes it straightforward to properly account for energy consumed for the expensive sensor reads, even if those reads occur as part of `gpsd` on behalf of another application. Being the only form of inter-process communication in Cinder, gates are simply named entry points into an address space that threads can "jump" into. Therefore, in this case, when `maps` makes a call to request GPS data from `gpsd`, the thread belonging to the `maps` application is executing in the address space of `gpsd`. In this way, all the energy consumed as a result of work completed by that thread is billed to its capacitor.

In order to conserve energy, one of the threads may frequently receive cached data after calling into `gpsd`. If `maps`, being unlucky, is always forced to pay for the actual sensor read, then it is unfairly bearing the full cost while `loc` benefits. To prevent such inequity, `gpsd` may require that all calls into it pay a certain amortized energy fee. Cinder makes this easy, since any application can subdivide and delegate its resources. Upon the `maps` thread calling `gps_read`, `gpsd` moves some number of joules from the `maps` thread's capacitor into its own capacitor. In the kernel, this is achieved by first ensuring that along the path from the `maps` capacitor to the root capacitor, the requisite amount of potential energy is available in each of the capacitors. If it is available, the system subtracts that amount of potential energy from each capacitor along that path, and then traverses the path from `gpsd`'s capacitor to the root capacitor adding that amount of potential energy to each capacitor. Being satisfied, `gpsd` returns the sensor value to `maps` (perhaps from a cached value or from a fresh read of the sensor), and on future calls it simply adds its capacitor as one of the parents of the client's thread, allowing it to use the energy within. As a result, `gpsd` can wait to accumulate enough energy from client threads before activating the GPS sensor again.

4.4 Background Applications

Capacitors also naturally encapsulate reduced frequency or reduced fidelity services. For example, continuing with the previous example, `gpsd` may have its threshold adjusted dynamically, allowing it to operate at reduced frequency when the battery is low or the system is under unexpected load. Likewise, the user may want the `loc` service to run with reduced performance while she performs other tasks or when the phone is idle. This is easily enabled by placing a low input rate on the capacitor, representing the entire `loc` background task. To enable `loc` to deal with its presumably bursty workload, she will grant it a low decay rate as well. However, when she opens the user interface corresponding to the `loc` service she may want increased responsiveness. The user interface's capacitor, therefore, supplies a subcapacitor with additional resources the `loc` background daemon can take advantage of.

5. RELATED WORK

The HiStar [8] operating system, on which Cinder is based, is designed from the ground up to minimize the amount of code that must be trusted. The system allows applications to express data security policies in terms of information flow, which the kernel enforces. HiStar has no notion of superuser, which complicates resource revocation, and makes it critical that all resources are carefully accounted for. For this reason, it uses hierarchical containers to account for all storage. Cinder's capacitors augment HiStar's kernel object type system to consider energy as well, creating a complete and coherent platform for resource management. The result is the ability to run the complex software stacks of today, while providing simple, transparent mechanisms that allow flexible and understandable policies for all aspects of the system.

Recognizing a need for resource accounting and policies independent of the process abstraction, Linux has recently incorporated "cgroups" [6] into the mainline kernel, which fills a similar role as resource containers [2] do. Though cgroups is extensible to support a wide variety of resources, it currently does not address energy. Furthermore, the Linux kernel contains many tasks for which energy use is not clearly

attributable to specific principals, complicating the problem. However, cgroups serves to demonstrate that the need for resource accounting, isolation, and subdivision is already being acknowledged by developers. In Cinder *all* allocation will be done in terms of capacitors and containers, guaranteeing that resources are properly accounted for, both in the kernel and in userspace.

ECOSystem [9, 10] presents an abstraction for energy, "currentcy", which unifies a system's device power states. It represents logical tasks using a flat form of resource containers [2] by grouping related processes in the same container. Such a model suffers under realistic workloads, as tasks have little control over how resources are used within subtasks, since it forces the parent process to compete against its (possibly ill-behaved) children for its own resources. Also, being implemented atop Linux, ECOSystem suffers from the same accounting issues as cgroups. Finally, ECOSystem distributes energy proportionally among applications, whereas Cinder leaves such policy up to the scheduler. This will allow Cinder to have different allocation policies for each capacitor, giving the system, and applications, additional flexibility.

ECOSystem must be configured with an energy model of the system to map runtime power states to power draw. This makes it susceptible to variability that occurs outside of the models for the devices, such as temperature or radio interference. Since Quanto [5] tracks actual energy use at runtime, it accounts for such dynamism. In any case, tracking energy consumption to device states remains an area of research and difficulty as systems become more decentalized, increasingly delegating work to specialized chips and devices. Cinder addresses the problem of presenting this low-level accounting to applications in a meaningful way while providing fine-grained control. Though we expect to face the same challenges found in other approaches, Cinder will benefit directly from all work in this area.

Acknowledgments

Some research is funded by NSF Cybertrust award CNS-0716806. This research was performed under an appointment to the U.S. Department of Homeland Security (DHS) Scholarship and Fellowship Program, administered by the Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement between the U.S. Department of Energy (DOE) and DHS. ORISE is managed by Oak Ridge Associated Universities (ORAU) under DOE contract number DE-AC05-06OR23100. All opinions expressed in this paper are the author's and do not necessarily reflect the policies and views of DHS, DOE, or ORAU/ORISE.

6. REFERENCES

[1] Apple previews developer beta of iphone os 3.0, Mar. 2009. http://www.apple.com/pr/library/2009/03/17iphone.html.

[2] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. *Operating Systems Review*, 33:45–58, 1998.

[3] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 48–63, New York, NY, USA, 1999. ACM.

[4] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[5] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In R. Draves and R. van Renesse, editors, *OSDI*, pages 323–338. USENIX Association, 2008.

[6] P. Menage. cgroups, Oct. 2008. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/cgroups/cgroups.txt;hb=b851ee7921fabdd7dfc96ffc4e9609f5062bd12.

[7] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical report, 1995.

[8] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

[9] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. pages 123–132, 2002.

[10] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *In Proceedings of the USENIX Annual Technical Conference*, pages 43–56, 2003.