Energy Management in Mobile Devices with the Cinder Operating System

Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières
Stanford University

Nickolai Zeldovich
MIT CSAIL

ABSTRACT

We present Cinder, an operating system for mobile phones and devices, which allows users and applications to control and manage limited device resources such as energy. Cinder introduces two new abstractions, *reserves* and *taps*. Unlike prior approaches, Cinder accurately tracks principals responsible for resource consumption even across interprocess communication, and allows applications to delegate their resources either in terms of rates or quantities. Rates can enforce system lifetime, while quantities can enforce dataplan or talk time limits. Proportional taps allow threads to prevent their descendants from hoarding unused energy. Cinder additionally institutes a global half-life to prevent malicious applications from starving the rest of the system.

We explore these abstractions, demonstrating their usefulness in a variety of applications running on the HTC Dream (a.k.a. Google G1). We show how Cinder maintains system lifetime in the presence of malicious applications, reserves energy for critical functions such as 911, supports energy-aware applications, easily augments existing Unix applications with energy polices, properly amortizes costs across multiple principals, and allows applications to sandbox untrusted subcomponents (such as browser plugins).

1 INTRODUCTION

In the past decade, mobile phones have emerged as a dominant computing platform for end users. These very personal computers depend heavily on graphical user interfaces, always-on connectivity, and long battery life, yet in essence run operating systems originally designed for workstations (Mac OS X/Mach) or timesharing systems (Linux/Unix).

Historically, such operating systems have had poor energy management or accounting support. This is not surprising: the first commodity laptop with performance similar to a desktop, the Compaq SLT/286 [1], was released just one year before the C API POSIX standard. The limitations of POSIX have prompted a large body of research to manage and control energy, ranging from CPU scheduling [9] to accounting [21] to offloading networking. Despite all of this work, however, current systems still provide little if any application control or feedback.

In the meantime, mobile devices are shifting from low-function proprietary applications to robust multiprogrammed systems with applications from thousands of sources. Recently Apple announced that their App Store now houses 185,000 apps [3] for the iPhone with more than 4 billion individual application downloads. This explosion of mobile software complexity and application availability makes it difficult to reason about the energy requirements and expenditures of mobile systems. As applications are bound to shift from simply being buggy to being outright malicious it will be critical that mobile operating systems provide mechanisms to protect the user's data and resources.

Recently, Android added the ability to track estimates of individual application energy consumption. This tracking uses a static cost model to various calls and counters like CPU time, packet counts, and Global Positioning System (GPS) requests. This represents a large step forward for helping consumers understand the mysteries of mobile device lifetime. But while Android provides improved *visibility* into system power use power use, it does not provide *control*. Besides manually configuring applications and manually checking power use every once in a while, there is no way for a user to enforce a maximum power draw or control system lifetime.

If we were to design a mobile phone operating system kernel from scratch, what would it look like? This paper presents an answer to this question: Cinder, an operating system designed around security as well as fine-grained resource accounting and control. To provide strong security, Cinder builds on top of the information-flow control facilities of the HiStar exokernel [19]. For resource management and control, Cinder takes advantage of device-level accounting and modeling, and accurately tracks parties responsible for resource use even across interprocess communication calls serviced in other address spaces. It provides *reserves* for resource delegation, which act as an allotment from which applications can draw resources, and *taps*, which place rate limits on the

consumption of applications and provide a point for fine-grained accounting. Taps connect reserves to one another allowing resources to flow through a graph to applications. Together, these abstractions allow users and applications to express their intentions, enabling the system to compose those concerns for policy enforcement.

We leverage Cinder's new abstractions in libraries, tools, and applications which enforce many typical and atypical energy policies. Cinder provides a working Unix-like environment running on AMD64, i386, and, most recently, ARM, beginning with our implementation on the HTC Dream (a.k.a. Google G1). We also provide utilities that allow crafting energy policies for existing Unix applications that are unaware of these abstractions, and make expressing and composing many policies a simple matter of shell scripting.

2 A Case for Energy Control

There is rich prior work on addressing the visibility problem [20, 21, 12, 11] of attributing consumption to application principals. Control, in contrast, has seen much less effort. Early systems like EcoSystem [20] proposed high-level application power limits. Mobile applications today, however, are much more complex: they spawn and invoke other services and have a much richer set of peripherals to manage.

We believe that for users and applications to effectively control power, an operating system must provide three mechanisms: *isolation*, *subdivision*, and *delegation*. We motivate these mechanisms through three application examples that we follow through the rest of the paper.

Isolation is a fundamental part of an operating system. Memory and IPC isolation provide security, while cpu and disk space isolation ensure that processes cannot starve others by hogging needed resources. Isolating energy consumption is similarly important. An application, whether rogue or buggy, should not be permitted to consume inordinate energy or the energy of others. Consider two processes in a system, each with some share of system energy. To improve system reliability and simplify system design, the operating system should isolate each process' share from the other's. If one process forks several additional processes, these children must not be able to steal the energy of the other. As a more concrete example, the energy a phone reserves for an emergency 911 call should be isolated from the rest of the system, so that other programs cannot use it.

Web browsers run (sometimes untrusted) plugins. Given that a browser receives a finite amount of power, it might want to protect itself from buggy or poorly written plugins that waste CPU energy. The browser would like to subdivide its energy so that it can give plugins a small fraction, knowing that isolation will prevent them from using its own lion's share. The ability to subdivide energy is critical for applications to be able to invoke other services without sacrificing all of their own resources.

Finally, there are times when applications need to allow others to use their energy, but do not want to carve off a reserved, isolated subdivision. The ability to delegate resources is an important enabler of inter-application cooperation. For example, the Cinder *netd* networking stack implicitly transfers energy into a common radio activation pool when an application cannot afford the high initial expense of powering up the radio. By delegating their energy to the radio, multiple processes can contribute to expensive operations; this can not only improve quality of service, but even reduce energy consumption.

Prior systems like EcoSystem [20] and Currentcy [21] provide isolation, but not subdivision or delegation. Isolation is sufficient when applications are static entities, but not when they themselves spawn new processes or invoke complex services. Subdivision lends naturally to standard abstractions such as process trees, resource containers and quotas, while delegation is akin to priority inheritance.

3 Design

Cinder is based on the HiStar operating system [19] which is a secure exokernel [8] that controls information flow using a label mechanism. The kernel provides a small set of kernel object types to applications, from which the rest of system is built: threads, address spaces, segments, gates, containers, and devices. Cinder adds two new kernel object types: *reserves* and *taps*. This section gives an overview of HiStar, describes *reserves* and *taps*, gives examples of how they can be used, and provides details on their security and information flow.

3.1 HiStar

The HiStar exokernel provides data-centric security policies on top of a set of six first-class kernel objects. Its segments, threads, address spaces, and devices are similar to conventional kernels. *Containers* provide hierarchical control over deallocation of kernel objects – objects must be referenced by a container or face garbage collection. *Gates* provide protected control transfer of a thread from one address space to a named point in another address space, and provide the basis for inter-process communication (IPC). Invoking a gate requires a privilege check and can optionally grant privileges to the calling thread.

More complex abstractions (files, processes etc.) are composed of these objects. The system has Unix-like compatibility through a library which runs stock software packages (bash, Xorg, xpdf, etc.). HiStar provides a single security primitive, *labels*, which makes these compositions safe while allowing fine-grained protection (e.g. on behalf of users, applications, processes, web origins, etc.).

Although the Unix security model of users and groups was not designed with mobile applications in mind, plat-

forms such as Android have managed to contort the system into providing some protection between applications [4]. HiStar's flexible security model allows Cinder to provide fine-grained, application-specific policies which are enforced by the small trusted kernel without re-purposing complex legacy mechanisms.

In HiStar, every kernel object is tagged with an immutable label at the time of its creation. Labels determine which threads can observe and which can modify the object. Roughly speaking, an object's label consists of a set of *categories* that restrict which threads can observe or modify the object. Threads *own* particular categories, which gives them privileges over objects labeled with those categories. Specifically, to observe an object, a thread must own all of the read categories in the object's label. To write the object, the thread must additionally own all of the write categories in the label. Any thread can create a new read or write category and place that category in the labels of objects it creates. It can also grant ownership of the category to other threads, so as to share privilege.

There is a third option besides read and write permissions: HiStar's information flow control allows one to label an object such that a thread may observe the object but only on condition that it give up the ability to communicate with the network or other processes. We believe such policies are crucial for protecting users' privacy and system integrity in an environment like mobile phones where users run untrusted applets. However, the original HiStar paper [19] already extensively discusses such uses of information flow control, so this paper concentrates on Cinder's contributions in resource management.

HiStar provides containers that allow effective accounting and revocation of storage resources. But the container hierarchy is insufficient for dealing with the complexities of energy management or resources where control over consumption *rate* is as important as *quantity*. Furthermore, powerful, hierarchical delegation of resources along a *single* hierarchy is insufficient. A system may want to group applications in one way for partitioning storage, another way for distributing energy, and yet another for distributing networking quotas. These shortcomings led us to rethink resource management, ultimately producing two new kernel object types: reserves and taps.

3.2 Reserves

A reserve describes a right to use a given quantity of a resource, such as energy. When an application consumes a resource the Cinder kernel reduces the values in the corresponding reserve. The kernel prevents applications from performing actions for which the reserve does not have sufficient resources. Reserves, like all other kernel objects, are protected by a security label (§3.6), that controls which threads can use and manipulate it.

All threads are associated with a reserve from which they draw energy. Cinder's scheduler is energy-aware and allows threads to run only if they have the required energy resources. Threads that have depleted their energy reserve cannot run which prevents new energy spending. This alone is sufficient to throttle energy consumption.

Reserves allow threads to delegate and subdivide their available resources. For example, an application granted 1000 mJ of energy can subdivide this reserve into 800 mJ and 200 mJ reserves, allowing another thread to connect to the 200 mJ reserve. Threads can also perform a reserve-to-reserve transfer of resources provided the thread is permitted to modify the level of both reserves.

Reserves track resources consumed from them by applications in order to provide accounting information to the user and applications, allowing applications to be made energy-aware. Finally, reserves can be deleted directly or indirectly when some ancestor of their storage container is deleted. Global decay allows the system to eventually recover the resources.

3.3 Taps

In theory, reserves are sufficient to control the system-wide use of resources. Transfer of resources between reserves could be implemented by special-purpose threads that explicitly move resources between reserves, and implement any rate-limiting and accounting policies that the resource owner requires. For example, suppose there are five applications, each of which should be able to consume an average of 1W of energy. To implement this, the administrator could create five reserves, one for each application, along with 5 threads, each of which is responsible for slowly transferring energy into one application reserve. Each thread would transfer energy until the application's reserve fills up to some maximum level (e.g. 2J), and would keep track of how much energy was transferred to that application.

However, this approach is likely to be inefficient in practice. If we want to perform fine-grained resource transfers and accounting, these special-purpose threads could consume a large fraction of the energy they transfer, if not more. Thus, we introduce a tap abstraction, which is conceptually an efficient, special-purpose thread whose only job is to transfer energy between reserves. While a reserve provides a quantity of a resource that can be consumed, a tap controls the rate at which a resource can be consumed. A tap has four pieces of state: a rate, a source reserve, a sink reserve, and a security label containing the privileges necessary to transfer the resources between the source and sink (§3.6).

Taps support two types of rates: constant and proportional. A constant tap transfers a fixed quantity of resources per unit time. For example, an application may be connected to the system battery via a tap supplying 1 mJ/s

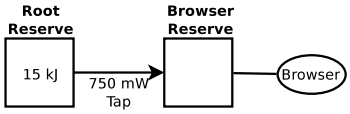

Figure 1: A 15 kJ battery or "root reserve" connected to a reserve via a tap. The battery of the device is protected from being misused by the web browser. The web browser draws energy from an isolated reserve which is fed by a 750 mW tap.

Figure 2: A common idiom in Cinder. The reserve on the left sources 1 mW of energy to the reserve on the right. Unused energy is taxed at a rate of 10%/sec and returned to the reserve on the left. The right reserve, while unused, converges on 10 mJ naturally as a product of the in and out taps.

which is 1 mW. Proportional taps transfer a portion of their source's resource per unit time. The principal use of proportional taps is to force a reserve to return unused resources and prevent hoarding.

Reserves and taps form a graph of resource consumption rights. In Cinder the system battery is represented in the resource graph as the *root reserve* of which the energy in all other reserves are a subdivision. Figure 1 shows a simple example of a web browser whose consumption is rate limited. This limit guarantees that even if the browser is aggressively using energy the battery will last at least 6 hours.

3.4 Reclaiming Unused Resources

Users and applications need to be able to give a task the ability to consume resources at a high rate but make sure that the task returns unused resources. For example, a foreground application on a phone may need to fully utilize many peripherals and drive the device at peak power. This requires giving the application a reserve fed with a very high tap. But this raises a problem: if the application draws less than the tap, the reserve will slowly fill with energy that no other application can use.

Cinder applications can solve this problem using proportional taps, as shown in Figure 2. In this configuration, the reserve on the right is limited to a maximum average power draw of 1mW. The backwards proportional tap means the right reserve can store up to 10s of this power (10mJ) for bursty operations. Once the reserve reaches 10mJ, the backwards proportional tap drains the reserve as quickly as the forward constant tap fills it.

3.5 Hoarding and Resource Decay

Backwards proportional taps allow applications to prevent energy from accumulating into reserves where it cannot be used. However, these backward taps do not prevent an application from hoarding energy. A thread can sidestep the taxation of a reverse tap by creating a new reserve with no proportional taps and periodically transferring resources to it. A malicious application could, over time, accumulate energy equal to the battery and starve the rest of the system.

While sophisticated constraints on reserve creation and resource transfer can eliminate this problem, these constraints significantly complicate energy-aware applications. Instead, Cinder prevents hoarding by imposing a global, long-term decay of resources across all reserves. Every reserve has an implicit proportional backwards tap to the battery.

This long-term decay prevents unbounded hoarding while keeping the application interface simple. Cinder computes decay using a system-wide half-life. Analogous to radioactive decay, each half-life period of time results in 50% of the energy stored in each reserve to be returned to the root (battery) reserve. In Cinder, the half-life is set to be shorter than the time between renewals of the resource (e.g. charging the device's battery), but long compared to the time quantum of an application. By default our system is configured with a 10 minute half-life. A long half-life allows applications to accumulate and store energy for significant periods, but allows the system to make large-scale, long-term hoarding impossible.

3.6 Access Control & Security

Cinder's reserves and taps are egalitarian: any thread can create a reserve or tap to subdivide and delegate its resources. Securing them is a matter of making information flow explicit in the resource graph formed by the reserves and taps, which requires ensuring the new kernel object types are protected by appropriate security labels and that all interactions with them obey those labels.

Reserves, being a passive construct, are easy to protect. Like other objects in HiStar, a reserve has a simple security label that describes which privileges are needed to observe (read) it or modify (write) it. Using resources from a reserve requires both read and write privileges: read because a failed draw provides information on the value (zero) and write for when it succeeds.

A tap actively moves resources between a source reserve and a sink reserve. Therefore, the tap needs permission to read and write both the source and sink, in order to decrease and increase the level, respectively. It may be the case that the source and sink's labels restrict information flow between the two objects, in which case any thread would need privilege to write both of them. For this reason, taps may own categories just like threads [19]. (A thread must own any categories that it grants to a tap it creates.) Storing privileges in taps conveniently and safely allows taps to transfer resources between reserves even involving complex security policies.

3.7 Accurate Accounting via Gates

HiStar's gate object type forms the basis of inter-process communication in Cinder. A gate is a named entry point in an address space, typically corresponding to a daemon or system service available over IPC. When a thread invokes a gate, it transitions from its original address space to the address space of the service it is invoking. Unlike traditional IPC, in which a thread in a client process sends a message to a thread in a server process, with HiStar the calling thread itself enters the server's address space. Since Cinder tracks resource consumption by the active reserve of a thread, the caller of a system-wide service is billed for resource consumption it causes, even while executing in the other address space.

In contrast, other systems such as Linux or OS X for the iPhone would need some form of message tracking during inter-process communication in order to heuristically bill the correct principals for resource consumption. Cinder provides more accurate accounting naturally. Cinder's `netd` networking stack, described in more detail in §6.3, takes advantage of this making it easy to bill threads for energy consumption due to network access.

3.8 Atomic Transfers and Debits

Many system service operations need to be performed in an all-or-nothing way. For example, work is wasted if an application begins a TCP flow but cannot finish it due to a lack of resources. Cinder's user space networking stack, therefore, ensures a thread has a threshold of energy before allowing a thread to queue a packet (e.g. enough energy to turn on the radio). Furthermore, being a user space networking stack, the kernel is unaware of how much energy to bill threads for received packets, hence threads should be able to debit their own reserves even if the debit causes the reserve to go into debt to perform this accounting in user space (likely while under control of a gate, as in the case of the networking stack).

To provide for these cases, Cinder's system calls for transferring resources can optionally fail if there are inadequate resources to complete the transfer or can transfer as much as is available from the source. Likewise, a debit of a reserve can fail if the reserve's level is too low or can be forced even if it causes its level to become negative. These options allow a service to guarantee that a thread has enough resources to complete an action beforehand and set aside some amount for payment, forming primitives for a sort of resource-aware locks. Forcing debits also allows the actual cost for the action to be billed even if it can only be determined after-the-fact and the debit would leave the reserve with a negative level.

3.9 Resource Inversion

Resource limits can lead to resource inversions, where a thread with plenty of resources available cannot run because a thread holding a lock has run out of resources. The solution is similar to more traditional priority inversion. The thread blocked on the lock can donate resources to the thread holding the lock; just as a thread can lend its priority, a thread can lend its resources.

```
reserve_create(container, label)
reserve_get_level(reserve)
reserve_get_total_consumed(reserve)
self_get_active_reserve()
self_set_active_reserve(reserve)
tap_create(container, source, sink, label)
tap_set_rate(tap, tap_type, rate)
reserve_transfer(source, sink, amount, can_fail)
reserve_debit(reserve, amount, can_fail)
```

Figure 3: System call interface for reserves and taps.

4 IMPLEMENTATION

We implemented our abstractions in the Cinder kernel, which runs on AMD64, i386, SPARC, and ARM architectures. The kernel is available under the GNU General Public License version 2 and is freely available via the Internet. Our principal experimental platform is the HTC Dream (Google G1), a modern smartphone based on the Qualcomm MSM7201A chipset. We have ported Cinder to the HTC Dream and profiled its energy usage. Because porting a kernel to a mobile phone platform is a non-trivial task that is rarely attempted, we describe our process here.

4.1 HTC Dream Port

To run Cinder on the HTC Dream we first ported the kernel to the generic ARM architecture (2,380 additional lines of C and assembly). MSM7201A-specific kernel device support for timers, serial ports, a simple framebuffer, interrupts, GPIO pins, and keypad required another 1,690 lines of C. Cinder implements the GSM/GPRS/EDGE radio functionality in userspace with Android driver ports.

The Qualcomm chipset includes two ARM cores: the ARM11 runs application code (Cinder), while a secure ARM9 controls the radio and other sensitive features. The two cores communicate over shared memory and interrupt lines. To access these undocumented facilities, we mapped the shared memory segment with a privileged process and ported the Linux shared memory device to userspace. This *smdd* daemon (4,756 lines) provides services via gate calls to other consumers, including the radio interface library (RIL).

In Android, the radio interface library consists of two parts: an open source generic interface library that provides common radio functions regardless of hardware platform, and a device-specific, Android-centric shared object that interfaces with the modem hardware (libril.so). Unfortunately, libril.so is closed-source and precompiled

for Android: this makes it excessively difficult to incorporate into another operating system. Without hardware documentation or tremendous reverse engineering, using the radio requires running this shared object in Cinder. To do so, we wrote a compatibility shim layer to emulate both Android's "bionic" libc interface, as well as the various /dev devices it normally uses to talk to the ARM9 (1,302 lines of C). We rewrote the library's symbol table to link against our compatibility calls, rather than the binary-incompatible uClibc functions and syscalls that regular Cinder applications use. Finally, we wrote a port of the radio interface library front-end that provides gates that service requests.

Cinder currently supports the radio data path (IP), and can send and receive SMS text messages. Cinder can also initiative and receive voice calls, but as it does not yet have a port of the audio library, calls are silent. In retrospect, since hardware documentation is unavailable, basing our solution on Android would have been far simpler from a device support perspective. However, that would have traded off the simplicity and accuracy of IPC resource accounting in Cinder and the powerful security abstractions inherited from HiStar. We felt that a cleaner slate justified the additional burden and reduced functionality.

Operating the radio is a quite complicated, requiring about 12,000 lines of userspace code along with the 263KB closed libril.so. In comparison, the entire Cinder kernel consists of about 27,000 lines of C for all four CPU architectures and all device drivers. The kernel is only 644KB - less than 2.5 times the size of libril.so.

4.2 HTC Dream Power Model

Energy accounting for a device as complex as the HTC Dream is a difficult task, compounded by the closed nature of the hardware. The closed ARM9 manages the state of the most energy hungry, dynamic, and informative components (e.g. GPS, radio, and battery sensors) and provides limited visibility to the rest of the system. The battery level, for example, is exposed to Cinder as an integer from 0 to 100.

Cinder's resource management mechanisms and policies are independent of its accounting mechanisms. Poor accuracy means that Cinder may overestimate or underestimate system lifetime. Recent work on processors has shown that fine-grained performance counters can enable accurate energy estimates within a few percent [18, 7]. Without access to such state in the HTC Dream, however, we rely the simpler, well-tested technique based on offline-measurement of device power states in a controlled setting [11, 20, 12]. This is the common approach used in phones today, and so has equivalent accuracy to commodity applications.

We evaluated the Dream's energy consumption during various states and operations to understand the device's behavior and statically model it. Our energy model is a function of device states and duration to an estimated amount of consumed energy. All measurements were taken using an Agilent Technologies E3644A, a simple DC power supply equipped with a current sense resistor that can be sampled remotely via an RS-232 interface. We sampled both voltage and current approximately every 200 ms, and aggregated our results from this data.

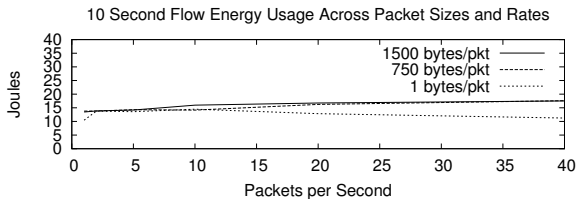

Figure 4: Radio data path power consumption for 10 second flows across six different packet rates and three packet sizes. Short flows are dominated by the 9.5J baseline cost shown in Figure 5. Data rate has only a small effect on the total energy consumption. The average cost is 14.3J (minimum: 10.5, maximum: 17.6).

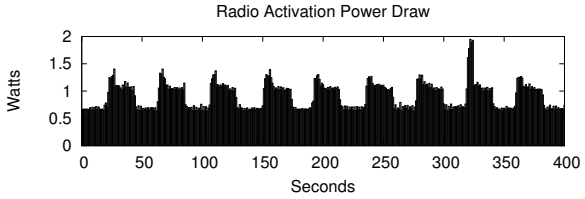

Figure 5: Cost of transitioning from the lowest radio power state to active. One UDP packet is transmitted approximately every 40 seconds to enable the radio. The device fully sleeps after 20 seconds, but the average interval consumes an additional 9.5J of energy (minimum 8.8J, maximum 11.9J). Power consumption for a stationary device can often be predicted with reasonable accuracy, but outliers such as the penultimate transition occur unpredictably.

Our HTC Dream's stock battery has a claimed capacity of 1150 mAh, which at the device's rated 3.7 V works out to about 15 kJ. While idling in Cinder the Dream used about 699 mW and another 555 mW when the backlight was on. Spinning the CPU caused another 137 mW of consumption. Our testing showed memory-intensive applications do increase power (a 13% increase over CPU power while just spinning); however, quantifying memory accesses is difficult without hardware support, may introduce significant overheads, and only has the potential to marginally increase the model accuracy. The ARM processor also lacks a floating point unit, leaving us with only integer, control flow, and memory instructions. For these reasons, our CPU model currently does not take instruction mix into account.

Data path power consumption is a particularly interesting case in that the baseline cost of activating the radio is very high. Small isolated transfers are therefore especially expensive. Unlike bulk transfers, they are unable to

amortize the radio power up cost. Figure 4 demonstrates the cost of activating the radio and sending UDP packets to an echo server that returns the same contents. Results demonstrate that the overhead involved dominates the total power cost for flows lasting less than 10 seconds in duration, regardless of the bitrate.

Figure 5 shows what this activation cost is. We power up the radio by sending a single 1-byte UDP packet to a sink host. The secure ARM9 controls the radio power states and automatically returns to a low power mode indistinguishable from our baseline power consumption after 20 seconds of inactivity. Cinder is unable to control this state on our closed hardware platform and, as a consequence, the inactivity timeout cannot be reduced.

The entire transition costs 9.5 Joules, making it especially expensive for short network transactions. It is desirable that applications coordinate their actions to amortize their costs and use the network in as simultaneous a fashion as possible. For example, if a set of applications that each access the network only occasionally are staggered, the effect on energy draw would be far more severe than if they were coordinated. In Section 6.3 we demonstrate how the abstractions presented in the previous section can be used for exactly this purpose.

5 APPLICATIONS

In order to gain experience with Cinder's abstractions we developed a number of applications using reserves and taps. This section describes the design of these applications including a task manager application that limits the energy consumption of background applications, a command-line utility that augments existing applications with energy policies, and an energy constrained web browser that further isolates itself from its browser plugins. We also discuss how Cinder could be used to reserve energy, particularly for emergency phone use.

5.1 Background Applications

Background applications complicate resource management on mobile phones substantially. Problematically, despite not being visible to the user, an application may still be using resources. This discontinuity between reality and what the user perceives tends to make the user suspicious of the foreground applications they have used most frequently, which may not be the responsible ones. Cinder provides not only a means to understand which applications are using resources, but also a means to manage those resources to meet the user's expectations. For example, since the user naturally suspects foreground applications of using energy the user can easily understand and manage their use of those applications. Cinder's job, then, is to manage background applications to prevent them from interfering with the user's natural intuition.

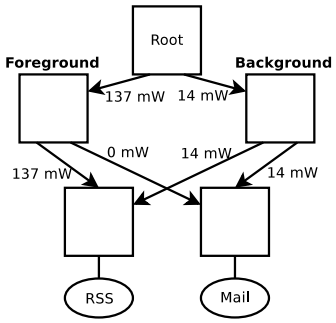

Figure 6: RSS is currently running in the foreground so the task manager has set its tap to give it access to additional energy. Mail is running in the background and can only draw energy from the background reserve. This enforces that actual battery consumption matches the user's expectation that the visible application is responsible for most energy consumption.

Figure 6 shows just how Cinder can accomplish this. Each application has a reserve allocated to it from which it draws energy. Also, each application's reserve is connected to two other reserves via taps. The first reserve is the foreground reserve which is connected to the battery via a high rate tap. The second is a low rate reserve connected to the battery via a low rate tap. An application's tap to the background reserve always allows energy to flow; however, the foreground tap is set to a rate of 0 while the application is running in the background and is set to a high value when the application is running in the foreground. A task manager is the creator of the tap connecting the application to the foreground reserve and, by default, is the only thread privileged to modify the parameters on the tap. Since programs are confined to low power while in the background, the user's expectations are respected. Section 6.2 evaluates this configuration in more detail.

5.2 `energywrap`

Taking advantage of the composability of Cinder's resource graph, the `energywrap` utility allows any application to be easily sandboxed even if it is buggy or malicious. `energywrap` takes a rate limit and a path to an application binary. The utility creates a new reserve and attaches it to the reserve in which `energywrap` started by a tap with the rate given as input. After forking, `energywrap` begins drawing resources from the newly allocated reserve rather than the original reserve of the parent process and then executes the specified program. This allows an application, even if it is entirely energy-unaware, to be augmented with energy policies.

The simple sandboxing policy provided by `energywrap` is implemented in about 100 lines of C++. An excerpt is shown in Figure 7. HiStar provides a `wrap` utility designed to isolate applications with respect to privileges and storage resources. Coupling this utility

```
// Create a reserve
object_id_t res_id;
res_id = reserve_create(container_id, res_label);
objref res = OBJREF(container_id, res_id);

// Create a tap and connect it between
// the battery and the new reserve
object_id_t tap_id;
tap_id = tap_create(container_id, root_reserve,
                    res, tap_label);
objref tap = OBJREF(container_id, tap_id);
// Limit the child to 1 mW
tap_set_rate(tap, TAP_TYPE_CONST, 1);

if (fork() == 0) {
    // child process: switch to new reserve before exec
    self_set_active_reserve(res);
    execv(args[0], args);
}
```

Figure 7: `energywrap` excerpt without error handling.

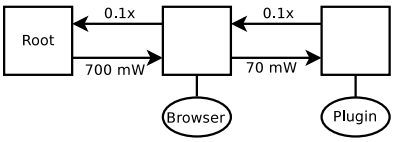

Figure 8: A web browser configured to run for at least 6 hours on a 15kJ battery. The web browser further ensures that its plugin cannot use more than 10% of its energy. 0.1x proportional taps prevent the browser and the plugin from hoarding energy.

with `energywrap` allows any application or user to provide a virtualized environment to other threads and applications. Section 6.1 evaluates the effectiveness of energy sandboxing and isolation.

`energywrap` has proved quite useful in implementing policies while designing and testing Cinder, particularly for legacy applications that have no notion of reserves or taps. Since `energywrap` executes an arbitrary executable it is possible to use `energywrap` to wrap invocations of itself or shell scripts which may invoke `energywrap` with other scripts or applications. This allows a wide class of ad-hoc policies to scripted using standard shell scripting or on-the-fly at the command line.

5.3 Fine-grained Control

Mobile web browsers now support plugins like Adobe Flash [2] and we can expect more plugins and extensions to follow. However, on a mobile device where resources are precious it is especially important to have control over how these plugins use resources.

In Cinder, a user may allocate some fixed rate or quota of energy for web browsing using reserves and taps. The web browser, then, may want to run a browser plugin while ensuring the plugin cannot starve other plugins or even the browser itself. The browser, as shown in Figure 8 allocates a separate reserve for the plugin and connects it to its own energy via a low rate tap.

Often a single plugin (e.g. Flash) may be handling a number of applications or requests all in a single process. In order to easily scale the amount of energy given to the plugin with the number of applications it is handling, the browser can simply add an additional tap per application. Then when a particular application is no longer being handled (e.g. the user navigates away from the page) the taps associated with that page can be automatically garbage collected, effectively revoking those energy sources.

Cinder includes a simple graphical web browser based on links2 that runs either in Xorg or standalone against the framebuffer. The browser is augmented with an extension running in a separate process whose resource usage is subdivided and isolated from the browser process itself. The browser, then, can send requests to the extension process (for ad blocking, etc.) and if the extension is unresponsive due to lack of energy the browser can display the unaugmented page.

5.4 Energy-Aware Applications

Resource consumption for applications can be limited in Cinder by running the application within energywrap, whether or not the application was written with Cinder's abstractions in mind. In addition, developers can use the abstractions provided by Cinder to gain fine grained control of resource usage within their applications to provide a better experience to end users.

For example, consider a GPS based application that lets a user know his precise location, and as an extra service pulls up location based information over the network. When battery life is low, the user might prefer that remaining energy be diverted towards location updates and not expended on finding location based data. On a platform that does not provide Cinder's abstractions, the application would need to present an interface to the user that supports modifying the priority of each operation. The user would need to use this interface each time he wants to modify the resource consumption behavior of the application, which is inconvenient. Alternatively, the application could query battery status and automatically limit the rate at which it makes network requests, solving the issue of convenience. However, controlling the resource consumption of the resulting application relative to other applications would still not be possible.

With Cinder, however, the developer could move the GPS and network related operations to separate threads, each possessing an independent reserve. The application could dynamically assign resources to each thread based on the resources available in its primary reserve. The resulting application would possess the benefits of automatically adapting its behavior based on available resources, and its resource consumption relative to other applications would be controllable by means of modifying the

parameters of its primary reserve. Thus a Cinder-aware application could reduce its overall energy usage while still providing important data to the user without requiring any ongoing configuration, while the Cinder platform ensures that it does not use up too many resources compared to other applications on the device.

Another energy usage optimization can be made when writing programs where the partial or degraded results of a computation are still of use to the user, and offer a good compromise between battery usage and user experience. For example, the quality of streaming video viewed on the phone can be scaled back, or texture quality in a game can be reduced, when the energy available to the application is low. Both outcomes are preferable to not being able to watch a video or play a game at all when insufficient resources are available to run the necessary computations at full fidelity.

As a concrete example, we have implemented an energy aware networked picture gallery viewer. The application maintains a separate thread for downloading images off of a server using a separate energy reserve. The main application checks the levels in the downloader's reserve periodically. The rate of energy flowing into the reserve could potentially be determined by system policy, by the parent process of the application, or the user depending on the exact use case. Energy flows out of the reserve at a rate dependent upon the operations performed by the application.

If the energy in the reserve drops below a threshold value, the main thread signals the downloader thread to obtain lower quality pictures. Note that a drop below the threshold value signals that the rate at which the application is expending energy is larger than the rate at which it the system provides it with energy. In the current implementation, the downloader thread complies with such a signal by requesting partial downloads of interlaced PNG images which yield a lower quality image in exchange for a smaller amount of bytes transferred over the network (and thus lower energy consumption by the network device). Depending on the application, different approaches can be used; for non-interlaced images the server might store a lower resolution copy of the image that is served instead, for example.

We performed tests with the application with energy-sensitive image downloading enabled and disabled. Our test case consisted of the application alternating between downloading a batch of images and sleeping. Each image in a batch was of similar size (approximately 2.7MB) and each batch contained the same number of images. Sleeping allowed the energy reserve for the download thread to fill with more energy at a constant rate. The first sleep cycle lasted for 40 seconds; the length of each successive sleep cycle diminished by 5 seconds per cycle, so a smaller amount of energy built up in the reserve after

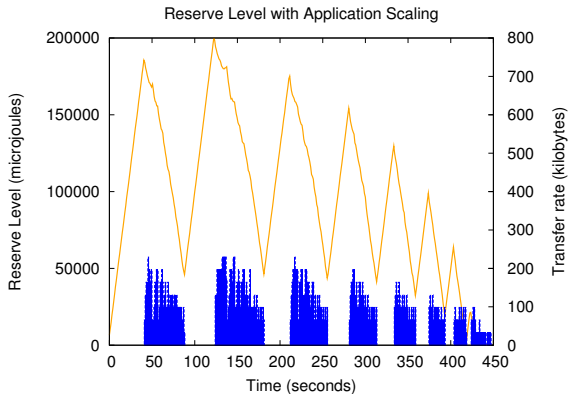

Figure 9: Image viewer with energy-aware scaling of image quality enabled. The orange line represents the energy in the downloader thread's reserve while the blue lines represent the amount of data downloaded. As energy become scarce (the reserve empties), quality is lowered and less data is downloaded per image.

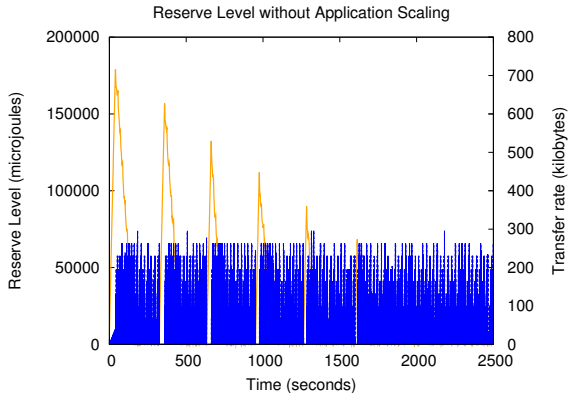

Figure 10: The same image viewer application as in Figure 9, but without dynamic scaling of image quality. The experiment takes over five times as long to complete within the energy budget since the application cannot adapt to reduced available energy.

each batch was downloaded. The rate at which energy was fed into the reserve was significantly smaller than the rate at which energy was used by the application. The test simulates a user loading a page of images, pausing a while to view the images, and then requesting more. The rate of energy flow into the download thread's reserve is low to simulate a low battery life environment. We track the energy reserve levels, the amount of bytes transferred over the network interface, the download time for each batch of images and the average bytes transferred per image over time.

The difference in behavior between the two operating modes for the application is pronounced; when image download requests are scaled in an energy aware fashion as in Figure 9, the quality of images and bytes transferred for each image progressively drops as resources are exhausted below the threshold point, while the time

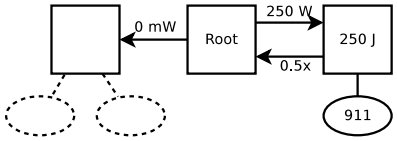

Figure 11: About 5 minutes of talk time (250 J) worth of energy set aside for an emergency call and transmission of the caller's current location. When the device detects an emergency call it sets the tap feeding normal applications to 0 mW. The proportional tap prevents 911 from hoarding energy during normal operation.

per transfer per image slowly decreases since the amount of bytes needed to transfer per image drops. Over the course of the test, the level of energy present in the reserve dropped below the threshold but never to zero.

When image download sizes are not scaled back as in Figure 10, the amount of bytes transferred stays constant per batch. With each successive batch, the amount of energy in the reserve at the start of the batch decreases since the thread sleeps for a smaller amount of time after downloading each batch. Thus the reserve runs out soon after the start of each batch in this case, with the image transfers stalling until enough energy is available to the thread to continue. This yields a significantly larger run time.

One can argue that the large runtime for the non-energy aware application is due in part to the Cinder platform purposely stalling it as it runs out of energy, while a non-Cinder platform would run the application faster since it wouldn't be blocked by the reserve aware scheduling mechanism. However, energy aware applications running on Cinder could be both fast and use a lower amount of energy while still providing an acceptable end user experience. Energy-oblivious applications that must be run in Cinder could still run at full speed even in low battery situations if run in the root energy reserve of the system; while that would sacrifice the energy management features that Cinder brings to the table, it would not yield battery life or performance any worse than running the application on other platforms.

5.5 Emergencies

Figure 11 shows how Cinder could be used to ensure that the device always has enough energy for a 5 minute emergency phone call. The taps between the battery and emergency reserve ensure the application always has 250 J set aside. The amount set aside can be computed from the capacity of the battery and the rated talk time for the phone. When an emergency call is placed the device clamps off all other applications by setting the tap that feeds the rest of the system to a rate of 0 mW. This guarantees not only a 5 minute talk time, but also that the rest of the phone's applications cannot interfere or consume energy while an emergency call is in progress. We have yet to implement and evaluate this application.

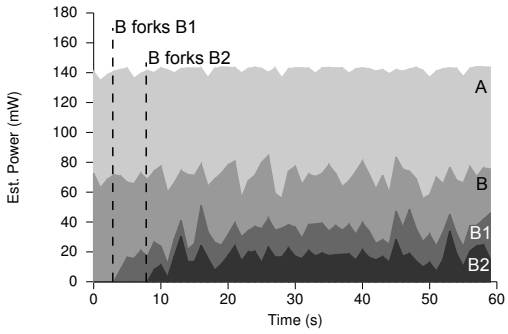

Figure 12: Stacked graph of Cinder's software-estimated CPU power during energy isolated process execution. Process A's energy consumption is isolated from other processes' energy use despite B's periodic spawning of child processes (B1 and B2). The sum of the estimated power of the individual processes closely matches the measured true power consumption of the CPU of about 139 mW during this experiment.

6 Evaluation

Using the set of applications described in §2, we evaluate whether Cinder can control power through subdivision, delegation, and isolation as well as whether it provides visibility into the energy and power of a running system. Furthermore, by examining how applications use the phone datapath, we evaluate whether Cinder can improve a system's energy efficiency by managing complex devices with non-linear power consumption.

All experiments in this section are evaluated using Cinder running on an HTC Dream. To measure power draw, we connect the Dream to the Agilent E3644A DC power supply. To monitor reserve energy levels we use the Dream's serial port output.

6.1 Isolation, Subdivision and Delegation: Buggy and Malicious Applications

We first show how a very simple use case – preventing the system from a buggy or malicious energy hog – requires isolation, subdivision, and delegation. Figure 12 shows a stacked plot of Cinder's power consumption estimates of two processes, A and B. Both are configured with separate reserves and taps each at a rate high enough to utilize the CPU at 50% apiece (about 68 mW each for the CPU which costs 137 mW to spin).

Process B spawns a new child process at about 5 seconds (B1) and again at about 10 seconds (B2). In a traditional system without reserves, limits, or some other form of hierarchical resource container, these additional processes would typically cause A to receive a smaller share of the CPU. With Cinder, however, Process A is isolated from these forks and still consumes about 50% of the CPU (and energy share).

Rather than have B1 and B2 draw from B's own reserve,

B creates two new reserves subdividing and delegating its own power to each using two taps. Each of the two taps is equal to one quarter of B's tap, such that after spawning both they are using half of its energy. This shows that Process B is free to subdivide and delegate energy from its reserve by creating new reserves, taps, and child processes which are guaranteed to be unaffected by Process A's energy consumption. Figure 12 shows B1 and B2 enjoy their resources isolated from A.

6.2 Delegation and Subdivision: Background Applications

Section 5.1 presented a configuration where system power is subdivided into a high power task manager reserve and a low power background reserve. These reserves delegate their energy to applications running in the foreground and background respectively. This delegation causes background applications to continue to make slow forward progress but keeps foreground applications responsive.

We set up a reserve and tap graph identical to Figure 6. We also use a similar graph that provides 300 mW in the foreground instead of 137 mW to highlight the expected behavior of applications with respect to the task manager.

Figure 13a shows two processes spinning on the CPU while in the background. The background provides the two of them 14mW, just enough to keep the 137mW CPU at 10% utilization. At about 10 seconds the task manager selects Process A as the foreground process, granting it enough energy to fully utilize the CPU (137 mW). Process B continues to run according to its background energy share of 14mW. At the 20 second mark the task manager retires Process A to the background by setting its foreground tap rate to 0 mW. At 30 seconds the task manager gives Process B access to the foreground resources and, similarly, returns it to the background at 40 seconds.

Figure 13b is the same configuration when foreground gives 300 mW of foreground energy. Because 300mW is greater than the CPU cost of 137mW, applications in the foreground can accumulate excess energy. The two processes move in and out of the foreground in the same way as before, but this accumulated energy changes their behavior. When B is moved to the foreground. A still has plenty of energy, and so competes with B for the CPU, such that each receives a 50% share. After A exhausts its energy, it returns to its original 14mW. Shortly thereafter, B moves to the background as well. But just as A did, it accumulated energy during its time in the foreground and so is able to use ≈90% of the CPU until it exhausts its reserve.

The system-wide half-life both caps the total energy hoarding possible during foreground operation and returns applications to the natural background energy level over a several minute period. This allows a process to perform an elevated amount of work briefly after returning to background status provided it underutilized its resources while in the foreground.

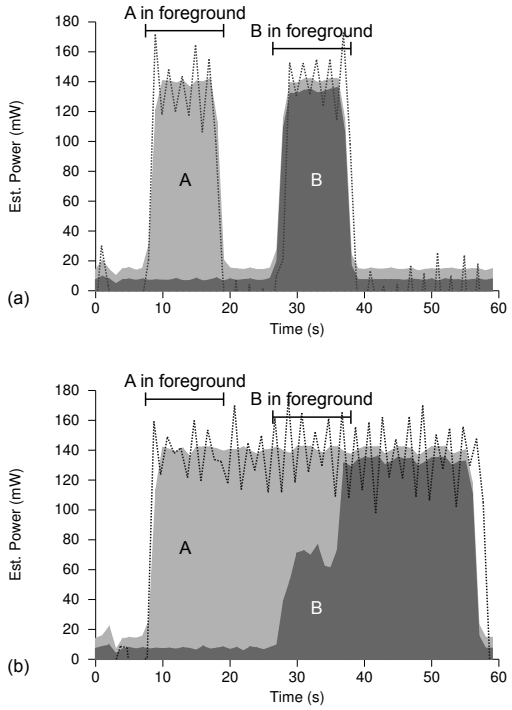

Figure 13: Stacked graph of Cinder's software-estimated CPU power as processes A and B both spin on the CPU. Together they are allowed 14 mW while in the background. The task manager runs A in the foreground in the 10 - 20 second interval and B in the foreground during the 30 - 40 second interval. (a) shows the results for the foreground process with 137 mW (the precise cost of using the CPU at 100%). (b) shows the foreground process with 300 mW. The dotted line shows actual power measurements compensated for baseline power draw with an idle CPU and averaged over 1 second intervals.

6.3 Delegation: Cooperative Networking Stack

Some of the most energy-hungry devices on a mobile device have complex, non-linear power models (e.g. the data path and the GPS). Cinder's abstractions, particularly reserves and gates, readily allow sophisticated policies that can reduce overall system energy consumption with such devices through delegation.

Section 4.2 showed the radio has a high initial cost and a much smaller amortized price for bulk transfers. This power profile is a problem for many background applications like email checkers, RSS feed downloaders, weather widgets, and time synchronization daemons.

Cinder can solve this problem using reserves and taps. The networking stack, *netd*, contains a reserve that saves energy for a radio power up event. If a thread makes a network system call but the sum of it and *netd*'s reserve are not sufficient, the call blocks. Blocked threads contribute joules acquired by their taps to the communal *netd* reserve. When there is sufficient energy to turn the radio

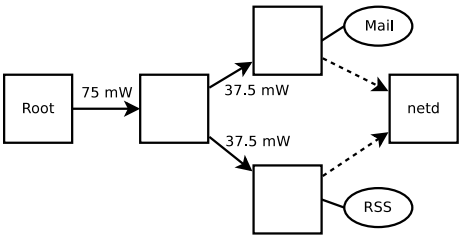

Figure 14: The mail checker and RSS feed downloader are constrained to use up to 37.5 mW apiece. When making network requests, `netd` transfers energy from their reserves into its own reserve. Once the requesting application's reserve combined with the `netd` reserve have enough energy the radio will turn on. This simple policy helps synchronize applications' network access, reducing active radio time and saving energy.

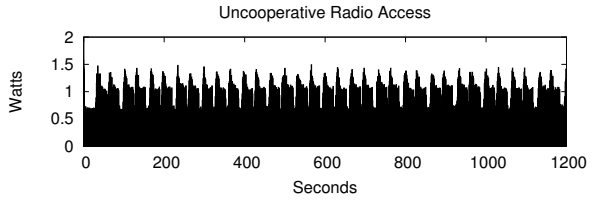

Figure 15: Two background applications, a pop3 mail and an RSS fetcher, each poll every sixty seconds. Since they are not coordinated, their use of the radio is staggered, resulting in increased power consumption. Each application uses the radio for at most a few seconds, but neither takes advantage of the other having brought the radio out of the low power idle state.

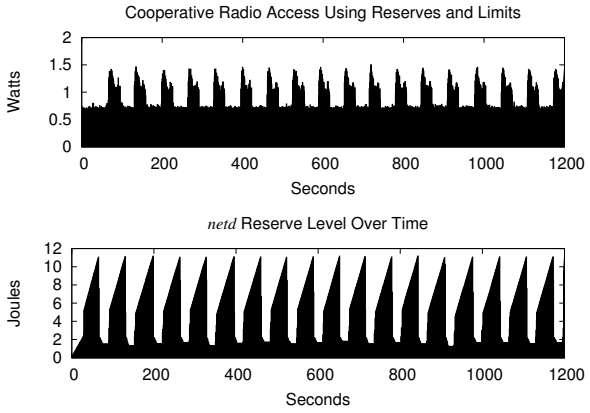

Figure 16: *Top*: The same mail and RSS background applications using reserves and limits to coordinate their access to the radio data path. Enough energy is allocated to each application to turn the radio on every two minutes. By pooling their resources, they are able to turn the radio on at most every sixty seconds. *Bottom*: The level of the reserve into which the two background applications transfer their allotted joules. When the reserve reaches a level sufficient to pay for the cost of transitioning the radio to the active state, it is debited, the radio is turned on, and the processes proceed to use the network. Although Figure 5 showed an average 9.5J cost to power up the radio, *netd* requires 125% of this level before turning the radio on, essentially mandating that applications have extra energy to transmit and receive subsequent packets. Therefore, the reserve does not empty to 0.

on and perform the system call, Cinder debits the reserve and permits the thread to proceed. The *netd* reserve is not subject to the system global half-life, as the process is trusted and only ever stores enough energy to activate the radio.

Cinder estimates the cost of radio access by tracking when network transmit and receive events occurred. For instance, if the radio has been idle for 20 seconds or more, threads wishing to use the network must contribute enough energy to turn the radio on and maintain the active power state until it again idles. Once the radio is on, however, additional operations are billed in proportion to the active period. That is, back-to-back actions are cheaper than ones with more delay between them because they extend the active period (delay the next idle period) less significantly.

For example, if the radio has been active for one second, it will automatically idle again 19 seconds later, so transmitting now only extends the active period by 1 second. However, if the radio is active but no packets have been sent or received for 15 seconds, transmitting now will extend the active period by an additional 15 seconds - a much costlier proposition.

Multiple applications can take advantage of the high initial cost/low amortized cost property by synchronizing their network accesses so as to effectively pool their energy resources. That is, they can do more work at lower cost in a manner reminiscent of low-power link layers [17]. We run two experiments: one with unrestricted energy consumption, and the other using Cinder's abstractions and modified *netd* stack. In both experiments, an RSS feed downloader starts with a poll interval of 60 seconds. Fifteen seconds later, a mail fetcher daemon starts, also with a 60 second poll interval. The second experiment uses the configuration shown in Figure 14. Both applications are provided enough energy to power up the radio every 60 seconds, if they work in unison.

Figure 15 shows the uncooperative applications wasting energy unnecessarily – each runs when the radio is idle and powers it up independently. Neither combine their efforts to amortize costs.

In comparison, Figure 16 shows what happens when the applications have constrained taps. Each application receives enough energy to activate the radio every two minutes. However, when they initiate network operations, their threads block in *netd* and contribute acquired energy to the radio activation pool (bottom of Figure 16). Every 60 seconds, enough energy has been saved to use the radio and both applications proceed simultaneously. This opportunistic schedule is similar to the low-power SP link layer [17], although Cinder is much more powerful: SP only seeks to minimize cost and does not actually control or limit energy consumption.

	Non-Coop	Coop	Improv
Total Time	1201s	1201s	N/A
Total Energy	1238J	1083J	12.5%
Active Time	949s	510s	46.3%
Active Energy	1064J	594J	44.2%

Table 1: Improvements in energy consumption and radio active state time using cooperative resource sharing in Cinder. Energy use due to the radio is significantly reduced, resulting in a 12.5% total system power reduction over the 20 minute experiment.

Figure 15 and the top of Figure 16 shows the independent and shared radio activation events. By supporting delegation, Cinder allows two independent applications to collaboratively improve quality of service by a factor of two while remaining in their respective power budgets.

This automatic collaboration doubles quality of service while actually spending less energy (each daemon would alone be only able to afford network access every 120 seconds). Table 1 shows the power savings of using Cinder's abstractions. In total, 12.5% less energy is used in the same time interval for an equivalent amount of work. While 12.5% is significant, we stress that in such background application examples, our baseline power consumption is artificially dominant as Cinder does nothing to place the hardware into lower power states while idle. We therefore expect Cinder to provide even greater improvement on a mature mobile platform that makes full use of the chipset's power savings features.

7 Related Work

We group related work into three categories: resource management, energy accounting, and energy efficiency.

7.1 Resource Management

Cinder's taps and reserves build on the abstraction of resource containers [5]. Like resource containers, they provide a platform for attributing resource consumption to a specific principal. By separating resource management into rates and quantities, however, Cinder allows applications to delegate with reservations yet reclaim unused resources. This separation also makes policy decisions much easier. As resource containers serve both as limits and reservations, hierarchical composition either requires a single policy (limit or reserve) or ad-hoc rules (a guaranteed CPU slot cannot be the child of a CPU usage limit).

Linux has recently incorporated "cgroups" [16] into the mainline kernel, which are similar to resource containers but group processes rather than threads. They are hierarchical and rely on "subsystem" modules which schedule particular resources (CPU time, CPU cores, memory).

ECOSystem [20, 21] 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 [5] by grouping related processes in the same container. This flat approach makes it impossible for an application to delegate, as it must either share its container with a child or put it in a new container that competes for resources. Like ECOSystem, Cinder estimates energy consumption with a software-based model that ties runtime power states to power draw.

7.2 Measurement, Modeling, and Accounting

Accurately estimating a device's energy consumption is an ongoing area of research. Early systems such as ECOSystem [20] use a simple linear combination of device states. Most modern phone operating systems, such as Symbian and OS X, follow this approach.

PowerScope improves CPU energy accuracy by correlating instrumented traces of basic blocks with program execution [11]. A more recent system, Koala, explores how modern architectures can have counter-intuitive energy/performance tradeoffs, presenting a model based on performance counters and other state [18]. A Koala-enabled system can use these estimates to specify a range of policies, including minimizing energy, maximizing performance, and minimizing the energy-delay product. The Mantis system achieves similar measurement accuracy to Koala using CPU performance counters [7].

Quanto [12] extends the TinyOS operating system to support fine-grained energy accounting across activities. Using a custom measurement circuit, Quanto generates an energy model of a device and its peripherals using a linear regression of power measurements. By monitoring the power state of each peripheral and dynamically tracking which activity is active, Quanto can give very precise breakdowns of where a device is spending energy.

Cinder complements this work on modeling and accounting. Improved hardware support to determine where energy is going would make its accounting and resource control more accurate. On top of these models, Cinder provides a pair of abstractions that allow applications and users to flexibly and easily enforce a range of policies.

7.3 Energy Efficiency

There is rich prior work on improving the energy efficiency of individual components, such as the voltage and frequency scaling a CPU [9, 13], spinning down disks [6, 14], or carefully selecting memory pages [15]. Phone operating systems today tend to depend on much simpler but still effective optimization schemes than in the research literature, such as hard timeouts for turning off devices. The exact models or mechanisms used for energy efficiency are orthogonal to Cinder: they allow applications to complete more work within a given power budget. The image viewer described in §5.4 is an example of an energy-adaptive application, as is typical in the Odyssey system [10].

8 Future Work

We believe that the reserve and tap abstractions may be fruitfully applied to other resource allocation problems beyond energy consumption. For instance, the high cost of mobile data plans makes network bits a precious resource. Applications, whether they be buggy, malicious, or simply poorly implemented, should not be able to run up a user's bill due to expensive data tariffs, just as they should not be able to run down the battery unexpectedly. Since data plans are frequently offered in terms of megabyte quotas, Cinder's mechanisms could be repurposed to limit application network access by replacing the logical battery with a pool of network bytes. Reserves could also act as quotas on text messages sent by a particular application.

Using the HTC Dream's limited battery level information Cinder could adapt its energy model based on past component and application usage, dynamically refining its costs. Though Cinder can easily facilitate this and we have made some adjustments to test this, evaluating the complex and dynamical system this would yield will require additional research.

Since we engineered the system, we did not find it unnatural to construct our examples and experiments using joule and watt units. However, we do not expect users and application developers to interact with the system in this way. By taking into account power model profiles for mobile devices, we expect to automatically derive the power and energy quantities necessary. Users should only need to to think in terms of minutes, network bytes, percentages of battery reserved, etc.

Finally, being a working Unix-like environment, Cinder is a candidate to run the Android platform in place of the Linux kernel. Since most of the facilities in Android are provided through the `dalvik` virtual machine Cinder would only have to satisfy a narrow interface to support the thousands of applications that consumers enjoy through Android Market.

9 Conclusion

Cinder is an operating system for modern mobile devices. It uses techniques similar to existing systems to model device energy use, while going beyond the capabilities of current operating systems by providing an IPC system that fundamentally accounts for resource usage on behalf of principals. It extends this accounting to add subdivision and delegation using its reserve and tap abstractions. We have described and applied this system to a variety of applications demonstrating, in particular, their ability to partition applications to energy bounds even with complex policies. Additionally, we showed Cinder facilitates policies which enable efficient use of expensive peripherals despite non-linear power models.

References

[1] The executive computer; compaq finally makes a laptop. `http://www.nytimes.com/1988/10/23/business/the-executive-computer-compaq-finally-makes-a-laptop.html`.

[2] Adobe and HTC Bring Flash Platform to Android, June 2009. `http://www.adobe.com/aboutadobe/pressroom/pressreleases/pdfs/200906/062409AdobeandHTC.pdf`.

[3] Apple Previews iPhone OS 4, Apr. 2010. `http://www.apple.com/pr/library/2010/04/08iphoneos.html`.

[4] Security and permissions, Apr. 2010. `http://developer.android.com/guide/topics/security/security.html`.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[6] F. Douglis, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, Berkeley, CA, USA, 1995. USENIX Association.

[7] D. Economou, S. Rivoire, and C. Kozyrakis. Full-system power analysis and modeling for server environments. In *In Workshop on Modeling Benchmarking and Simulation (MOBS)*, 2006.

[8] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management.

[9] K. Flautner and T. Mudge. Vertigo: automatic performance-setting for linux. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 105–116, New York, NY, USA, 2002. ACM Press.

[10] 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.

[11] 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.

[12] 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.

[13] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM Press.

[14] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *MobiCom '96: Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 130–142, New York, NY, USA, 1996. ACM Press.

[15] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 105–116, New York, NY, USA, 2000. ACM Press.

[16] 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`.

[17] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 76–89, New York, NY, USA, 2005. ACM Press.

[18] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: a platform for os-level power management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302, New York, NY, USA, 2009. ACM.

[19] 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.

[20] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 123–132, New York, NY, USA, 2002. ACM.

[21] 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.