Energy Management in Mobile Devices with the Cinder Operating System

Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, Nickolai Zeldovich†
Stanford University and MIT CSAIL†

Abstract

We argue that controlling energy allocation is an increasingly useful and important feature for operating systems, especially on mobile devices. We present two new low-level abstractions in the Cinder operating system, *reserves* and *taps*, which store and distribute energy for application use. We identify three key properties of control – *isolation*, *delegation*, and *subdivision* – and show how using these abstractions can achieve them. We also show how the architecture of the HiStar information-flow control kernel lends itself well to energy control. We prototype and evaluate Cinder on a popular smartphone, the Android G1.

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

General Terms Design

Keywords energy, mobile phones, power management

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 time-sharing systems (Linux/Unix).

Historically, operating systems have had poor energy management and accounting. This is not surprising, as their APIs standardized before energy was an issue. For example, the first commodity laptop with performance similar to a desktop, the Compaq SLT/286 [Com 1988], was released just one year before the C API POSIX standard. The resulting energy management limitations of POSIX have prompted a large body of research, ranging from CPU scheduling [Flautner 2002] to accounting [Zeng 2003] to offloading networking. Despite this work, current systems still provide little, if any, application control or feedback: users have some simple high-level sliders or toggles.

This limited control and visibility of energy is especially problematic for mobile phones, where energy and power define system lifetime. In the past decade, phones have evolved from low-function proprietary applications to robust multiprogrammed systems with applications from thousands of sources. Apple announced that as of April 2010 their App Store houses 185,000 apps [App 2010] for the iPhone with more than 4 billion application downloads. This shift away from single-vendor software to complex application platforms means that the phone's software must provide effective mechanisms to manage and control energy as a resource. Such control will be even more important as the danger grows from buggy or poorly designed applications to potentially malicious ones.

In the past year, mobile phone operating systems began providing better support for understanding system energy use. Android, for example, added a UI that estimates application energy consumption with system call and event instrumentation, such as processor scheduling and packet counts. This is a step forward, helping users understand the mysteries of mobile device lifetime. However, while Android provides improved *visibility* into system power use, it does not provide *control*. Outside of manually configuring applications and periodically checking battery use, today's systems cannot do something as simple as controlling email polling to ensure a full day of device use.

This paper presents Cinder, a new operating system designed for mobile phones and other energy-constrained computing devices. Cinder extends the HiStar secure kernel [Zeldovich 2006] to provide new abstractions for controlling and accounting for energy: reserves and taps. *Reserves* are a mechanism for resource delegation, providing fine-grained accounting and acting as an allotment from which applications draw resources. Where reserves describe a quantity of a resource, *taps* place rate limits on resources flowing between reserves. By connecting reserves to one another, taps allow resources to flow to applications. Taps and reserves compose

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

together to allow applications to express their intentions, enabling policy enforcement by the operating system.

Cinder estimates energy consumption using standard device-level accounting and modeling [Zeng 2002]. HiStar's explicit information flow control allows Cinder to track which parties are responsible for resource use, even across interprocess communication calls serviced in other address spaces. Without needing any additional state or support code, Cinder can accurately amortize costs across principals, such as the energy cost of turning on the radio to multiple applications that simultaneously need Internet access.

While Cinder runs on a variety of hardware platforms (AMD64, i386, ARM), the most notable is the HTC Dream, a.k.a. the Android G1. To the best of our knowledge, other than extensions to Linux, Cinder is the first research operating system that runs on a mobile phone. The reason for such a first is simple: the closed nature of phone platforms makes porting an operating system exceedingly difficult.

This paper makes three research contributions. First, it proposes reserves and taps as new operating system mechanisms for managing and controlling energy consumption. Second, it evaluates the effectiveness and power of these mechanisms in a variety of realistic and complex application scenarios running on a real mobile phone. Third, it describes experiences in writing a mobile phone operating system, outlining the challenges and impediments faced when conducting systems research on the dominant end-user computing platform of this decade.

2. A Case for Energy Control

This section motivates the need for low-level, fine-grained energy control in a mobile device operating system. It starts by reviewing some of the prior work on energy visibility and the few examples of coarse energy control. Using several application examples as motivation, it describes three mechanisms an OS needs to provide for energy: isolation, delegation, and subdivision. The next section describes reserves and taps, abstractions which provide these mechanisms at a fine granularity.

2.1 Prior Work on Visibility and Control

Managing energy requires accurately measuring its consumption. A great deal of prior work has examined this problem for mobile systems, including ECOSystem [Zeng 2002], Currentcy [Zeng 2003], PowerScope [Flinn 1999b], and PowerBooter [Zhang 2010]. These systems use a model of the power draw of hardware components based on hardware states. For example, an 802.11b card draws only slightly more power while transmitting than receiving, whereas a CPU's power draw increases with utilization. Current mobile phone energy accounting systems, such as Android's, use this approach. Cinder also does as well; Section 4 provides the details.

Early systems like ECOSystem [Zeng 2002] proposed mechanisms by which a user could control per-application energy expenditure. ECOSystem, in particular, introduced an abstraction called Currentcy, which gives an application the ability to spend a certain amount of energy, up to a fixed cap. This flat hierarchy of energy principals – applications – is reasonable for simple large applications. Mobile applications and systems today, however, are far more complex and involve multiple principals. For example, web browsers run active code as well as possibly untrusted plugins, network daemons control access to the cellular data network, and peripherals have complex energy profiles.

2.2 Isolation, Delegation, and Subdivision

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

The first mechanism is isolation. Isolation is a fundamental part of an operating system. Memory and inter-process communication (IPC) isolation provide security, while CPU and disk space isolation ensure that processes cannot starve others. Isolating energy consumption is similarly important. An application should not be permitted to consume inordinate amounts of energy, nor should it be able to deprive other applications. 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 additional processes, these children must not be able to consume the energy of the other.

The second mechanism is delegation. Delegation allows a principal to loan any of its available energy and power to another principal. After delegation, either the resource donor or the recipient can freely consume the delegated resources. Furthermore, if there are multiple donors delegating to this recipient, the resources are pooled for use by the recipient. Resource delegation is an important enabler of inter-application cooperation. For example, the Cinder *netd* networking stack 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 may not only improve quality of service, but even reduce energy consumption.

The third mechanism is subdivision. Subdivision allows applications to partition their available energy. Combined with isolation, subdivision allows an application to give another principal a partial share of its energy, while being assured that sure that the rest will remain for its own use. For example, modern web browsers commonly run plugins, some of which may even be untrusted. If a browser is granted a finite amount of power, it might want to protect itself from buggy or poorly written plugins that could waste CPU en-

ergy. Subdivision lets the browser give full control over a fraction of its energy allotment to plugins. Isolation further ensures that each plugin component does not consume more than its share.

2.3 Prior Systems

Prior systems like ECOSystem [Zeng 2002, 2003] only partially support isolation and subdivision: child processes share the resources of their parent. This is sufficient when applications are static entities, but not when they spawn new processes and invoke complex services. The web browser demonstrates the problem: it has no way to prevent its plugins from consuming its own resources once they are spawned. Cinder's subdivision lends naturally to familiar and standard abstractions such as process trees, resource containers, and quotas.

Furthermore, prior systems do not permit delegation, which is akin to priority inheritance. For always-on systems which have small variations in power draw, such as the laptops for which they were designed, this is not a serious limitation. On mobile phones, however, which have almost two orders of magnitude difference in active and sleep power, the cost of powering up peripherals, such as the wireless data interface, can be significant. Delegation provides a means to facilitate application cooperation.

3. Design

Cinder is based on HiStar [Zeldovich 2006], a secure operating system built upon information flow control. Cinder adds two new fundamental kernel object types: *reserves* and *taps*. This section gives a brief overview of HiStar and key features related to resource management, describes reserves and taps, gives examples of how they can be used, and details how they are secured.

3.1 HiStar

HiStar is composed of six first-class kernel objects, all protected by a security *label*. Its segments, threads, address spaces, and devices are similar to those of conventional kernels. *Containers* enable 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 offset in another; they are the basis for all IPC.

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 threads from performing actions for which their reserves do not have sufficient resources. Reserves, like all other kernel objects, are protected by a security label (§3.5) that controls which threads can observe, use, and manipulate it.

All threads draw from one or more energy reserves. Cinder's CPU scheduler is energy-aware and allows a thread to run only when at least one of its energy reserves is not empty. Threads that have depleted their energy reserves cannot run. Tying energy reserves to the scheduler prevents new spending, which is sufficient to throttle energy consumption.

Reserves allow threads to delegate and subdivide resources. As a simple example, an application granted 1000 mJ of energy can subdivide its reserve into an 800 mJ and a 200 mJ reserve, allowing another thread to connect to the 200 mJ reserve. However, threads rarely manage energy in such concrete quantities, preferring instead to use taps (§3.3). A thread can also perform a reserve-to-reserve transfer provided it is permitted to modify both reserves.

Reserves also provide accounting by tracking application resource consumption. Applications may access this accounting information in order to provide energy-aware features. Finally, reserves can be deleted directly or indirectly when some ancestor of their container is deleted, just as a file can be deleted either directly or indirectly when a directory containing it is deleted in a Unix system.

3.3 Taps

A tap transfers a fixed quantity of resources between two reserves per unit time, which controls the maximum rate at which a resource can be consumed. For example, an application reserve may be connected to the system battery via a tap supplying 1 mJ/s (1 mW).

Taps aid in subdividing resources between applications since partitioning fixed quantities is impractical for most policies. A user may want her phone to last at least 5 hours if she is surfing the web; the amount of energy the browser should receive is relative to the length of time it is used. Providing resources as a rate naturally addresses this.

Another approach, which Cinder does not take, would be to implement transfer rates between reserves through threads that explicitly move resources and enforce rate-limiting as well as accounting. Given five applications, each to be limited to consume an average of 1 W, the system could create five application reserves and threads, with each thread transferring while tracking and limiting energy into each of these applications' reserves. However, this fine-grained control would cause a proliferation of these special-purpose threads, adding overhead and decreasing energy efficiency.

Taps are made up of 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 (§3.5). Conceptually, it is an efficient, special-purpose thread whose only job is to transfer energy between reserves. In practice, transfers are executed in batch periodically to minimize scheduling and context-switch overheads.

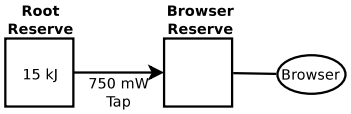

Figure 1. A 15 kJ battery, or *root reserve*, connected to a reserve via a tap. The battery is protected from being misused by the web browser. The web browser draws energy from an isolated reserve which is fed by a 750 mW tap.

3.4 Resource Consumption Graph

Reserves and taps form a directed graph of resource consumption rights. The root of the graph is a reserve representing the system battery; all other reserves are a subdivision of this root reserve. Figure 1 shows a simple example of a web browser whose consumption is rate limited using a tap. The tap guarantees that even if the browser is aggressively using energy the battery will last at least 5 hours (15,000 J at 0.750 J/s is about 5.6 hours).

3.5 Access Control & Security

Any thread can create and share reserves or taps to subdivide and delegate its resources. This ability introduces a problem of fine-grained access control. To solve this, reserves and taps are protected by a security label, like all other kernel objects. The label describes the privileges needed to observe, modify, and use the reserve or tap.

Using resources from a reserve requires both observe and modify privileges: observe because failed consumption indicates the reserve level (zero) and modify for when consumption succeeds. Since a tap actively moves resources between a source and sink reserve, it needs privileges to observe and modify both reserve levels; to aid with this, taps can have privileges embedded in them.

4. Cinder on the HTC Dream

Controlling energy requires measuring or estimating its consumption. This section describes Cinder's implementation and its energy model. The Cinder kernel runs on AMD64, i386, and ARM architectures. All source code is freely available under open-source licenses. Our principal experimental platform is the HTC Dream (Google G1), a modern smartphone based on the Qualcomm MSM7201A chipset.

4.1 Energy accounting

Energy accounting on the HTC Dream is difficult due to the closed nature of its hardware. It has a two-processor design, as shown in Figure 2. The operating system and applications run on an ARM11 processor. A secure, closed ARM9 coprocessor manages the most energy hungry, dynamic, and informative components (e.g. GPS, radio, and battery sensors). The ARM9, for example, exposes the battery level as an integer from 0 to 100.

Recent work on processors has shown that fine-grained performance counters can enable accurate energy estimates

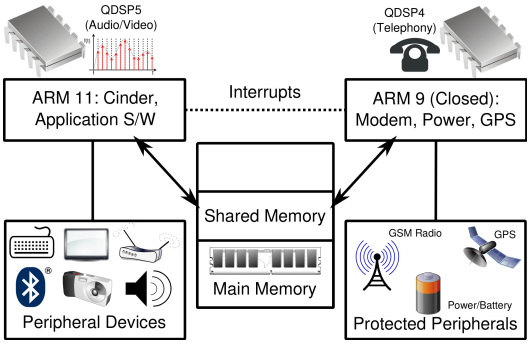

Figure 2. The two ARM cores in the MSM7201A chipset. Cinder runs on the ARM11, whereas the ARM9 controls access to sensitive hardware including the radio and GPS. The two communicate via shared memory and interrupt lines.

within a few percent [Economou 2006; Snowdon 2009]. Without access to such state in the HTC Dream, however, Cinder relies on the simpler well-tested technique of building a model from offline-measurements of device power states in a controlled setting [Flinn 1999b; Fonseca 2008; Zeng 2002]. Phones today use this approach, and so Cinder has equivalent accuracy to commodity systems.

4.2 Power Model

Our energy model uses device states and their duration to estimate energy consumption. We measured the Dream's energy consumption during various states and operations. All measurements were taken using an Agilent Technologies E3644A, a DC power supply 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.

While idling in Cinder, the Dream uses about 699 mW and another 555 mW when the backlight is on. Spinning the CPU increases consumption by 137 mW. Memory-intensive instruction streams increase CPU power draw by 13% over a simple arithmetic loop. However, the HTC Dream does not have hardware support to estimate what percentage of instructions are memory accesses. 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 and assumes the worst case power draw (all memory intensive operations).

4.3 Peripheral Power

The baseline cost of activating the radio is exceptionally high: small isolated transfers are about 1000 times more expensive, per byte, than large transfers. Figure 3 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

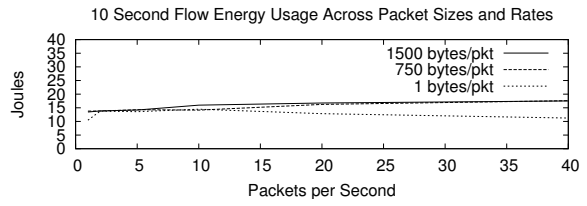

Figure 3. Radio data path power consumption for 10 second flows across six different packet rates and three packet sizes. Short flows are dominated by the 9.5 J baseline cost shown in Figure 4. For this simple static test, data rate has only a small effect on the total energy consumption. The average cost is 14.3 J (minimum: 10.5, maximum: 17.6).

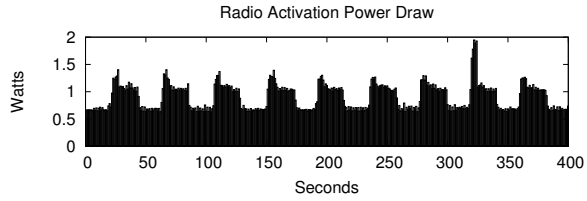

Figure 4. Cost of transitioning from the lowest radio power state to active. One UDP packet is transmitted approximately every 40 seconds to enable the radio. The device fully sleeps after 20 seconds, but the average plateau consumes an additional 9.5 J of energy over baseline (minimum 8.8 J, maximum 11.9 J). Power consumption for a stationary device can often be predicted with reasonable accuracy, but outliers, such as the penultimate transition, occur unpredictably.

power cost for flows lasting less than 10 seconds in duration, regardless of the bitrate.

Figure 4 shows this activation cost. An application powers up the radio by sending a single 1-byte UDP packet. The secure ARM9 automatically returns to a low power mode after 20 seconds of inactivity. Because the ARM9 is closed, Cinder cannot change this inactivity timeout.

With this workload, it costs 9.5 joules to send a single byte! One lesson from this is that coordinating applications to amortize energy start-up costs could greatly improve energy efficiency. In §5.5 we demonstrate how Cinder can use reserves and taps for exactly this purpose.

4.4 Mobility & Power Model Improvements

Cinder's aim is to leverage advances in energy accounting (see §8.2) to allow users and applications to provision and manage their limited budgets. Accurate energy accounting is an orthogonal and active area of research. Cinder is adaptable and can take advantage of new accounting techniques or information exposed by device manufacturers.

```
// 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 5. `energywrap` excerpt without error handling.

5. Applications

To gain experience with Cinder's abstractions, we developed applications using reserves and taps. This section describes these applications, including a command-line utility that augments existing applications with energy policies, an energy constrained web browser that further isolates itself from its browser plugins, and a task manager application that limits energy consumption of background applications.

5.1 `energywrap`

Taking advantage of the composability of Cinder's resource graph, the `energywrap` utility allows any application to be 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 executes the specified program. This allows even energy-unaware applications to be augmented with energy policies.

The sandboxing policy provided by `energywrap` is implemented in about 100 lines of C++. An excerpt is shown in Figure 5. HiStar provides a `wrap` utility designed to isolate applications with respect to privileges and storage resources. Coupling this utility with `energywrap` allows any application or user to provide a virtualized environment to any thread or application. Section 6.1 evaluates the effectiveness of energy sandboxing and isolation.

`energywrap` has proved useful in implementing policies while designing and testing Cinder, particularly for legacy applications that have no notion of reserves or taps. Since `energywrap` runs an arbitrary executable, it is possible to use `energywrap` to wrap itself or shell scripts, which may invoke `energywrap` with other scripts or applications. This

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.