End-to-End Energy Management of Mobile Devices

Ranveer Chandra†, Omid Fatemieh*, Parya Moinzadeh‡
Chandramohan A. Thekkath†, Yinglian Xie†
†Microsoft Research
*Microsoft Corporation
‡University of Illinois at Urbana-Champaign

ABSTRACT

Energy drain in mobile devices is well recognized to be a serious problem. Most mobile operating systems provide facilities to mitigate energy drain, but they are usually heavy handed and often require shutting down an offending application or uninstalling it. In this paper, we describe an alternative that controls the behavior of an energy hungry application rather than kill it. Our system offers a finer-grained approach to energy drain and is cognizant of specific application energy characteristics as well as interactions amongst multiple applications that can affect energy drain in unexpected ways. Using our system, we believe users can avoid the annoyance of sudden and unexpected battery loss, particularly when operating in standby mode.

Our system, called E-Loupe, consists of components running on each device as well as in a centralized data center. E-Loupe gathers per-device data, analyzes this data, and implements *energy sandboxing* in the device kernel to control how often an energy hungry application is run and what resources it is allowed to consume. Our experimental results on data from over 73,000 users shows that we can identify the cause of more than 85% of the energy spikes, and upper bound the energy drain in nearly all of these cases.

1. INTRODUCTION

Mobile devices such as phones and tablets are now commonplace. At the same time, there is an abundant supply of applications ("apps") for these devices that are viewed as practically indispensable by their users. Many of these apps are sophisticated, and like applications that execute on PCs, make non-trivial demands on the host device's resources. One such resource is the charge in the device battery, which unlike its PC counterpart, is quite limited and therefore a significant resource that must be husbanded.

Much prior work on energy management for mobile devices has focused on energy savings at the lowest level of the system, e.g., the radio, or the CPU, or the system-on-a-chip of the mobile device. These efforts are indeed very important, but they provide only limited help to app developers in making their apps more energy efficient or to users in avoiding energy-hungry apps. In fact, manufacturers of mobile devices complain that their hardware and low-level systems are energy efficient and that the apps are the culprits [5, 4]. However, characterizing the end-to-end energy used by an app is difficult for multiple reasons.

First, and perhaps surprisingly, the straightforward approach of estimating an app's energy by executing it in isolation and measuring its energy usage does not work well. This is primarily because covering all usage and execution scenarios is non-trivial due to the following:

- Configuration. For example, one instance of an app may be configured to poll the network more frequently than another instance.
- User-specific variation. For instance, one user of a social media app may have many more contacts than another that must be updated by the app.
- Concurrent apps. An an example, two apps that simultaneously use a resource like the GPS can consume less power than if they ran separately because of the fixed cost of powering up the GPS is not amortized.

Second, estimating energy usage of an app by measuring resource (e.g., CPU, display, disk, network etc.,) usage does not work satisfactorily because high energy consumption does not necessarily imply high resource usage. We demonstrate this effect using energy and resource usage data that we sampled from a large collection of mobile devices (Table 1) . Figure 1 shows the cumulative distribution (CDF) of our samples versus the average power. We plot two separate CDFs: one for samples with high resource usage (network, disk and CPU usage above median) and one for those with low resource usage (network, disk, and CPU usage

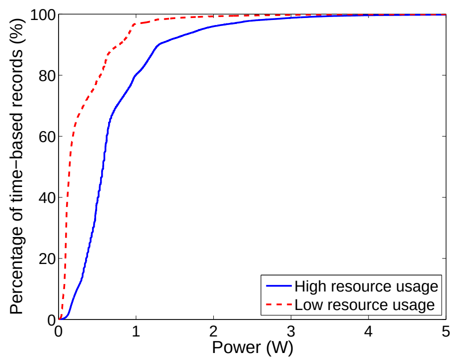

Figure 1: Cumulative distributions of samples with high resource consumption and low resource consumption. Values greater than 0.4W denote high energy drain for tablet device with a 42Wh battery.

below median). The CDF for the low resource samples indicate that over 30% of these consume high power. Likewise, the CDF for the high resource samples shows that 42% of these, in fact, consume low power. If resource usage were a good indicator of energy drain, we would expect almost all of the high resource records to show large average power (and mutatis mutandis for low resource records).

Finally, we must account for both foreground and background jobs. Mobile platforms run a foreground task that uses the screen and interacts with the user, as well as multiple background tasks that do not interact with the user, but consume a significant fraction of the total energy. Energy profiling background tasks is prohibitively expensive and as far as we are aware, existing systems only measure the foreground task and do not fully account for the energy drain. Instead, existing operating systems restrict the set of processes that are allowed to run in the background(iOS, Android, and Windows) or limit the amount of resources these background jobs can use (iOS and Windows). Yet, despite these approaches, these systems can incur significant energy drain when there are background jobs because the restrictions imposed by the OS are only indirectly addressing the problem of energy drain.

The goal of our system, called E-Loupe, is to provide fine-grained, on-demand control over energy hungry apps while overcoming the challenges mentioned above. It effectively characterizes the energy consumed by apps and uses a technique, *energy sandboxing*, to control when energy hungry apps are scheduled, and how much resources they are allowed to consume. This is unlike current mobile operating systems that provide relatively coarse-grained control over apps [1, 2]. For example, in most cases, energy hungry apps are typically killed, rather than controlled, to reclaim resources. A novel aspect of our approach is that it can account for multi-app interactions. Another attractive feature is that it offloads some of the work to each mobile device and thus accommodates scale. It is important to note that our system does not per se. improve the energy consumption of a specific app (the app must consume a certain quantity of energy to perform a certain task), but we reduce the rate of energy consumption and prolong battery life to meet a targeted standby battery lifetime. This feature is instrumental in avoiding sudden and unexpected battery loss, particularly because of background processes.

E-Loupe has three components

1. The first component runs on each user's device and collects energy and resource usage samples. Our current population has about 73,000 users. The collected data is uploaded to a service in a datacenter.
2. A second component running in the datacenter uses statistical inferencing techniques to diagnose the cause of an energy anomaly, and determines kernel policies to isolate and contain it. As others have independently observed [14], statistical analyses of energy data from a large population yields signficant insights.
3. A final component, running in the device kernel implements the policies to limit the impact of the errant app by restricting its resource usage to meet a desired energy goal.

We implemented our system for Windows 8 mobile devices. Using data collected from over 73,000 devices, we show that E-Loupe can successfully isolate the causes of high energy drain for 87.5-92.3% reported energy spikes. The energy sandboxing mechanism is able to reduce the average power consumption by 5-6 times from the peak. Finally, our trace-based simulation suggests that E-Loupe can reduce the energy discharge rate by 3 times for real user reported energy spikes.

2. RELATED WORK

E-Loupe manages the end-to-end energy consumption of mobile devices – from monitoring their energy consumption, to isolating the culprit processes, and when possible, recovering from anomalous energy behavior (or *energy spike*). To the best of our knowledge, prior work has only looked at subsets of the problem without an obvious way to combine them.

A common technique to identify energy hungry processes is resource-based modeling [13, 15, 16, 21]. The system monitors the resources (CPU, display, network, disk, etc.,) used by the process, and then applies per-resource models, e.g. CPU [12], network [17, 8], display [10], to additively determine the total energy consumed by the process. This technique has also been used in commercial products, such as BatterBatteryStats for Android [6].

Although simple, this technique cannot always identify the culprit process(es). First, resource models are only as accurate as the granularity of monitoring. Although one could monitor CPU DVFS states every few milliseconds, and the display pixels 60 times a second, such a monitoring framework will itself consume significant energy. In contrast, measuring resource consumption at a coarser granularity leads to false negatives (Figure 1). Second, developing models for all system components, under all possible operating conditions is difficult. For example, the same network load will draw different amount of energy depending on the cellular chipset and signal strength [13, 21]. Third, when resources are shared, e.g. a shared GPS, or a common service is used in background mode (`BroadcastReceiver` in Android, or the `svchost` process in Windows), per-process resource consumption does not capture the energy consumed by other processes on its behalf.

An alternative to computing energy using resource-based modeling is to do the reverse: measure energy consumption and attribute it, using statistical techniques, to the active processes. Carat [14], a concurrent work, applies statistical techniques on energy data collected across several devices to identify the faulty app. However, Carat is unable to detect cases where a cluster of processes cause the energy spike, which as we show is common in screen-off mode. (Section 7).

In fact, E-Loupe goes further, and tries to recover from the energy spike. Modern OSes use several techniques to reduce the energy consumed by each component, such as CPU DVFS, shutting off cores, memory frequency scaling, display brightness, Wi-Fi power save mode, 3G fast dormancy, and many others. They can also be configured to implement power policies based on the battery level. However, these preventive actions are unable to contain high energy drain. The OS is unable to react since it cannot attribute energy drain to a faulting process (or processes). The only work we are aware of that enables per-process energy accounting is Cinder [18]. However, since this would require an OS redesign and a knowledge of existing app energy requirements, we instead propose a new concept, called energy sandboxing, that requires small changes to the OS, and can yet upper bound the amount of energy the entire system drains in screen off mode.

3. SYSTEM OVERVIEW

E-Loupe consists of two privileged components that run on the client device and a server-side component that runs in a datacenter. A lightweight client-side data logger gathers coarse-grained energy consumption data samples from the mobile device, batches them, and uploads them to the server whenever it has access to Wi-Fi. This datalogger also feeds a client-side module that constructs a model for energy consumption that is specific to the device. A server-side module stores the uploaded data, processes them to detect energy spikes, and helps infer the cause of energy drain. This information is sent back to the client, where a client-side module calls into our kernel API to effect resource and energy sandboxing.

In this section we give an overall architecture of our system. The two sections that follow describe the functioning of our system in greater detail.

Data Logger

To minimize overhead, the E-Loupe data logger typically gathers only coarse-grained data (on the order of tens of minutes). We refer to these as *long-term* reports. However, when the logger detects that energy discharge rate exceeds a threshold, currently set at 0.4W (Figure 8), it transmits a set of special *short-term* reports, which occur on the order of minutes.

Both data reports are created by sampling the "fuel gauge" on the battery systems. We do not collect finer-grained data because the fuel gauges on existing systems does not report accurate battery drain over short periods. We are actively considering logging more fine-grained information while maintaining the lightweight nature of the data logger. However, in this paper we show that such coarse-grained information is enough to get us most of the way in energy diagnosis and recovery.

Both long-term and short-term records consist of the following fields

- `timestamp:` The end time of the interval.
- `process list:` List of processes that were active in the sample interval (system and non-system processes). We focus on processes rather than apps because in many cases, on nearly all mobile platforms, the system processes perform tasks on behalf of the apps, particularly when the apps are in the background. Our goal is to learn relationships between these processes, and attribute energy drain to the appropriate process, or group of processes, so

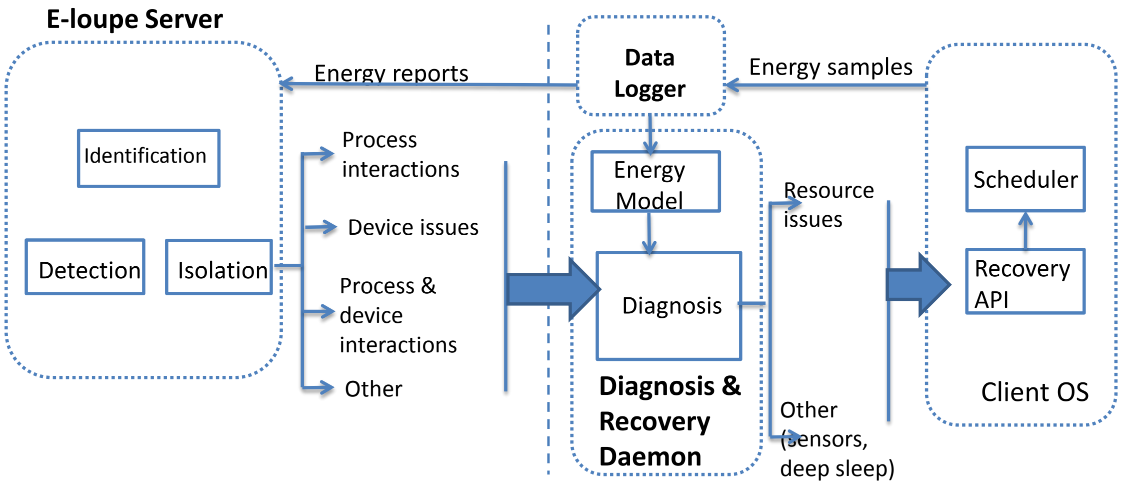

Figure 2: The system architecture

that we can be specific in our recovery actions.

- `energy`: Energy drained during this interval, retrieved by polling the battery fuel gauge. This is a cumulative figure attributed to all the processes in the list above. It does not measure instantaneous power, only the energy e_i drained during each internal Δt_i denoted by the ith report. The energy discharge rate $e_i/\Delta t_i$ has the same unit as power, and we refer to this as "power" in the sequel.
- `cpu usage`: For each process, the foreground and background CPU cycles used by each process during the interval.
- `disk usage`: For each process, the number of disk reads, writes, and flushes, in terms of bytes, during the interval.
- `network activity`: For each process, the number of bytes sent and received on each network interface during the sample interval.

Server

This service runs in the datacenter and stores *energy reports* from the data logger in a database. The server process statistically analyzes these records for energy anomalies.

The power we compute based on energy and time is the mean of the instantaneous power for a given duration. With a large number of samples drawn from random i.i.d variables, we can therefore use a normal distribution to approximate the distribution of the computed power values, in accordance with the Central Limit Theorem [20]. Likewise, whenever we have sufficient samples from a specific device, we also use a normal distribution to approximate the power distributions from that device.

Using our samples, the server calculate two metrics that are useful for users and publishes on a Web site. The first, called the *Consumed Power* is a per-process indication of how much power the device consumes when that process is active. The second metric, called the *Occurrence Frequency* is a per-process indication of how often a process is likely to be active if it is installed. These metrics are useful in two ways: first, they provide a ranking of apps and second, they also help in troubleshooting energy spikes. In the interest of space we do not describe these further in this paper.

Detecting Energy Spikes

When the data logger transmits a set of short-term reports to indicate the potential for an energy spike, the server analyzes the data to confirm it.

The analysis is fairly straight forward: Assuming that the average power follows a normal distribution $\mathcal{N}(\mu, \sigma^2)$ (with μ and σ^2 estimated from data), then for each report with power P_i, E-Loupe computes the probability of P_i coming from $\mathcal{N}(\mu, \sigma^2)$. If this probability is smaller than a predefined threshold (e.g., 0.05 in our experiment), E-Loupe marks this event as an energy spike.

Isolating the Cause of Energy Spikes

When an energy spike is confirmed, the isolation step addresses *what* caused it. The server categorizes the causes of an energy spike: e.g., is this a problem specific to the device, or is it because there is a problem with a specific process, or is a group of processes responsible for it. We describe this approach in greater detail in Section 4. After isolating the potential cause, the server communicates this information back to the client.

We use a centralized service to detect and isolate the cause of an energy spike rather than doing this on a client because it allows us to aggregate energy records from a very large ensemble and use statistically significant techniques. As a simple example, an energy anomaly that persistently occurs only on a specific device would be hard to identify as such if the analysis were done only at the device.

Client Diagnostic & Recovery Daemon

When a client receives feedback from the server about the cause of an energy spike, it further diagnoses the cause and effects recovery from the spike where possible. We describe the details in Section 5.

Diagnosis of Energy Spikes

The diagnosis function answers *why* the energy spike occurred. Based on server inputs and a locally derived energy model (described in Section 5.1), the client determines if the energy spike is caused by (i) high resource consumption (CPU, network, disk), or (ii) if the device is unable to enter low power state, or (iii) if the lowest power state itself consumes excessive power, a situation that commonly arises when there are peripherals attached to the device.

Recovery

The result of the diagnosis is used by the daemon to invoke a set of OS APIs. This allows the OS to either slow down the entire system, or slow down the resource consumption of the relevant processes, so as to contain the spikes in energy consumption. We describe this technique in detail in Section 5.2. Our technique has the desirable property that the faulty processes don't necessarily have to be killed.

Users need not be involved during data collection and diagnosis, except in the recovery step. Depending on the diagnosis results, if a recovery policy must be applied to a particular app, a user will be prompted to enable this "energy-aware" OS policy so that E-Loupe does not violate application semantics.

4. ISOLATING THE CAUSE OF AN ENERGY SPIKE

The server classifies the cause of an energy spike into four categories: process issues or the interactions among different processes, device issues, the specific combinations of processes and devices, and others. E-Loupe tests each factor separately in sequential order, as explained below.

A typical mobile platform has three class of processes running on it: ordinary processes, worker processes, and system processes. Worker processes are started by either ordinary or systems processes to do work on their behalf, and are often shared amongst multiple ordinary processes. System processes are OS daemons that execute on the device on behalf of the OS and consume some power in all energy records. Thus any energy attribution scheme must be cognizant of the interactions between these process types. We exclude system processes from our analysis of causes and only consider individual ordinary and worker processes and their mutual interactions.

Individual Processes and Mutual Process Interactions

The first step is to identify individual ordinary processes, since limiting a specific process is less likely to have an adverse impact on the user than limiting the whole device. However, achieving this is not straightforward due to two factors. The first is coverage. Not all energy spikes are caused by a small number of pre-classified energy-hungry processes. Second, the presence of worker processes complicate matters. A process may cause energy spikes only when it is performing a subset of tasks, often with the help of worker processes. For example, `svchost.exe` is such a worker process that does work on behalf of others.

E-Loupe examines all the processes that are present when an energy spike occurs, excluding system processes. For each process j, we categorize the energy reports into two classes: one that contains j, denoted by D_j, and one without j, denoted by $D_{\bar{j}}$, across all devices and over time. We use two normal distributions $\mathcal{N}_j(\mu_1, \sigma_1^2)$ and $\mathcal{N}_{\bar{j}}(\mu_2, \sigma_2^2)$ to approximate D_j and $D_{\bar{j}}$, and estimate the means and the variances from data.

We adopt Student's t-test to determine whether D_j and $D_{\bar{j}}$, with unequal sizes and unequal variances, are drawn from the same distribution, with 95% confidence error bounds. If the two distributions are different and $\mu_1 - \mu_2 \geq \theta$, then E-Loupe outputs process j as a possible suspect. Here θ is an adjustable threshold based on the desired granularity of isolation, and we set θ to 10% of the normal platform power in our current implementation.

If a non-worker process is identified in this step, then we know that this process alone will trigger higher energy consumption. So E-Loupe does not further consider its interactions with other processes.

If instead, a worker process W is identified in this step, then we need to identify the process (typically an ordinary process) that W is working for. Without application semantics, we use a heuristic to infer the process that may have triggered W.

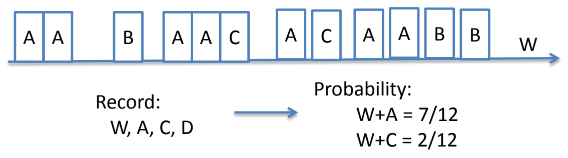

Figure 3: An example to identify energy-hungry processes that triggered a helper process.

The idea is that if a process j often relies on W to perform work, then j and W are likely to co-occur frequently. Thus, we consider all ordinary processes in the report and examine their co-occurrences with W in history. We then pick the top processes that co-occurred most frequently with W as candidate factors, each with a probability. Figure 3 illustrates this inference process. In this example, the worker process W can co-occur with process A, B, or C. Only A and C occurred in the report. So we compute the probability of W co-occurring with A and C, respectively. According to the probability, A is the most likely real factor behind W, but C is also a possible candidate. Picking the co-occuring processes can be time-consuming if the history is long. To avoid long histories, the server periodically summarizes and caches the results in the background.

If no individual process is identified as a cause by itself, we examine all process pairs (j,k), where at least one of j or k is an ordinary process. For each such pair treated as a unit, we apply Student's t-test as above to compare the two sets of powers with and without the pair. Notice that we need $O(N^2)$ such tests. N is on the order of 10 and the tests can be done quite efficiently in these cases.

Device Issues

After excluding process and process- interaction issues, E-Loupe proceeds to check whether an energy spike has a device specific cause.

If so, we expect to observe consistent high average power from this device d, modeled as a normal distribution $\mathcal{N}_d(\mu_d, \sigma_d^2)$ with sufficiently many data points. E-Loupe then tests whether the energy spike with reported power P_i is just a normal variation that fits into $\mathcal{N}_d(\mu_d, \sigma_d^2)$, based on estimated μ_d and σ_d. If so, we classify the cause as device specific.

Process and Device Interactions

In some cases, a process does not generally consume much energy, but becomes energy-hungry when running on certain devices, either because of misconfigurations, or because the specific usage pattern triggers a bug. Therefore, for all non-system processes[1] that are not isolated as energy-hungry in the reported spike, E-Loupe additionally examines the interactions between the process and the device, using Student's t-test in a similar way.

Others

If none of the above steps identifies a likely cause, E-Loupe classifies the issues as the "other" category. For example, an energy spike could just be caused by a large number of concurrently running processes, causing high aggregated resource consumptions. We leave this case to the diagnosis process for further analysis in future work.

5. DIAGNOSIS AND RECOVERY

On receiving the likely reason from the E-Loupe server, the client daemon: (i) diagnoses the cause of the energy spike as either a resource or device issue, and (ii) accordingly attempts to recover and improve energy efficiency.

If the energy spike happened while the screen is on, it is difficult to recover since the user is often actively interacting with the mobile device, and recovery is likely to adversely affect user experience. For example, a CPU-intensive game may legitimately use high energy, and attempts to restrict its CPU usage (to save energy) will have unintended consequences. Thus, in cases when the screen is on, if the high energy drain is caused by a foreground process, this information is logged for the user to revisit at a later point in time. When perceived to be effective, we provide generic suggestions to the user, such as dimming the display and closing unused open applications. In addition, if an app is using substantial network resources, we recommend switching to Wi-Fi instead of the cellular network.

We therefore focus our diagnosis and recovery activities when the screen is off, where background processes can consume significant amount of energy that can materially affect battery life.

Mobile operating systems try to maximize energy usage. Typically, when the screen is off and there are no active processes, the operating system takes steps to maximize the mobile device's standby lifetime [2, 1, 3]. In particular, it puts the device in the lowest power state, where it consumes very low power (called the *power floor*, P_l). Ideally the device stays in the lowest power state whenever the screen is off. However, in order to support background tasks, the device needs to frequently come out of the lowest power state (see Figure 4).

[1]If a system process has this issue, then we expect to output device issues in the previous step, since system processes tend to trigger persistent energy spikes.

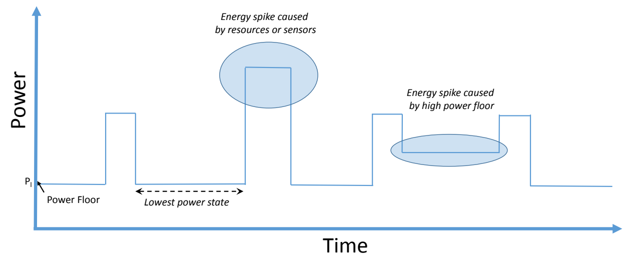

Figure 4: An example of the system power consumption when the screen is off.

When there is an energy spike, as shown in Figure 4, it is usually caused by (i) background apps that either consume too many resources, (ii) apps that retain locks on resources that prevents the device from entering the lowest power state, and/or (iii) the power floor itself is high because of a peripheral, such as a headset or keyboard.

5.1 Diagnosis

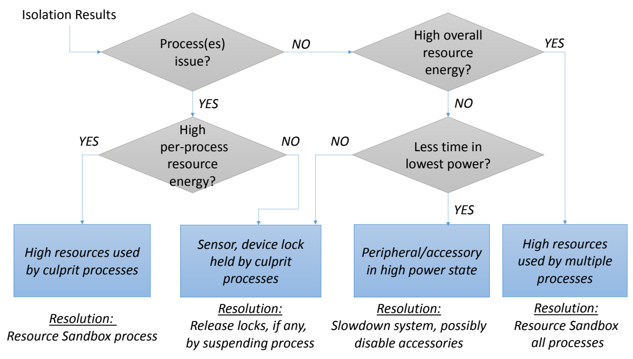
Isolation Results
Process(es) issue?
NO
High overall resource energy?
YES
YES
NO
High per-process resource energy?
YES
NO
NO
Less time in lowest power?
YES
High resources used by culprit processes
Sensor, device lock held by culprit processes
Peripheral/accessory in high power state
High resources used by multiple processes
Resolution: Resource Sandbox process
Resolution: Release locks, if any, by suspending process
Resolution: Slowdown system, possibly disable accessories
Resolution: Resource Sandbox all processes

Figure 5: The decision tree used by the E-Loupe diagnosis routine.

Figure 5 presents a simple flow chart that E-Loupe uses to diagnose an energy spike. Our basic strategy is to resolve energy spikes through a recovery process (described later) wherever possible, and if that is not possible, to mitigate the effect.

If results of the isolation step from the server indicate that one or more processes are responsible for the spike, we first try to ascertain if these processes consume excessive resources. The rationale for this is that it is relatively easy to constrain high resource usage. To identify excessive energy due to resource usage, we use the resource utilization figures from the energy report to determine if they would have resulted in high energy usage. In order to make this determination, we need a model of energy usage. Fortunately, we do not need a highly accurate model for our purposes because prior work indicates that accurate models for the energy consumed by CPU, network, and disk are difficult to construct(Section 2). If we are not able to account for energy due to resource usage, we categorize the problem as a device specific problem, where either a process is holding on to a lock, or peripherals, or the device is not in the lowest power state.

We use an approximate model, tailored for E-Loupe's recovery module, where we only need to model the worst case energy that could have been consumed by CPU, network, and disk. For this purpose, we use a simple mathematical model, where the energy E for a given duration is computed as: $E = E_l + E_r$. Here E_l is the energy that a device spends in the lowest power state, and E_r is the energy caused by active computation. Specifically, $E_l = \Delta t_l * P_l$, where Δt_l is the duration of the lowest power-state, and P_l refers to the power floor for this case. The energy used by active computation E_r can be computed as

$$E_r = \Delta t_r * a + b * \text{cpu} + c * \text{disk} + d * \text{network}$$

where Δt_r is the active computation time and $\Delta t = \Delta t_l + \Delta t_r$ is the total duration in which E is reported. Also note here we need to use separate values of network for cellular and Wi-Fi.

One approach for determining the model coefficients is to measure them using controlled experiments. A more lightweight approach, which we take, is to leverage the collected energy reports on the mobile clients to derive the coefficients from the data on each device. This approach has the advantage that the derived model naturally captures the individual device deviations from platform averages, and thus is more accurate compared with a general model.

In the above formulas, the power floor P_l is a constant parameter that can be estimated from reports when no process is running, i.e., with zero CPU cycles. The time spent in the lowest power state Δt_l can be approximately set using the worst case assumption so that we derive larger than actual coefficients. We set Δt_l to a large value (99% of the time in our case) so that the energy consumed by CPU, disk and network is the upper bound of the actual value. Each E-Loupe clients derives the coefficients (a, b, c, d) using linear regression. E-Loupe then uses the derived model to gauge the worst-case energy consumed due to common resource used by processes, process groups, or the entire device.

5.2 Recovery

We propose a new mechanism, called *energy sandboxing*, that reduces the maximum amount of energy a system draws in a given interval in screen-off mode. This preserves the user perception that the battery is not draining too quickly, while still getting the work done, albeit slowly. It prevents the user from being surprised by unexpected drainage of his system's battery when the screen is off.

We implement energy sandboxing using two techniques:

Resource Sandboxing: If the energy spike is caused by high resource consumption by a process or group of processes, we limit the maximum amount of resources these processes are allowed to use within a time interval. For example, using this technique, we could limit the culprit processes to use at most 10 seconds of CPU time in a 15 minute interval, or 1 MB of data download in a 15 minute interval.

To determine how much to sandbox, we use the energy model described previously. We compute the maximum amount of resources the culprit process(es) is allowed to consume in a given time period, while meeting the desired power requirement of the device. Note that maximum coefficients in the approximate model (Section 5.1) prevents us from overly constraining the maximum resource limit for the processes. Once the processes reach their specified limit, we suspend them until the next time period when its quota is refreshed. We release the resource limit on these processes once the user turns on the screen. Note that other well-behaved processes are not affected by this approach.

Slowdown: When the energy spike is caused by a device problem, we slow down the frequency of OS timers. This reduces the frequency with which the OS wakes up to service the processes, thereby prolonging the time it is in the lowest power state. Linux introduced the concept of tickless timers, using which the OS would poll the system at a dynamic frequency [19]. We use a similar approach in screen off mode, to help the system stay in the lowest power state for longer. Note that system interrupts are still delivered in a timely fashion; for example, an incoming VoIP call would still wake up the OS and complete the call. Slowdown only affects software timers that the OS uses, for example, for periodic wakeups, or for checking the state of connected devices.

To determine the new software timer frequency, we use the approximate energy model and note that Δt_l is inversely proportional to software timer polling frequency; so a higher polling frequency reduces the amount of time the system is in the lowest power state.

Discussion: We note that both these approaches might affect user experience. They will prolong the standby time, but may also delay the completion of jobs. To avoid an adverse user experience, we do not trigger energy sandboxing unless the user opts in for this policy. Second, we show the slowed-down processes on the user's screen next time they turn the screen on. This enables them to exempt a process from this policy. Finally, we do not sandbox real-time processes, such as IM, VoIP, radio, and music clients.

As we see in Figure 5, there are cases when the energy spike is caused by a process that keeps the system in high power state for long, or when the power floor itself is high. While *slowdown* helps slightly mitigate the problem, it does not address the root cause of the problem: a device lock, a wakelock, or a connected accessory. Therefore, when the diagnosis module flags that resource consumption is not the cause, we inspect all resource locks and enabled accessories, and try to disable the devices or suspend processes that hold the locks for that time interval. Our current technique is ad hoc, and only works for sensors or accessories we have blacklisted. We plan to explore this resolution space in more detail in future work.

6. IMPLEMENTATION

We have implemented the E-Loupe client on Windows 8, and the E-Loupe server using a combination of a SQL server and a machine cluster.[2] The client consists of about 150 lines of code in the kernel, and approximately 1000 lines of code at the user level. The web service is approximately 5000 lines of code.

Data Logger

We leveraged the existing data collection framework in Windows 8 to obtain the energy consumed in an interval, and the names of running processes. The OS also maintains the CPU cycles, network bytes and disk operations consumed by each process in foreground and background mode, and for different network types. The data logger runs every minute in screen on mode, and every 15 minutes in screen off mode, and captures the battery level, set of processes, and their respective resource utilization over the entire duration.

Server

The server is implemented as two separate interacting components that work with each other. The first is a back end data-processing component, written in C# and PLINQ, which periodically computes

[2]Since the Windows Phone system uses the same Windows 8 kernel, we believe it should be easy to port our system to the phone.

the isolation results and outputs metrics (the *Consumed Power* and *Occurrence Frequency*). The second component, written in ASP.NET, is a Web service component that generates and updates Web contents to a Microsoft IIS Web server and interacts with the clients. It handles incoming energy reports as well as outgoing results of the statistical analysis.

Unlike detection, which is relatively easy to compute, the isolation stage is more expensive as we need to perform many Student t-tests, which are both I/O and computation intensive. To improve efficiency, we aggressively cache the t-test results from processing different reports to avoid redundant computation. In addition, we leverage PLINQ to parallelize the computations across different cores for efficiency.

Note that we do not restrict the number of reports submitted by a client for it to benefit from the service. In fact, since we aggregate data from a large number of clients, a client that submits a single energy report can still reap the benefit. To ensure meaningful statistics with enough data points for each process, we exclude from our analysis processes that are installed on fewer than five devices. An energy spike caused by such processes cannot be identified as factors by our system. Likewise, apps that are newly available in the marketplace are in a similar situation. In Section 7.2.4, we study how different amount of data impacts the detection and isolation results.

6.1 Client Diagnosis & Recovery Daemon

We implemented this component using a user-level daemon and modifications to the kernel scheduler. The mechanisms are implemented in the kernel, while the policies are implemented in the daemon, as shown in Figure 6. The user daemon computes the amount of sandboxing to be implemented and calls into kernel APIs to implement the sandboxing.

We implement the following mechanisms in the kernel.

Per-process resource sandboxing: The Windows 8 kernel maintains per-process resource consumption (cpu, network, disk) in the process control block. Each process is allocated the same fixed resource quota (CPU, network). When this quota is exhausted, processes are allowed to access a shared global pool of resources when available [3]. We modified this implementation to (i) significantly increase the global pool for all processes, so that non-culprit processes do not run out of resources (ii) enable each process to have an independent resource quota, and

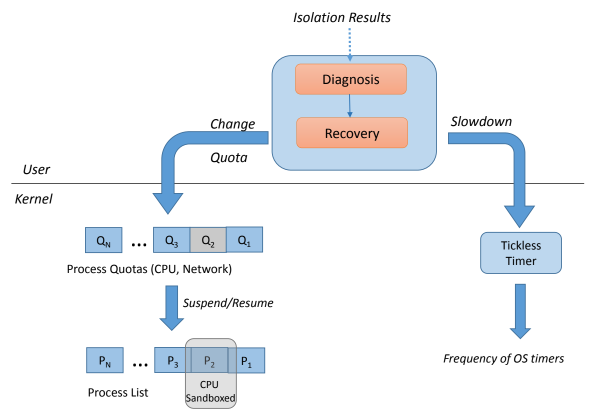

Figure 6: The user level energy recovery service inputs from the E-Loupe service and either applies resource sandboxing, or the slowdown policy, which are implemented by the kernel.

(iii) prevent the sandboxed processes from accessing the global pool. Every time the process is scheduled to run, the scheduler checks its current utilization with its permissible quota. If it exceeds the assigned quota, the process is put in a suspended state. The per-process quota is assigned for a 15 minute interval (synchronized across processes) and the process's utilization is reset to 0 at the beginning of each interval.

Slowdown: We use the dynamic timer implementation in the kernel to implement slowdown. We exposed this as a configurable parameter, which is called by the user level daemon to change the timer frequency. Processes continue to run as normal during the slowdown. However, all OS timers are dilated in time. As soon as the user turns on the screen, the timer is set to the normal value.

7. EXPERIMENTAL RESULTS

We first present details of our dataset, and then show that E-Loupe can (i) accurately isolate the faulting processes, and (ii) mitigate the energy spike.

7.1 Datasets

# platforms	# devices	# reports	# popular apps
16	73,119	16,119,233	13,190

Table 1: Dataset statistics.

Our dataset consists of about 16 million reports from over 73K devices (Table 1) spanning 14 weeks. These device were configured to submit energy reports at different collection intervals and each de-

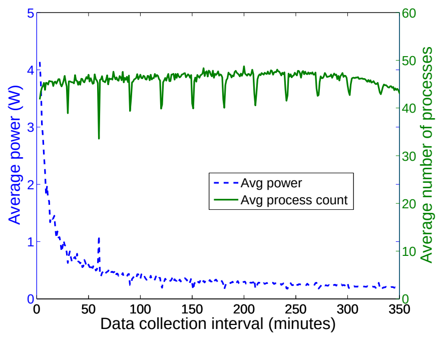

Figure 7: Average power and number of processes for different data collection window.

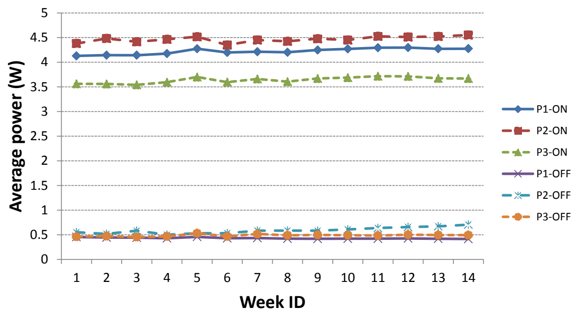

Figure 8: Average power consumption of three most popular platforms over time.

vice submitted only a subset of its reports. Our dataset contains over 13K processes that were active on more than 5 machines.

A concern when collecting coarse-grained data is the loss of fidelity with increasing collection intervals. To understand this variation we plot (Figure 7) the power value and the number of processes in the record vs. the collection interval. Indeed, as we increase the interval, the power value tends to become smaller, and so we may miss energy spikes in the detection phase. On the other hand, increasing the window size does not make our isolation ineffective because the number of candidate processes for each report does not increase much, implying that the scope of isolation remains relatively constant.

7.2 Isolating the Causes of Energy Spikes

We apply the E-Loupe server to detect and isolate energy spikes in our dataset. We then present experiments to determine the validity of our results. Figure 8 plots the average power for the three most popular platforms (named P1, P2, and P3) over 14 weeks.. The screen-on power (P1-ON, P2-ON, and P3-ON) figures are roughly 10 times that in the screen-off mode (P1-OFF, P2-OFF, and P3-OFF), but the results are similar and consistent over time. In the rest of this section we present results from one representative week on the most popular platform.

7.2.1 *Breakdown of Isolation Results*

Using the detection method described in Section 4, we identify around 2.4-3.8% reports as having energy spikes from total 731,865 reports. We perform the isolation process on these reports and present the results in Table 2. Overall, E-Loupe is able to attribute energy spikes to specific factors such as device or process in the majority of cases—87.5% for screen-on reports and 92.3% for screen-off reports. The remaining 12.5% and 7.7% cannot be classified, and we can use the local energy model based on the cumulative resource consumptions (e.g., CPU, disk, network) to further diagnose them.

When the screen is on, we find that device itself is a major factor of high energy drain (47.1%), followed by a combination of process and device (31.9%). This observation suggests that the device settings (e.g., display brightness) or usage patterns (active gaming activities) are more likely the dominant cause of high energy drain. In contrast, when the screen is off, only 14.6% of energy spikes are device-related; most are caused by process issues. In particular, 76.7% of them are due to individual processes.

7.2.2 *Top Processes Causing High Energy Drain*

We now examine the types of processes or process groups that caused high energy drain.

Screen-On Cases

In the screen-on case, E-Loupe outputs 34 individual processes as causing high energy drain, with no systems processes amongst them. The biggest contributors to energy are games (e.g., Hydro thunder hurricane, Solitaire), typically with intensive CPU, network, and disk usage as well as high display dynamics. Apart from games, a few communication apps, such as QQ, also cause high energy drain, probably because of their video chat feature.

Some communication apps, such as Skype and Lync, are not identified as a significant factor by themselves, but in conjunction with game apps (e.g, Solitaire), these are selected, possibly because both of them tend to use more energy than regular apps and their combined usage is significant. Such process combinations are in fact the most common factors for explaining energy spikes in the screen-on case.

	Device issues	Process related				Other
		Total	Single process	Process groups	Process and device	
Screen on	47.1%	44.4%	7.8%	20.8%	31.9%	12.5%
Screen off	14.6%	77.7%	76.7%	7.1%	38.8%	7.7%

Table 2: Isolation result breakdown. A high energy drain can be isolated into a combination of process-related factors, around 3-4 in the average case. Each column refers to the percentage of energy spikes that had the corresponding factors output by the isolation analysis.

Worker process	Corresponding app description
FlashUtil_ActiveX.exe	Internet Explorer
WSHost.exe	Windows store
svchost.exe [WerSvcGroup]	Skype app
IMEbroker.exe	QQ app
msfeedssync.exe	Skype app

Table 3: Example popular process groups with worker processes.

	High-power reports	Low-power reports
Screen on	15.48%	1.32%
Screen off	33.80 %	3.16%

Table 4: Validation.

Screen-Off Cases

In the screen-off case, however, the most commonly observed energy spikes are from a few worker-processes, which perform tasks for other applications. In total, we identified 10 such worker-processes and they are correlated with 114 other processes that may have triggered them to work. We list a few of them that occur frequently in Table 3. Thus when the screen is off, a common source of energy drain are those workers that silently perform tasks in the background for long-lasting apps that users may leave to run.

In addition to worker processes, there also exist 74 non-worker, app processes identified as responsible factors. Some top frequent ones include productivity processes, music, video, and radio streaming apps, news and social network apps and a finance app. A common characteristic of these apps is that they often require network communications and may also use additional hardware resources (e.g., audio).

The isolation also outputs in total 181 process interactions as potential factors, and similar to the screen-on cases, many of them include instant messenger and VoIP apps (34 out of 181 process groups). We also observe combinations of worker and app processes, where the worker-processes alone are not selected in isolation. For example, we find 3 combinations where `audiodg.exe` working with music or radio apps, and 15 combinations involving different instances of `svchost.exe`.

7.2.3 Validating the Isolation Results

To evaluate the isolation results, ideally, we would like to use ground truth information regarding known causes of high energy drain (e.g., via user reports). Lacking such ground truth, we validate our approach by testing the results generated from one portion of the dataset, and applying them to new part of the dataset. We perform two sets of experiments – to validate the set of energy-hungry processes, and to validate the device issues.

First, we validate the set of processes or process groups as high energy factors. We apply the isolation results derived using data from three consecutive weeks (i.e., training data) to the data from the fourth week data (i.e., testing data). To be general, we pick only the set of processes or process groups from the training data that have triggered at least two energy spikes. For testing, we select top 2% of high-power testing reports and sample 2% of low-power testing reports for comparison. For each testing report, we examine if the flagged process or process groups occur and if they consume CPU cycles. If so, we mark this report as flagged by our previous isolation results.

Table 4 shows the percentage of flagged testing records. The process-related factors occur in 15.48-33.80% of high power reports, but the chance of them showing up in low power reports is around 10 times lower, suggesting that these are likely the correct high-energy causes. Although these factors do not explain all high-energy reports, we note that not all energy spikes are caused by process issues, as presented in Section 7.2.1. Furthermore, we picked only the most commonly occurring processes and so ignored cases where an energy spike is caused by an unusual process or process combinations.

The second experiment attempts to validate the reported device issues or process and device combinations. In our dataset, the device IDs are unique only within a week. So we take half of a week's data as training, and apply the results on the second half of the week. We focus on only the set of devices that have submitted sufficient reports (≥ 30). The validation results are highly encouraging. Once we flag a device as a cause by itself (or device and process issues), all its future reports indeed exhibit consistent high-energy drain and we flag 100% them in our experiments.

7.2.4 *Data Sensitivity*

The premise of our isolation technique is that we have access to a large amount of data for statistical tests. A natural question is "how does the amount of data affect isolation results?". There are two cases where the amount of data matters: First, we require a process to be observed across at least 5 devices for it to be included in analysis. Second, in performing t-tests, we require at least 5 sample points from each distribution.

To answer the above question, we run isolation on differently sampled datasets. As expected, we found that less data decreased the success rate, but we could still isolate a large fraction of energy spikes. Even with 20% of the samples, E-Loupe is able to find causes for over 40% of high energy drain reports. Therefore, we believe that E-Loupe can be incrementally deployed, and its effectiveness will keep getting better with more data.

7.3 Diagnosis and Recovery

We first show that the approximate energy model works as expected. We then show the effectiveness of our recovery techniques on real machines, and then use a large trace based simulation to evaluate the expected improvement in the wild.

7.3.1 *Diagnosis*

We first validate the approximate energy model, and then evaluate the diagnosis on our trace data.

We derive the model on short term reports, since they are over shorter periods of time and are more accurate. We aggregate these reports for the entire platform so that we have sufficient data for our experiment. Table 5 shows the coefficients of resources derived using linear regression. We separate the coefficients for disk read and write, as well as network send and receive, as these activities each consume different amount of energy. To test regression accuracy, we divide the data into training and testing data. Our testing errors are around 14% - 19%. Since high energy consumption can be caused by reasons other than high resource usage, the linear resource model is expected to have errors. However, the less than 20% error is sufficient for E-Loupe to determine if an energy spike is related to high resource consumption.

7.3.2 *Recovery on the Real Device*

We first study the effect of Resource Sandboxing on the power consumption of the device, and the

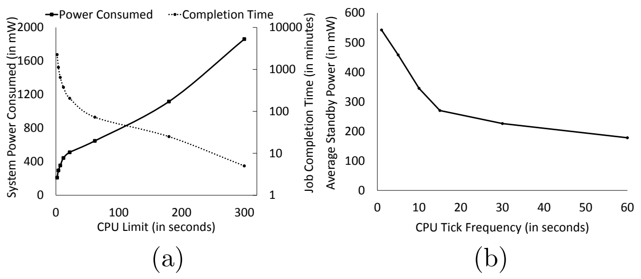

Figure 9: (a) Impact of CPU sandboxing on the power consumption and task completion time. The results are from a prototype Windows RT device. (b) Impact of system slowdown on average power consumption.

completion time of the task. We wrote a custom app that created background CPU load on the device when the screen was on. We then let the E-Loupe recovery mechanism take over but with different, carefully chosen, target power levels. This resulted in our custom process getting anywhere from 2 to 300 seconds of execution time every 15 minutes. We used the fuel gauge to measure the power consumed across runs that lasted several hours. We also accounted for non-linearities in reported battery values by starting each fresh experiment at 100% charge.

In Figure 9 (a), we show the total completion time and the actual power that was consumed for each of the power levels. At the lowest power level, our sandboxed process gets only 2 seconds of execution time every 15 minutes. While this works well for increasing the standby battery life, the process takes very long to complete. This policy is best for rogue apps, or when the battery is running low. When the target power is high, the process completes much faster, but at a cost of significantly higher power consumption.

We next study the impact of slowdown on energy consumption. On a system with several installed processes, but in airplane mode, we changed the OS tick frequency (described in Section 5.2). We measured the battery level in the beginning and at the end of run, where each run lasted at least a couple of hours. As we see in Figure 9 (b), changing the frequency from 1s to 60s reduced the average power consumed by more than one-third. However, as mentioned earlier, this reduction in power comes at a cost – a few OS services get delayed notifications.

7.3.3 *Expected Improvement on Trace Data*

We perform trace-based simulation to evaluate the improvement on energy spikes detected in real

	Coefficients						Regression error	
P_l	a	CPU (Mcycle)	Disk read (KB)	Disk write (KB)	Net send (KB)	Net rcvd (KB)	Median	Mean
200	1667	0.38	0.054	0.150	13.341	3.048	14%	19%

Table 5: Energy model derived from data. All the units are in mJ, except P_l is in the unit of mW.

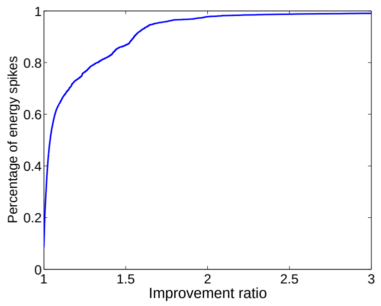

Figure 10: The improvement ratio of using energy sandboxing. We compare only the portion of power coming from resource consumptions (CPU, disk, network).

data. We use the average platform power (420mW) as our target power for sandboxing, and use this target to compute the upper bound for CPU cycles that a device can consume based on data in Figure 9, which is 400Mcycles/min. To estimate the energy consumed from resources, we use the energy model derived in Section 7.3.1.

We consider two cases in our simulation, depending on the isolation results: (1) if we see a device issue, then we simulate device slowdown. The slowdown factor is determined by the actual aggregated CPU cycles observed and the targeted CPU cycles; (2) if the problem is caused by processes, then each process may have a different slowdown factor. For simplicity, we compute a fair share of CPU for each process. Well-behaved processes, not reported by the isolation strategy, are not sandboxed. The remaining culprit processes are then sandboxed so that they do not exceed their fair share.

We then use the energy model to compute just the portion of power due to resource consumptions, using coefficients b, c, d. Figure 10 shows the ratio of energy before sandboxing vs. after sandboxing. We can reduce resource-based power by 1-3 times for most of the spikes, sometimes the benefit can be more than that. The remaining 18% of energy spikes do not benefit. They have process-related issues but these processes do not trigger high resource consumptions. This early result is promising in suggesting that energy sandboxing is an effective primitive to control energy usage.

8. SUMMARY AND FUTURE WORK

E-Loupe takes the first step in end-to-end energy management. We leverage the ability to gather device energy readings and statistical techniques to provide nuanced, on-demand control over energy hungry applications. Since a part of the work is offloaded to individual clients, and the statistical analysis in the centralized service is parallelized, we believe our approach will scale adequately. Our approach does not make a particular application more energy efficient, but it prevents the battery from unexpectedly and prematurely draining.

There exist prototypes to monitor the energy consumed by various components on a mobile device [7, 11]. In the future, we will likely be able to more accurately attribute energy consumption to some components. We also need a way to attribute aggregate energy consumption to various processes. Moving forward, we plan to expand the isolation techniques in Section 4 to work with the new features that might become available.

Consulting with the cloud on every energy spike might consume energy, and might also consume cellular data bandwidth. We realize this limitation and are investigating a local mini-E-Loupe that can try to make a first cut guess and apply recovery mechanisms locally, and consult the cloud service only if the local techniques do not work.

Research and industry are proposing new techniques to save energy, such as ARM's Big.Little low-power processor architecture, core-gating to shut off unused cores, cloud offloading [9], etc. We are actively exploring these and other strategies as part of system's recovery component, such as offloading the culprit processes to a low power core, or to the cloud.

9. REFERENCES

[1] Android developer: Processes and threads. http://developer.android.com/.
[2] ios programming guide: App states and multitasking. http://developer.apple.com/.
[3] Windows 8: Introduction to Background Tasks, Guidelines to Developers. http://go.microsoft.com/fwlink/?LinkID=227329&clcid=0x409.
[4] Android apps and news: Google blames android battery woes on user practices and poorly-designed apps. http://androinica.com/2010/05/google-blames-android-battery-woes-on-user-practices-and-poorly-designed-apps/, 2010.

[5] Intomobile: Motorola says android apps negatively affecting device performance. http://www.intomobile.com/2011/06/03/motorola-says-android-apps-negatively-affecting-device-performance/, 2011.

[6] Betterbatterystats. https://play.google.com/store/, 2012.

[7] Qualcomm developer network: Trepn profiler. https://developer.qualcomm.com/mobile-development/development-devices/trepn-profiler, 2013.

[8] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[9] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[10] M. Dong and L. Zhong. Chameleon: A color-adaptive web browser for mobile OLED displays. In *ACM MobiSys*, pages 85–98, 2011.

[11] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 323–338, Berkeley, CA, USA, 2008. USENIX Association.

[12] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. In *First Workshop on Hot Topics in Measurement and Modeling of Computer Systems*, HotMetrics08. ACM, 2008.

[13] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[14] A. J. Oliner, A. Iyer, E. Lagerspetz, S. Tarkoma, and I. Stoica. Collaborative energy debugging for mobile devices. In *Proceedings of the Eighth USENIX conference on Hot Topics in System Dependability*, pages 6–6, 2012.

[15] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[16] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[17] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: A cross-layer approach. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 321–334, New York, NY, USA, 2011. ACM.

[18] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the Cinder operating system. In *EuroSys*, pages 139–152, 2011.

[19] S. Siddha, V. Pallipadi, and A. V. D. Ven. Getting maximum mileage out of tickless. In *Intel Open Source Technology Center, Linux Symposium*, 2007.

[20] L. Wasserman. *All of statistics: A concise course in statistical inference*. Springer Verlag, 2004.

[21] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.