Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof

Abhinav Pathak
Purdue University
pathaka@purdue.edu

Y. Charlie Hu
Purdue University
ychu@purdue.edu

Ming Zhang
Microsoft Research
mzh@microsoft.com

Abstract

Where is the energy spent inside my app? Despite the immense popularity of smartphones and the fact that energy is the most crucial aspect in smartphone programming, the answer to the above question remains elusive. This paper first presents *eprof*, the first fine-grained energy profiler for smartphone apps. Compared to profiling the runtime of applications running on conventional computers, profiling energy consumption of applications running on smartphones faces a unique challenge, asynchronous power behavior, where the effect on a component's power state due to a program entity lasts beyond the end of that program entity. We present the design, implementation and evaluation of *eprof* on two mobile OSes, Android and Windows Mobile.

We then present an in-depth case study, the first of its kind, of six popular smartphones apps (including Angry-Birds, Facebook and Browser). *Eprof* sheds lights on internal energy dissipation of these apps and exposes surprising findings like 65%-75% of energy in free apps is spent in third-party advertisement modules. *Eprof* also reveals several "wakelock bugs", a family of "energy bugs" in smartphone apps, and effectively pinpoints their location in the source code. The case study highlights the fact that most of the energy in smartphone apps is spent in I/O, and I/O events are clustered, often due to a few routines. This motivates us to propose bundles, a new accounting presentation of app I/O energy, which helps the developer to quickly understand and optimize the energy drain of her app. Using the bundle presentation, we reduced the energy consumption of four apps by 20% to 65%.

Categories and Subject Descriptors D.4.8 [Operating Systems]: Performance–Modeling and Prediction.
General Terms Design, Experimentation, Measurement.
Keywords Smartphones, Mobile, Energy, Eprof.

1. Introduction

Smartphones run complete OSes which provide full-fledged "app" development platforms, and coupled with "exotic" components such as Camera and GPS, have unleashed the imagination of app developers. According to a new report [1], the app market will explode exponentially to a $38 billion industry by 2015, riding the huge growth in popularity of smartphones. Despite the incredible market penetration of smartphones and exponential growth of the app market, their utility has been and will remain severely limited by the battery life. As such, optimizing the energy consumption of millions of smartphone apps is of critical importance. However, the quarter million apps [2] developed so far were largely developed in an energy oblivious manner. The key enabler for energy-aware smartphone app development is an energy profiler, that can answer the fundamental question of *where is the energy spent inside an app*? Such a tool can be used by an app developer to profile and consequently optimize the energy consumption of smartphone apps, much like how performance profiling enabled by *gprof* [3] has facilitated performance optimization in the past several decades.

Designing an energy profiler for modern smartphones faces three challenges. First, it needs to track the activities of *program entities* at the granularity that a developer is interested in. For example, some developers may be interested in energy drain at the level of threads, while others may desire to understand the energy breakdown of an app at the granularity of routines, which are the natural building blocks following the modular programming design principle.

Second, energy accounting requires tracking of power draw activities of various smartphone hardware components. Third, the power draw and consequently energy consumption activities need to be mapped to the program entities responsible for them. Performing the above two tasks for smartphones faces several major challenges. First, modern smartphones do not come with built-in power meters. Second, and more importantly, smartphone components exhibit asynchronous power behavior, *i.e.,* the instantaneous power draw of a component may not be related to the current utilization of that component. Such asynchronous behavior include: **(a) Tail power state:** Several components (GPS, WiFi, SDCard, 3G) have tail power states [4, 5]; **(b) Per-**

sistent power state wakelocks: Smartphone OSes employ aggressive CPU/Screen sleeping policies and export wakelock APIs for use by apps to prevent them from sleeping. In a typical usage, the power drain due to a wakelock persists beyond a program entity (*e.g.,* a routine); **(c) Exotic components:** Newer components like camera and GPS start consuming high power once switched on in one entity, and often continue till switched off by some other entity [4, 6]. Such asynchronous power behavior pose challenges to correctly attributing the energy consumption of the whole phone to individual program entities.

In this paper, we study the problem of energy profiling and accounting of smartphone apps and make three concrete contributions towards enabling *energy-aware app development* on smartphones. First, we present the design of *eprof*, the first (to the best of our knowledge) fine-grained energy profiler for modern smartphones, and its implementation on two popular mobile OSes, Android and Windows Mobile. Our design leverages a recently proposed fine-grained online power modeling technique [4], which accurately captures complicated power behavior of modern smartphone components in a system-call-driven Finite State Machine (FSM). *Eprof* design focuses on energy accounting policies: how to map the power draw and energy consumption back to program entities. We explore alternate accounting policies and adopt in *eprof* the *last-trigger* policy which attributes lingering energy drain (*e.g.,* tail) to the last trigger, as it more intuitively reflects asynchronous power behavior in mapping energy activities to the responsible program entities.

Second, we report on our experience with using *eprof* to analyze, for the first time, the energy consumption of six of the top 10 most popular apps from Android Market including AngryBirds, Android Browser, and Facebook. *Eprof* exposes many surprising findings about these popular apps: (a) third-party advertisement modules in free apps could consume 65-75% of the total app energy (*e.g.,* AngryBirds, popular chess app); (b) clean termination of long lived TCP sockets could consume 10-50% of the total energy (*e.g.,* browser doing google search, CNN surfing, AngryBirds, NYTimes app, mapquest app), (c) tracking user data (*e.g.,* location, phone stats) consumes 20-30% of the total energy (*e.g.,* NYTimes). In a nut shell, *eprof* shows that, in most popular free apps, performing the task related to the purpose of the app (*e.g.,* chess algorithms in chess apps) consumes only a small fraction (10-30%) of the total app energy.

Our experience with profiling these popular apps using *eprof* revealed several key observations. (1) Our experience confirms with ample evidence that smartphone apps spend a major portion of energy in I/O components such as 3G, WiFi, and GPS. This suggests that compared to desktop apps, optimizing the energy consumption of smartphone apps should have a new focus: the I/O energy. This is especially true since CPU energy optimization techniques have been well studied and mature techniques like frequency scaling have already been incorporated in smartphones. (2) The asynchronous power behavior of smartphone I/O components is indeed triggered often in smartphone apps, in fact in all 21 apps we tested, including popular ones such as Angrybirds and the Android browser. (3) Over the duration of an app execution, there are typically a few, long periods of time when I/O components continuously stay in some high power state, which we term as *I/O energy bundles*. (4) Further, the I/O energy of an app is often due to just a few routines that are called by different callers in the app source code, most intuitively a consequence of modular programming practice for I/O operations. This is in stark contrast with CPU time profiling (*e.g.,* using *gprof*) where all routines in the app consume some CPU time. Together observations (3) and (4) suggest that there are often only a few routines that are responsible for I/O bundles.

The above observations suggest that a flat per-entity energy split presentation (similar to time split reported by *gprof*) does not immediately help the programmer to curtail the app energy. A presentation that is more informative and constructive, which aims to reduce I/O energy consumption, is to identify each I/O energy bundle and present its I/O energy profile. In the third part of the paper, we develop such an energy accounting presentation which captures the routines and their causal execution order within each energy bundle. We show how such a bundle-oriented presentation facilitates quick understanding of the energy consumption of an app beyond individual routines and exposes ways of program restructuring to optimize the app's energy consumption. Using the bundle accounting information, we restructured a few apps running on the two OSes, reducing their energy consumption by 20-65%.

2. Accounting Granularity

Energy accounting for smartphone apps answers the essential question for energy optimization and debugging: *where is the energy spent inside an app?* In answering this question, we need to (1) break an app into energy accounting entities, (2) track the power draw and energy activities of each hardware component, and (3) map the energy activities to the entities responsible for them. We discuss the first task of how to track entities in this section.

Granularity of Energy Accounting. The granularity of accounting entities depends on the level at which a developer desires to isolate the energy bottleneck and optimize energy drain, *e.g.,* by restructuring the source code. An entity could be one of the four conventional, well-understood program entities, a process, a thread, a subroutine, and a system call. In principle, an entity can be made more elaborate by the programmer, *e.g.,* a collection of above program entities (*e.g.,* all routines doing networking). In this paper, we focus on the four conventional program entities and leave accounting for more general entity definitions as future work.

Energy accounting at the system call or routine granularity directly exposes the root causes for energy consumption to the developer. Splitting energy among various threads of a process is also important as modern smartphone apps often consist of a collection of code written by third-party service providers (*e.g.,* AngryBirds runs the third-party Flurry [7] program as a separate thread for data aggregation and advertisement.) Finally, per-process accounting is relevant as all new smartphone OSes support multitasking and concurrently running apps affect each other's energy consumption.

Tracking Program Entities. Since system calls are what trigger I/O components into different power states, the key to tracking all four program entities for energy accounting is to log I/O system calls (which is already done by the online power modeling scheme [4]) and their call stacks which allow us to map a system call to the calling routine, thread, and process during postprocessing. To enable accounting for CPU energy drain at the routine level, we use instrumentation to either log the exact routine boundaries or sample the stack periodically to estimate CPU utilization per routine [3]. Finally, we need to log the process and thread ids at each CPU context switch to enable CPU accounting per thread and per process.

3. Asynchronous Power Behavior

Modern smartphones come with a wide variety of I/O hardware *components* embedded in them. Typical components include CPU, memory, Secure Digital card (sdcard for short), WiFi NIC, cellular (3G), bluetooth, GPS, camera (may be multiple), accelerometer, digital compass, LCD, touch sensors, microphone, and speakers. It is common for apps to utilize several components simultaneously to offer richer user experience. Unlike in desktops and servers, in smartphones, the power consumed by each I/O component is often comparable to or higher than that by the CPU.

Each component can be in several operating modes, known as *power states* for that component, each draining a different amount of power. Each component has its own base state which is the power state where that particular component consumes zero power (irrespective of other components). A component can have one or more levels of productive power states (*e.g.,* low and high for WiFi NIC), and the tail power state, which typically consumes less power than a productive power state, *e.g.,* WiFi, sdcard, 3G radio.[1] Finally, the idle power state corresponds to the *system-wide* power state where the phone drains near zero power: the CPU is shut off, the screen is off, and all other components are turned down, except the network components which respond to periodic beacons.

Modern smartphones exhibit *asynchronous power behavior* where an entity's impact on the power consumption of the phone may persist until long after the entity is completed.

Tail energy. Several components, *e.g.,* disk, WiFi, 3G, GPS, in smartphones exhibit the tail power behavior [4–6], where activities in one entity, *e.g.,* a routine, can trigger a component to enter a high power state and stay in that power state long beyond the end of the routine. This is in stark contrast with the execution time metric profiled by *gprof* which ends promptly when the routine returns.

Wakelocks. Smartphone OSes apply aggressive sleeping policies which make smartphones sleep after a brief period of user inactivity, and export APIs which apps need to use to ensure the components stay awake, irrespective of user activities, so that apps can perform their intermittent activities in the background (*e.g.,* network sync). Figure 1 shows the power state changes due to wakelocks [8] on Android on passion (Table 1 lists the mobile phones we use throughout the paper). For example, when wakelock `PARTIAL_WAKE_LOCK` exported by the PowerManager class in Android is acquired, the CPU is turned on, consuming 25mA.[2]

Wakelocks thus present another example of asynchronous power behavior of smartphones. A wakelock acquired by a caller entity,[3] *e.g.,* a routine, triggers a component into a high power state. The component continues to consume power after the entity is completed and other entities start using the component. The component is returned back to the idle power state when the wakelock is released, possibly by another entity. Correctly accounting energy due to wakelocks is particularly important as it can help to track down wakelock bugs [9] (*e.g.,* Facebook bug [10], Android eMail bug [11, 12], and Location Listener bug [13]).

Exotic components. Today's smartphones contain several exotic components, such as GPS, camera, accelerometer, and sensors, which consume energy differently than traditional components like CPU [4, 6]. Once these components are switched on by an entity, they continue to drain power until the moment they are switched off, often by another entity.

The above asynchronous power behavior pose challenges to the second task of developing an energy accounting tool, *i.e.,* tracking energy activities of the components. We overcome these challenges by leveraging a recently proposed online power model for smartphones [4], which captures the above intricate asynchronous power behavior of modern smartphones in a finite state machine (FSM). The FSM consists of power states as the nodes and system calls as the triggers for transitions among the power states. Using the FSM power model, system calls issued during the app execution drive the FSM to different power states. For a productive power state, linear regression is used to correlate the duration the component stays in that state with the parameters (workload) of the system call that drove the FSM to the state, and energy consumption at that state is deduced [4]. The duration and hence the energy consumed at tail states and states due to wakelock acquires and releases are straight-forward.

[1] Special cases such as CPU frequency scaling and wireless signal strength are handled by altering the magnitude of the power consumed in the respective states as a function of these state parameter values.

[2] In this paper, for power measurement we directly report the current drawn in milli-Amperes (mA). The actual power consumed would be the current drawn multiplied by 3.7V, the voltage supply of the battery. Similarly, for energy we directly report micro Ampere Hours (μAH); the actual energy would be the μAH value multiplied by 3.7V. The smartphone batteries are rated using these metrics and hence are easy to cross reference.

[3] Usually wakelocks are held by framework entities in Android, which control the inactivity timeouts, based on user level policies.

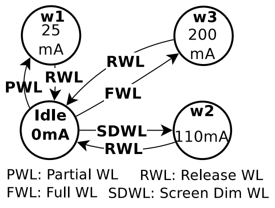


Fig. 1: Wakelock FSM (passion /Android).

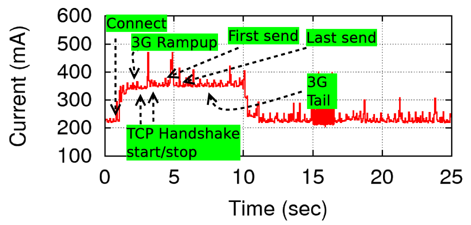


Fig. 2: Send happens right after connect.

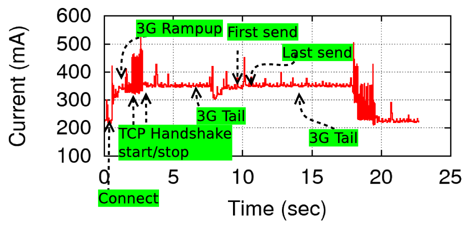


Fig. 3: Send happens 5 seconds after connect.

Table 1: Mobile handsets used throughout the paper.

Name	HTC-	MHz	OS (kernel)
magic	Magic	528	Android 2.0 (Linux 2.6.34)
tytn2	Tytn II	400	WM6.5 (CE5.2)
passion	Passion	1024	Android 2.3 (Linux 2.6.38)

4. Accounting Policies on Smartphones

In this section, we first use an example to show how the above asynchronous power behavior of smartphones poses unique challenges to the third task of energy accounting, *i.e.,* how to attribute energy activities to the responsible program entities. We discuss alternate accounting policies and then present the energy accounting policy used in *eprof*.

4.1 Accounting Policy Challenge: A Simple Example

The accounting policy complications due to the three asynchronous power behavior share the same nature: how to attribute an energy activity that persists beyond the triggering program entity or entities. We focus on the tail energy behavior, to illustrate the complication and design choices.

Consider a simple app that connects (in routine `net connect()`), and uploads data via five sends with 10KB each (in routine `netsend()`), to a server over the 3G network. Figure 2 plots the current draw of passion running Android during the app execution. The app consumes a total of 314 μAH of energy. The moment the connect system call is issued, the 3G radio ramps up [5, 14] power draw for 2.5 seconds before the TCP handshake is started. The rampup consumes 61 μAH (19.5% of the entire app energy). After the handshake which consumes 11 μAH (3.5%), routine `netconnect()` is completed, `netsend()` starts and performs the five sends (which together consumes 55 μAH (17.5%)), and the app is completed. However, even after the app completion, the device continues to draw high power due to the 3G radio staying in the tail power state for 6 seconds, consuming 187 μAH, 59.6% of the total app energy.

Figure 3 plots the power draw of the same app except a single difference, the `netsend()` routine is performed 5 seconds after `netconnect()`. This program consumes 520 μAH (65% more than the original version) with the following energy breakdown: rampup (60 μAH, 11.53%), connect (15 μAH, 2.88%), tail 1 (183 μAH, 35.19%), send (60 μAH, 11.53%), and tail 2 (200 μAH, 38.46%).

The above examples show that the tail energy in Figure 2 would have existed even if the second routine did not exist, and hence intuitively the first routine should be held accountable for the tail energy somehow. One simple policy is to split the tail energy among the two routines either equally or weighted based on the workload generated. Such a policy faces several problems: (1) It is not always easy to define the weights based on the workload generated, *e.g.,* in this app, should the weight assigned to `netconnect()` be 3 handshake packets and to `netsend()` be 5*10KB of packets? (2) This splitting policy becomes more complicated to implement and more obscure in understanding the profiling output in the presence of intermittent component accesses which result in interleaved productive states and tail states. (3) Splitting the tail energy may misinform the developer that if a certain entity, *e.g.,* `netsend()`, is removed, its part of tail energy could be saved.

An alternative accounting policy, termed last-trigger policy, is to account the tail energy to the last entity, out of all the entities, each of which *would have* triggered the tail, *i.e.,* routine `netsend()` in the case of Figure 2. This approach avoids the first two problems above, which makes it not only easier to implement, but more importantly, much easier to understand by the programmer. However, this approach still may misinform the developer that if the last trigger, *e.g.,* `netsend()`, is removed, the tail energy would be removed. In reality, the same amount of tail energy would have been consumed irrespective of whether the last trigger existed. For example, in Figure 2 if `netsend()` did not exist, `netconnect()` would have also been followed by a similar 3G tail.

We also considered other possible policies such as first-trigger, which accounts the tail energy to the first entity, out of all the consecutive entities, each of which *would have* triggered the tail. Such a policy shares with last-trigger in encouraging triggers to draft behind each other to save energy, and in misleading developers that removing the first trigger would remove the tail. Out of the two, last-trigger appears slightly more intuitive; the developer can start with optimizing the last trigger.

Finally, we argue this last "misinforming" problem exists no matter what accounting policy is used. Hence ultimately, for an accounting tool to be informative to the developer, the profiling output needs to make explicit how the energy due to asynchronous power behavior such as tail energy

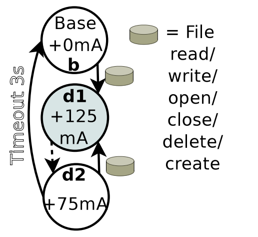


Fig. 4: Sdcard FSM for tytn2 on WM6.

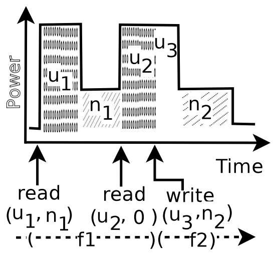


Fig. 5: Assign energy to last system call.

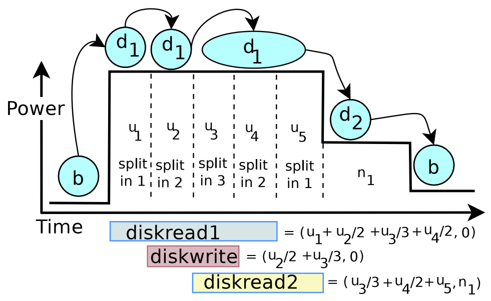

Power
Time
d1
d1
d1
d2
b
b
u1
u2
u3
u4
u5
split in 1
split in 2
split in 3
split in 2
split in 1
n1
diskread1 = (u1+ u2/2 +u3/3 +u4/2, 0)
diskwrite = (u2/2 +u3/3, 0)
diskread2 = (u3/3 +u4/2+u5, n1)

Fig. 6: Splitting energy of a component among concurrent system calls.

is accounted, and the developer needs to understand such asynchronous power behavior to make meaningful use of such energy accounting tools.

4.2 Accounting Policies for Asynchronous Power

Following the above discussion, we adopt the last-trigger policy in *eprof*: always account the energy lingering beyond a program entity due to asynchronous power behavior (*e.g.,* tail energy) to the last entity, out of all the entities that *would have* triggered the power behavior. The policy will be stated explicitly in the profiling output.

4.2.1 Tail Power State

Since tail energy is wasted as the component is not doing any productive work, many potential optimizations (*e.g.,* aggregation [5]) are being studied to reduce tail energy. For this reason, *eprof* explicitly separates tail energy from the rest, and reports an "energy tuple" (u, n), where u and n represent the utilization energy and the tail energy consumption, respectively, in its profiling output.

We illustrate how the accounting policy is applied to the tail power state behavior using an example. Figure 4 shows an example of the tail power state in the FSM power model of sdcard on the tytn2 phone. Any file operation sends sdcard into a high power state *d1* followed by a tail state *d2* which continues until 3 seconds of disk inactivity and then sdcard returns to the base state. Figure 5 shows an example containing two entities f1 and f2. Entity f1 invokes the first $read$ call which sends the component to state *d1*, consuming u_1 energy, followed by a tail consuming n_1 which is cut short by a $read$ call, which again sends the component to *d1*, consuming u_2. Right after entity f1 ends, f2 starts and invokes a $write$ call, causing the component to stay in state *d1*, consuming u_3, followed by a tail state consuming n_2. The tail state lasts beyond the completion of f2.

It is clear (u_1, n_1), u_2 and u_3 should be accounted to the first $read$ call, second $read$ call and the $write$ call, respectively. Following the last-trigger policy, n_2 is charged to the last system call before the tail state, *i.e.,* $write$. In summary, the three system calls get energy tuples (u_1, n_1), $(u_2, 0)$ and (u_3, n_2), respectively.

4.2.2 Wakelocks and Exotic Components

WakeLocks and exotic components exhibit similar asynchronous energy drain patterns. Each of them has an on/off switch which when turned on (a wakelock is acquired or GPS/camera is started) starts draining energy and the energy drain stops only when it is switched off (*e.g.,* the wakelock is released). We discuss accounting for wakelocks below. Accounting for exotic components is similar.

Figure 1 shows the FSM that models the power state transitions due to wakelocks on passion running Android. An entity that acquires a wakelock triggers a component into a high power state, which can persist after the entity exits and another entity starts, until the wakelock is released by this other entity. Following the last-trigger policy, the energy consumed by the component during the period when the wakelock was held is attributed to the entity that acquired the wakelock. Accounting this way helps the developer to track "wakelock bugs", an important class of energy bugs in mobile apps [9] due to missing wakelock releases (§7.3).

4.3 Concurrent Accesses

When multiple threads access a component, there can be concurrent system calls issued to the component. Figure 6 shows an example where three threads simultaneously access sdcard for reading and writing files. $diskread1$ triggers a power state change from base to $d1$. While the component is serving this request, two other threads invoke two more requests $diskwrite$ and $diskread2$.

To perform energy accounting, we first apply linear regression inside each productive power state to estimate the total duration that component stays in that state based on the total workload of all system calls. We then divide up the total energy in that state among the multiple system calls as follows: we first estimate the completion time of each system call assuming they have the same rate of making progress, then split the whole duration into intervals, each with a different number of concurrent system calls, and then split the energy consumed in each interval evenly among those system calls. Such a policy is justified as follows. First, we observed using microbenchmarking that the time to complete I/O system calls are roughly proportional to their workload, suggesting the hardware component is mostly fair in carrying out concurrent system calls. Second, smartphone hardware does not export internal information about workload processing order and hence it is difficult to develop a more refined policy.

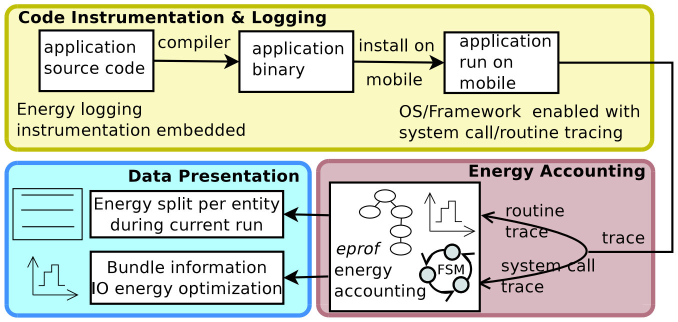

Code Instrumentation & Logging
application source code
compiler
application binary
install on mobile
application run on mobile
Energy logging instrumentation embedded
OS/Framework enabled with system call/routine tracing
Data Presentation
Energy split per entity during current run
Bundle information IO energy optimization
Energy Accounting
eprof energy accounting
FSM
routine trace
trace
system call trace

Fig. 7: ***Eprof*** **architecture overview.**

Following the above split policy, the duration while in power state $d1$ is split into five intervals with varying numbers of active system calls, and $d1$ is split evenly within each interval. The tail energy is charged to the last system call served by the component. The final accounting of sdcard energy consumption for the three calls is shown in Figure 6.

4.4 Accounting for High Rate Components

The FSM power model [4] does not cover RAM and Organic LED screen (OLED) since these components are accessed at much higher rates (and hence called high rate components) resulting in high overheads in event based modeling. Traditionally RAM power is modeled using LLC (Last Level Cache) Misses [15, 16], periodically polled from hardware (CPU registers). Power draw of OLED screens is dictated by pixel colors and hence can be modeled by periodically scrapping the screen buffer and computing the energy using sampled pixels [17]. However, the HTC magic does not export LLC Misses information to the kernel, and perf_events [18], the Linux performance counter system which is still new on ARM architectures, does not yet support the HTC passion handset. Also, Google stopped shipping developer phones with OLED screen in 2011 due to a supply shortage [19]. Hence, we leave RAM/OLED accounting as future work.

5. *Eprof* Implementation

We describe *eprof* implementation at the routine granularity. Accounting at the thread and process granularities follows naturally.

5.1 *Eprof* Operations

Figure 7 shows the three components of *eprof*: (1) code instrumentation and logging, (2) power modeling and energy accounting, and (3) profile presentation. In the first phase, the app source code is instrumented for system-call tracing and routine tracing. We also discuss in §5.2 how apps built on top of the Android SDK can be logged without source code. The instrumented binary is then run on the smartphone OS/framework with system call logging enabled, to gather both detailed routine invocation trace and system call trace at runtime. During the second phase, the routine invocation trace is played back while at the same time the system call trace is used to drive the FSM power model to replay the energy activities. The energy activities are mapped to the routines according to the accounting policy described in §4. Finally, *eprof* outputs the energy profile.

5.2 Implementation

We have implemented *eprof* on two smartphone OSes: Android and Windows Mobile 6.5 (WM6). Due to page limit, we only describe our implementation on Android below.

SDK Routine Tracing. Routing tracing logs routing invocations and the time spent per invocation. Apps written with the Android SDK run inside the Dalvik VM. For such apps, Android provides a routine profiling framework [20] which *at runtime* marks routine boundaries with timestamps and calculates the runtime of each routine. To reduce the overhead of retrieving timestamps, we modified the current profiling framework to only count all caller-callee invocations, and perform periodic sampling to log the routine call stack and the time at each sampled interval, just as in *gprof* [3].

NDK Routine Tracing. Android also provides developers with Native Development Kit (NDK) using which they can run performance critical parts of their apps outside the VM. For the NDK part of apps, we used the *gprof* port of NDK profiler [21] to perform routine tracing, which requires linking with the Android *gprof* library.

System-Call Tracing. System-call tracing logs the time and the call stack of each system call. This is performed in the framework, the bionic C library, and the kernel. First, apps written with SDK invoke both traditional system calls such as network and disk and special framework events, *e.g.,* sensors, location tracking, and camera. We log such system calls by inserting ADB (Android Debugger) logging APIs where they are implemented in the framework code [22] to log the calls (time and parameters) and call stacks. Second, apps written with NDK only use traditional system calls. However, since Arm Linux does not support userspace backtracing from inside the kernel [23], we log the calls and call stacks at the bionic C library interface. Finally, for both SDK and NDK apps, we log CPU (sched.switch) scheduling events in the kernel using Systemtap [24].

Logging without Source Code. In general, a recompile is required after instrumentation for routing tracing. For the evaluation in this paper, we modified the framework to automatically start and stop *eprof* routine and system-call tracing for the SDK part of all apps. This allows us to perform energy profiling without needing a recompile and hence the source code which is often not available (*e.g.,* the Angrybirds app). The source code is still required for the NDK part of apps.

Accounting. The logs collected during an app run are post-processed for accounting. We extended Traceview [25] in Android SDK, which currently performs runtime accounting, to perform energy accounting and data presentation. We added 3K LOC to the existing 5K LOC in Traceview.

Data Presentation. *Eprof* outputs energy tuple per entity in the sorted order (with inclusive/exclusive energy for hierarchical entities). When routines are the entities, *eprof* be-

Table 2: Apps used throughout the paper.

App	Description	App	Description
Windows Mobile (on tytn2)		Android (on magic)	
sd	Skin Detection [26]	syncdroid	Mobile file sync
lchess	Local Chess [27]	streamer	Photo streaming
pup	Upload photo albums	andoku	Sudoku game [28]
cchess	Cloud Chess (offload)	goOut	Location app
pdf2txt	PDF to text [29]	k9mail	Email Client
pslide	Photo Slide show	wordsrc	Game [28]
fft	speech recog. [30]	andtweet	Twitter client [28]
Android (on passion)			
browser	Google on Browser	cnn	CNN on Browser
fb	Facebook	pup	Photo uploading
ab	AngryBirds	mq	MapQuest
nyt	New York Times app	fchess	Free Chess [31]

comes a call-graph energy profiler; it mimics the output of *gprof* [3] by replacing each time value with a (time, energy) value tuple. It also outputs a breakdown of the total energy consumed into per-component energy consumption.

6. Evaluation

In this section, we compare *eprof*'s accuracy with previous accounting approaches and measure its overhead.

Applications. Table 2 lists the set of 21 apps used in the rest of the paper. Some of them are among the top 10 most popular apps in Android Market while others were downloaded from several open-source projects [26–30].

6.1 Related Work: Previous Accounting Approaches

The energy accounting problem has been previously studied in different context. We summarize the two best known policies proposed: split-time and utilization-based.

The split-time energy accounting scheme simply splits the time into fine-grained time bins, and accounts the energy spent (typically obtained directly from a power meter) in a bin to the sampled running entity (process/thread/routine) in that bin. Powerscope [32, 33] measures power using an external power meter and accounts energy for mobile systems like laptops at the routine granularity using split-time accounting. Li *et al.* [34] use split-time to account OS energy on commodity hardware, using a system-wide cycle accurate power model to estimate instantaneous power consumption. Quanto [35] also uses the split-time policy to measure and account system-wide energy in sensor networks for programmer defined entities.

The recently proposed Cinder [36] and PowerTutor [6, 37] also perform smartphone energy accounting. They differ from *eprof* in several aspects. First, they support processes as the finest accounting granularity. Second, both systems use utilization-based power models to model and account energy of each component to the processes. As shown in [4], utilization-based power models do not capture asynchronous power behavior found in modern smartphones.

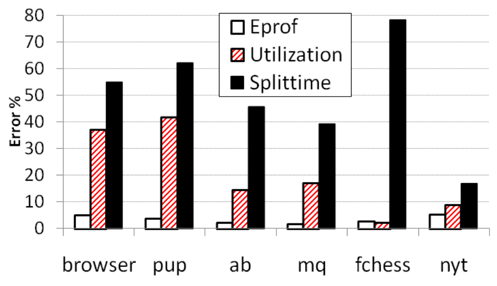


Fig. 8: Accuracy of different accounting policies.

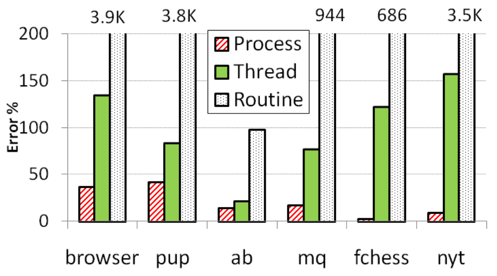


Fig. 9: Accuracy of utilization-based model at different granularities.

6.2 Accounting Accuracy

It is difficult to measure per-entity accounting accuracy since there is no easy way to measure the ground truth in the presence of asynchronous power behavior. We expect the per-entity accounting accuracy of *eprof* to be the same as that of the system-call-based power model it is based on, since the triggers for the power model, system calls, also form the finest granularity among the four program entities that *eprof* profiles (§2). To compare different accounting schemes, we compare their aggregate accounting accuracy: how does the sum of per-entity energy breakdown under different accounting schemes approximate that of the ground truth, *i.e.,* the total energy spent as measured using a power meter [38]? We define accounting "error" as the percentage difference of the sum of all entity energies except process 0 (which does not use any hardware component) with ground truth energy measured.

Figure 8 plots the accounting error of the three schemes, at the process granularity, for a few apps from Table 2 on Android on passion (results are similar for others). We see that the error in *eprof* is under 6% for all apps while that of utilization-based accounting ranges from 3% to 50% and of split-time ranges from 15% to 80%. The higher error for utilization-based accounting is a direct consequence of the error in utilization-based power models [4]. Split-time accounting, which though utilizes direct power meter readings, performs the worst since it accounts most of the energy due to asynchronous power behavior to PID 0 (the null process), which performs no hardware activity and should be attributed zero energy.

For system-wide energy accounting at the thread and the routine granularities, split-time and *eprof* report the same errors as at the process granularity, because split-time is largely oblivious to the accounting granularity as it divides the time into fixed-sized bins and accounts each bin energy to the sampled entity, and *eprof* accounts energy at the system-call level, which is finer-grained than at the routine/thread level. In contrast, utilization-based accounting shows larger error when estimating energy at finer granularities, as shown in Figure 9, since utilization-based power models incur larger errors in finer-grained estimation [4].

6.3 Logging Overhead

Measuring the logging overhead of *eprof* on the smartphone app runtime and energy consumption is tricky since smartphone apps are interactive, *i.e.,* their execution involve periods of inactivities waiting for human input. To prevent such inactivity periods from diluting the measured overhead, for each app in Table 2, we isolated its core part performed in-between human interactions in calculating the logging overhead, *e.g.,* the code in lchess that corresponds to computing each `computer_move`, in between the `moves` made by the human. The logging overhead of *eprof* falls between 2-15% for the apps on WM6 and between 4-11% for the apps on Android on the two handsets, out of which about 1-8% is due to system call tracing alone. Microbenchmarking reveals that logging each entry in *eprof* (syscall or routine) consumes $2.5\pm0.5\mu$s on passion (1GHz CPU), including $1.5\pm0.2\mu$s overhead of `getClock()`, and consumes 30μs on tytn2 (400MHz CPU) with 10μs for reading the clock. Since the logging only incurs overhead on CPU and memory, the energy overhead for logging is the runtime overhead multiplied by the CPU power, which comes down to 0.69-12.99% for the apps on WM6 and between 0.40-7.35% for the apps on Android. Finally, the logging rate (including system call tracing) for the apps varies between 60-70 KB/s.

7. Applications

We report on our experience with using *eprof* to understand the energy consumption of the 21 apps in Table 2. Due to page limit, we first briefly summarize the energy bottleneck of all the apps identified by *eprof*, and then present an in-depth analysis of the most popular 5 apps.

7.1 Identifying Energy Hotspots

Figure 10 shows the percentage time and energy of the energy hotspot routine in each of the 14 apps in Table 2, listed under WM (tytn2) and Android (magic). Already, this summary exposes several interesting observations about the energy consumption of these apps. (1) There is a stark contrast in the percentage runtime and the percentage energy drain for some of the hotspot routines, *e.g.,* goOut spends over 20% of its energy on GPS routine `attachlistener` which runs for under 3% of runtime. (2) The energy consumption behavior of two versions of the same app differ significantly. Specifically, lchess which runs purely on mobile consumes 30% of its energy in checking the human move, while cchess spends 27% energy packing and unpacking program state for offloading the computation to the cloud (as in [39, 40]). (3) The profiling results of andoku and wordsearch, each containing thousands of routines, reveal that their energy bottleneck routines are for building the UI, *i.e.,* `setTextColorView()` and `AddRow()`, respectively.

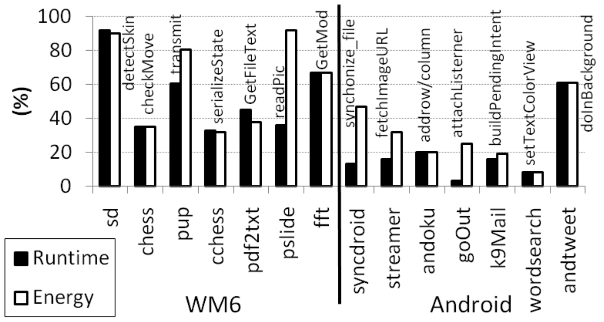


Fig. 10: Percentage runtime and energy consumption of energy hotspots.

Table 3: Session description for the apps used in case study.

App	Session Description
browser	User opens browser, performs a Google search, scrolls the HTML page and closes the app.
angrybirds	User plays a full game of AngryBirds hitting all three birds and then closes the app.
fchess	User plays two moves of chess game with computer.
nytimes	User opens the NYTimes app, app downloads and displays contents, user scrolls the front page.
mapquest	User starts app, app finds location, fetches map tiles and renders, user then clicks “gas station” button.

7.2 Case Studies

We now present an in-depth analysis of 5 popular apps running on Android on passion. All the apps were run on 3G; we skip the WiFi runs due to page limit. Table 3 describes the session scenario of each app used in the case study. Table 4 summarizes the statistics of the profiling runs and where most of the energy is spent in these apps as identified by *eprof*. It shows that running these apps for about half a minute can invoke 29–47 threads, many of which are third-party modules, and 200K–6M routine calls. The complexity of these apps is daunting; without *eprof*, it would be difficult to understand their energy profile. Overall, the about 30-second run of these apps drain 0.35%-0.75% of a full battery charge, a rate which could discharge the entire battery in a couple of hours.

7.2.1 Android Browser – Google Search vs. CNN

Google search. The Android browser comes with Android and is arguably one of the most frequently used apps on Android. We first profiled a 30-second run of the browser for one dominant usage: Google search, where the user opens the browser, performs a Google search over 3G, and closes

Table 4: Summary of energy drain of 5 popular apps.

App	Run-time	#Routine calls (#Threads)	% Battery	3rd-Party Modules Used	*Where is the energy spent inside an app?*
browser	30s	1M (34)	0.35%	-	38% HTTP; 5% GUI; 16% user tracking; 25% TCP cond.
angrybirds	28s	200K (47)	0.37%	Flurry[7],Khronos[41]	20% game rendering; 45% user tracking; 28% TCP cond.
fchess	33s	742K (37)	0.60%	AdWhirl[42]	50% advertisement; 20% GUI; 20% AI; 2% screen touch
nytimes	41s	7.4M (29)	0.75%	Flurry[7],JSON[43]	65% database building; 15% user tracking; 18% TCP cond.
mapquest	29s	6M (43)	0.60%	SHW[44],AOL,JSON[43]	28% map tracking; 20% map download; 27% rendering

the browser. The Google search page triggers the GPS to determine user location. The browser process consumes a total of 2000 μAH out of which about 53%, 31%, and 16% are spent in CPU, 3G, and GPS, respectively.

The browser forks a total of 34 threads, including 4 http worker threads, a main thread, and a Webviewcore thread besides GC (garbage collector), DNS resolver, and other threads. Less than 500KB of data is transfered over 3G. Figure 11(a) plots the split of the total browser energy among different threads with each thread's energy consumption further split by phone components. We gain the following insight into how the energy is spent in the browser. (1) Thread http0 consumes the most energy (28%), 24% of which is spent in 3G tail. This thread performs the bulk of http I/O (request and response). Thread http1 consumes another 10% energy. Together, the two http threads consume 38% energy. (2) Two generic Android threads, HeapWorker and IdleReaper, consume 14% and 10% energy respectively. Most of their energy are spent in 3G tails as follows. IdleReaper reaps idle TCP connections after a configured timeout, each of which leads to a 3G tail. HeapWorker cleans up each network connection upon app exit by sending a TCP FIN packet, which also often leads to an isolated 3G tail. The two threads are used in any apps that access the web, and we term them *TCP conditioning* utilities. (3) Threads main and Webviewcore are responsible for loading the browser and building its GUI. The main thread consumes 10% energy which is entirely CPU. Webviewcore, which also starts GPS to track user location, consumes 24% of the total energy, with 11% and 5% spent in GPS and GPS tails, respectively. Webviewcore spends most of its energy (24%) in routine `JavaWebCoreJavaBridge.handleMsg()` (18%).

To understand where the energy is spent at the routine level, we plot in Figure 11(b) *per-routine* energy breakdown for a few selected routines. The energy includes that of callee routines to better capture the whole function performed by the routine. The per-routine profiling clearly shows the energy breakdown among the 3 major steps of a Google search. (1) Routine `android/net/http/Connection.processRequests()` which processes network requests on behalf of the browser and hence involves networking, consumes 35% of the browser energy (7% in CPU for processing http). (2) Processing compressed http response after downloading consumes 15% energy, out of which 5% is spent in decompressing the compressed html response (routine `java/util/zip/GZIPInputStream.read()`). (3) Routines from class `android/view/ViewRoot.java` which renders GUI consume about 5% energy.

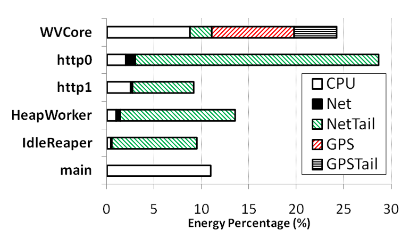
Browsing a CNN page. When the user surfs CNN, the browser spawns 30 threads, and consumes a total of 2400 μAH out of which about 40%, 60%, and 0% are spent in CPU, 3G and GPS, respectively. Figures 12(a)-12(b) again plot the per-thread and per-routine energy split, which draw contrast with the Google search scenario. (1) Surfing the CNN page results in higher data download (1200 KB) and invokes four different http threads to share downloading and parsing, which consume 26%, 9%, 11% and 8% energy, respectively, for a total of 54%, higher than the 38% by http0 and http1 in Google search. (2) Thread IdleReaper, which reaps idle TCP connections through routine `IdleCache.IdleReaper.run()`, consumes more energy (15%) than in Google search due to reaping more sockets. (3) Webviewcore consumes only 10% energy in CPU, as it no longer starts GPS to track user location.

These profiling results of the Android browser suggest that TCP *conditioning* (reaping and proper shutdown) over 3G can waste significant energy in 3G tails. We discuss strategies to reduce this energy drain in §8.3.

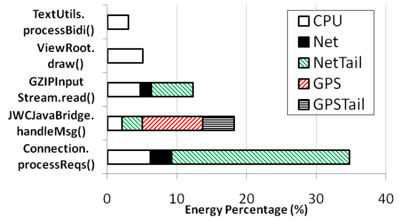
7.2.2 AngryBirds

We next profiled one of the most popular smartphone games, downloaded over 50M times from Android Market, angrybirds. In the profile run, the user plays a single instance of the game over 3G, and the app spawns 35 threads. The "GLThread" thread handles gameplay and the touch events, and invokes the third-party Khronos EGL interface [41] to paint the screen for game events. It also comes bundled with Flurry [7], a third-party mobile data aggregator and ad generator. Flurry runs as a separate thread, collects various statistics about the phone including its location, OS, and software version, and uploads the data to its server. Later, it downloads and renders ads during gameplay.

Figures 13(a)-13(b) show the energy breakdown of the top 5 threads and routines, which provides the following insight. (1) The core part of the app, thread GLThread, though CPU intensive, consumes only 18% of the total app energy. Within the thread, the Khronos API consumes 9% energy over 1K calls made to the API routine, and the rovio renderer spends another 9% energy in over 1K calls. Rendering the ad consumes 1% energy. (2) The Flurry thread consumes most of the energy (45%). Within the thread, GPS location tracking consumes 15% energy and its tail consumes addi-

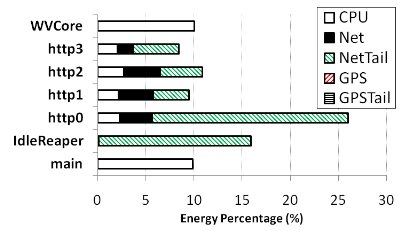


(a) Per-thread

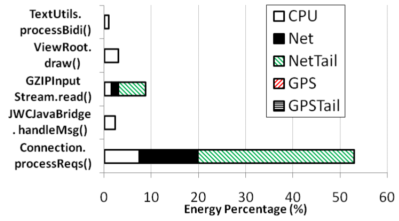


(b) Per-routine

Fig. 11: Google search on browser.

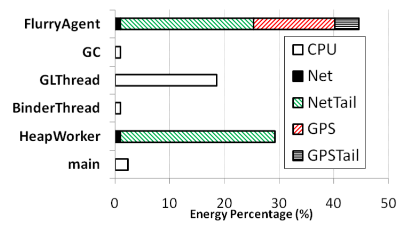


(a) Per-thread

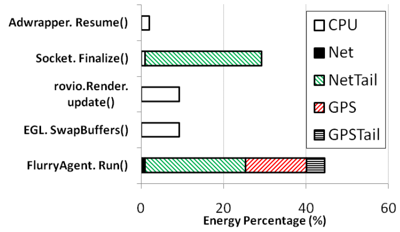


(b) Per-routine

Fig. 12: CNN on browser.

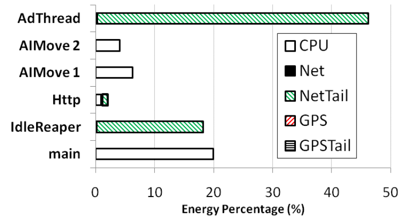


(a) Per-thread

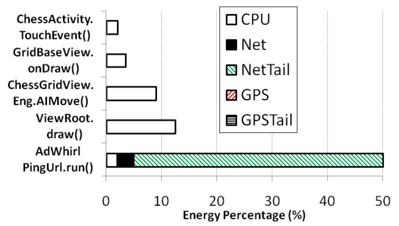


(b) Per-routine

Fig. 13: AngryBirds.

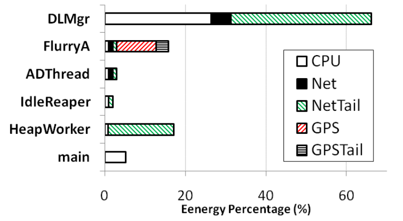


(a) Per-thread

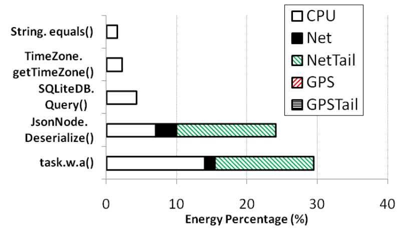


(b) Per-routine

Fig. 14: Free Chess.

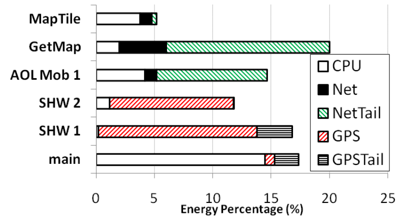


(a) Per-thread

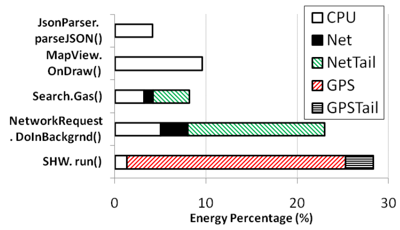


(b) Per-routine

Fig. 15: NYTimes.



(a) Per-thread



(b) Per-routine

Fig. 16: MapQuest.

tional 4% energy; collecting the handset information consumes less than 1% energy (CPU only); uploading the information and downloading the ads consume 1% energy with only under 2KB data transfered over 3G; but the 3G tail consumes 24% energy. (3) When the app is closed, thread HeapWorker performs cleanup, closing an unclosed socket as part of the finalize method (Figure 13(b)), which creates a 3G tail consuming 28% of the app energy.

7.2.3 Free Chess

We next profiled the most popular free chess game [31] on Android Market, downloaded over 10M times. Like angrybirds, this app downloads ads over 3G which consumes most of its energy. It spawns 37 threads during the 33-second profile run. The main thread is responsible for the gameplay, AdThread fetches ads over the network, and IdleReaper reaps remote server TCP connections after timeout.

Figures 14(a)-14(b) show a clear four-way energy breakdown. (1) AdThread which runs third-party AdLibrary AdWhirl [42] through routine `com/adwhirl/PingUrl.run()`, consumes 50% energy, almost entirely spent in 3G tail. (2) The main thread which paints the board consumes only 20% energy entirely in CPU through routines `android/view/ViewRoot.draw()` and `uk/co/aifactory/fireballUI/GridBaseView.onDraw()`. The user plays 2 moves which are responded by the computer's AIMoves. (3) The AIMoves are computed through two different threads (AIMove1 and AIMove2), each calling routine `uk/co/aifactory/chessfree`

/ChessGridView.Eng.AIMove(), consuming a total of 10% energy. (4) IdleReaper consumes 18% energy, again almost entirely in 3G tail.

The above energy profiling provides an important insight: free apps like fchess and angrybirds spend under 25-35% of their energy on gameplay, but over 65-75% on user tracking, uploading user information, and downloading ads.

7.2.4 NYTimes

We next profiled the Android app nytimes which has been downloaded over 10M times and is representative of the family of publisher provided viewing apps. The app spawns 29 threads during the profile run to fetch news and display the news. It uses Proguard [45] to obfuscate its class and method names. As a result, understanding *eprof* output was slightly complicated.

Figure 15(a) shows a clear four-way energy breakdown. (1) The main thread which activates GUI and displays the news downloaded, consumes only 5.2% energy. (2) The DownloadManager thread consumes the bulk of the app energy (65%). It downloads about 1MB of data over 3G and stores it in a local SQL database. Interestingly, we observe after the main thread finished displaying the news, until when the app consumed only 25% of its total energy, DownloadManager continues to utilize CPU and network, draining the remaining 75% energy. (3) Like angrybirds, nytimes also runs Flurry consuming 16% of the app energy. (4) Heapworker consumes 15% energy, again mostly in 3G tail.

Figure 15(b) shows the energy split for the top 3 energy consuming routines inside DownloadManager. The app spends 30% of its energy in routine task.w.a(), which has an obfuscated name and hence we could not infer its function, 24% in deserializing the fetched content (Jackson JSON library), and 7% in the SQL database.

7.2.5 MapQuest

Finally we profiled the MapQuest location tracking app, which is representative of the family of location-oriented search apps. Upon starting, the app locates user location using the third-party SkyhookWireless (SHW) [44] engine, downloads and deserializes (using Jackson JSON [43]) map tiles, and renders the map. The user then searches for gas stations nearby. The app consumes a total of 3600 μAH energy, split as 28%, 42%, and 30% among CPU, 3G and GPS, respectively. Figures 16(a)-16(b) show that SHW consumes 29% energy via two threads through routine SkyHook.run(), the main thread consumes 18% energy performing GUI and map rendering (via routine MapView.OnDraw() and JSON parsing), and routine search.gas(), invoked when the user clicks the gas station search button, consumes 8% of the app energy, 4% of which is spent in its own 3G tail.

The energy breakdown reveals that the ratios of 3G and GPS energy over their tails differ drastically: 3G spends 82% in its tail while GPS spends only 15% in its tail. The cause of such different tail energy footprint is the way these components are used. GPS is used for continuous tracking and is typically turned on once to start tracking, and turned off to stop tracking, generating one GPS tail. Network transfers are often performed via intermittent sending/receiving small amount of data, incurring many tail periods in between.

7.3 Detecting Energy Bugs

We show how *eprof* helps to find an instance of the class of wakelock energy bugs [9] in FaceBook (FB). As discussed in §3, apps with background services typically use the wakelock acquire/release APIs exposed by the smartphone OS to keep the phone awake, *e.g.,* to perform intermittent I/O activities. A wakelock energy bug happens when a wakelock is held longer than necessary due to a missing lock release.

facebook.katana.HomeActivity is one of the main activities of the FB app. In a typical run of the app, the user launches the app, HomeActivity downloads and displays the FB home page, while the user navigates. When using *eprof* to profile a 30-second run of the FB app (v1.3.0, released Oct 2010), which spawned 50 threads, including background services, with over 2M routing calls, and consumed a total of 1200 μAH energy, we observed from the per-routine profiling output of *eprof* that routine com/facebook/katana/service/FacebookService.onStart() which starts the background service consumed 25% of the app energy, out of which 18% was attributed to routine com/facebook/katana/binding/AppSession.acquireWakeLock(). This much energy due to a wakelock is suspiciously high and is typically a symptom of wakelock bugs. A close look at the call-graph output of *eprof* shows the service routine never called the release API to release the wakelock until the app completion. Apparently the wakelock held by the app continued to drain power even after the app termination, by not allowing the CPU to sleep.

We decompiled the FB installer to Java source code using ded [46], and confirmed that indeed the said routine acquired the wakelock and never released the wakelock due to a programming error. FB fixed the bug in its next release (v1.3.1) which we verified as by inserting a release call of the wakelock as indicated by *eprof*.

8. Optimizing I/O Energy using Bundles

Our experience with profiling popular apps using *eprof* reveals several key observations about the energy consumption of modern smartphone apps. The observations motivate us to propose a new, aggregate accounting presentation called I/O *energy bundle*, which is at a higher level than the default per-entity output of *eprof*, yet more concisely captures *where the energy is spent in a smartphone app and more importantly, why?* Such a presentation offers more direct help to the developer in optimizing the app energy.

8.1 Observations

Our extensive experience with profiling popular apps using *eprof* in §7 reveals the following key observations.

(1) I/O consumes the most energy. Most of the energy in an app is spent in accessing I/O components, and tail energy

Table 5: Energy breakdown summary per app.

App	Total I/O Energy	Bundles	#I/O Routines /total routines
Handset:tytn2 running WM6.5			
pslide	92%	3 (3 Disk)	2/21
pup	57%	3 (3 NET)	3/32
Handset:magic running Android			
syncdroid	50%	4 (1 NET, 3 DISK)	8/0.9K
streamer	31%	3 (3 NET)	4/1.1K
Handset:passion running Android			
browser	69%	3 (2 Net, 1 GPS)	5/3.4K
angrybirds	80%	4 (3 NET, 1 GPS)	5/2.2K
fchess	75%	2 (2 NET)	7/3.7K
nytimes	67%	2 (1 NET, 1 GPS)	16/6.8K
mapquest	72%	3 (2 NET, 1 GPS)	14/7.1K
pup	70%	1 (1 NET)	3/1.1K

typically accounts for the largest fraction of the I/O energy. CPU consumes a small fraction of the app energy, most of which is spent in building up the GUI of the app. The second column of Table 5 shows that most apps spend 50-90% of their energy in I/O.

(2) I/O energy is spent in a few bundles. We observe that apps typically consume I/O energy in a few, distinct lumps. Within each lump, an I/O component actively and continuously consumes power, *i.e.,* it stays in a high power state or the tail power state. For example, Figure 2 shows a lump which consists of several network events – a connect and 5 sends which together drive the 3G FSM from the base state to active states, and back to the base state. The 3G energy spent in the lump consists of ramp-up energy (for connect), energy consumed for TCP handshake and sends, and tail energy. Similarly, in browser performing a Google search (§7.2), there are two overlapping I/O lumps, one of 3G consisting of network connects and sends by the http threads, and the other of GPS consisting of GPS start/stop.

We define an I/O *energy bundle* as a continuous period of an I/O component actively consuming power, which corresponds to the duration in traversing from one instance of the base power state to the next in the component's power FSM. Table 5 (third column) shows that the high I/O energy of apps is typically spread across very few (1 to 4) bundles.

(3) Very few routines perform I/O. We further observe a stark contrast between the way the CPU and I/O components are utilized by smartphone apps: CPU usage is typically split between thousands of routines of an app, though with varying amount, whereas I/O activities arise from very few routines, called by many callers. The intuition behind this finding is that modular programming dictates implementing a few generic routines to perform I/O activities, rather than dispersing them throughout the code. For example, in event based I/O programming with *select()*, the routine containing the select loop performs nearly all the network I/O of the app. In MapQuest, routine `runRequest()` in `com/mapquest/android/util/HttpUtil.java` performs all the HTTP requests. Table 5 (last column) shows that the number of routines performing I/O versus the total number of routines called by each app (on Android this includes framework routines called by the app). We observe that very few routines, between 4 to 8, are responsible for driving I/O components. MapQuest and NYTimes show higher numbers as third-party threads perform their own I/O.

8.2 Bundle Presentation

The above three observations reveal a key insight into how energy is spent in an app: I/O energy accounts for the bulk of an app's energy, and it arises in a few bundles, each of which involves a few I/O performing routines. This insight suggests that a more direct way of helping a developer to understand and optimize the energy consumption of an app is to focus on its I/O energy bundles. We thus propose a bundle-centric accounting presentation which consists of an FSM of the I/O component for each bundle during the app execution, annotated with the relevant routines triggered during that bundle. We show in our case study below that one FSM often captures multiple occurrences of identical bundles.

The bundle presentation is generated as follows. For each bundle captured during the app execution, the productive power states of the FSM of the component are first annotated with the syscall events and hence routines that drove the FSM to those states. Since very few routines are responsible for I/O activities, it is easy to visualize this small set of routines in the annotated FSM. Next, for each instance the component spends in the tail state, we annotate the tail state with the routines called by the app during that period, including routines that use other components, usually CPU. Since the app can call several (possibly thousands) routines during a tail state, we only include the top three most time-consuming routines during the tail state.

8.3 Case Studies

Now understanding the I/O energy of an app boils down to two questions: why are there so many bundles and why is each bundle so long? We have used the bundle accounting presentation to quickly gain insights to these questions and consequently hints on how to optimize the I/O energy of nearly all the apps in Table 5. Due to page limit, we present our experience with four apps below.

8.3.1 Why is a bundle Long?

Pup. Figure 17 shows the bundle presentation for pup during a 30-second app run, which consists of a single 3G bundle that lasts 25 seconds, consuming 70% of the app energy. The bundle presentation clearly shows *why* the bundle consumes 70% energy. It shows that once one photo is sent (in Net High state), the FSM returns to the 3G tail state, during which time it reads the next photo, computes a hash for it, and again uploads it over the network. The app performs CPU computation during the 3G tail which elongates the 3G tail; the tail could have been shorter if the app uploaded the next photo sooner. Further, the above interleaving of network and computation activities happens three times. Such

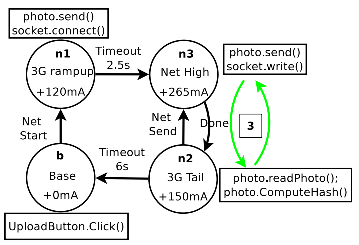


Fig. 17: Bundles in Pup.

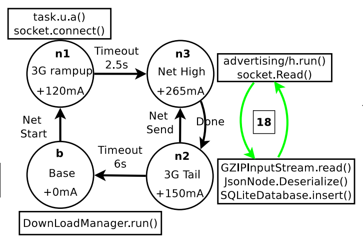


Fig. 18: Bundles in NYTimes.

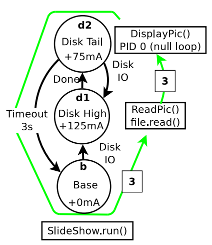


Fig. 19: Bundles in PSlide.

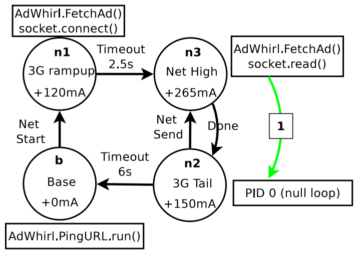


Fig. 20: A bundle in FChess.

information gives the programmer the hint that the app's I/O energy can be cut down by aggregating network activities which would reduce the three 3G tails into one.

NYTimes. Figure 18 shows the single 3G bundle of DownloadManager thread. Similarly as pup, this bundle performs periodic I/O and computation 18 times to build its database. In each iteration, it reads one chunk of data and stores it into its database after deserializing.

8.3.2 Why Are There So Many bundles ?

Pslide. Figure 19 shows three similar looking bundles during the app run. Routine `ReadPic()` reads a photo from sdcard which triggers sdcard into a high power state followed by the tail state consuming 75mA. During the tail state, the app displays the photo and sleeps for 5 seconds, during which (after 3 seconds) the FSM returns to the base state. This process is repeated three times. The bundle presentation shows that the three separate bundles waste three tail energies. The three bundles could be merged into one which incurs only one tail by aggregating the reading of sdcard photos.

FChess. Figure 20 shows the first bundle where app component Adwhirl [42] fetches ads over 3G. Once the ad is fetched and displayed, the thread goes to sleep and the 3G FSM returns to tail. The second bundle (not shown) involving IdleReaper and its 3G tail (§7.2.3) can be avoided if this thread cleans up its TCP connections.

8.3.3 Optimizing I/O Energy

The case studies above show how bundle analysis gives hints on restructuring the source code to minimize the number of bundles and the length of each bundles. For the apps for which we had source code, we reorganized the code structure by following these hints. Rerunning the restructured apps shows pslide, pup, streamer, and syncdroid reduced their total energy by 65%, 27%, 23% and 20%, respectively,

9. Related Work

Application profilers. Performance profiling is a long studied topic. Running time profiling has been proposed at the application level [3, 47, 48] to monitor the call graph trace and estimate the running time of routines, for object oriented languages [49, 50], and at the kernel level [51]. *Eprof* is concerned with profiling energy consumption which is not linear as time. Several energy profiling schemes have been proposed for desktops [34], for mobile devices [52], and for sensor networks [53]. These schemes estimate the energy consumption of a routine based on strict time boundaries of the routines and hence can incur significant error when applied to profiling smartphone apps (§6).

Characterizing smartphone energy consumption. Carroll and Heiser [54] measured the power consumed by different phone components under different application loads by hardwiring individual power meters to different phone components. Shye *et al.* [55] and Zhang *et al.* [6] built linear regression based models for modeling app level power consumption and profiled several apps including Google Map and Browser. All these work measure per-app or per-component energy drain on smartphones. *Eprof* is capable of measuring intra-app energy consumption and gives insights into energy breakdown per thread and per routine of the app.

Mobile energy optimization. Finally, a number of specialized energy saving techniques on mobiles have been proposed, *e.g.,* for specific applications on mobile systems [56, 57], for a specific protocol [58, 59], via offloading [39, 40], and via delaying communication [60]. *Eprof* is a general-purpose fine-grained energy profiler that directly assists an app developer in the app energy optimization cycle.

10. Conclusion

This paper makes three contributions towards answering the ultimate question faced by millions of smartphone users and developers today: *Where is the energy spent inside my app?* We first present *eprof*, the first fine-grained energy profiler for smartphone apps and its implementation on Android and Windows Mobile. *Eprof* adopts the last-trigger accounting policy to most intuitively capture asynchronous power behavior of modern smartphone components in mapping energy activities to the responsible program entities. We then present an extensive, in-depth study using *eprof* to gain insight of energy usage of smartphone apps using a suite of 21 apps. Finally, we propose bundles, a new presentation of energy accounting, that helps app developers to quickly understand and optimize the I/O energy drain of their apps.

Eprof opens up new avenues for studying smartphone energy consumption. It can be readily used to compare the energy efficiency of different implementations of the same app (*e.g.,* Firefox vs. the Android browser). The energy accounting engine of *eprof* can be combined with compiler techniques such as static analysis to develop energy optimizers that automate the process of restructuring app source code to reduce their energy footprint, and with the OS scheduler to develop energy-aware process scheduling algorithms.

Acknowledgments

We thank the reviewers for their helpful comments, and especially our shepherd, George Candea, whose detailed feedback significantly improved the paper and its presentation. Abhinav Pathak was supported in part by a 2011 Intel PhD Fellowship.

References

[1] "Mobile app internet recasts the software and services landscape." URL: http://tinyurl.com/5s3hhx6

[2] "Apples app store downloads top 10 billion." URL: http://www.apple.com/pr/library/2011/01/22appstore.html

[3] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *Proc. of PLDI*, 1982.

[4] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system-call tracing," in *Proc. of EuroSys*, 2011.

[5] N. Balasubramanian and et.al., "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proc of IMC*, 2009.

[6] L. Zhang and et.al., "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proc. of CODES+ISSS*, 2010.

[7] "Flurry: Mobile analytics." URL: http://www.flurry.com/

[8] "Android powermanager: Wakelocks." URL: http://developer.android.com/reference/android/os/PowerManager.html

[9] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging for smartphones: A first look at energy bugs in mobile devices," in *Proc. of Hotnets*, 2011.

[10] "Facebook 1.3 not releasing partial wake lock." URL: http://geekfor.me/news/facebook-1-3-wakelock/

[11] "Email 2.3 app keeps awake when no data connection is available." URL: http://www.google.com/support/forum/p/Google+Mobile/thread?tid=53bfe134321358e8

[12] "Email application partial wake lock." URL: http://code.google.com/p/android/issues/detail?id=9307

[13] "Using a locationlistener is generally unsafe for leaving a permanent partial_wake_lock." URL: http://code.google.com/p/android/issues/detail?id=4333

[14] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Characterizing radio resource allocation for 3g networks," in *Proc of IMC*, 2010.

[15] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya, "Virtual machine power metering and provisioning," in *Proc. of SOCC*, 2010.

[16] F. Rawson, "MEMPOWER: A simple memory power analysis tool set," *IBM Austin Research Laboratory*, 2004.

[17] M. Dong, Y. Choi, and L. Zhong, "Power modeling of graphical user interfaces on OLED displays," in *Proc. of DAC*, 2009.

[18] "perf: Linux profiling with performance counters." URL: https://perf.wiki.kernel.org/

[19] "Android debug class." URL: http://en.wikipedia.org/wiki/Nexus_One#Hardware

[20] "Android debug class." URL: http://developer.android.com/reference/android/os/Debug.html

[21] "Android ndk profiler." URL: http://code.google.com/p/android-ndk-profiler/

[22] "Cyanogenmod." URL: http://www.cyanogenmod.com/

[23] "Introducing utrace." URL: http://lwn.net/Articles/224772/

[24] "System tap." URL: http://sourceware.org/systemtap/

[25] "Profiling with traceview." URL: http://developer.android.com/guide/developing/debugging/debugging-tracing.html

[26] "Skin recognition in c#." URL: http://www.codeproject.com/KB/cs/Skin_RecC_.aspx

[27] "C# micro chess (huo chess)." URL: http://archive.msdn.microsoft.com/cshuochess

[28] "Open source Android app." URL: http://en.wikipedia.org/wiki/List_of_open_source_Android_applications

[29] "itextsharp." URL: http://itextsharp.sourceforge.net/

[30] "Exocortex.dsp: C# complex number and fft library for microsoft .net." URL: http://www.exocortex.org/dsp/

[31] "Chess free: Ai factory limited." URL: https://market.android.com/details?id=uk.co.aifactory.chessfree

[32] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proc. of WMCSA*, 1999.

[33] F. Jason and S. Mahadev, "Energy-aware adaptation for mobile applications," in *Proc. of SOSP*, 1999.

[34] T. Li and L. John, "Run-time modeling and estimation of operating system power consumption," *SIGMETRICS*, 2003.

[35] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems," in *OSDI*, 2008.

[36] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich, "Energy management in mobile devices with the Cinder operating system," in *Proc. of EuroSys*, 2011.

[37] "Power monitor for Android." URL: http://powertutor.org/

[38] "Monsoon power monitor." URL: http://www.msoon.com/LabEquipment/PowerMonitor/

[39] E. Cuervo, B. Aruna, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *MobiSys*, 2010.

[40] B.-G. Chun and P. Maniatis, "Augmented Smartphone Applications Through Clone Cloud Execution ," in *HotOs*, 2009.

[41] "Khronos: Egl interface." URL: http://www.khronos.org/

[42] "Adwhirl by admod." URL: https://www.adwhirl.com/

[43] "Jackson: Json processor." URL: http://jackson.codehaus.org/

[44] "Skyhook: Location positioning, context and intelligence." URL: http://www.skyhookwireless.com/

[45] "Android proguard." URL: http://developer.android.com/guide/developing/tools/proguard.html

[46] "Decompiling apps." URL: http://siis.cse.psu.edu/ded/

[47] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, April 1998.

[48] J. Spivey, "Fast, accurate call graph profiling," *Software: Practice and Experience*, 2004.

[49] M. Dmitriev, "Profiling Java applications using code hotswapping and dynamic call graph revelation," in *Proceedings of the 4th International Workshop on Software and Performance*. ACM, 2004, pp. 139–150.

[50] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 108–124, 1997.

[51] "Oprofile." URL: http://oprofile.sourceforge.net/news/

[52] K. Asanovic and K. Koskelin, "EProf: an energy profiler for the iPAQ," MS Thesis, MIT 2004.

[53] T. Stathopoulos, D. McIntire, and W. Kaiser, "The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes," in *IPSN*, 2008.

[54] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. of USENIX ATC*, 2010.

[55] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *Proc. of MICRO*, 2009.

[56] Y. Wang, J. Lin, M. Annavaram, Q. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh, "A framework of energy efficient mobile sensing for automatic user state recognition," in *Proc. of Mobisys*, 2009.

[57] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments," in *Proc. of Mobisys*, 2008.

[58] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta, "Wireless wakeups revisited: energy management for voip over wi-fi smartphones," in *Proc. of Mobisys*, 2007.

[59] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," in *Proc. of Mobisys*, 2011.

[60] M. Ra, J. Paek, A. Sharma, R. Govindan, M. Krieger, and M. Neely, "Energy-delay tradeoffs in smartphone applications," in *Proc. of Mobisys*, 2010.