Fine-Grained Power Modeling for Smartphones Using System Call Tracing

Abhinav Pathak
Purdue University
pathaka@purdue.edu

Y. Charlie Hu
Purdue University
ychu@purdue.edu

Ming Zhang
Microsoft Research
mzh@microsoft.com

Paramvir Bahl
Microsoft Research
bahl@microsoft.com

Yi-Min Wang
Microsoft Research
ymwang@microsoft.com

Abstract

Accurate, fine-grained online energy estimation and accounting of mobile devices such as smartphones is of critical importance to understanding and debugging the energy consumption of mobile applications. We observe that state-of-the-art, utilization-based power modeling correlates the (actual) utilization of a hardware component with its power state, and hence is insufficient in capturing several power behavior not directly related to the component utilization in modern smartphones. Such behavior arise due to various low level power optimizations programmed in the device drivers. We propose a new, system-call-based power modeling approach which gracefully encompasses both utilization-based and non-utilization-based power behavior. We present the detailed design of such a power modeling scheme and its implementation on Android and Windows Mobile. Our experimental results using a diverse set of applications confirm that the new model significantly improves the fine-grained as well as whole-application energy consumption accuracy. We further demonstrate fine-grained energy accounting enabled by such a fined-grained power model, via a manually implemented *eprof*, the energy counterpart of the classic *gprof* tool, for profiling application energy drain.

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

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

1. Introduction

Mobile devices such as smartphones provide significant convenience and capability to the users. A recent market analysis [Com] shows that the smartphone market is the fastest growing segment of the mobile phone market; in 2010 over 45.5 million people in the United States owned smartphones. Despite the incredible market penetration of smartphones, their utility has been and will remain severely limited by their battery life. As such, understanding the power consumption of applications running on mobile devices has attracted much research effort. Early research [Flinn 1999a;b, Mahesri 2005] has focused on power measurement, i.e., measuring the power consumption of the mobile device during the execution of an application using a power meter, with the goal of understanding energy consumption by individual applications. These studies directly rely on the availability of power meters and do not develop a power estimation model for use in the "wild" without a power meter.

More recent efforts have focused on developing online power models for mobile devices. Typically, during the training phase, a power consumption model is developed by running sample applications, and correlating certain application behavior, or *triggers*, with specific power states or power state transitions, of individual components or the entire system, measured using an external power meter. The generated power model during this training phase can then be used online, without any measurement from a power meter, for estimating the energy consumption in running any application. Thus such an online power model enables application developers to develop energy profiling tools to profile and consequently optimize the energy consumption of mobile applications, without the expensive power meters, much like how performance profiling enabled by *gprof* [Graham 1982] has facilitated performance optimization in the past several decades.

There are two desirable features for such an online power model: (1) It should incur low overhead in logging the triggers, so that the energy estimation based on the model can be performed online; (2) It should enable accurate fine-grained energy accounting, e.g., on a per-subroutine basis, and in the presence of multiple threads and processes. This is because as in performance profiling and optimization, the natural granularity in profiling and optimizing the energy consumption of an application is at the subroutine level.

The large body of work on power modeling on smartphones [Shye 2009, Zhang 2010], and more generally for desktops [Zeng 2002] and servers [Fan 2007, Kansal 2010], are based on the fundamental yet intuitive assumption that the (actual) utilization of a hardware component (e.g., disk, NIC) corresponds to a certain power state and the change of utilization is what triggers the power state change of that component. Consequently, their design all use the notion of utilization of a hardware component as the "trigger" in modeling power states and state transitions. The usage statistics of each hardware component (e.g. disk) are typically provided by the OS, for example, /proc on Linux.

In this paper, we make a key observation that the fundamental assumption behind utilization-based power modeling does not hold in several scenarios on smartphones, due to the increasingly sophisticated power optimization in the device drivers and OS-level power management.

- Several components (e.g., NIC, drive, and GPS) have tail power states.
- System calls that do not imply utilization (e.g., file open, close, socket close) can change power states.
- Several components (e.g. GPS, camera) do not have quantitative utilization.

An immediate consequence of these non-utilization-based power behavior of smartphone components is that while a utilization-based model may still achieve reasonable accuracies in estimating the energy consumption of whole applications due to the cancellation of per-interval estimation errors in both directions [Shye 2009], they suffer poor accuracy in fine-grained energy estimation. For example, they are incapable of modeling the energy consumption due to lingering tail power states which can last up to several seconds, far beyond the completion of the triggering subroutine. Consequently, such models cannot be used to develop profiling tools that support accurate energy accounting on a per-subroutine basis.

In this paper, we propose a new, *system-call-based power model* that overcomes the above limitations of utilization-based power modeling. Our design is motivated by the following observations. First, system calls provide the only means via which applications gain access to the hardware (I/O) components. As such their names along with the parameters give clear indication of the components and the level of utilization being requested. Hence, they already encompass the triggers used in utilization-based power modeling. Second, such a model can capture all the power behavior of I/O system calls that do not imply component workload or utilization. Finally, a system call can be naturally related back to the calling subroutine and the hosting thread and process. Together, the above observations suggest system-call-based power modeling can achieve accurate fine-gained energy estimation, and enable fine-gained energy accounting, e.g., on a per-subroutine, per-thread, and per-process basis.

The design of our system-call-based power modeling scheme consists of two major components. First, it uses Finite State Machines (FSM) to model the power states and state transitions of each component as well as the whole smartphone. Some states have constant power consumption; they capture non-utilization-based power behavior. Other states leverage a linear regression (LR) model to capture the power consumption due to system calls that generate workload. Second, it uses a carefully designed testing application suite which leverages the domain knowledge of system calls (i.e., their semantics and causal invocation ordering) to systematically uncover the FSM transition rules. We have implemented the system-call-based power modeling scheme on two smartphone OSes and validated that it significantly improves fine-grained power estimation as well as whole-application energy consumption estimation for a diverse set of applications.

In summary, this paper makes following contributions.

- We make the observation that the fundamental assumption behind utilization-based power modeling, that the power state of a component is correlated with its utilization, often does not hold on modern day smartphones, due to the increasingly sophisticated power optimizations in the device drivers. Consequently, utilization-based power models can suffer poor accuracy in fine-grained power estimation.
- We propose a new power modeling approach based on tracing system calls of the applications, which gracefully captures both utilization-based and non-utilization-based power behavior of I/O components, and hence can achieve accurate fine-grained energy estimation.
- We present the detailed design and implementation of our new approach on two smartphone operating systems, Windows Mobile and Android.
- We present experimental results that demonstrate the accuracy of our new modeling scheme using a diverse set applications. When estimating the energy consumption for 50ms time intervals, the 80th percentile intervals across the applications report an error under 10%, but vary between 16% and 52% across the applications under the utilization-based modeling. Further, the whole-application energy estimation error varies in the range of 0.2% to 3.6% under our scheme, compared to between 0.4% to 20.3% under utilization-based modeling.

- We further demonstrate fine-grained energy accounting enabled by such a fined-grained power model, via a manually implemented *eprof*, the energy counterpart of the classic *gprof* tool, for profiling application energy consumption on a per-subroutine basis.

2. Background: Power Management in Smartphones

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

Each component can be in several operating modes, each draining a different amount of power. We call such different modes different *power states* for that component. We note that in our work and in all previous work, power modeling is concerned with estimating the power consumption of the whole phone as the components switch between operating modes, not when they are turned off by the user or the OS power manager, i.e., optimizations of their sleep-wakeup cycles. The later cases can be easily incorporated into energy accounting. In principle, the power state of the component should simply correspond to the throughput of its work done, i.e., the actual utilization of the component. For example, we observed that for a fixed channel condition, the WiFi NICs of the smartphones studied in this paper transmit at 5.5 Mbps at a lower power state, but may switch to a higher power state in order to transmit at 11 Mbps. We denote this assumption as the *utilization-power-state correlation assumption*.

However, as the device drivers in modern day smartphone operating systems incorporate more and more power management "smarts", the above simple assumption on the tight correlation between the utilization of a component and its power state often does not hold. In particular, the power state of a component could potentially depend on non-utilization-based factors. These include external conditions (e.g., signal strength for WiFi and Cellular) and semantics of system calls such as initiating a component or terminating the usage of a component (e.g., closing a socket). Using these factors as input, several power saving optimizations can be programmed in a component's device driver which decides the component's power transition rules. For example, the device drivers for the wireless NICs on the smartphones we have studied adjust the transmission power when the signal strength changes. In principle, all the information about power states and transition rules of a component can be uncovered from reading the device drivers of the component. However, most of the device drivers for smartphones, including Android handsets, are proprietary.

3. State-of-the-Art: Utilization-based Modeling

The large body of work on power modeling for desktops, servers, and more recently for smartphones are based on the *utilization-power-state correlation assumption* stated in Section 2. Consequently, their designs all use the notion of utilization of a hardware component as the "trigger" in modeling power states and state transitions. We call this class of modeling as *utilization-based modeling*.

Early utilization-based models focused on estimating the power consumption of individual components, using the corresponding performance counters, e.g., CPU [Bellosa 2000, Snowdon 2009, Stanley-Marbell 2001, Tiwari 1996], memory [Rawson 2004], disk [Zedlewski 2003], and of the entire system [Bircher 2007, Flinn 1999b]. These models do not relate the power consumption of the system with the applications. More recent utilization-based power models tried to estimate the power consumption of applications running on desktops [Zeng 2002], sensor nodes [Shnayder 2004], virtual machines [Kansal 2010], data center servers [Fan 2007], and most recently, mobile devices [Shye 2009, Zhang 2010].

There are two key ingredients in constructing a power estimation model in a utilization-based power modeling scheme. First, during the training phase, it collects the utilization of individual hardware components, typically via OS-provided utilization statistics (e.g. usage statistics in /proc in Linux in [Shye 2009, Zhang 2010] for power modeling in smartphones) while running some sample applications, and measures the corresponding power consumption of those components, e.g., using power meters. Second, it typically develops an LR model to correlate the sampled trigger values, i.e., the utilization, and the measured power consumption at that sampling moment (e.g. [Shye 2009, Zhang 2010]). Once the model is constructed, it can be used to perform online estimation of power consumption, by continuously collecting the utilization of the components and feeding them into the model.

4. New Challenges

Utilization-based power modeling, however, cannot handle the following non-utilization-based power behavior of smartphone components.

Several components have tail power states. Components such as NICs, sdcard and GPS on many smartphones exhibit the so-called "tail" power state phenomenon; they stay at high power state for a period of time after active I/O activities. Figure 1(a) shows after a read (same is true for write) system call of 10 bytes which sent the sdcard to high power state and lasted about 10 milliseconds, the sdcard stays in high power state for 5 seconds, on the HTC Touch Pro phone (touch for short) running Windows Mobile 6.1 (WM6 for

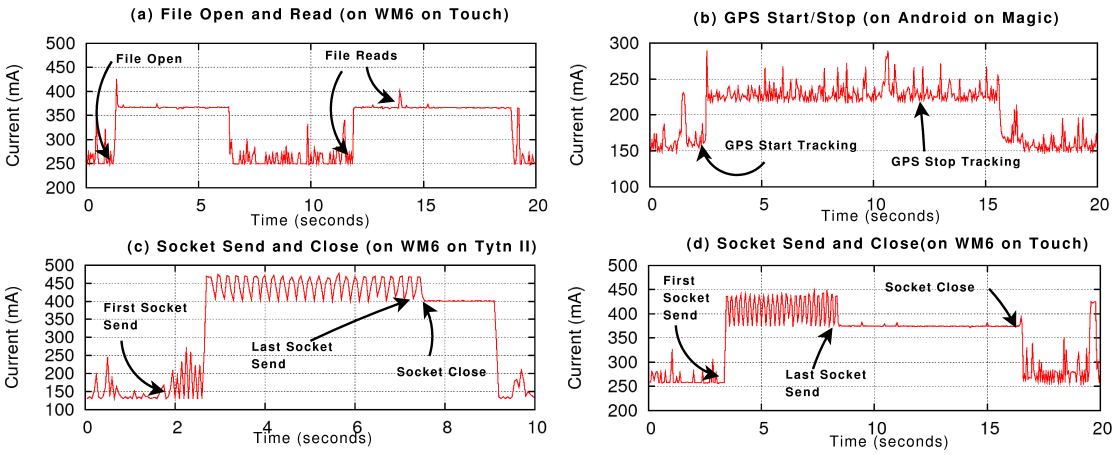

Figure 1: System call based power transitions: Top row: Disk on touch , GPS on magic , bottom row: network on touch and tytn2.

short). Similar tail power state lasts for 3 seconds on the HTC Tytn 2 (tytn2 for short) phone running WM6. On all the smartphones (WM6 on touch and tytn2, and Android on HTC Magic (magic for short)) we have tested, the NIC continues to be in high power state for a few seconds after an active send/recv is completed. Figure 1(c) shows the tail power state lasts about 1.7 second on the tytn2 phone. Clearly the utilization of the components (NIC or sdcard) is zero during the tail power states. This breaks the fundamental assumption behind utilization-based power models that the usage of a components determines its power state.

System calls that do not imply utilization can change power states. Many systems calls that do not imply high utilization of components can send the components to high or low power states. This could be due to power optimizations programmed in device drivers. For example, on windows mobile, file open, close, create, delete system calls trigger a power level change in the sdcard device driver; after these calls the component remains in a high power state for a few seconds. Figure 1(a) gives the example of file open on the touch phone. Similarly, a socket close system call immediately ends the high tail power state of the NIC on windows mobile touch smartphones, shown in Figure 1(d). Such low level power optimizations are typically done in the device drivers and they also break the utilization-power-state correlation assumption in utilization-based power models.

Several components do not have quantitative utilization. Several "exotic" components on smartphones, such as GPS and camera, do not have a notion of quantitative utilization, that is parallel to the amount of data sent or received by a NIC. These components typically are turned on or off by system calls. For example, the "requestLocationUpdate" call in Android sets GPS in a high power state, shown in Figure 1(b). The "opencamerahardware" system call in Android sets camera in a high power state. In principle, one can add a counter (e.g. to /proc) to record the binary state of these components to facilitate utilization-based power modeling of these components. However, since utilization-based modeling fundamentally assumes periodic sampling of performance counters, it can suffer delay in reading the change of power states of such components.

An immediate consequence of these limitations is that utilization-based modeling can suffer poor accuracy in fine-grained energy estimation, i.e., for small intervals such as the duration of subroutines in an application, for two reasons. First, utilization-based modeling relies on periodic sampling of the usage counters (e.g. reading /proc). The time interval at which the sampling is performed can be too course-grained compared to the duration of subroutines, or the sampling can become costly if done at a very fine granularity. Second, more importantly, a tail power state triggered by a system call in a subroutine can last till long after that subroutine has returned. In addition, using a diverse set of applications we show in Section 7 that the error in per-interval power estimation can add up and lead to poor accuracy in whole-application energy consumption estimation for many applications.

5. System-Call-Based Power Modeling

5.1 Key Idea

In this paper, we propose *system-call-based power modeling* which overcomes the above limitations of utilization-based power modeling. Our new approach is based on the following five observations:

- System calls provide the only means via which applications gain access to the hardware (I/O) components. As such their names along with the parameters give clear indication of which components and what level of utilization are being requested. Hence, they already encompass the triggers used in utilization-based power modeling. In fact, the utilization statistics used in utilization-based power modeling such as for network and disk activities in

/proc in Linux are exactly updated based on the parameters in selected system calls such as read/write. To model CPU power consumption, we log context switch events in the kernel.

- Using system calls as triggers to power state transitions naturally solves the second limitation of utilization-based power modeling. Those system calls that do not imply utilization but trigger power state changes can be identified during the training phase and incorporated in the power model.
- Similarly, the invocation of system calls that turn on and off "exotic" components immediately triggers power state change for those components, which avoids the delay due to periodic sampling of performance counters in utilization-based modeling.
- Using system calls as triggers naturally suggests using a Finite State Machine (FSM) to model the state transitions. The states in an FSM can be easily annotated with the timing and workload of recent events to accurately model state transitions due to non-utilization-based power behavior (e.g. tail states) and accumulated-utilization-based power behavior (e.g., transmitting enough packets causes the WiFi NIC to go to a higher power state to increase the bitrate.)
- System calls can be easily related back to the calling subroutine, the hosting thread and process. This, combined with the above observations that they capture non-utilization-based power behavior, enables fine-grained energy accounting, e.g., on a per-subroutine, per-thread, or per-process basis.

In the following, we first analyze the power states and state transitions involved in a single system call to motivate the Finite State Machine (FSM) implementation of system-call-based power modeling. To systematically uncover the FSM of power states for a given phone, we develop the CTester application suite, based on the domain knowledge of system calls for each OS, to automate the construction of the FSM in three incremental steps. Step 1 uncovers the FSM for individual system calls for a component. Steps 2 and 3 then exercise superposition of different power states for one and multiple components, respectively.

5.2 Modeling Single System Call Power Consumption

The actual mechanism of system-call-based modeling is motivated by the typical sequence of power states and state transitions involved in a single system call.

5.2.1 Power Consumption Behavior of System Calls

The first column of Figure 2 plots the actual power levels of a smartphone, measured using a power meter, from the start of a typical system call and during the corresponding power states and state transitions resulted. The three rows are for a single disk read, a network send with a few packets, and a network send with many packets, respectively. A system call that specifies a certain amount of workload, e.g., sending X bytes to the NIC, can send the component into one of several possible base power states. For example, on tytn2, touch and magic, sending a few packets within a short period sends the NIC to a lower power base state, while sending many packets, via one or multiple back-to-back send system calls, sends the NIC to a high power base state. The workload specified by a system call eventually is carried out by the component via a sequence of atomic I/O operations, e.g., a sequence of packet transmissions by the NIC, each corresponding to an instantaneous burst of power consumption from the current base power state.

5.2.2 Abstraction of Power States

Since a system call is often turned into a sequence of low-level hardware component operations, e.g., packet transmissions, the key to developing a system-call-based power model which only uses system call information as input is to devise power states that abstract away the low-level power behavior, e.g., in between individual packet transmissions, yet still capture the power consumption of that component due to the system calls.

We denote the power state spanning the duration covering the sequence of the component-level I/O operations due to a system call a *productive power state* of that component. Such a productive power state is characterized by the burst power of individual I/O operations, and the duration of the sequence of I/O operations. Each component can potentially have multiple such productive states. Since the duration of a productive state depends on the workload specified in the input parameter of the system call, and the power bursts due to individual packet transmissions cannot be exactly captured (drivers are closed-source), we develop an LR-based power model for that *productive state* which correlates the input workload (e.g., bytes to be sent) with the total duration of the power burst for that system call. We note this usage of LR is different from in utilization-based power modeling (e.g. [Shye 2009]) where LR is used to estimate the power drain of the whole phone.

Next, we observe after a productive power state, i.e., when there are no more I/O operations, a component can stay at the base power of the productive power state for a period of time. This is previously known for cellular NICs [Balasubramanian 2009], but we discovered it can also hold true for disks and GPS on smartphones. Since the component is not doing useful work, we call this state the *tail power state*. A tail power state is characterized by its base power and its duration.

Continuing on our example in Figure 2, the three figures in the middle column depict the abstract productive and tail power states and the transitions that model the measured power states and transitions in the first column. The above abstraction of two (types of) power states and state transitions triggered by system calls is general in that it cap-

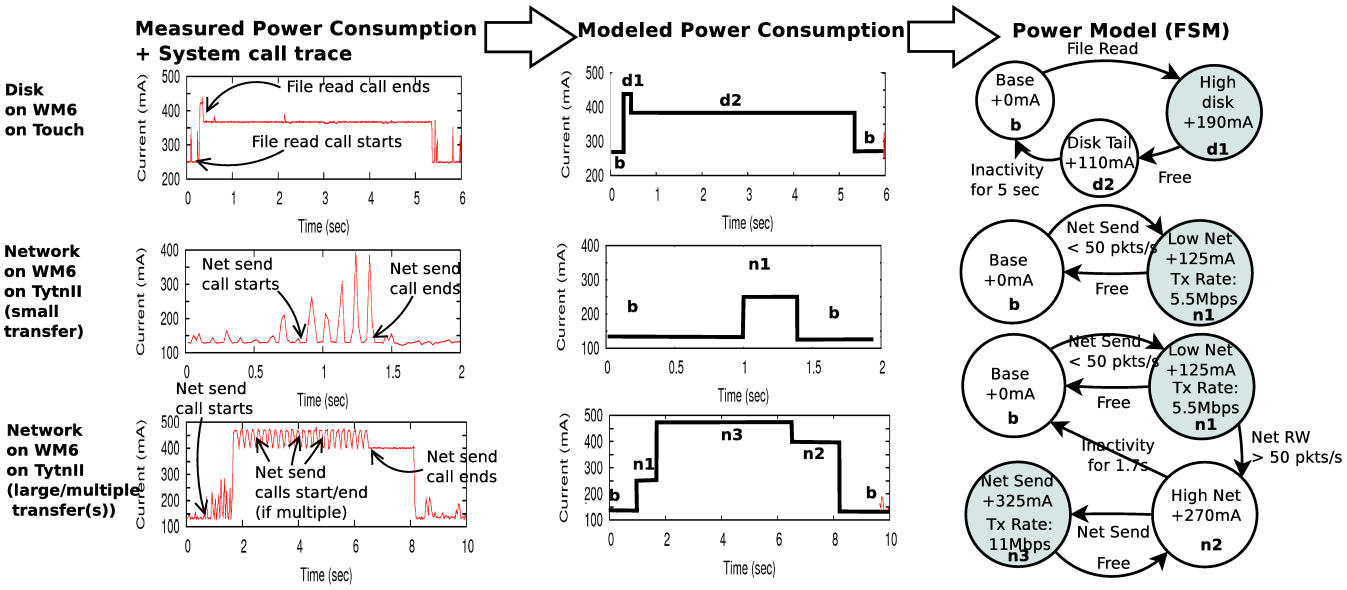

Figure 2: Modeling power states following a system call.

tures that different I/O system calls can have different duration/power in either state, or may experience only one of the two states. Finally, we observe that even consecutive burst powers during a productive power state can fluctuate, as observed in other work [Bellosa 2000] as well. We suspect this is largely due to the sampling nature of power measurement by the power meter. We calculate the average of all the burst powers measured during the training phase and use it as the burst power of that productive state.

5.2.3 System-Call-based Power Modeling using FSMs

The above abstraction of power states, which captures the power behavior of individual I/O system calls, naturally suggests a Finite State Machine (FSM)-based implementation to capture the transitions between the power states. Formally, in our FSM-based power model, each state represents a power state of a component, or of the set of all components when extended to model the power of the entire smartphone as discussed below. Each state is annotated with a (power, timeout duration) tuple, and the timing and workload of recent events of the component. The transitions between states capture the conditions that trigger state changes. There are three types of conditions: a timeout activity (e.g., the duration of the tail power state is over), a new system call, or a combination of timeout and past history of device utilization. As an example for the third condition above, a new send system call of fewer than 50 packets only takes the NIC to a low productive power state on WM6 on tytn2. A subsequent system call that together with previous send system call generates more than 50 packets per second will cause the NIC to enter a higher productive state followed by a tail state. To capture this history information of device utilization, at each state in the FSM, we need to keep information of all system calls in the recent past. We find in our experiments with two operating systems and three smartphone handsets that storing system call information up to the past 60 seconds while staying in a power state is sufficient to capture all state transition conditions from that state due to recent device utilization.

Continuing our example in Figure 2, the last column shows the three FSMs that model the power states and state transitions of the three system calls, respectively. Unless otherwise stated, all power measures in FSMs in the rest of paper refer to the additional power consumption on top of an idle phone, i.e., with no application activities. For simplicity, we represent power consumed at an instant using the current drawn by the phone in milliAmperes. The actual power consumed would be the current drawn multiplied by 3.7V, the voltage supply of the battery. Similarly, energy is reported in uAH (micro Ampere Hours), and the actual energy would be the uAH value multiplied by 3.7V. These metrics are used since smartphone batteries are rated using these metrics and hence is easy to correlate.

5.3 Modeling Multiple System Calls

Once the FSMs for individual system calls are generated for a component, we systematically integrate them to model the power consumption when there are multiple concurrent system calls to the same component. Concurrent system calls here are defined as where the second system call is invoked before the component is out of the productive or tail state due to the first system call. Concurrent system calls can be issued from the same process or from concurrently running processes. For example, when multiple processes are running, a system call causes the context switch from the calling process (which is now blocked) to another process,

which then issues another system call before the first one is completed.

We first observe that since a component can only be in a small number of possible power states, taking the union of all power states discovered by modeling individual system calls is sufficient to discover all possible power states for that component. We next focus on modeling state transitions.

There are two possible timings in which two concurrent system calls can arrive. In the first case, a subsequent system call arrives after the previous one is out of its productive power state (it could still be in the subsequent tail power state). In this case, the resulting power behavior when the second system call arrives can be modeled by simply superimposing the FSM of the second system call on top of the first, i.e., taking the maximum of the power states due to them for the overlapping time period.

In the second case, the second system call arrives while the first is still in its productive state. Since a productive state models the power consumption when the component is performing actual I/O operations, if the second system call is a workload-based system call, e.g., read/write, the effect on the component power behavior will be as if the first system call was invoked with the total workload of both system calls. If the second call is an initialization-based system call, e.g., open, the component will first come out of the productive state of the previous read/write, enters the tail state of that read/write, and then starts the FSM due to the open system call. From this time on, the FSM of the open call will be superimposed on the tail state of the previous call.

5.4 Modeling Multiple Components

After we generate the FSM models for the individual components, we develop one FSM model for the entire smartphone, by driving all components simultaneously. One may expect the total power consumption of multiple components to be a simple summation of those of individual components when active in isolation. Our experiments show that this was indeed the case on Android, but not on Windows Mobile. In particular, the tail states of different components interfere with each other on Windows Mobile. The basic idea behind combining FSMs of different smartphone components is to try out all possible combinations of the sets of conditions, each set for driving one component into all possible states of that component, and measure the corresponding total power consumption. This process is automated via the CTester application suite described later.

Complexity. While the above approach can result in a combinatorial number of power states and testing runs, in practice it remains practical. First, the major components we tested (CPU, disk, network, GPS, camera) typically have only up to three power states each (one or two productive states and one tail state), and hence even the total number of combinations is still under 20 for the three major components (CPU, disk, network). Second, since energy modeling is a one-time procedure per smartphone per OS, it is acceptable for this procedure to take some time.

Completeness of the methodology. We expect the methodology above can capture all the possible states and transition conditions in practice. All the power states are likely to be accounted for as each component can have a finite number of power states and our CTester application suite exercises all combinations of the states across components. Furthermore, we run the CTester application suite multiple times to ensure the the derived FSMs are consistent. This repeatability serves as an assurance of the completeness of the FSM power model. Finally, we validate the derived FSM power model by performing fine-grained energy estimation against actual measurement from a power meter, using a diverse set of applications (Section 7.3).

We note the above completeness issue also exists in utilization-based modeling. Since our methodology for constructing the FSM-based power model offers a systematic way of searching for all possible states and state transitions, it is more likely to result in a more complete model than utilization-based modeling, which is typically based on trying random sample applications (e.g. [Shye 2009]).

5.5 The CTester Applications

To support the above methodology for uncovering the power states and state transitions of various components, we design a testing benchmark suite, called CTester, which includes an application for each component carefully designed to exercise all the relevant system calls, and a wrapper application that invokes individual applications at predetermined timing to create scenarios of concurrent system calls on the same or multiple components. During a CTester application run, we measure the power dissipated through an external power meter which reports fine-grained power consumption of the smartphone. The difference from the base power when the phone is in idle state is the extra power due to running the application.

Creating the benchmark applications requires the knowledge of all the possible system calls and their ordering (e.g. a read call cannot proceed an open system call) through which applications or the kernel can access a component. The first step is to classify all I/O system calls into two categories: initialization-based which includes calls that initialize or uninitialize a component such as file open, close, create, delete, and workload-based which includes calls that generate the actual workload to the component, such as file read and write. Next, for each component, we develop a testing application to exercise the relevant system calls by interleaving initialization and workload system calls. Figure 3 shows the generic structure of a CTester application for one component which consists of a sequence of initialization (e.g. file open/close) and workload (e.g. send/recv) system calls, which respects their correct usage ordering, i.e., a workload-

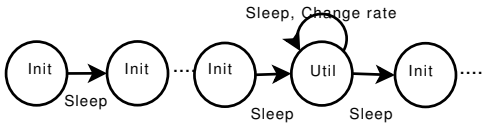

Figure 3: Structure of a CTester program.

based call has to follow an open system call. We introduce sufficient interval (on the order of several seconds) between the system calls.

The range of input parameters to the consecutive workload-based calls is decided based on the component's throughput, with the goal of uncovering all productive states and the threshold on the workload that triggers the transition from a low productive power state to a high productive power state if there are multiple. For example, the WiFi NIC of some smartphones has two productive states and the switching happens when there are more than 50pkts/sec transmitted or received. To search for such transitions efficiently, we increase the input parameters exponentially and perform binary search between two consecutive parameters if they lead to different productive states. In principle, the search process should continue until reaching the limits of the argument space. In practice, the thresholds for power transitions usually occur at small parameter values, e.g. 50 packets/sec for the WiFi NICs. Hence the search finishes very quickly, i.e., after observing that there is no more power state change after a few iterated doubling of the input parameter values.

6. Implementation

We have implemented system call tracing on Windows Mobile 6.5 running Windows CE 5.2 kernel and Android 2.2 running Linux Kernel 2.6.34. We note such system call tracing is previously available in desktop OSes [etw, str] but not in smartphone OSes.

6.1 Tracing System Calls in Windows Mobile 6

WM6 provides a logging mechanism called CeLog [cel], which only tracks all the events related to CPU (context switch and thread suspend/resume), memory (virtual and heap page fault), TLB, and interrupt. CElog is implemented by instrumenting the corresponding functions in the kernel source code. In principle, we could extend CElog to log other system calls needed for our power modeling, such as file system, GUI, and network interface, by instrumenting other parts of the kernel. However, this requires kernel source code and is tedious.

Instead, we leveraged the mechanism used by WM6 to implement system calls, a process called *thunking*. A system call is made to a special invalid address, which will result in a prefetch-abort trap, and the trap handler recognizes which system call was being made based on the invalid address. To log the needed additional system calls, we reroute the system calls in `coredll.dll` and `ws2.dll` (for networking) to enable logging. Essentially, we replace the thunks (e.g. `xxx_send`) with our own functions (e.g. `celog_send`),

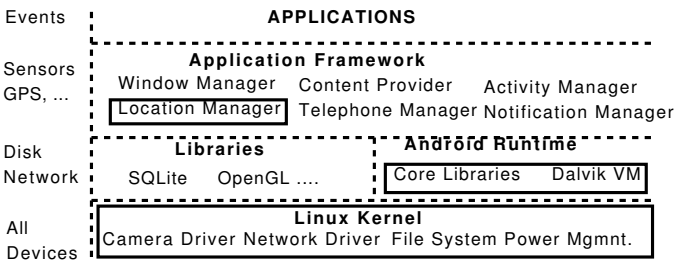

Figure 4: System call instrumentation in Android.

which are simply wrappers that log the system calls before calling the thunk exported by `coredll.dll` and `ws2.dll`. The new functions are implemented as a part of a library that is statically linked with `coredll.dll` and `ws2.dll` during compilation. System calls to GPS and various sensors are logged in a similar way.

6.2 Tracing System Calls in Android

Figure 4 shows the Android framework. It consists of an application framework which is built on top of a few C/C++ libraries and Android runtime. The runtime consists of core java libraries and Dalvik Virtual Machine which are optimized for mobile platforms. The entire Android stack sits on top of the standard Linux kernel. Applications on Android can be either written using Android SDK in java which will run in isolation in Dalvik VM, or written in native C/C++ compiled using the arm compiler which will run outside Dalvik, or written in both using Google's Native Development Kit (NDK). NDK assists compute-intensive applications written in java on Android SDK, to run part of their code (usually the compute-intensive part) outside Dalvik to improve their performance.

Ideally we just need to log all the events at the kernel layer. However, since Android uses many daemon processes that run on Dalvik to coalesce requests from different application processes, it would be difficult to attribute the cause of a system call back to the specific caller, which is needed to perform accurate energy accounting, e.g., on a per-process basis. For this reason, we log events not only at the kernel level, but also the framework level and the VM level (the three boxes in Figure 4). In particular, we use SystemTap [sys] to log system calls and CPU scheduling events in the `sched.switch` function in the kernel, use the logging library that comes with Android to additionally log different sensors, GPS, accelerometer, camera accesses at the framework level, and log disk/network at the DalvikVM level. We changed the default timer granularity of 10 ms to 1ms.

6.3 Flashing Customized Kernel Images

We built a WM6 and Android image with our modifications using platform builder [pb] and shared WM kernel source code [cec] for WM6 and cynogenmod tools [cyn] for Android. The customized images can directly work in corre-

Table 1: Mobile handsets used throughout the paper.

Name	Handset HTC-	CPU (MHz)	OS (kernel)	Base power
magic	Magic	528	Android 2.0 (Linux 2.6.34)	160mA
touch	Touch	528	WM6.1 (CE5.2)	250mA
tytn2	Tytn II	400	WM6.1 (CE5.2)	130mA

sponding emulators running on a desktop PC. We flashed commodity smartphones with customized kernel images.

7. Evaluation

We now present the FSM power models using our system-call-based power modeling approach for Android and WM running on three smartphones and validate their accuracy in energy estimation on a set of diverse applications.

7.1 Hardware Platform

Table 1 lists the hardware, the OS, and the base power of three smartphones used throughout this paper. We use out-of-box settings for LCD brightness and other system parameters. We use PowerMonitor [pow] to measure the power consumed by the smartphone. The smartphone battery is bypassed; the power terminals of the phone are connected to the power supply from the power monitor and the phone is powered through the monitor. PowerMonitor samples current draw once every 200 microseconds.

7.2 FSM Models Constructed

We start with the FSMs constructed for individual components of the three smartphones, followed by the FSM models for the entire smartphones.

7.2.1 FSMs for Individual Components

Figure 5 depicts the FSMs for CPU, sdcard and WiFi NIC for the three smartphones. We also modeled LCD, GPS, and camera, but omit them from the FSMs for clarify. The state transitions are labeled by the system calls or history-based conditions. The productive states in the FSMs are color shaded while the tail/base states are not shaded.

For CPU, we observe a simple FSM for all three smartphones. A process (thread) scheduled to run on the processor consumes a constant current. When the null loop executes on the CPU (thread 0x0 (idle) in process NK.EXE on WM6 and process pid 0 on Android), the system goes to the base state. The productive state for magic, tytn2 and touch consumes +100mA, +130mA and +200mA, respectively.

For sdcard, we observe a simple FSM for Android on magic where disk read/write system calls trigger the state change to a productive state with an average power consumption of +105mA. Other disk system calls do not have any noticeable effect on power consumption. The duration of the productive state is determined by the amount of data written or read and the sdcard throughput. On WM6, we find that system calls file open, close, create, delete, read and write transit the sdcard to a high power state which consumes +125mA on tytn2 and +190mA on touch. The duration of read/write system calls is determined similarly for Android. A productive state is followed by a tail state which consumes +75mA and +110mA and lasts for 3 and 6 seconds for tytn2 and touch, respectively, before going back to the base state. If another system call is observed in tail state, it jumps into productive state and the above cycle repeats.

For wireless NIC, we observe a few power optimizations implemented by the device drivers. For all the handsets, there were two productive states and one tail state. We see that state transitions occur only on network send/recv system calls in all the handsets, except touch where a socket close call also triggers a transition. For all handsets, we find that transmitting (or receiving) packets in the base state causes a state transition to a low productive state (called "low net"). If the transmission rate increases beyond a threshold (12 packets in the past 1 second), the component transits to another base state (called "high net"), which is the same as the tail state. Any further workload-based system calls, i.e., send/recv, cause the NIC to transit to the second productive state. When the sending rate drops below 12 packets in the past 1.5 seconds, the NIC transits to the base state in magic. When there is no network activity for 1.7 seconds, similar transition occurs on tytn2. On touch, a socket close call or 12 seconds of no activity, whichever occurs first, causes the NIC to transit to the base state.

The signal strength of the wireless network affects the power consumption of the productive states. The signal strength can vary from -20dBm to -90dBm (best to worst), and increases the productive state power consumption by a maximum of 20% from best to worst (nearly linearly). Using simple network API calls, the model periodically samples the network signal strength and corrects the power consumption accordingly.

We also developed FSMs for LCD, GPS and camera. The later two are turned on and off using system calls. For example, GPS and camera on Android can be accessed by applications on the Android framework through requestLocationUpdates and OpenCameraHardware system calls. One interesting observation about the FSM for GPS is that after the GPS is turned off, a tail state of +70mA follows for 3 seconds on Android (Figure 1(b)).

7.2.2 FSMs for the Entire Smartphone

Figure 6 plots the FSMs for entire phones for all three phones. For clarity, we only include CPU, sdcard and WiFi NIC, and omit GPS and camera. The names of the states are simply composed from the state names in the FSMs of individual components in Figure 5 (separated by ","). The composite states that exhibit similar characteristics have been merged; their names contain names before merging separated by a "/". As in Figure 5, the power consumed in each state in Figure 6 represents the additional current (in mA) consumed in that state above the base current. Specifically,

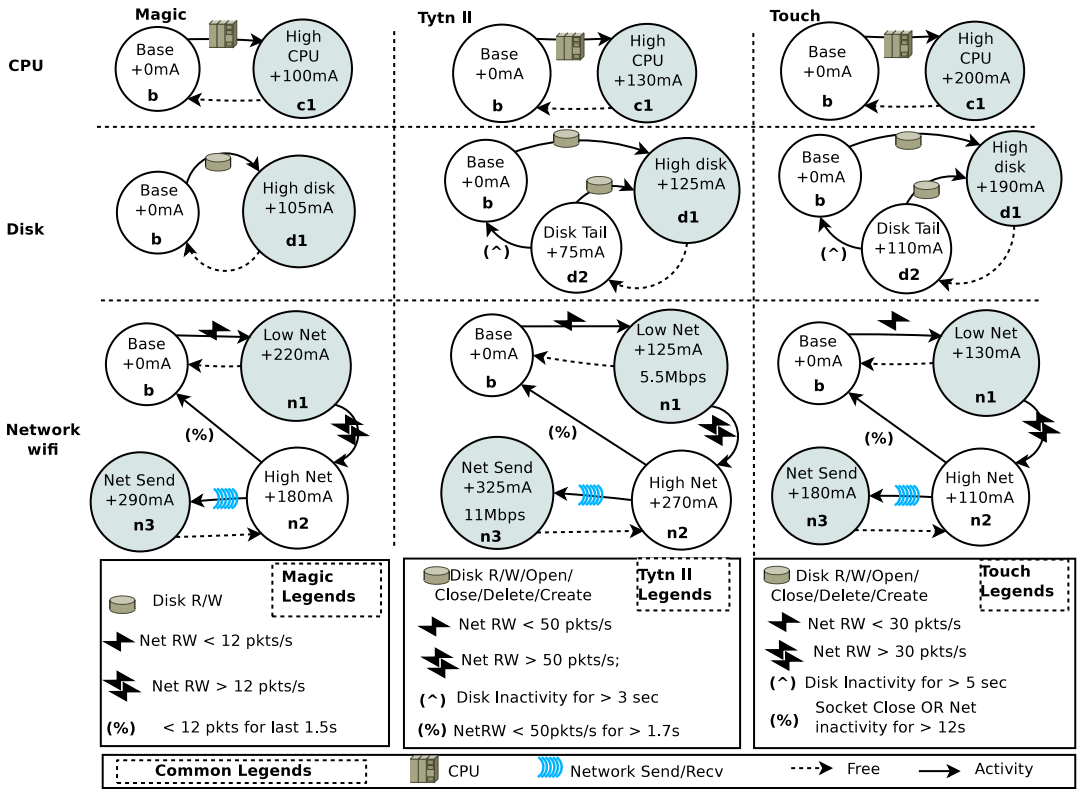

Figure 5: FSM for CPU, Disk and WiFi NIC for three smartphones. The circles in shades are productive states.

we run the cTester benchmark programs to drive different components to reach the desired states (e.g. CPU busy and WiFi NIC in tail) and measure the current consumed by the entire smartphone above the base level.

Android on magic has the simplest FSM. It has a base state, one productive state for each component (CPU, sdcard, network), and a network tail state. When multiple components are active, the total power consumption equals the summation of those of individual components when active alone. For WM6 on the other two handsets, there are two tail states, one of sdcard and the other of the network. We find that these states interact with each other; they do not add up as in case of Android. For example, an application consuming high CPU consumes +200mA (+130mA) irrespective of whether the transition came from the tail state of disk (which consumes +110mA (+75mA)) or from the base state (at 0mA) on the touch (tytn2) handset. A similar interaction exists between network and CPU on the touch handset. Another feature we observe is that when the tail states of sdcard and network overlap, the power draw does not equal the sum. For example, when sdcard and network are in their tail states, the power consumed on tytn2 will be the maximum of the power of the two states (in this case +270mA which is that of network). When the network tail state expires, the system falls back to the disk tail state (+75 mA), which lasts for 3 seconds from the last disk activity.

7.3 Energy Consumption Estimation

We now compare the accuracy of fine-grained and whole-application energy consumption estimation under our system-call-based modeling (FSM model for short) and under utilization-based modeling. For utilization-based modeling, we implemented the LR model for smartphones as described in [Shye 2009], by considering four components: LCD, CPU, sdcard, and NIC. We do not compare applications that use GPS or camera, since they are not modeled in [Shye 2009]. LR is developed by training with random application runs (as in [Shye 2009]).

For both OSes, we created /proc like utilization entries using our system call tracing, since our tracing captures non-utilization-based calls and hence forms a superset of the /proc-like logging utilities. However, the real benefit of using the same /proc under FSM and LR is that it allows us to run each application once, and perform offline comparison of their energy estimation accuracy for that run. This is important as most of the smartphone applications are interactive; the run time (and energy consumption) in different runs of the same application can easily fluctuate by over 10%. Further, if we ignore the extra overhead of periodically sampling /proc in LR for now, we can isolate and study the impact of /proc sampling frequency on LR's accuracy. We compare the overhead of FSM and LR in Section 7.4.

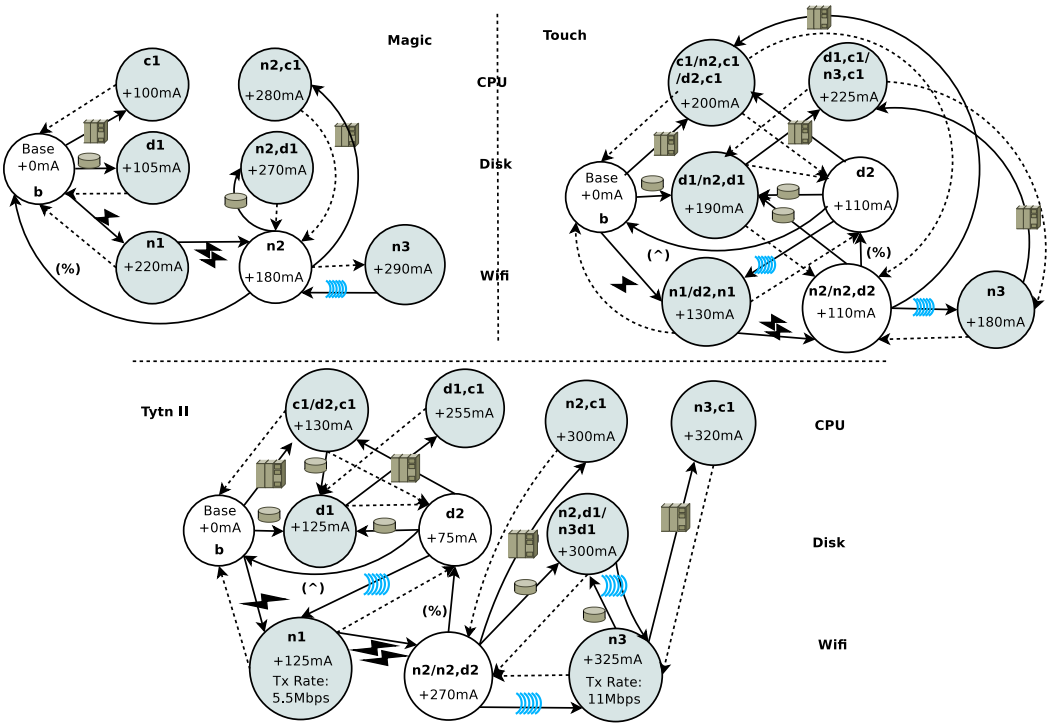

Figure 6: FSMs for entire smartphones when considering only CPU, Disk and WiFi NIC. The legends are the same as in Figure 5.

Table 2: Applications used throughout the paper.

App.	Description
Windows Mobile (on tytn2)	
game	First person shooter game (graphical)
chess	Mobile optimized version of chess
diskB	Open a file, sleep for 10 sec, write to file for 5 sec
netB	Connect, sleep for 10 sec, write over network for 5 sec
ie.cnn	User browsing mobile version of CNN on IE
pviewer	Slideshow of photos from SD card
docConv	A document converter (from .abc to .def)
virusScan	Performs an antivirus scan on SDcard
youtube	User watching a youtube video of 150 sec
puploader	Upload 8 photos to desktop using NIC from SDcard
Android (on magic)	
csort	sort program in native C
dropbear	ssh server process in native C, user listing all files
maps	google maps using GPS and network
facebook	facebook logging process
youtube	User watching a youtube video of 41 sec

Table 2 describes the applications used for both WM6 and Android, running on one handset each. We selected the tytn2 handset for evaluating WM6 applications since it has a more complex FSM when compared to touch, and the magic handset for Android applications. We selected a mixture of applications that utilize either a single, two, or more components at a time. For Android, we also selected a mixture of applications that are either written in native C (csort and dropbear), or in java based on Android SDK and Google NDK (youtube, facebook), and included ones which use GPS (maps).

7.3.1 Fine-grained Energy Estimation

We first measure the error in fine-grained energy estimation. For each application, we split its execution time into 50ms intervals and calculate the absolute error of per-interval energy estimation under FSM and LR. Figure 7 shows the CDF of these errors for eight applications on WM6 and Android. We omitted simpler applications from these figures. They have comparable or better accuracy. Figure 7(a) shows that for all of the applications, the 80th percentile energy estimation error under FSM is less than 10%, and is less than 5% if we increase the time interval to 1s (not shown).

In contrast, Figure 7(b) shows that LR with 1s intervals (LR/1s) performs far worse than FSM. The error at the 80th percentile varies between 16% and 52%. A close look at the ie.cnn and youtube applications on WM6 shows why LR incurs much larger error. The ie.cnn application is CPU-intensive, with minimal network activity (about 20 packets of data). For each 1s interval, LR attributes a uniform power value for all 50ms intervals based on the average CPU utilization in that 1s, as shown in Figure 8(d). For youtube, when streaming a video from the youtube server, packet arrival is spread over the entire time duration of the video. There are several periods lasting between a few hundred ms to 1s during which there is no network activity (no network receive). During a specific 1s time period, if there is no network activity, LR attributes 0 power for the NIC. But in re-

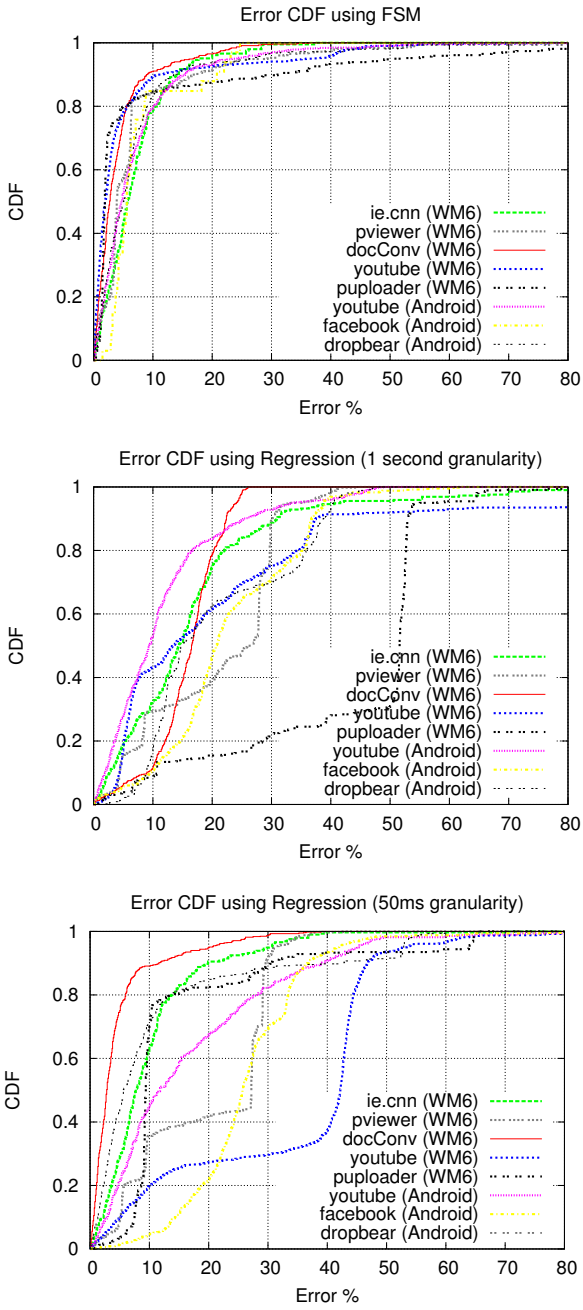

Figure 7: CDF of absolute error percentage between measured and estimated energy consumption per 50 ms for applications using (a) FSM, (b) LR with 1s granularity, (c) LR with 50ms granularity.

ality, due to recent high network activity ($>$ 12 packets in past 1.5 second), the NIC is in tail power state consuming +180mA. As a result, all 20 50ms time intervals in this second have high error, as shown in Figure 8(f).

7.3.2 Impact of Fine-grained Sampling in LR

An intuitive way to improve LR's estimation accuracy is to improve its frequency in sampling utilization counters. However, high sampling frequency can lead to very high overhead, as we show in Section 7.4. Ignoring the overhead for now, we increase the sampling frequency from once per second to once every 50ms, and the CDF of the resulting energy estimation error of LR (LR/50ms) is shown in Figure 7(c). We see that compared to LR/1s, the accuracy (a) increases for ie.cnn, docConv, puploader, and dropbear, (b) is mostly unaffected for pviewer, and (c) degrades for youtube (on WM6 and Android) and facebook.

Figure 8(g)-(i) plot the energy estimation under LR/50ms for the same three applications. Compared to LR/1s, LR/50ms performs quit well for the CPU-intensive ie.cnn since the more frequent sampling captures the fine-grained CPU utilization. However, since LR can not capture the disk tail states exhibited in running pviewer, like LR/1s (Figure 8(e)), LR/50ms (Figure 8(h)) also incurs about 30-40% error during the tail states. LR/50ms gives worse accuracy for applications in category (c) because of the network tail phenomenon. As explained earlier, packet arrival to NIC is spread throughout the time period of video. When we increase the sampling frequency of LR, LR only reports NIC utilization for the 50ms durations when there is actually a network receive, and zero NIC utilization in all other 50ms intervals. Relatively few 50ms intervals contain a recv call. As a result, though more precisely capturing the timing of network activities, LR's accuracy actually degrades. Puploader and dropbear are unaffected by this phenomenon as their destination servers are nearby (1ms RTT) and hence the gap between consecutive packets is only a few ms.

7.3.3 Whole-application Energy Estimation

Finally, we also compare the whole-application energy estimation accuracy under FSM and LR. Figure 9 plots the percentage error for all applications on both OSes under FSM and under LR/1s (using 1 second time granularity for training and estimation as in [Shye 2009]). We see that the estimation error under FSM varies between 0.2% and 3.6%. We note that if we just use the FSM for each component and add up the estimated energy consumption, the estimation error would be 22%, 45%, 11% and 36% for docConv, virusScan, youtube and puploader, respectively, on WM6 (not shown). In contrast, LR performs well for the applications that do not have dominant tail power patterns, but the estimation error varies between 3.5% to 20.3% for the rest, including diskB, pviewer, youtube, facebook, and dropbear. These results show system-call-based power modeling is not only far more accurate in fine-grained energy estimation, but can also significantly improve the accuracy of whole-application energy estimation.

7.4 Logging Overhead

Since /proc comes from logging system calls (though a subset of those in FSM), in principle both FSM and LR incur system call tracing overhead. In addition, LR incurs the overhead of periodic sampling of /proc. Below, we conser-

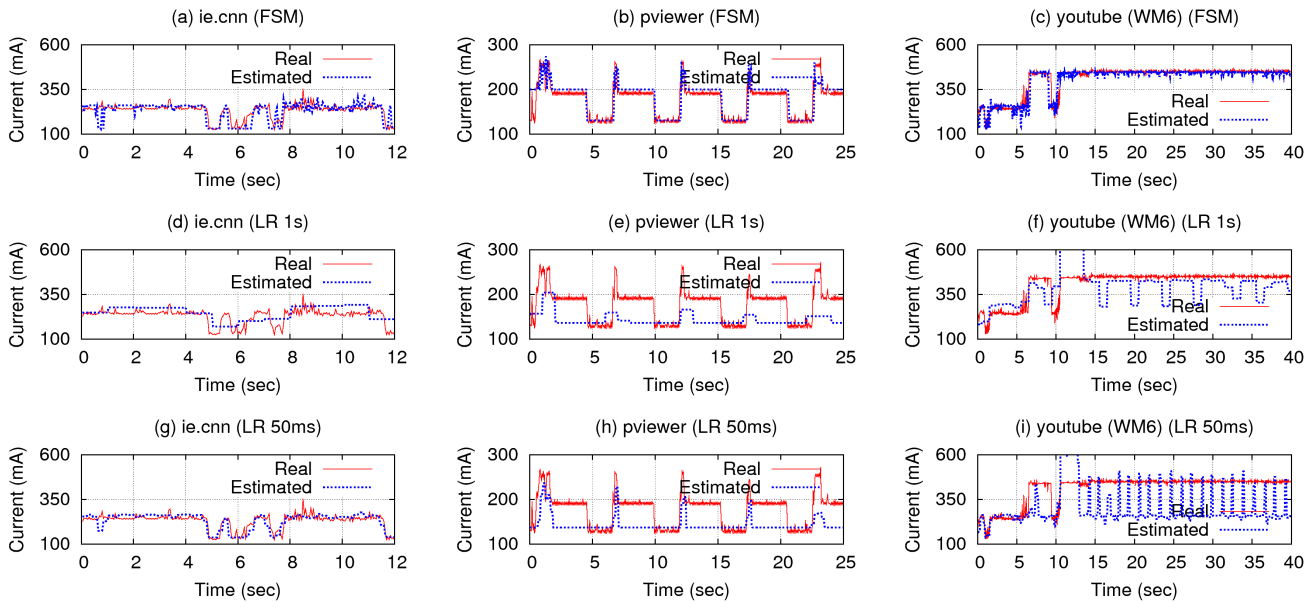

Figure 8: Power Profile: Measured vs estimated energy consumption over time on WM6 (tytn2 handset) under FSM, LR/1s, and LR/50ms.

vatively compare the system call tracing overhead in FSM with only the periodic sampling of /proc in LR.

We measure the system call logging overhead of FSM by comparing the execution time of the applications when logging is turned on versus off. Since re-execution of an interactive application (such as youtube) can experience significant variability in execution time due to external factors such as variability in server response time, network delay, and authentication schemes, we focus on applications that are not affected by external factors in measuring the logging overhead. The logging overhead under FSM varies between 5.4%-9.8% on Android and 1.1%-8.9% on WM6.

For LR, note the overhead of sampling /proc is proportional to the sampling frequency and independent of the frequency of system calls triggered by the application. We measured the overhead of reading /proc (CPU and IO (per-process network utilization is not available in /proc)) stats values for all processes from a user-level C application on Android. The overhead ranges between 7.5%-10.0% when sampling once per second, 15.2%-17.5% once every 500ms, and 35.3%-52.5% once every 200ms. The results show that due to high overhead, it is not even practical to obtain fine-grained utilization information using /proc.

8. Applications: Energy Profiler

We give a proof-of-concept demonstration of an important application of fine-grained energy estimation enabled by our system-call-based power modeling, by showing a manually implemented *eprof*, the energy counterpart of the classic *gprof* tool [Graham 1982], for profiling application energy consumption.

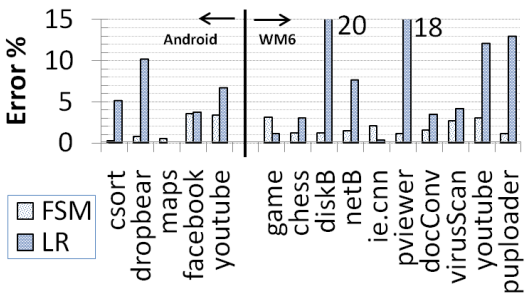

Figure 9: Absolute error in end-to-end energy consumption estimation.

Building on top of our FSM power model, there are two remaining challenges in the design of *eprof*. First, as in gprof, we need to construct the call graph and perform bookkeeping of each invocation of each subroutine, i.e., execution counts and energy consumption. Currently, we manually annotate every subroutine in the source code to log the entry and exit points. This also allows us to trace each system call to the caller subroutine.

The second challenge is per-subroutine energy accounting. A major difference between per-subroutine (or per-process, per-thread) execution time accounting in gprof and energy accounting in eprof comes from the tail energy consumption by many components. Figure 10 shows an example where a network system call in subroutine F1 sends the NIC to a tail state which lasts beyond the completion of both F1 and subroutine F2. The energy consumption during F2 was an accumulative effect of both CPU activities of F2 and the tail state of the NIC due to F1. A complete solution to the above problem of fine-grained energy accounting is beyond the scope of this paper. In our proof-of-concept implemen-

Table 3: Eprof output for three applications: The columns of the table are: % time and % energy spent in self, cumulative time and energy spent by all the function descendants, actual time spent in a function itself, number of times a function is called, time and energy spent in self per call, time and energy spent non-cumulative per call, and name of the function.

%		cum.		self		#	self		total		name
time	energy	s	uAH	s	uAH	calls	ms/call	nAH/call	ms/call	nAH/call	
chess											
30.4	30.4	34.4	1243.0	21.6	779.9	61931	0.4	12.6	0.56	20.1	Is_Move_Legal
19.7	19.7	22.9	825.9	14.0	505.0	73665	0.2	6.9	0.3	11.2	Is_Move_Valid
14.9	14.9	75.1	2713.0	10.6	382.1	60416	0.2	6.3	1.2	44.9	CheckHumanMove
12.0	12.0	8.6	308.8	8.6	308.8	919	9.3	336.0	9.3	336.0	CountScore_Human
5.9	5.9	4.2	150.4	4.2	150.4	433	9.6	347.3	9.6	347.3	CountScore
docConv											
42.0	36.6	34.0	1063.0	24.9	762.9	77	323.4	9907.3	441.9	13805.6	ExtractTextFromFile
28.3	25.9	45.8	1443.6	16.8	539.6	148133	0.1	3.6	0.3	9.8	CheckToken
19.5	28.0	55.1	1951.9	11.5	583.6	1	11533.7	583569.2	55120.9	1951933.5	ExtractText
7.0	6.3	59.3	2082.6	4.2	130.6	1	4186.6	130639.5	59307.5	2082573.0	main
3.2	3.2	1.88	65.9	1.9	65.9	77	24.5	855.8	24.5	855.8	getpagecontent
puploader											
50.2	67.9	2.0	167.9	2.0	167.9	8	245.8	20987.6	245.8	20987.6	sendfile
18.0	15.0	0.7	37.0	0.7	37.0	8	88.2	4628.0	88.2	4628.0	readfile
17.5	9.0	0.7	22.2	0.7	22.1	8	85.9	2769.7	85.9	2769.7	calchash
5.7	3.3	0.2	8.1	0.2	8.1	1	224.8	8118.7	224.8	8118.7	GUI_Form
5.2	3.0	0.2	7.4	0.2	7.4	1	203.4	7345.7	203.4	7345.7	initnet

tation, we take the following simple approach: we always break up the power consumed in a power state into power consumed per component, by assigning a continuing power state (e.g. tail state) the same power when occurring alone. For example in Figure 10, F1 and F2 will be charged P1 and (P2-P1) during T2 to T3, respectively. We leave more complicated cases where multiple subroutines (threads or processes) in the past intervals are responsible for the total power as future work.

Table 3 shows the gprof-style time and energy breakdown for top 6 functions (sorted by runtime) in three applications (chess, docConv and puploader on WM6 on tytn2). We observe that the time and energy percentages are about the same for all the functions in chess (only CPU), but differ in docConv (CPU, sdcard). The most interesting application is puploader, where three functions, calchash, readfile and sendfile, use CPU, disk and network respectively. Sendfile consumes the most time (50.2%) and energy (67.9%). Calchash, though consuming 17.5% of the time, spends only 9.0% of the energy. If we did not apply fine-grained energy accounting as discussed above, sendfile and calchash would have been reported to consume 57.5% and 13.0% of energy.

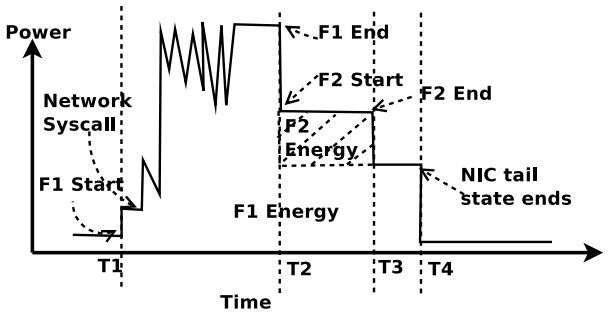

Figure 10: Eprof: Function level breakdown.

9. Related Work

System call tracing has been exploited to develop useful tools on desktop and server machines, in particular, for accounting resource utilization [Barham 2004], building debuggers/replay tools [Guo 2008], automatic fault detection and diagnosis [Yuan 2006], and energy measurement and accounting [Zeng 2002]. It has also been used on mobile devices for malware detection [Bose 2008], and on sensor nodes for energy measurement [Shnayder 2004].

Our system-call-based power modeling is related to [Shnayder 2004, Zeng 2002], which also exploit events as opposed to utilization rates of components, but for desktop machines and sensor nodes. Further, these work do not use system calls to capture non-utilization-based power behavior. In contrast, we discover non-utilization-based power behavior on smartphones and use system call tracing to capture both utilization-based and non-utilization-based power behavior to achieve accurate fine-grained energy estimation.

Finally, RevNIC [Chipounov 2010] aims to reverse-engineer the exact behavior of device drivers, which is a harder task than reverse-engineering device drivers' power optimization behavior. Our approach works well in practice as fundamentally the power behavior of the drivers are correlated with the intention and consequence of system calls that trigger them, and there are only a small number of power states per component in practice.

10. Conclusion

In this paper, we have presented the design and implementation of a system-call-based power modeling approach which gracefully captures both utilization-based and non-utilization-based power behavior. Our experimental results on Android and Windows Mobile using a diverse set of applications show that the new model drastically improves the accuracy of fine-grained energy estimation as well as whole-application energy estimation. We further presented a proof-of-concept demonstration of *eprof*, the energy-counterpart of *gprof*, for optimizing the energy usage of application programs. We are developing a full-fledged *eprof* and plan to release the modified Android image and tools to the public. Our power modeling study also exposed significant diversity of power behavior of different OSes and smartphone handsets. As future work, we plan to develop detailed classification of power behavior of different OSes and handsets and for different applications.

Acknowledgments

We thank the program committee and reviewers for their helpful comments, and especially our shepherd, M. Satyanarayanan, whose detailed feedback significantly improved the paper and its presentation.

References

[Com] Android phones steal market share. URL http://bmighty.informationweek.com/mobile/showArticle.jhtml?articleID=224201881.

[cel] Celog event tracking. URL http://msdn.microsoft.com/en-us/library/aa462467.aspx.

[cyn] Cyanogenmod: Android community rom based on froyo. URL http://www.cyanogenmod.com/.

[etw] Event tracing for windows (etw). URL http://msdn.microsoft.com/en-us/library/ms751538.aspx.

[pb] Microsoft platform builder. URL http://msdn.microsoft.com/en-us/library/ms938344.aspx.

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

[str] Strace. URL http://linux.die.net/man/1/strace.

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

[cec] Windows embedded ce shared source. URL http://msdn.microsoft.com/en-us/windowsembedded/ce/dd567722.aspx.

[Balasubramanian 2009] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc of IMC*, 2009.

[Barham 2004] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proc. of OSDI*, 2004.

[Bellosa 2000] F. Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proc. ACM SIGOPS European workshop*, 2000.

[Bircher 2007] W.L. Bircher and L.K. John. Complete system power estimation: A trickle-down approach based on performance events. In *Proc. of ISPASS*, 2007.

[Bose 2008] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *Proc. of MobiSys*, 2008.

[Chipounov 2010] Vitaly Chipounov and George Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proc. of EuroSys*, 2010.

[Fan 2007] X. Fan, W.D. Weber, and L.A. Barroso. Power provisioning for a warehouse-sized computer. In *Proc. of ISCA*, 2007.

[Flinn 1999a] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. of SOSP*, 1999.

[Flinn 1999b] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proc. of WMCSA*, 1999.

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

[Guo 2008] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *OSDI*, pages 193–208, 2008.

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

[Mahesri 2005] A. Mahesri and V. Vardhan. Power consumption breakdown on a modern laptop. *Proc. of PACS*, 2005.

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

[Shnayder 2004] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of Sensys*, 2004.

[Shye 2009] 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.

[Snowdon 2009] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for os-level power management. In *Proc. of EuroSys*, 2009.

[Stanley-Marbell 2001] P. Stanley-Marbell and M. Hsiao. Fast, flexible, cycle-accurate energy estimation. In *Proc. of ISLPED*, 2001.

[Tiwari 1996] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee. Instruction level power analysis and optimization of software. *The Journal of VLSI Signal Processing*, 13(2), 1996.

[Yuan 2006] C Yuan, N Lao, J Wen, J Li, Z Zhang, Y Wang, and W Ma. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.

[Zedlewski 2003] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proc. of FAST*. USENIX Association, 2003.

[Zeng 2002] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proc. of ASPLOS*, 2002.

[Zhang 2010] 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 *Proc. of CODES+ISSS*, 2010.