Cellular Networks and Mobile Computing
COMS 6998-11, Spring 2012

Instructor: Li Erran Li
(lierranli@cs.columbia.edu)

http://www.cs.columbia.edu/~lierranli/coms6998-11Fall2012/

9/25/2012: Ebug debugging, Power Models, and Profiling
Announcements

• Contact TA Jaimin Shah (jps2178@columbia.edu) to provision your iOS devices

• Programming assignment 2 will be due on Tuesday, Oct 2nd
Outline

• The Rise of Ebugs
• Debugging no-sleep Ebugs
• Methods of Measuring Power Usage
• Power Models
  – Usage based
  – System call trace based
• Profiling
• Conclusion
The Rise of Energy Bugs

Single Symptom:
**Severe, Unexpected** Battery Drain

Apps Need Not Crash
No Blue Screen Of Death

Common Perception:
Kill some apps to fix

Courtesy: Pathak et al
User Frustration  
(Dialer App EBug)

Comment 24 by mgil...@gmail.com, Aug 14, 2011
This defect is a real P.I.T.A. - I don't want to use my phone as a phone because I have to restart it every time. If I forget then it's usually 30-40% battery gone by the end of the day.

Comment 30 by hansheng...@gmail.com, Aug 15, 2011
Bring your charger with you and keep it charged!!! That's the only way the phone can last a day. It's a irritating bug!!!

Comment 239 by egork...@gmail.com, Nov 6 (6 days ago)
GOOGLE!!!!! DO SOMETHING WITH THIS ISSUE!!! FASTER PLEASE!!!!
Crawling Internet Forums

1. 4 Online Mobile Forums
   (~400 mobile devices; Several mobile OSes)
   1.2M Posts
   grep “power drain”, “battery drain”, etc.
   39K Posts
   K-means
   1000 Clusters

   EBug Taxonomy
   Human

2. 2 Mobile Bugs/Issues Repository
   Indepth Analysis

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
Ebug Taxonomy

- **Hardware** (23%)
  - **OS**
  - **Apps**
    - No Sleep Bug
    - Loop Bug
    - Immortality Bug

- **Software** (35%)
  - **External Service**
  - **Network Signal Strength**
  - **Wireless Handovers**

- **External** (12%)

- **Unknown** (30%)

*Courtesy: Pathak et al.*

Cellular Networks and Mobile Computing (COMS 6998-11)
Hardware EBug

- Battery
- Sim Card
- SDCard
- Exterior Hardware Damage
- Hardware: 23%
- Software: 35%
- External: 12%
- Unknown: 30%

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
Ebug Taxonomy

- Hardware (23%)
- Software (35%)
  - OS
  - Apps
    - No Sleep Bug
      - Loop Bug
      - Immortality Bug
  
- External (12%)
  - External Service
  - Network Signal Strength

- Unknown (30%)
  - Wireless Handovers

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Pathak et al
OS Ebugs

Why does OS Leak Energy?

– Hard to infer

– OS Processes

– System Configuration

iPhone 4S Battery Life Bugs Got You Down?

Battery life on the iPhone 4S: the new 'death grip'?

By Doug Gross, CNN
updated 4:17 PM EST, Tue November 1, 2011 | Filed under: Mobile

iPhone battery fix coming 'in a few weeks'

By Doug Gross, CNN
updated 11:19 AM EST, Thu November 3, 2011 | Filed under: Mobile

IOS Version: 4.0 – 4.3.3 (5% posts)

2.5% posts

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
Apps EBug: No Sleep Bug

• Aggressive Sleeping Policies: Smartphone OSes freeze system after brief inactivity

• Power encumbered Programming: Programmer has to manage sleep/wake cycle of components

• No Sleep Bug: At least one component is kept awake due to mismanagement
External Conditions

- External Services (<1%)
- Network Signal Strength (11%)
- Wireless Handovers (<1%)
EDB: Energy Debugging Framework
Mobile Programming EcoSystem: The EBug Blame Game

- App Developers
- Network Operators
- Framework Developers
- Kernel Developers
- Hardware Manufacturers
- Firmware/OEM Developers

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Pathak et al
Outline

• The Rise of Ebugs
• Debugging no-sleep Bugs
• Methods of Measuring Power Usage
• Power Models
  – Usage based
  – System call trace based
• Profiling
• Conclusion
Smartphones are Energy Constrained

Battery energy density only doubled in last 15 years

CPU

3G/4G

WiFi

Screen

Camera

GPS

Courtesy: Jindal et al

Cellular Networks and Mobile Computing (COMS 6998-11)
Paradigm Shift in Power Management

• Desktop/Server/Laptop: Default ON
  – CPU turned off when idle for long time

• Smartphones: Default OFF
  – Smartphone OSes aggressively turn off Screen/
    CPU after brief user inactivity
  – Helps increasing standby time period

Courtesy: Jindal et al
Consequences of Aggressive Sleeping Policy

public void DoNetwork ( )
{

    Sync_Over_Network ( ); //Sync state with remote server
    //Can take up to a minute

}

• Background jobs
• User is not always touching phone

Need a mechanism to explicitly keep components awake.

Keep the screen on !
Android API to Explicitly Keep Components Awake

PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
PowerManager_WAKE_LOCK wakelock = pm.newWakeLock(
(PowerManager.PARTIAL_WAKE_LOCK, null);

<table>
<thead>
<tr>
<th>API</th>
<th>Component (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL_WAKE_LOCK</td>
<td>CPU</td>
</tr>
<tr>
<td>SCREEN_DIM_WAKE_LOCK</td>
<td>CPU and Screen (DIM)</td>
</tr>
<tr>
<td>SCREEN_BRIGHT_WAKE_LOCK</td>
<td>CPU and Screen (Bright)</td>
</tr>
<tr>
<td>Camera.startPreview() Camera.release()</td>
<td>Camera</td>
</tr>
</tbody>
</table>
public void DoNetwork () {

  wakelock.acquire (); //Don’t let CPU sleep till I say so
  Sync_Over_Network (); //Sync state with remote server
  //Can take up to a minute

  wakelock.release (); //CPU is now free to sleep
}
iOS API to Explicitly Keep Components Awake

- **Screen:** `idleTimerDisabled` property in `UIApplication`
  - If NO, the idle timer turns off the device’s screen after a specified period of inactivity

- **GPS:** enable/disable location updates, and set the distance filter and accuracy levels
  - Core Location uses the available GPS, cell, and Wi-Fi networks to determine the user’s location
  - Core Location minimizes the use of these radios
  - Setting the accuracy and filter values gives Core Location the option to turn off hardware altogether in situations where it is not needed.

- **WiFi:** `UIRequiresPersistentWiFi` key in the app’s Info.plist file
  - System will not shut down the Wi-Fi hardware while your app is running.
App programmer has to manage sleep/wake cycle of components
NoSleep Energy Bugs

An error in smartphone app, where a component is woken up, but not put to sleep

<table>
<thead>
<tr>
<th>Component</th>
<th>Impact (Google Nexus)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>50-60% battery in 12 hrs</td>
</tr>
<tr>
<td>Screen</td>
<td>100% battery in 4-6 hrs</td>
</tr>
<tr>
<td>GPS</td>
<td>100% battery in 3-5 hrs</td>
</tr>
</tbody>
</table>

Courtesy: Jindal et al
Collecting and Characterizing Bugs

• Online mobile forums
• Bug lists
• Open source code repositories
• Running the detection tool

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Jindal et al
Types of NoSleep Bugs

- NoSleep Code Paths
- NoSleep Race Conditions
- NoSleep Dilations

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Jindal et al
(a) NoSleep CodePaths

At least one path in program wakes up a component and does not put it to sleep

```java
public void DoNetwork()
{
    try {
        wakelock.acquire();
        Sync_Over_Network();
        wakelock.release();
    } catch (SomeException e) {
        Print.Error(e);
    }
}
```

CPU sleep till I say so

Sync state with remote server

Can take up to a minute

throws exception

At least one path in program wakes up a component and does not put it to sleep

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Jindal et al
(b) NoSleep Race Condition (1)

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Jindal et al
(b) NoSleep Race Conditions (2)

public class MainThread {
    boolean KillFlag = false; //Kill Flag
    //Infinite Loop
    while(true){
        //CPU Cant Sleep
        wakelock.acquire();
        WorkerThread.start(); //start worker thread
        WorkerThread.interrupt(); //stop worker thread
        wakelock.release(); //CPU can sleep now
        KillFlag = true; //Kill the worker thread
        Break if(KillFlag) break; //Exit
        NetSync(); //Critical Section
        Thread.Sleep(3 Mins); //Sleep for 3 mins
        wakelock.release(); //CPU can sleep
        wakelock.acquire(); //CPU Cant Sleep
        ....
    }
}

public class WorkerThread {
    //Infinite Loop
    ....
    ....
    ....
}

Buggy Execution
(c) NoSleep Dilations

Components are eventually turned off but are kept on for unusually long duration of time

```java
public void onCreate() {
    wakelock.acquire();
}

public long startNewTrack() {
    wakelock.acquire();
    ..
}
```
Detecting NoSleep Bugs

• NoSleep Code Paths
• NoSleep Race Conditions

• Static Data Flow Analysis
  – Reaching Definitions Data Flow Problem
Problem Statement: For each point in the program, determine which definition of variable are reachable.
NoSleep Code Path DataFlow Analysis

Entry

\[ \textbf{d0}: \text{wakelock} = 1; \]
\[ \text{Sync\_over\_network()} \]

B0

B1

\[ \textbf{d1}: \text{wakelock} = 0; \]

Print.error(e);

IN[exit]: \{d0, d1\}

exit

Apply reaching definitions to NoSleep CodePaths

Entry

\[ \textbf{d0}: \text{wakelock}. \text{acquire}(); \]
\[ \text{Sync\_over\_network()} \]

B0

B1

\[ \textbf{d1}: \text{wakelock}. \text{release}(); \]

Print.Error(e)

exit

B2

CELLULAR NETWORKS AND MOBILE COMPUTING
(COMS 6998-11)

Courtesy: Jindal et al
Additional Details of Analysis in Paper

• Event based programming

• Java runtime issues

• Special code paths: reducing false positives

• Multi-threaded programs
Implementation

• Soot Implementation
  – Open source framework for analyzing java bytecode from Sable research group at McGill University
  – Added ~2 KLOC

• Runs on Java byte code
  – No need for source code
Evaluation: NoSleep Code Paths

500 Apps ➔ 187 ➔ 86 ➔ 55

Apps Manipulating components ➔ Apps which could be decompiled ➔ Find NoSleep Bugs

<table>
<thead>
<tr>
<th>Total Apps</th>
<th>BUG in reality</th>
<th>No BUG in reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>86</td>
<td>True Positive 42</td>
<td>False Positive 13</td>
</tr>
<tr>
<td>Tool reports BUG (55)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool reports no BUG (31)</td>
<td>False Negative 0</td>
<td>True Negative 31</td>
</tr>
</tbody>
</table>

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Jindal et al
## Causes of Detected NoSleep Bugs

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of Apps</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect Wakelock Handling in Events</td>
<td>26</td>
<td>MyTracks</td>
</tr>
<tr>
<td>If, else + exception Paths</td>
<td>12</td>
<td>Facebook</td>
</tr>
<tr>
<td>Forgot Release</td>
<td>4</td>
<td>K9Mail</td>
</tr>
</tbody>
</table>
Common Cause of False Positives

If(caller != BLACKLISTED)

Wl.acquire();

. .

If(caller != BLACKLISTED)

Wl.release();

EXIT

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Jindal et al
Summary

• Paradigm shift in power management
  – Power encumbered programming

• NoSleep energy bugs
  – Code paths, race, dilation

• Reaching definition data flow analysis
  – Found 30 new previously unreported bugs
Future Work

• Comprehensive approaches to No-Sleep bug
  – Detection: e.g. Symbolic execution

  – Prevention: e.g. Better abstraction for power management API

  – Mitigation: e.g. Runtime detection, Limiting damage of the bug

Courtesy: Jindal et al
Measuring Power Usage

• Approach 1: Use power meter (offline)
  – Buy an expensive equipment ($770)
  – Problems:
    • Only reports entire device energy consumption

• Approach 2: Use built-in battery sensor (online)
iOS Battery API

- Use `UIDevice` class to obtain information and notifications about
  - charging state (property `batteryState`)  
  - charging level (property `batteryLevel`)  

```swift
1. [[UIDevice currentDevice] setBatteryMonitoringEnabled:YES];
2. NSArray *batteryStatus = [NSArray arrayWithObjects:
   @"Battery status is unknown.",
   @"Battery is in use (discharging).",
   @"Battery is charging.",
   @"Battery is fully charged.", nil];
3. if ([[UIDevice currentDevice] batteryState] == UIDeviceBatteryStateUnknown)
   NSLog(@"%@", [batteryStatus objectAtIndex:0]);
4. else {
   NSString *msg = [NSString stringWithFormat:
   @"Battery charge level: %0.2f%%\n%",
   [[UIDevice currentDevice] batteryLevel] * 100,
   [batteryStatus objectAtIndex:[[UIDevice currentDevice] batteryState]] ];
5. NSLog(@"%@", msg);
6. }
```
Android Battery API

• Sample updates stored in files:
  – Current: /sys/class/power_supply/battery/batt_chg_current
  – Voltage: /sys/class/power_supply/battery/batt_vol
  – Capacity: /sys/class/power_supply/battery/capacity

1. File fcur = new File("/sys/class/power_supply/battery/batt_chg_current");
2. if (fcur.exists())
3.  ...

• File names are vendor dependent

• **Access using Android Debug Bridge (adb)**
  – <sdk>platform-tools
  – Command: adb shell
Outline

• The Rise of Ebugs
• Debugging no-sleep Bugs
• Methods of Measuring Power Usage
  • Power Models
    – Usage based
    – System call trace based
• Profiling
• Conclusion
Smartphone is Energy Constrained

• Energy: One of the most critical issues in smartphones
  – Limited battery lifetime

• Battery energy density only doubled in last 15 yrs

• Smartphone capability has increased drastically
  – Multiple Components: GPS, 3G, retina display, ....
Towards Understanding Energy Drain

• **Key Question: Where is energy being spent?**
  – Which component/process/thread/function(?)
Generic Power Modeling

- Triggers
- Model
- Power meter
- Actual Power Consumption
- Training Phase
- Predicted Power Consumption
- Prediction Phase

Courtesy: Pathak et al
Smartphone Power Modeling: Utilization Based (1/3)

Model = \((\text{Util}_{\text{Net}}) \times E_{\text{Net}} + (\text{Util}_{\text{CPU}}) \times E_{\text{CPU}} + (\text{Util}_{\text{Disk}}) \times E_{\text{Disk}}\)

Courtesy: Pathak et al
Smartphone Power Modeling: Utilization Based (2/3)

• PowerTutor model

\[(\beta_{uh} \times \text{freq}_h + \beta_{ul} \times \text{freq}_l) \times \text{util} + \beta_{CPU} \times \text{CPU\_on} + \beta_{br} \times \text{brightness} + \beta_{Gon} \times \text{GPS\_on} + \beta_{Gsl} \times \text{GPS\_sl} + \beta_{Wi\_Fi\_l} \times \text{Wi\_Fi\_l} + \beta_{Wi\_Fi\_h} \times \text{Wi\_Fi\_h} + \beta_{3G\_idle} \times \text{3G\_idle} + \beta_{3G\_FACH} \times \text{3G\_FACH} + \beta_{3G\_DCH} \times \text{3G\_DCH}\]

\(\beta\) : power coefficient.
util, brightness and etc.: system variables.

• Sesame paper has two optimizations: model molding, principle component analysis (PCA)
Smartphone Power Modeling: Utilization Based (3/3)

Model = (Util_{Net}) \cdot E_{Net} + (Util_{CPU}) \cdot E_{CPU} + (Util_{Disk}) \cdot E_{Disk}

Fundamental (yet intuitive) assumption

(Only active) Utilization \Rightarrow power consumption

Second assumption
Energy scales linearly with amount of work

Third assumption
Components power consumption add linearly

Desired Feature
Which process/thread/function? Hard to correlate

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Pathak et al
(Only active) Utilization => Power Consumption

File open/delete/close/create change power state

Several components have tail states (3G, disk, wifi, gps)
Energy scales linearly with amount of work

(1) Send packets @ < 50 pkts/s

(2) Send packets @ > 50 pkts/s

WM6.5 on Tytn II

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
Components power consumption adds linearly

WM6.5 on HTC Touch

1. Send(10mb); sleep(); Socket.close();
2. Spin_CPU(10M);
3. for (i in 1 to 5){
   Send(2mb); Spin_CPU(2M);
}
   Sleep(); Socket.close();

Cellular Networks and Mobile Computing (COMS 6998-11)  
Courtesy: Pathak et al
What have we learnt so far?
Simple (state-of-art) energy modeling assumptions are wrong
There exists a notion of power states

What have we hinted so far?
Device drivers have intelligent power control rules
System calls play a role in power consumption

Challenges in fine-grained power modeling?
Device drivers are closed source (no code/no information)
System Calls As Power Triggers

Key observation: System call is the interface through which an application communicates with the underlying system (hardware) and outside world (Internet, GPS, etc.)

Key Idea: Use System Calls as triggers in power modeling

Advantages:

– Encapsulates utilization based triggers
  • Parameters of system calls
– Captures power behavior of ones that do not necessarily imply utilization
– Can be traced back to process, thread, function
  • Eases energy accounting
Finite-State-Machine (FSM) as Power Model Representation

We use Finite-State-Machine (FSM)

- **Nodes**: Power states
  - Base State: No activity on phone
  - Productive state: Actual utilization
  - Tail state: No-useful work

- **Edges**: Transition rules
  - System calls (start/completion)
  - Workload (Ex: 50 pkts/sec)
  - Timeout
FSM Power Model Construction

• Systematic ‘Brute Force’ Approach
  – Step 1: Model Single System Call
  – Step 2: Model Multiple System Calls for Same Component
  – Step 3: Model Multiple Components (Entire Phone)

• Requires domain knowledge
  – Semantics of system calls

Courtesy: Pathak et al
Step 1: Single System Call FSM

WM6.5 on HTC Touch

System call: `read (fd, buf, size);`

Measured power consumption + system calls (trigger)

Modeled power consumption

File Read
- Base +0mA
- Disk Tail +110mA
- High disk +190mA

 FSM

Inactivity for 5 sec
Free

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Pathak et al
Step 2: Modeling Multiple System Calls of Same Component

• Observation: A component can only have a small finite number of power states

• Methodology
  – Identify and merge similar power states
  – Obey programming order
  – Model concurrent system calls
WM6.5 on HTC Tytn II

**Step 2: WiFi NIC**

- **Base State +0mA**
  - Send: Send < 50 Pkts/sec
  - Send: Send done
- **Low Net +125 mA**
  - Send: Send > 50 Pkts/sec
  - Send: Send done
- **High Net +325 mA**
  - Send: Send
  - Send: Send done
- **Net Tail +280 mA**
  - Send: Send < 50 Pkts/sec
  - Send: Send done
  - Send: Send > 50 Pkts/sec
  - Send: Send done

- **Base State +0mA**
  - Socket close
- **Net Tail +280 mA**
  - Socket close

**SEND**

**CLOSE**

Cellular Networks and Mobile Computing (COMS 6998-11) 

Courtesy: Pathak et al
Step 3: Modeling Multiple Components

• Observation: Different components may interact with each other’s power consumption

• Methodology
  – Try to reach different combination of states
  – Construct new states and transitions in FSM
Implementation

• Windows Mobile 6.5
  – Extended CeLog

• Android
  – System Tap: Logs kernel events
  – Android debugging framework: Custom logging in Dalvik VM
Evaluation: Handsets Used

- HTC Tytn II
  - Win 6.5 (CE 5.2)

- HTC Touch
  - Win 6.5 (CE 5.2)

- HTC Magic
  - Android (Linux 2.6.34)

Courtesy: Pathak et al
Snapshot of FSM for Entire Phone

WM6.5 on HTC Tytn II

CPU (ctx_in) -- ctx_out

Base State +0mA

High Disk +125 mA

Disk: Read/write/open/close/create/delete

Timeout 3s

Call completed

Disk Tail +75 mA

Timeout 1.7s

Send < 50 Pkts/sec

Send done

Low Net +125 mA

Send > 50 Pkts/sec

Net Tail +270 mA

Net Tail + CPU +300 mA

Send

Send done

High Net +325 mA

CPU

High CPU +130 mA

CPU (ctx_in)
End-To-End Energy Estimation Error

- **FSM:** under 4%
- **LR:** 1% – 20%

**Courtesy:** Pathak et al
Fine-Grained Energy Estimation

CDF of energy estimation error per 50ms time interval

- FSM: 80th percentile error less than 10% for all apps
- LR: 10th percentile error less than 10% for all apps
Outline

• The Rise of Ebugs
• Methods of Measuring Power Usage
• Power Models
  – Usage based
  – System call trace based
• Profiling
• Conclusion
Energy Profiling

• eprof published in Eurosys 2012
• **QCOM Trepn Profiler**
  – Trepn leverages hardware sensors built into the Snapdragon MDP
  – Analyze power consumption of hardware blocks in the Snapdragon MDP, including:
    • CPU (system and auxiliary)
    • GPS
    • Bluetooth
    • Camera
    • Audio
    • Memory
    • Network data (optimizes data transfer frequency)
Towards Energy Profiling

- Performance

- Energy: One of the most critical issues in smartphones
- Battery energy density only doubled in last 15 yrs

1970’s-80’s

Gnu Profiler (gprof PLDI’82)

2010

Energy Profiler...?

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
Approach (a) : Power Meters

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
Approach (a): Power Meters

Cellular Networks and Mobile Computing (COMS 6998-11)

Courtesy: Pathak et al
Approach (b) : Software Energy Profiling

• Tracking power activities

• Tracking app activities

• Mapping power activities to app activities
Tracking Power Activities
Power Modeling

• State-of-art ‘utilization based’ power models are inaccurate on smartphones
  – Only active utilization => power consumption
  – Energy is consumed linearly w.r.t utilization
  – Hard to map power triggers to fine grained app activities

• System call triggered FSM based fine-grained power model [Eurosys ‘11]
  – Use system calls as power triggers
  – System calls drive Finite-State-Machine

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
Tracking App Activities

• Granularity of Energy Accounting

Multi Threading:

Third Party Ad Module

Multi Processing:

Collect information
Upload information
Download ads

Courtesy: Pathak et al
Cellular Networks and Mobile Computing (COMS 6998-11)
Tracking App Activities

• Granularity of Energy Accounting
  – eprof supports per Process/Thread/Routine granularity

• I/O Devices
  – Track system call to program entity
    • Process – getpid()
    • Thread – gettid()
    • Routine – backtrace()

• CPU
  – Just like gprof [PLDI ‘82]
  – Periodic sampling of routine call stack
Where is Energy Spent Inside My App?

- Tracking power activities
  - System call based online power model

- Tracking app activities
  - IO: Backtrace system calls
  - CPU: Just like gprof [PLDI ‘82]

- Mapping power activities to app activities
  - Accounting Policy: Lingering Energy Consumption
Lingering Energy Consumption

(a) Tail Energy

Effect on power/energy consumed by a component because of an activity lasts beyond the end of the activity

Components with tail:
- Sdcard
- 3G
- WiFi
- GPS

Power Consumed (mW)

Cellular Networks and Mobile Computing (COMS 6998-11)
Lingering Energy Consumption (b) Persistent State Wakelock

- **Aggressive Sleeping Policies:** Smartphone OSes freeze system after brief inactivity

- **Power encumbered Programming:** Programmer has to manage sleep/wake cycle of components

---

Keep the screen on!
Lingering Energy Consumption

Case 1: Single Call Single Tail

1. Energy represented in terms of an energy tuple \((U, T)\)

2. \((U, T)\) is attributed to entity \((s)\) containing send system call
Case 2: Multiple Calls Multiple Tails

How to split tail $T_2$ among?

**Average Policy:** Split tail energy $T_2$ in weighted ratio

1. Not easy to define weights
2. Policy gets complicated in presence of multiple system calls
Lingering Energy Consumption
Case 2: Multiple Calls Multiple Tails

_Last-Trigger-Policy:_ Assign asynchronous (tail) energy to the last active system call

\[
\text{send}_1 : (U_1, T_1) \\
\text{send}_2 : (U_2, T_2)
\]

1. Not easy to define weights
2. Policy gets complicated in presence of multiple system calls
eprof System
(Andriod and Windows Mobile)

Logging Overhead:
2-15% Run Time,
1-13% Run Energy

Courtesy: Pathak et al
eprof Implementation

- SDK routine tracing: extend Android routine profiling framework
- NDK routine tracing: use gprof port of NDK
- System call tracing: insert ADB logging APIs in framework code and log CPU (sched.switch) scheduling event in kernel using systemtap
  - [http://www.cyanogenmod.com/](http://www.cyanogenmod.com/)

```
AndroidManifest.xml
<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

MainActivity.java
import android.os.Debug;
---
public void onCreate(Bundle savedInstanceState) {
    ...
    Debug.startMethodTracing("HelloTest");
    ...
    Debug.stopMethodTracing();
}  

adb pull /mnt/sdcard/HelloTest.trace .
traceview HelloTest.trace
```
eprof Implementation (Cont’d)

• Augmented TraceView in Dalvik
  – gprof-like tracing + syscall tracing

• eprof API: \texttt{StartEnergyTracing()} \texttt{StopEnergyTracing()}
  – Needs app recompile

Modified Android framework to trace each app by default
(no need for app source)
## Using eprof

<table>
<thead>
<tr>
<th>App</th>
<th>Num Routine calls</th>
<th>Num Threads</th>
<th>3rd Party Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser</td>
<td>1M</td>
<td>34</td>
<td>--</td>
</tr>
<tr>
<td>AngryBirds</td>
<td>200K</td>
<td>47</td>
<td>Flurry, KHRONOS</td>
</tr>
<tr>
<td>FChess</td>
<td>742K</td>
<td>37</td>
<td>adwhirl</td>
</tr>
<tr>
<td>NYTimes</td>
<td>7.4M</td>
<td>29</td>
<td>Flurry</td>
</tr>
<tr>
<td>MapQuest</td>
<td>6M</td>
<td>43</td>
<td>Skyhook Wireless, AOL</td>
</tr>
</tbody>
</table>
Case Studies: (a) Android Browser Google Search

<table>
<thead>
<tr>
<th>Activity</th>
<th>Energy %</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>38%</td>
</tr>
<tr>
<td>TCP Conditioning</td>
<td>25%</td>
</tr>
<tr>
<td>User Tracking</td>
<td>16%</td>
</tr>
<tr>
<td>GUI Rendering</td>
<td>5%</td>
</tr>
</tbody>
</table>

 Courtesy: Pathak et al
Case Studies: (b) Map Quest

Parser. parseJSON()
MapView. OnDraw()
Search.Gas()
NetworkRequest. DoInBackgrnd()
SHW. run()

Energy Percentage (%)

CPU
Net
NetTail
GPS
GPSTail

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al
### Case Studies: (c) – Angry Birds

<table>
<thead>
<tr>
<th>Activity</th>
<th>Energy %</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Tracking &amp; Uploading Information</td>
<td>45%</td>
</tr>
<tr>
<td>Fetching Ads</td>
<td>28%</td>
</tr>
<tr>
<td>Game Play</td>
<td>20%</td>
</tr>
</tbody>
</table>

Google Nexus (3G)

---

Courtesy: Pathak et al
Case Studies (d): Facebook Wakelock Bug

FaceBookService: 25%

AppSession.acquireWakelock()
App Energy Drain Characteristics

• IO consumes the most energy
  – Most apps spent 50% - 90% of their energy in IO
  – A linear energy presentation does not help with debugging

• IO energy is spent in bursts, called bundles
  – A bundle is defined as a continuous period where IO component actively consumes energy
  – Very few IO bundles per app

Power Consumed (mW)

% of Total Energy

Browser  AngryBirds  Fchess  NYTimes  Mapquest  Photo  Upload

IO Energy Bundles

Cellular Networks and Mobile Computing
(COMS 6998-11)

Courtesy: Pathak et al.
**IO Bundle Representation**

What is the app doing here?

- **Time**: $T_1, T_2, T_3$
- **Power Consumed (mW)**: $U_1, U_2, U_3$
- **Send Bundles**: $send_1, send_2, send_3$

**Power Consumption**
- **Base State**: $+0\text{mA}$
- **Net Send**: $+265\text{ mA}$
- **Net Tail**: $+150\text{ mA}$

**5 seconds inactivity**

**Very Few Routines**

**SendPacket()**

Cellular Networks and Mobile Computing

Courtesy: Pathak et al
Optimizing IO Energy using Bundles

Why is a bundle so long?

Base State +0mA  \rightarrow Net send \rightarrow Net Send +265 mA  \rightarrow Net send \rightarrow Net Tail +150 mA  \rightarrow 5 seconds inactivity

DownloadMgr. Read()

GZipStream.read()

JsonNode.Deserialize()

SQLiteDB.insert()

Why are there so many bundles?

Base State +0mA  \rightarrow Net send \rightarrow Net Send +265 mA  \rightarrow Net send \rightarrow Net Tail +150 mA  \rightarrow 5 seconds inactivity

AdWhirl.FetchAd()

PID 0 (null loop)

Reduced energy consumption of 4 apps by 20-65% by minimizing number of bundles and reducing bundle lengths
Summary

• eprof: fine-grained energy profiler
  – Enables opportunities for in-depth study of app energy consumption

• Case studies of popular apps energy consumption
  – 65-75% of app energy spent in tracking user and fetching ads (for example angrybirds)

• Bundles: IO energy representation
  – Helps debugging smartphone app energy
Conclusion and Future work

• Ebugs need to be dealt with
  – No-sleep ebug debugging studied

• Fine-grained energy modeling and profiling very important to pinpoint energy bottleneck and ebugs
  – Accounting is tricky
  – I/O energy consumption is a major part

• Display energy modeling and profiling is still lacking
Online Resources

• Soot: open source framework for analyzing java bytecode
• ded: decompiling android application tool
  – http://siis.cse.psu.edu/ded/
• SDK routine tracing
• gprof: NDK routine tracing
  – http://code.google.com/p/android-ndk-profiler/
• systemtap: system call tracing
Questions?