Energy-Efficient Software Development

Mobile Applications

Mobile Applications: Trend

Energy Efficient Software in Mobile World

Software is getting slower more rapidly than hardware becomes faster.
Mobile Applications: Trend

90% of Time on Mobile is Spent in Apps

A system approach: transfer scheduling based on app traffic profiles

Energy Use of Mobile Applications

The average tail energy is 80% of the total energy consumption in HTC phones.

Researches in Software Community

Detecting Energy Bugs and Hotspots in Mobile Apps

Energy Bugs and Hotspots in Mobile Apps

- **Energy hotspots**
  - Application execution cause smartphone to consume abnormally high amount of battery power
  - Utilization of hardware resources is low

- **Energy bugs**
  - Malfunctioning application prevent smartphone from becoming idle
  - No user activity ongoing

---

How to Detect Energy Hotspots/Bugs?

- Running test inputs systematically generated
  - Capture a sequence of user interactions that leads to hotspot/bug situations
  - Manual validation of results

- Systematically... But, it *is* challenging!
  - No code properties relevant to energy consumption
  - No appropriate coverage for energy hotspots/bugs

---

How to Detect Energy Hotspots/Bugs?

**Which parts of software code to check?**

- System calls

**How to capture interaction sequences?**

- Event Flow Graph

*Generate EFG traces that are likely to lead to undesirable energy consumption!*

---

Energy Bugs and Hotspots in Mobile Apps

- **Hardware**
  - Resource Leak
  - Suboptimal Resource Binding

- **Sleep-State Transition**
  - WakeLock Bug
  - Tail-Energy Hotspot

- **Background Services**
  - Vacuous Background Services
  - Expensive Background Services

- **Defective Functionality**
  - Immortality Bug
  - Loop-Energy Hotspot
How to Detect Energy Hotspots/Bugs?

How can we know whether an execution trace involves a bug or hotspot?

- Executes test inputs on a off-the-shelf smartphone, measuring the power consumption
- Especially, before and after executing the application

Statistical dissimilarity to detect energy bugs
Anomaly detection technique to detect energy hotspots

- Generate trace difference

Life-Cycle of Android Activity

Sadly, not all real apps follow this setup and teardown sequence.

Complex GUI and user interactions make testing all scenarios infeasible.

Simple GUI with no user interactions

Complex GUI with user interactions

Code with Potential Energy Bug

```java
1 List<Integer> data;
2 LocationManager locationManager;
3 long Min_Update_Time = 18, Min_Distance = 1000 * 60 * 1;
4 public void onCreate(Bundle savedInstanceState) {
5     super.onCreate(savedInstanceState);
6     setContentView(R.layout.main);
7     locationManager = ((LocationManager) getSystemService(LOCATION_SERVICE));
8     locationManager.requestLocationUpdates(locationManager, 3000, 10, new LocationListener() {
9         public void onLocationChanged(Location location) {
10             Min_Update_Time = location.getTime();
11             Min_Distance = location.getDistanceTo(new Location());
12             log.v("Demo", "some exception happended");
13         }
14     });
15     data.clear(); //this can throw an exception
16     super.onStop();
17     try {
18         catch (Exception ex) {
19             log.v("Demo", "some exception happended");
20         }
21     }
22 }
23
14
```

Code with Potential Energy Hotspot

```java
1 public Object[] nonAggregatedComm() {
2     Object[] objectArray = new Object[10];
3     for(int i=0; i<10; i++) {
4         objectArray[i] = new Object();
5         log.v("Demo", "some exception happended");
6     }
7     return objectArray;
8 }
9
15
```

```java
1 public Object[] aggregatedComm() {
2     Object[] tempArray = new Object[10];
3     for(int i=0; i<10; i++) {
4         tempArray[i] = downloadObject(i);
5     }
6     return objectArray;
7 }
8
17
```

Interleaving would cause energy-inefficiencies due to Tail-Energy

fix!!
Power Consumption Rating for Hardware

- Power profile in XML
  - Provided by device manufacturers
  - Describes average power consumption ratings for hardware components
  - Used for displaying battery statistics by Android framework

Computing Energy Utilization Ratio

\[
\text{Utilization} = U_{\text{Screen}} + U_{\text{CPU}} + U_{\text{WiFi}} + U_{\text{Radio}} + U_{\text{GPS}}
\]

\[
U_{\text{CPU}} = \begin{cases} 
W_{\text{CPU200}} \cdot \text{Load}_{\text{CPU}}, & \text{if CPU is operating at 200MHz} \\
W_{\text{CPU400}} \cdot \text{Load}_{\text{CPU}}, & \text{if CPU is operating at 400MHz} \\
W_{\text{CPU600}} \cdot \text{Load}_{\text{CPU}}, & \text{if CPU is operating at 600MHz} \\
W_{\text{CPU800}} \cdot \text{Load}_{\text{CPU}}, & \text{if CPU is operating at 800MHz} 
\end{cases}
\]

\[
U_{\text{Screen}} = \begin{cases} 
W_{\text{ScreenON}} \cdot \text{Load}_{\text{screen}}, & \text{if screen on} \\
W_{\text{ScreenFULL}} \cdot \text{Load}_{\text{screen}}, & \text{if at full brightness} 
\end{cases}
\]

\[
U_x = W_x \cdot \text{Load}_x, \quad x \in \{\text{WiFi, Radio, GPS}\}
\]

Framework

Methodology

- Extracting event flows
  - Rely on Hierarchy Viewer and Dynodroid
  - Modify Dynodroid to generate EFG
  - Generate all event traces of length \( k \) and store into DB
  - Each starts from the root screen of the application

Example
Methodology (cont.)

- Extracting system calls
  - Instrument application code that invokes system calls
  - Obtain system call trace by executing application against event traces on the emulator
    - Instrumented and instrumentation-free code behavior are identical.

- Example system calls

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Number of APIs</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Management</td>
<td>3125</td>
<td>WakeLock.acquire()</td>
</tr>
<tr>
<td>Local Area</td>
<td>2116</td>
<td>WillLock.acquire()</td>
</tr>
<tr>
<td>Telecomm Network</td>
<td>1091</td>
<td>SmsManager.set Exxon(zzz)</td>
</tr>
<tr>
<td>Haptic Feedback</td>
<td>783</td>
<td>Vibrator.vibrate()</td>
</tr>
<tr>
<td>GPS</td>
<td>44</td>
<td>LocationManager.requestLocation()</td>
</tr>
<tr>
<td>Storage</td>
<td>66</td>
<td>DownloadManager.request()</td>
</tr>
<tr>
<td>Others</td>
<td>25</td>
<td>SensorManager.getAltitude()</td>
</tr>
</tbody>
</table>

Methodology (cont.)

- Detect hotspots and bugs
  - Obtain E/U ratio trace in stages during trace execution
  - Stages
    - PRE: before trace execution
    - EXC: under execution
    - REC: system returning to idle mode
    - POST: after trace execution

- Expected E/U ratio trace

Methodology (cont.)

- Compute dissimilarity between E/U ratio in PRE and POST stage and compare with the predefined threshold %.
- Detecting energy bugs
  - If the difference is greater than 50%

- Detecting energy Hotspots
  - Consider the energy trace as time-series data
  - Compute discord in the data (~ peak finding)
    - JMotif (an off-the-shelf data mining library) is integrated.
  - Rank the detected hotspots based on their magnitudes

Methodology (cont.)

- Guidance heuristics for test generation
  - Prefers:
    - Unexplored trace
    - Traces with high likelihood of leading to a hotspot or a bug
  - Parameters (trace-related)
    - Num. of system calls ($a$)
    - Similarity to exploration history ($b$)
    - Starvation of event traces due to unexplored system calls ($\gamma$)
  - Refinement formula
    - $S_E = a \times G_a + b \times G_b + \gamma \times G_a$
    - $a + b + \gamma = 1$
Experimental Setup
- Experiment setup and Subject applications
  - Trace length of 4 (26~2,800 traces)
  - LoC between 400 ~ 12,000
  - Run test framework for 20 mins

Results
- Efficacy of finding energy bugs and hotspots
  - Energy bugs were reported for 1/3 of subject applications
  - Hotspots for 3 applications
- Artifacts provided for developers
  - MonkeyRunner script for each trace
  - Energy trace pattern
  - Energy issue details (hotspots)
  - Set of system calls invoked

Results: Energy bug
- Aripuca GPS tracker
  - Missing code: Forgot to remove location updates

Results: Energy Hotspots
- Montreal Transit
  - Duplicated location update calls:
    - one for the main feature
    - the other for advertisement
## Results: System Call vs. Code Coverage

- Measured LoC covered by the traces

<table>
<thead>
<tr>
<th>Application Name</th>
<th>System Call Coverage (%)</th>
<th>Code Coverage (%)</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argot</td>
<td>100</td>
<td>7</td>
<td>11,023</td>
</tr>
<tr>
<td>Android Battery Dog</td>
<td>100</td>
<td>17</td>
<td>463</td>
</tr>
<tr>
<td>Apps</td>
<td>100</td>
<td>15</td>
<td>4,357</td>
</tr>
<tr>
<td>Kitchen Timer</td>
<td>100</td>
<td>30</td>
<td>7,101</td>
</tr>
<tr>
<td>Mostrof Travel</td>
<td>89</td>
<td>11</td>
<td>10,925</td>
</tr>
<tr>
<td>NPR News</td>
<td>100</td>
<td>24</td>
<td>6,533</td>
</tr>
<tr>
<td>Omnomnom</td>
<td>83</td>
<td>36</td>
<td>6,120</td>
</tr>
<tr>
<td>Pedometer</td>
<td>100</td>
<td>56</td>
<td>849</td>
</tr>
<tr>
<td>Vanilla Music Player</td>
<td>86</td>
<td>20</td>
<td>4,881</td>
</tr>
<tr>
<td>Simple Chess Clock</td>
<td>100</td>
<td>49</td>
<td>448</td>
</tr>
<tr>
<td>WiFi 4CE</td>
<td>100</td>
<td>27</td>
<td>304</td>
</tr>
<tr>
<td>WorldClock</td>
<td>100</td>
<td>56</td>
<td>1,157</td>
</tr>
</tbody>
</table>

*Manual test created to increase code coverage did not find additional energy bugs or hotspots.*