Energy Efficient Applications for Low Powered Devices

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Anjali Singhal
10305919

under the kind guidance of
Prof. D. B. Phatak

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai-400076
## Contents

1 **Introduction** .............................. 1  
  1.1 Motivation .............................. 2

2 **Survey on Energy Conservation** .......... 3  
  2.1 Power Encumbered Programming ............. 3  
    2.1.1 Programming APIs for Traditional Components .......... 3  
    2.1.2 Programming APIs for Exotic Components .......... 4  
  2.2 Energy Bugs ............................ 4  
    2.2.1 No-Sleep Bugs ...................... 5  
    2.2.2 No Sleep Code Path .................. 6  
    2.2.3 Implementation ..................... 7  
  2.3 Modeling Energy Consumption ............. 8  
  2.4 Optimizations in Energy Consumption ...... 10  
    2.4.1 Power-aware Application Design ............ 10  
    2.4.2 Battery virtualization ................ 10  
    2.4.3 Network Applications ................ 11  
    2.4.4 Optimizations in OLED/LCD Display Power Consumption ..... 15  
    2.4.5 CPU Intensive Applications ............. 15

3 **Problem Formulation** ................. 17

4 **Energy Bugs** ............................ 19  
  4.1 Lifecycle of android applications ......... 19  
    4.1.1 Activity .......................... 20  
    4.1.2 Service .......................... 21  
    4.1.3 Broadcast Receiver ................ 21  
    4.1.4 Content Provider .................. 21  
  4.2 Interprocedural Dataflow Analysis ......... 22  
    4.2.1 IFDS Framework .................... 22  
    4.2.2 IFDS/IDE solver on Soot ............. 23

5 **Identification of Energy Bugs** .......... 26  
  5.1 Flowfunctions .......................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1 Normal flow function</td>
<td>26</td>
</tr>
<tr>
<td>5.1.2 Call flow function</td>
<td>27</td>
</tr>
<tr>
<td>5.1.3 Return flow function</td>
<td>27</td>
</tr>
<tr>
<td>5.1.4 Call-to-return flow function</td>
<td>27</td>
</tr>
<tr>
<td>6 Implementation Details</td>
<td>29</td>
</tr>
<tr>
<td>7 Results</td>
<td>31</td>
</tr>
<tr>
<td>7.0.5 Test Case</td>
<td>31</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>33</td>
</tr>
<tr>
<td>8.1 Future Work</td>
<td>33</td>
</tr>
</tbody>
</table>
Abstract

The smartphones, now a days, are not only used for the basic purposes such as calls, etc. but also for many different applications for many different purposes releasing everyday for navigation etc, hence constraint on energy consumption has become huge bottleneck. We study various techniques to improve the battery life focusing on application level so that developer of each application can build the application such that energy consumption of his application can be optimized reducing the burden on battery of the device. Researchers have provided many solutions to optimize energy consumption that includes detection of energy bugs, battery virtualization, also various optimizations for different kind of applications. We also study for network intensive applications, CPU intensive applications, etc many approaches to optimize the energy consumed, considering energy-performance trade-off. For these optimizations, developer should have clear idea of where the energy is spent inside his application so we also discuss various tools and techniques available to model energy consumption. However, none of these solutions are 100% accurate and remove all energy bugs or provide full optimizations. We identify improvements that can be done in existing approaches to make them more accurate. We focused on implementing techniques to detect energy bugs in the android applications. We presented and implemented algorithm to detect no sleep code paths by extending a generic IFDS/IDE solver for simple android applications having no asynchronous communication. In future it can be extended further to analyze and detect no sleep bugs in all android applications.
Chapter 1

Introduction

Energy efficiency and optimizations can be made at atleast three levels. At the architecture level through multi-core processors, increasing the sleep states energy efficiency can be achieved. At the system layer, through hard disk spin-down, multi-threading or compiler driven optimizations, energy efficiency can be achieved. Application layer has most of the information on the actual performance and energy trade-off. By changing algorithm or design at the application level, optimizations can be made. To allow the developer to make these optimizations, information on energy usage of the application is required and among various algorithm design options which need simple changes in source code like using different library functions, changing storage patterns etc. In smartphones, aggressive sleeping policy is applied that is, by default every component stays in idle state, unless the developer writes code to instruct the OS to keep it on. List of components broadly fall into two categories[15] :

- **Traditional Components** : CPU, WiFi, NIC, 3G Radio, memory, screen and storage that are also found in desktop and laptop machines.

- **Exotic Components** : GPS, camera and various sensors.

The power consumption of individual I/O components (camera, GPS, etc) is generally higher than the power consumption of CPU in case of smartphones. Smartphones employ an sleeping policy that puts the components to sleep state after a certain user inactivity period. Smartphones face a new class of abnormal system behavior, namely energy bugs[15], that causes unexpected amount of high energy drain by the system. Hence focus is to provide developer with different application level optimizations so as to reduce power consumption, to find energy bugs and their root cause i.e. in nutshell to study low power techniques for android based smartphones. Android has been chosen among others for its rapid increasingly popularity in the smartphone market and a promising software framework, which is open software. Also to study the detection energy bugs and providing optimizations in application design, reduce the energy consumption of network, LCD and CPU.
1.1 Motivation

In recent years, user preferences in mobile network usage and mobile phone have changed tremendously, and they are no more used just for making calls or messaging. Now its also used for mobile internet connectivity, gaming, multimedia applications, social networking sites and also for education. Due to constraint on size and weight of smartphones, etc their battery capacity is limited. So to meet the recent trend of mobile usage, energy efficiency of such devices is very important. At the same time, functionality of the device is also increasing. Smartphones combine the functionality of basic usage of mobile devices with capabilities of PC applications. Energy management is crucial as all these rich functionalities puts lot of pressure on battery life. Hence good understanding, of where and how the energy is used, is required.

Also optimizing the energy consumption of existing smartphone apps is of vital importance. There have not been significant improvements in battery technology. Furthermore there is little evidence that this situation will improve dramatically in next few generations. At the same time, these devices are running more applications with widely varying power consumption characteristics. Hence, managing the battery resource will continue to dominate the list of challenges for some time to continue. Designers of software-hardware platforms for smartphones have added power saving features, allowing developers to manipulate their power consumption based on different functionality. Many software developers generally focus on application functionality rather than efficient power consumption. As a result, many smartphone apps are drain more power. Android is the latest trend in mobile OS. Even though, android provides a complete set of application, middleware and linux kernel for the phone applications developer, it does not utilize several standard kernel features. Hence the need of optimizations and energy efficiency is critical.
Chapter 2

Survey on Energy Conservation

We surveyed all the techniques to conserve energy in smartphones by detecting the energy bugs, to model the power consumption of applications and various optimizations in design, network components, LCD and CPU.

2.1 Power Encumbered Programming

The default policy in smartphones is that every component, stays in idle state, unless the developer explicitly writes the code to keep it on. Due to such policy application developers have to explicitly use power manager APIs exported by the OS to keep component on. Hence the term power encumbered programming[15] is proposed. All smartphones employ an aggressive sleeping policy, that is, components are put to sleep or idle state after some inactivity period. This sleeping policy impacts apps if some app is interacting with external network.

To avoid such disruption, power manager APIs for app developers are provided. Android framework provides power management APIs for developers to acquire and release wakelocks. Such explicit management [2] of smartphone components has significantly burdened developer with more programming issues to deal with for less consumption and its called as power encumbered programming.

2.1.1 Programming APIs for Traditional Components

An android framework provides wakelock functions through PowerManager.Wakelock class with different flags for managing traditional components: CPU, Screen & keyboard backlight. A wakelock is instantiated using one of the options affecting different hardware component. The app declares and acquires wakelock, due to which irrespective of user activity OS does not put CPU, screen, etc to sleep state. Once the critical task is completed, the developer releases the wakelock in the application. Single wakelock controls one or more components. In case of non reference counted wakelocks, an acquire wakes up the component and single release will set the component free to sleep irrespective of no of times acquire is called due to non reference counted wakelocks.

In case of reference counted wakelocks, each acquire increments the counter associated with that particular wakelock and a release decrements it. The component is sleep only if the counter reaches to zero. Same component can be held by multiple entries( threads).
2.1.2 Programming APIs for Exotic Components

Components such as camera, GPS, several sensors are the biggest consumers in smartphones including few traditional components and drain battery at high rate. They are also turned on, off by programming APIs. The incorrect usage of power manager APIs can lead to increase in energy utilization of these components, wasting significant amount of battery power. Moreover, the event based nature of smartphones apps increases the complexity of the code even higher due to interactive nature of apps. Android application is event based. Hence, developer needs to manipulate wakelocks accordingly.

<table>
<thead>
<tr>
<th>Component lock/manager name (API to start/stop)</th>
<th>Component(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partial Wake Lock</strong> (acquire/release)</td>
<td>CPU</td>
<td>CPU runs despite any timers</td>
</tr>
<tr>
<td><strong>Screen Dim Wake Lock</strong> (acquire/release)</td>
<td>CPU and Screen (DIM)</td>
<td>No illumination if shutdown, else illuminates till lock release (Flag ACQUIRE_CAUSES_WAKEUP forces illumination in all cases)</td>
</tr>
<tr>
<td><strong>Screen Bright Wake Lock</strong> (acquire/release)</td>
<td>CPU and Screen (bright)</td>
<td></td>
</tr>
<tr>
<td><strong>Full Wake Lock</strong> (acquire/release)</td>
<td>CPU, Screen (bright) and Keyboard backlight</td>
<td></td>
</tr>
<tr>
<td><strong>Proximity Screen Off Wake Lock</strong> (acquire/release)</td>
<td>Screen, Proximity Sensor</td>
<td>Screen shuts if sensor activates</td>
</tr>
<tr>
<td><strong>LocationManager</strong> (requestLocationUpdate/removeUpdates)</td>
<td>GPS</td>
<td>Tracks user location</td>
</tr>
</tbody>
</table>

Table 2.1: Wakelock flags[15]

2.2 Energy Bugs

Energy bug[15], or ebug, is defined as state in system, when an application, OS, etc causes huge amount of energy drain. They may occur due to programming mistake, inappropriate use of power manager API, incorrect application design, asynchronous messages, remote server crash, or from hardware fault. Ebugs do not crash the application, the whole system keeps running normally, but only consuming unexpectedly large amount of energy making battery discharge very quickly. Due to this detection of energy bugs and their cause becomes even more difficult. Ebugs were reported in [15] from the bugs reports of android and error reports from discussion forums. From each posted thread started by a user who observes a specific problem with the mobile, followed by emails from other users/developers discussing the issue. Ebugs were extracted by authors of [15] from these forum postings by applying regex. Then k-means was applied which clustered the posts based on the text contained. They (manually) selected clusters which contained ebug symptoms. Ebugs were categorized as:
• **Hardware Ebugs:**
  
  – *Battery:* May occur due to faulty battery i.e. it does not hold complete charge upon charging and it drains internally which results in heating.
  
  – *Exterior Hardware Damage:* Damage to the external buttons of the phone causing random unlocking, etc.
  
  – *SIM:* Scratches on the SIM increased drain due to increase in internal resistance.
  
  – *SD Card:* Corrupted SD card could trigger buggy apps into looping state where they repeatedly try to access the hardware.
  
  – *External Hardware:* Erroneous phone chargers often partially charge the phone.

• **Software Ebugs:**

  – *OS Ebugs:* Root causes were buggy OS processes example some process may be running in the background and draining battery and configuration changes such as incorrect profile to SetCPU resulted in overclocking and underclocking.

  – *Application Ebugs:*
    
    *No-Sleep Bug:* “no-sleep bugs” in applications erroneously do not allow at least one component of the phone to sleep. A no-sleep bug is a situation where an application acquires a wakelock with a flag indicating which component to wakeup, but does not release or it is not released in of the possible path even when the task is finished. These bugs are detectable in cases where the locked component is easily noticed, e.g., screen (sms application), but very hard to detect where its activities are not easily noticed, e.g., GPS and CPU (map based apps). Root cause may be programming mistake, race conditions that prevented lock release and sleep conflicts.

    *Sleep Conflicts:* If a component is triggered into a high power state, and the CPU becomes idle due to user inactivity, the component remains in high power state as the program that triggers the component to a low power state can not execute until the CPU is woken up.

    *Loop Bug:* Loop bug happens where an application is waiting for message from remote client and remote client has crashed. Eg. If a remote server crashes, email password is changed, the clients repeatedly try to connect to the remote server, or perform email-authentication.

    *Immortality Bug:* An immortality bug is a situation where a buggy application upon being terminated by the user, respawns, and keeps draining battery.

2.2.1 No-Sleep Bugs

In the reference [16], ebugs were collected by crawling web forums. The apps containing these bugs (binary installers) were downloaded and decompiled to java source code using tool ded[3] by the authors[16] and then the root cause was analyzed. Not all the apps were decompiled as some of the apps were obfuscated with proguard. The analysis for no-sleep bug was done on the apps decompiled having no sleep bug and they characterized no-sleep bugs as:
• **No-Sleep Code Path:** The root cause found, for the existence of no sleep bugs, was due to presence of such a path where wakelock is acquired by the application but not released. Even existence of single such path would make the application buggy and induce energy bug. The causes for the existence of such code paths observed were:

1. One cause is that developer acquired the wakelock but forgot to release it, or he released it in if branch but did not release in the else branch.
2. Another cause is that developer acquired wakelock and released the wakelock but on one of the possible paths the wakelock was not released, and control took that path and due to which wakelock was kept acquired.
3. Application level deadlock might prevent the code to reach release statement.

When an activity is started, it remains in the activity stack and may come to foreground or pushed to background. When user exits app, the state of app is saved and returned back when user returns to it. The app is killed only when the android device has no memory left in RAM or when onDestroy is called by the app. When the app goes to background onPause event is called and when it is again moved to foreground onResume is called. onDestroy() is called only when the application component is about to be removed from memory. Tracking all possible paths for an application to detect wakelock bugs and acquire/ release statements is very difficult since due to event based system, some paths may not be taken into account. Hence the automated technique to find such code paths is required.

• **No Sleep Race Condition:** An activity may start a service to do some background tasks where UI is not required. In such a case service runs in separate thread. Developer may acquire wakelock in activity and release it in service. But, it may be case that the thread releasing the wakelock may get executed before the thread acquiring it. Developer would have released the wakelock but still the component is left waken draining the battery, resulting in energy bug. Finding energy bugs in source code with race condition is very difficult since all possible execution paths have to be considered.

• **No Sleep Dilation:** In this type of energy bug, the component is put to sleep state after long duration than expected. However, keeping the component in high power state may be intended by the app developer. Causes of this may be explained by examples such as after acquiring the wakelock waiting for an external event.

Debugging of the no sleep bugs is done at compile time as it incurs no runtime overhead, will determine facts that are applicable for every run. Acquiring and releasing of wakelock l is considered as definition of variable \( w_1 \) corresponding to value of l according to [16]. A definition of variable \( w_1 \) is said to have reached p, there is path from d to p where \( w_1 \) is not redefined. If the definition \( w_1 \) reached exit of code, then there exists no sleep bug. Hence it becomes reaching definition dataflow problem.

### 2.2.2 No Sleep Code Path

Due to event based system of android, many possible paths are present for an application. Dataflow analysis is performed which analyses the effect of each statement along all possible paths and a directed control flow graph is created. Every node in control flow graph represents...
basic block of statements. Every directed edge \( n_i \rightarrow n_j \) in the CFG connects block \( n_i \) to \( n_j \). This CFG also incorporates exception graph i.e., includes exception edges along with regular edges. For every node, two sets are computed, GEN and KILL. Gen set represents the new facts that are generated in the node and KILL set represents the facts which become false in the node. Each node also IN and OUT set which deals with the information flowing in from other nodes and updates it producing OUT. Forward analysis of CFG is considered for reaching definition analysis. Meet operation is applied if there are multiple incoming edges to any node, and then its IN is calculated. If there are cycles in CFG, then the algorithm is repeated till the analysis converges to a fixed point. Conservative approach is followed here by applying union as meet operator to all paths. Only non-reference counted wakelocks are analyzed by Reaching Definitions Problem.

The domain of the analysis is a set containing all wakelocks. Once the acquire/ release statements are transformed to assignment definitions (0/1), the dataflow problem becomes to finding Reaching Definitions, that is all the wakelock definitions reaching at the exit of the CFG. If the definitions with variables as 0 only reaches at the exit, the application is said to have no energy bug.

Null pointer exception, etc which are directly handled by JVM may also cause no sleep bug. Each unhandled RTE to exit node edge is created. This will create many false positives. For event based entry points, each event has its own handler and CFG with multiple entry points. The order of execution matters a lot which may be unknown at compile time. Hence for default events default ordering is considered and for other events, developer is asked to give order of invocation.

2.2.3 Implementation

The dataflow analysis is applied on the .class files of the code obtained by decompiling binary installers of the application using ded tool. Intents causes indirect control transfer in the Android framework which creates problems in static analysis as at runtime it is very difficult to determine which component it will call to handle the intent. Conservative approach can be applied by analyzing all possible components that may be referred. In case if developer checks if the wakelock is held then only release it, so no sleep bug is present but the else part will reach to exit node and report no sleep bug. So in these cases else branch with definition to release is added to remove false positive.

- True Positive : No sleep bug reported manually and also by the analysis
- True Negative : No bug is reported manually as well as by the analysis
- False Positive : No bug is reported manually but it is reported by the analysis
- False Negative : Bug is reported manually but not by the analysis

Hence we studied the analysis of energy bugs in the applications. Further after removing the bugs, still many optimizations can be made to these apps in order to conserve energy. For this sake, the developer must know where and exactly how much energy is been consumed by the application and each hardware component while the application is running. Hence modeling of energy consumption for apps is required.
2.3 Modeling Energy Consumption

To achieve energy efficiency, where and how energy is consumed, this information should be known. For achieving efficiency in applications it is very crucial that the information of where the energy is spent to be known by the developer. There have been several attempts in modeling the energy consumption by different techniques. Two approaches have been explained here:

2.3.1 Powertutor

Powertutor[4] is a tool that monitors power consumption of the application based on built-in voltage sensor and battery discharge curve. It requires no external measurement equipment. System variables such as LCD brightness, etc are used to by powertutor to estimate power consumption. It calculates function of system variables to determine power consumption of overall system.

Built-in voltage sensor is used to create power model for the system. In paper[20], they used HTC Dev Phone 1 and Monsoon FTA22D meter for power measurement. Two programs are executed, first one to control the system variables CPU utilization, LCD brightness, etc which may be imprecise. Second program captures most changes record logs at high frequency. By these two programs, in short time span many power states are collected, and hence at particular time system state is analyzed. To calculate power consumed by particular component, all states except the concerned activity are kept constant, and only one is varied to extreme values. By calculating power state of individual components independently, error found out was 6.27%[4]. Hence to estimate system power consumption, sum of independent component power consumption is done.

- **CPU**: It is strongly influenced by CPU utilization and frequency. Hence, to estimate CPU power consumption, system variables utilization and frequency-voltage settings were characterized. Power consumption difference of application processor’s active state and idle state was calculated.

- **LCD**: Power consumption of LCD display is modeled by varying its brightness.

- **WiFi**: Considered two network parameters: data rate and channel rate. Power consumption of WiFi is modeled by varying these system variables: no of packets sent and received per second, upload link channel rate, and upload link data rate. The card remains transmitting state for approximately 10-15ms for high data rates. For low data rates it is even shorter. When WiFi interface is operating on low data rates then it is in low power state. When the data rate crosses 15 packets per second then WiFi interface goes to high power state and it switches back to low power state when data rate becomes less than 8 packets per second.

![Figure 2.1: Wi-Fi interface power states[20]](image)
• **Cellular:** Its power consumption model depends on send, receive data rate and sizes of queue. Cellular device can have three states: \textit{CELL, DCH}: A dedicated channel to communicate with base station is provided and thus has high speed uplink/downlink rates. After the period of inactivity, \textit{CELL, FACH} state is reached. \textit{CELL, FACH}: Communication channel to the base station is shared, has less data rate and can also access other channels (\textit{CELL, RACH/CELL, FACH}). This channel, after period of inactivity goes to IDLE state. IDLE: Here it only receives but does not transmit.

• **Audio:** Power consumption is modeled by varying the volume.

**Battery Based Automated Power Model**

During discharge, the voltage of a lithium-ion battery varies, through which estimation of power consumption is done. The state of discharge (SOD) represents battery discharging percentage. Energy capacity & discharge curve varies with current, temperature. A battery is modeled as shown in figure 2.2. Thus by calculating voltage drop across the variable resistor, power consumption can be calculated. To convert the battery voltage readings into power consumption values, SOD variation is calculated.

\[ T \times (t_1 - t_2) = E \times (\text{SOD}(V_1) - \text{SOD}(V_2)) \]

where \( T \) is power consumed in \([t_1, t_2]\) time, \( E \) is the marked battery capacity\cite{4}, and \( \text{SOD}(V_i) \) is the SOD at voltage \( V_i \). There is variation in battery discharge curve for various batteries hence the same table for all batteries might be inaccurate. Hence, for each battery separate discharge curve has to be considered. For creating discharge curve for particular battery, it is completely discharged from full capacity to zero capacity with constant current. With temperature and age, the discharge curve of the same battery may vary. For a new battery, \( E \) can be obtained from the marked label itself. Due to battery’s internal resistance, the discharge curve depends upon the current. Hence, even if the voltage is constant, \( T \) depends upon discharge current. While taking voltage reading, all the components are put to low power or idle modes to that the impact of battery’s internal resistance is minimal. To measure accuracy comparison of battery based and meter based models were done.

\[ \frac{|\text{measured} - \text{predicted}|}{\text{measured}} \]

Constructing battery discharge curve requires three steps:
1. Each individual component’s discharge curve is constructed starting with fully charged battery each time.

2. Find out the power consumption of each component by varying the state of that particular component while keeping all other component’s state constant.

3. Construct regression model to get the power consumption of whole system.

This model has to constructed only once for every new phone and this tool removes the need for external equipment to measure the power consumption.

### 2.4 Optimizations in Energy Consumption

The application design stages are not fully power optimized. There are significant opportunities present for power optimization in that area. Virtualization of the battery resource of mobile devices across application classes is one of the optimization. Designing device power manager that allocates device resources to different applications in accordance with device level power management policies set by user. Networking itself is very expensive. A notion in cellular communications is called tail time, namely the high power state period can also be put to utilization.

#### 2.4.1 Power-aware Application Design

Developers perform energy profiling to select between different designs in energy-performance trade-off. The design choice is very important to manipulate the energy usage. Let us take an application that access file containing several data items. The developer can access the secondary storage in two ways: first one is to compresses the stored data and one that does not, as mentioned in [13]. The best choice needs to be determined by the developer. Both options has library routines exist to code them and the selection of the routine does not affect the application logic in any way. However, processing of different amount may reduce the reduce IO load but may increase CPU load. The amount data processed affects the cost. For particular well-known application domains it is useful due to constraints on the bottleneck resource, but in general, the application’s design choice is not obvious. As discussed above, The choice can be guided by the energy profiling. Compression increases the CPU usage dramatically. Compression reduces the disk usage significantly. Using the compressed file stream, in fact saves energy and improves the performance. If such detailed visibility into energy costs is available to the developer at design time with each design change then it becomes easier to optimize the energy consumption of the application. Profilers discussed earlier provide fine grained profiling of applications as in where the energy is spent on different hardware components. Different techniques or algorithms for any application can be compared using these profilers and based on energy-performance trade-off, most suitable technique can be chosen by developer.

#### 2.4.2 Battery virtualization

In PowerVisor[21], each application class is assigned a virtual battery based on user level policies. Providing guarantees on battery usage on per-application or application-class level provides unique benefit to mobile users so that they can be less worried about running out of power for critical applications. Having a battery allocation for each application (navigation, phone, game,
email) will ensure that they have access to a fraction of the battery as per user’s individual policy. Some users may allocate a higher fraction of the total battery to games or navigation, while others may prioritize email or voice calls. Some applications could be grouped into work, entertainment, and so on, with appropriate battery shares provided to each application class. First approach is based on the idea of thresholding. In this approach, the user assigns a battery level threshold to each application class, and the operating system ensures that an application from a particular class can run only if the battery level is above its assigned threshold. The second approach relies on the concept of virtualization. The operating system virtualizes the physical battery into several virtual batteries; one for each application class. Each virtual battery is allocated a fixed share of the physical battery, irrespective of how the other application classes utilize their virtual battery. Per-application energy usage can be estimated based on the techniques discussed above, battery can be virtualized. But the implementation of this virtualization scheme itself consumes significant power compared to other important applications, hence it is not put to action.

**Optimizations in different applications:** Major contributors for energy can be identified and the optimization algorithms for the optimal usage will help us a lot making applications energy efficient.

Major contributing hardware components in energy consumption are to be found so that optimizations in consumption of high energy consuming devices can be done. In the study done in [10] for different benchmarks consuming variable amount of energy comparison of the consumption is done and major role players are found out. National Instruments PCI-6229 DAQ on freerunner device is used to measure the voltages. This measurement approach yields the power directly consumed by each component.

**Baseline cases:**

- Suspended device: The communications processor performs while application processor is idle results in a low level of activity, as the network connection should not create problem in receiving calls, etc. The GSM subsystem power dominates.

- Idle device - If applications are inactive the device remains in idle state but the display consumes a large amount of energy at that time. GSM is also a large consumer

As per study in [10] for difference benchmarks such as in video applications, display and GSM are the biggest consumer, for text messaging display is biggest consumer, for phone call and emailing GSM dominates, GPRS dominates for web browsing. The power consumption attributes to the GSM module, display, LCD panel and touchscreen, graphics driver and the CPU. proper usage of the attributes can decrease the power consumption for example using light-on-dark color scheme.

### 2.4.3 Network Applications

Most commonly used mobile networking technologies are 3G, GSM, and WiFi. As we saw network applications are huge power drain and can considerably reduce battery life. In data transfer through network components through either 3G or GSM the energy consumption can be divided in three parts referring to figure 2.3. 1) Ramp energy: energy required to reach to the high-power state, 2) Transmission energy, and 3) Tail energy: energy used to be in high-power state being idle after the transfer ends. In case of WiFi it can be divided as: 1) Scan associated
to an access point and 2) Transfering data. From the study done in [7], in 3G, 60% of the energy is being wasted as the Tail energy. Ramp energy is very small with respective to the Tail energy. A similar trend exists in GSM too. The tail time is smaller in GSM, compared to 3G (6 vs. 12 secs). TailEnder[7] is an energy-efficient protocol to design the scheduling of data transfers.

![Figure 2.3: 3G/ GSM Power Consumption](image)

It considers two classes of applications: 1) delay-tolerant applications (eg. email and RSS feeds), and 2) applications (eg: web search and web browsing). After completing transfers TailEnder does outgoing transfers to minimize the overall time spent in high energy states, while respecting delay-tolerance deadlines. TailEnder determines what data to fetch so that the overall energy consumption can be minimized. This prefetching useful data can cause less energy consumption but if we fetch non-useful data then it will increase the consumption thats why TailEnder uses a probabilistic model to do it. In case of 3G, GPRS/GSM, inactive timers controls the transition between the different states. Inducing tail state serves benefits such as it reduces the overhead of switching back to high state again and again if there exists transmission very soon in future. In case of WiFi, association with an access point is high in terms of initial costs. In the above association, the transmission power level and the amount Of data transfer determines the total energy consumed. Wifi consumes less energy that 3G and GSM technology, especially when we transfer large amount of data. The measurement was done using Nokia Energy Profiler on HTC phone using hardware power meter. Their energy measurements were calculated as:

**3G Measurements:** Tail energy consumes significant amount of the total energy. Ramp energy consumed is significantly small compared to tail energy. The average energy per transfer is being reduced when the time takes in less than the tail time as it can’t get the full tail penalty. So TailEnder [7] is designed such that the use of the tail energy can be maximized in multiple transfers, only if the transfers occur within tail-time of each other.

**GSM Measurements:** The Tail energy is comparable to the transfer energy but it is less than the tail energy consumed in 3G. However, like 3G, the ramp energy is higher than the tail energy in GSM. Here data sizes impacts more than the inter-transfer times for power consumption.

**WiFi Measurements:** It was observed in [7] that the transfer energy is very less compare to the scanning and association energy. The energy consumption is proportional to the time duration between successive transfers due to the high maintenance energy in WiFi.

**3G vs WiFi vs GSM:** According to the analysis in [7] 3G, the rate of the power consumes is greater than the GSM and WiFi for data downloading. This is due to two reasons 1) the power
level is higher in 3G than GSM and (2) the Tail energy set for 3G is much higher than GSM. With increasing data sizes WiFi’s efficiency increases dramatically. In the above discussion we saw the WiFi is more efficient than the GSM but in case of scan and transfer of low amount of data, GSM is more efficient, but in compare with 3G, WiFi is still more efficient in this case also.

**TailEnder**

Applications can be classified under two categories:

1. **Delay tolerant applications**: If delay in sending email won’t make much problem to the user we can achieve power efficiency by sending multiple emails together, scheduled in a sequential manner. The goal is to minimize the tail energy of multiple scheduled transmission of data by putting delay in such a manner that one transmission’s tail overlaps with another’s data transfer energy keeping the deadline constraint in count. For example let us consider n requests where i(th) request has an arrival time ar_i and a deadline de_i. Let P = {p_1, . . . , p_n} denote a transmission schedule that transmits request i at time p_i. The schedule P is feasible if it transmits each request i after its arrival ar_i and before its deadline de_i. After each transmission the device remains in high power state for the tail-time(T). Generally, the request sequence is not known apriori. Tailender uses following mechanism: each incoming request is isolated until its deadline falls within ρ where ρ ∈ [0, 1]. According to TailEnder, ∆ is the last deadline for the packet transmitted at time t. Whenever any request re_i arrives at time t having arrival time ar_i having deadline de_i, if de_i > t and for the last request scheduled at ∆, the tail state is going on, then transmit re_i. Else if no request’s tail state is going on even if de_i > t then add re_i to queue. if (t == de_i) then transmit re_i and all requests in queue.

2. **Applications benefiting from prefetching**: By aggressively prefetching the documents lot of energy can be saved in the form of tail energy and on the other hand a large amount of energy will be wasted is the prefetched documents doesn’t required by the users. The TailEnder tries to solve the issue of the amount of prefetched documents to maximize the reduction of energy consumption using a statistical analysis of the user behavior. The prefetched documents size determines the amount of energy saving. Let f be the number of prefetched documents (in decreasing rank order) and P(f) be the probability that a user requests a document within rank r. Let E be the Tail energy, R(f) be the energy required to receive f documents, and TE be the total energy required to receive a document. The expected fraction of energy savings if the top f documents are prefetched is

\[
\frac{E * P(f) - R(f)}{TE}
\]

Research in the [7] from crawlers showed that prefetching of 10 web documents can save maximum energy. But we have to make a balance between the prefetched documents and the user requirements thus it maximize the cost reduction. Thats why, TailEnder prefetches 10 web documents for each user query[7].

The efficiency of WiFi is higher but availability is low, thats why we need to switch to 3G when WiFi is not available. Thats why the energy benefits can be achieved significantly if
we can combine WiFi and 3G for mobile without affecting the availability of mobile networks. Cross-application opportunities are not explored by tailender.

The issue of utilizing tail time has been attempted in two ways:

- **Traffic Aggregation:** TailEnder tries to aggregate small transmissions into large ones so that the occurrence of Tails (and thus energy consumption) can be reduced. Lack of a high prediction accuracy of future transmissions may result in unnecessary energy consumption in the aggregation. So there is chance of increasing the energy consumption rather reducing it. Furthermore, even after the aggregation, there are still a number of Tails with heavy energy waste.

- **Tail time tuning:** Tail time cannot be dynamically changed based on future traffic patterns. TOP dynamically terminate the tail time when it predicts there is no future transmission however very high prediction accuracy is required.

**Tail Theft**

1. TailTheft[14] steals the existing Tail Time for two tasks: (i) prefetch the data likely to be requested in the future (for news, video, etc.), and (ii) defer the delay-tolerant data to be transferred later (for email, etc.). So, a number of transmissions can be scheduled to the Tail Time of other transmissions. Thus the energy consumption is significantly reduced.

2. The stealing of Tail Time is achieved through a Virtual Tail mechanism and a Dual Queue Scheduling algorithm. The Virtual Tail mechanism: (i) maintains a virtual Tail Time of the same length with the physical one after the completion of a transmission; (ii) leverages the period of virtual Tail Time for prefetching and delayed transfer; and (iii) switches the device from the high to the low power state at the end of the virtual Tail Time. The Dual Queue Scheduling differentiates (i) the transmissions that could be prefetched/deferred and (ii) other transmissions, and schedules the former to the virtual Tail Time of the latter.

3. TailTheft does not tune the length of Tail Time. Hence, TailTheft does not increase energy consumption, even if the prefetched data would not be requested by the user in the future.

- **Virtual Tail:** If tail time is reduced or cut out then due to many state promotions energy consumption will increase. TailTheft maintains a virtual Tail Time of the same length with the physical one. Generally, this Tail Time consumes energy but does not transfer data. A timer is employed to decide whether it is in the virtual Tail Time now. During the virtual Tail Time, if there are delayed past transmissions or future requested transmissions, TailTheft schedules them. Prefetching or delaying is indicated by assigning value to the argument $t_{\text{delay}}$ in the TailTheft API call. If $t_{\text{delay}}$ is zero, the transmission cannot be prefetched/deferred; otherwise, if it has positive or negative value, the transmission could be deferred or prefetched respectively. Finally, at the end of the virtual Tail Time, TailTheft switches the device from the high to the low power state through Fast Dormancy. Fast Dormancy is feature defined in 3G standard specification 3GPP which can be employed to terminate the Tail immediately. If the transmission has not been finished at the end of the virtual Tail Time, then it goes in vain. TailTheft is more applicable for applications with periodical small traffics such as RSS, emails, news, and small data sync.
If needed, the virtual Tail Time could have an increased duration with a reasonable gap to the physical Tail Time, instead of a duration exactly the same as the physical Tail Time.

- **Dual Queue Scheduling:** TailTheft differentiates the two types of transmissions based on the value of $t_{\text{delay}}$ in the TailTheft API call. For normal transmissions, TailTheft schedules the corresponding packets as soon as a transmission request is received, and adds in a queue called normal queue. For TailTheft transmissions, TailTheft add in the queue called TailTheft queue, and do not schedule the TailTheft transmissions until there is no packet for normal transmissions. TailTheft employs a timer (i.e., $T_{\text{timer}}$) to quantify the time span after the transmission of the last normal packet. If there is no normal packet to be transferred and the time span is in the Tail threshold (i.e., $T_{\text{tail}}$), TailTheft schedules the first transmission in the TailTheft queue. If at the current time ($t_{\text{c}}$), an application decides it could prefetch a transmission before the deadline ($t_{\text{d}}$), the application could call the TailTheft API with the argument $t_{\text{delay}}$ set to be $t_{\text{c}} - t_{\text{d}}$. In this case, TailTheft schedules the transmission to a time before the deadline, and the prefetching and delayed transfer can share the same scheduling algorithm.

The queue for normal transmissions is checked. If it is not empty then the normal transmissions are scheduled. Else if the tail time of some normal transmission is going on then from the tail theft queue small transmissions are scheduled.

### 2.4.4 Optimizations in OLED/LCD Display Power Consumption

An LCD display system is composed of an LCD panel, a frame buffer memory, an LCD controller, and a backlight inverter and lamp.

The LCD backlight is the dominant power consumer, with the LCD panel and the frame buffer coming the second and third according to [11]. Dynamically setting the brightness of backlight in response to changes in the ambient luminance conserves energy. Frame buffer compression [18] can also be used to reduce the power consumption of a frame buffer memory and its associated buses. LCD controllers periodically refresh their display at 60Hz, or an even higher sweep rate, and the frame buffer is therefore very active. Frame buffer compression reduces the activity of the frame buffer using run-length encoding (RLE) and thus its power consumption.

An OLED display consumes different amount of energy for different colors. Hence many inversion techniques for non-interactive or content less area of application are applied which affects the energy consumption significantly.

### 2.4.5 CPU Intensive Applications

Processor has speed of (say) 2GHz, it does not mean that 2GHz is the one and only frequency at which processor can operate. The value 2GHz suggests that it is the maximum frequency at which processor can operate stably. The system will get maximum throughput if processor keeps functioning at this frequency. But higher frequency also means higher power consumption. As we know that mobile and tablet devices have limited power backup. If their CPU is kept at maximum frequency constantly, then batteries will drain quickly. The solution is to switch CPU frequency between available set of frequencies depending upon CPU load. Linux kernel allows CPU frequency scaling operations by means of ’cpufreq’ component[1].
(v2.6.35.13) has implementation of 5 governors. Governor, part of cpufreq, is responsible for deciding suitable CPU frequency. Every governor implements different logic on how to switch in-between available frequencies. The documentation about `cpufreq` can be found in kernel source code, located at ./Documentation/cpu-freq
Chapter 3

Problem Formulation

From above survey several enhancements and problem identified in the above mentioned techniques are discussed as follows:

- **Energy Bugs**: In the technique proposed in [16] as explained in section 2.2, the dataflow analysis is used for detecting no sleep bugs. The solution works very fine for single thread applications and even for multiple thread applications for which execution order is specified by developer on compile time. Several improvements and enhancements that be made in this technique are

  - Removing or decreasing the no of false positives in the solution proposed.
  - Taking in account explicit synchronization for multithreaded applications. Also taking care of intent resolutions so that the appropriate target method is only considered during analysis and not all possible methods that can be called, thereby reducing false positives as well.
  - In the future, most of the applications will be offloaded to external servers, i.e., most of the code of applications will reside on those external servers. Hence the detection of energy bugs due to external conditions will be of critical importance which is not handled anywhere yet as per our knowledge. We propose to provide a solution to detect energy bugs due to external conditions such as looping bugs, etc in the next stage of our project.

- **Modeling Energy Consumption**: In case of powertutor, different models for different types of phone has to be constructed. Even when model for particular phone is constructed, eventually battery’s capacity will change in few years or months, so again modeling has to be done. To solve this issue in future the discharge curve of same battery of different capacities can be analyzed. That is if the slope of the curve is depending on the resistance in some manner and in what pattern it is depending. The aim is, without actually discharging the battery, calculating the discharge curve. Powertutor does not take into account the asynchronous behavior of phone. It is utilization based model which only takes into account consumption by each individual component. Also the results by combining battery discharge curve along with the last trigger policy of eprof can be analyzed.
- **Optimizations in Energy Consumption**:

  - **Power Aware application design**: Many more options of application design can be presented to developer along with their energy-performance trade-off. So as to let the developer have a clear idea of what will be the right choice of design for which kind of application. There are many applications available to generate different kinds of workload e.g. stressapp. Hence for different workload each design can be individually tested. Example app having continuous but variable workload processor frequency scaling can be done, etc. Also some more optimizations which are explained in section 2.4.3, 2.4.4 and 2.4.5 can be added.

  - **Battery Virtualization**: The implementation of battery virtualization mentioned in [21] consumes lot of energy itself. To reduce this energy consumption different implementation technique can be proposed. This virtualization can be implemented as a policy enforcement through an Android service. This android service will periodically after some amount of time will check if the energy consumption of any application class has exceeded the fraction of battery allocated to it. This policy enforcement can be done as it is done for various security models. Comparison of energy consumed using this technique and the technique proposed in [21] can be done to find most efficient way. Also, according to user profile, fraction of energy consumption of application class can be updated based on its past usage using some training systems.

  - **Network Applications**: Three ways of utilizing tail time are discussed: 1) Tail Aggregation (TailEnder): Different tail time of transmissions are combined together and sent so that the tail time of previous transmission overlaps with the transmission time of next transmission by deferring or prefetching. But very high prediction accuracy is required else even more energy is consumed. 2) Tail Time Tuning: Using fast dormancy the tail time can be cut. But doing this will increase too many state promotions increasing the energy consumption even higher. 3) TailTheft: Stealing the tail time for transmissions. A virtual tail time is maintained along with physical and small transmissions are scheduled in it. Moreover, if tail time ends before the transmission, it is cancelled. Hence, if the virtual time could be adjusted long enough, keeping in mind all the deadlines, then no transmissions would have to be canceled. Also stripping large transmissions into smaller transmissions can be more efficient.
Chapter 4

Energy Bugs

As we discussed above in literature survey section, due to complexity of android model and use of PowerManager API to explicitly turn on/off the components by developers, special types termed as energy bugs or no sleep bugs are created [16]. These bugs do not crash application etc, instead consumes significant amount of energy. To solve this issue, first of all these bugs should be detected. For detecting these bugs, we try to find paths on which wakelocks are acquired but not released through static analysis of the applications installed. Authors in [16], present first compile time approach using reaching definition intraprocedural dataflow analysis to find wakelock definitions reaching at the end of procedure. Conservative approach is used to find wakelock definitions reaching at the end of code that are not released considering all possible paths.

Since there are multiple possible paths, and user may take any path, also for precise analysis, dataflow analysis should consider all possible paths. If there is no sleep bug in single path also, the application should be considered buggy. For precise analysis of android application, full lifecycle of application should be incorporated in analysis. Hence first we study full lifecycle of android application then incorporate it in interprocedural reaching definition analysis.

4.1 Lifecycle of android applications

Android application has four main components Activity, Service, Broadcast Receiver and Content Provider[5]. They are essential building blocks of an android application. Android applications do not have main method, instead have many entry points for each component. For each component, a complete lifecycle is defined by android framework. All the components of applications are to be mentioned in manifest.xml file and overriding the lifecycle methods to start/stop/resume/pause the component. Each component has different lifecycle that defines how it is created or destroyed. Additionally applications can also register callbacks that implement predefined interfaces that can be called asynchronously. These callbacks should be added to entry points also[5]. Flowdroid[12] models main method for taint analysis of android applications such that it considers every order of individual component lifecycle and callbacks.
4.1.1 Activity

Activity is a single screen with user interface. All the running activities are managed in the stack. The activity on the top runs in foreground, and when the callbacks are called, it may switch to pause, restart, resume, etc states as shown by the lifecycle diagram of activity. Entry is performed through onCreate, onResume callbacks. Similarly exit is performed through onDestroy, onPause callbacks. When a stopped activity is restarted, onRestart callback is called.

![Activity Lifecycle Diagram](image)

Figure 4.1: Activity lifecycle
4.1.2 Service

Service runs in background for performing long running tasks and does not have an user interface. Service can be started using either startservice or bindservice. It is setup in onCreate and started by another component using onStartCommand and upon using bindservice onBind callback is called. If it is started in intentservice, onHandleIntent is called. Thus the service should complete its task and release wakelocks till end of onStartCommand, onUnBind and onHandleIntent. If onDestroy callback is defined by developer, it is the last callback that the service receives.

![Service Lifecycle Diagram](image)

Figure 4.2: Service lifecycle

4.1.3 Broadcast Receiver

When system wide broadcasts are issued, applications defining broadcast receivers respond to the broadcast through onReceive callback and it returns when the task is finished. Hence must release all wakelocks by the end of onReceive callback. They need not be declared in manifest file, if the app registers broadcast receiver by calling registerreceiver().

4.1.4 Content Provider

They are used when application wants to share some data with other applications. It receives request from other clients and process the relevant callback (query, insert, delete, update, etc). Each exposed callback should release all wakelocks at the end of each exposed callback.

**Intent based component communication:** Intents are asynchronous messages through which different applications can communicate. They can be explicit or implicit. Explicit intents
specify target component’s name, whereas implicit only specifies the action to be performed. It should be handled while doing precise analysis so that all the wakelocks are released. Same is the case for runnable objects.

4.2 Interprocedural Dataflow Analysis

Interprocedural reaching definition analysis is formulated in terms of IFDS framework by authors in paper[17] and can be reduced to graph reachability problem. The IFDS framework is general framework for interprocedural analysis over finite facts and distributive functions. Reaching definition analysis can be expressed in IFDS framework since there are finite no of facts and it is also distributive over \( \cap \) (meet) operation which is union(\( \cup \)) for reaching definition analysis. An IFDS framework can be reduced to graph reachability problem. Hence we discuss the paper[17] for IFDS framework.

4.2.1 IFDS Framework

Precise interprocedural dataflow analysis algorithm must provide the “meet over all valid paths” solution. In [17], the authors provide polynomial time algorithm for finding precise solution to interprocedural analysis problem. To solve the interprocedural dataflow analysis, the program is first represented as supergraph \( G^* = (N^*, E^*) \), where \( G_1, G_2 \) etc are CFG’s for each procedure and \( G_{main} \) represents CFG of main procedure. Each \( G_i \) has unique start node \( s_p \), exit node \( e_p \), call site \( c_p \) and its corresponding return site \( r_p \). In addition to these edges few extra edges are added:

- Call to return edge
- Call to start edge
- Exit to return edge

One way to achieve interprocedurally valid paths is by labeling call edges by ‘(’and return edges by ‘)’ . A path from \( i \) to \( j \) is a same level valid path if the sequence of ‘(’ and ‘)’ has balanced parenthesis. Through CFL reachability, from each start node, interprocedural analysis can be performed. Time complexity of CFL reachability problem is \( O(N^3D^3) \) where \( N \) represents nodes and \( D \) represents set of dataflow facts or analysis domain. Upon performing the interprocedural analysis, exploded supergraph is constructed. It can be defined as \( (N^#, E^#) \) where \( N^# \) includes all nodes of supergraph and each node has a set of dataflow facts (0 to \( D \)) associated with it. \( E^# \) represents edges from start node to each reachable node. If there is a path edge from \( n_{i,k} \) to \( n_{j,l} \) i.e., if a dataflow fact \( d_k \) is valid at node \( n_i \) then dataflow fact \( d_l \) will be valid at node \( n_j \). Hence dataflow analysis on supergraph corresponds to graph reachability on exploded supergraph. This paper [17] uses a dynamic programming approach called tabulation algorithm that tabulates the same level realizable paths. It is faster than CFL reachability algorithm.

Tabulation algorithm calculates set called path edges which contains the same level realizable paths or valid paths. It also uses set called summary edges which represents the edge for each function/procedure that relates fact at entry and exit of the procedure. When a call node is reached, its summary function can be used to determine which facts reached at the end of corresponding return sites. In case of non recursion this method works, however in case
of recursion summary is calculated via worklist algorithm. When a calling node is reached, summary edge for called procedure is traversed. New summary edges for reachable nodes in p are traversed and added, and are copied to call sites of procedure p. The authors of [17] generalised IFDS framework to IDE framework where dataflow facts are maps of symbols and their corresponding values. The time complexity of tabulation algorithm is O(ED^3). We studied two tools to achieve interprocedural analysis:

- **WALA[6]:** Watson Libraries for Analysis (WALA) by IBM also uses tabulation algorithm[17]. Its implementation focuses on memory efficiency. They also provide example to compute context sensitive reaching definition analysis. The ability to disregard invalid path is referred as context sensitivity. It converts the input program to SSA intermediate representation form. Though WALA generate appropriate output for very simple java program containing static fields in bit vector form. It is not easily extensible and not very easy to be used by developers to customize the analysis when compared with soot implementation. Hence we shift to soot implementation for interprocedural reaching definition analysis.

- **Soot:** In the paper [9] the authors also implemented IFDS/IDE solver over soot using tabulation algorithm [17]. Initially soot only supported intraprocedural analysis. They converted the input java bytecode to jimple intermediate representation. This implementation can be extended as per developer’s requirement and can be customized. We discuss the extension of IFDS/IDE solver over soot in further sections.

### 4.2.2 IFDS/IDE solver on Soot

It is an open source implementation[9] of interprocedural dataflow analysis over soot that implements generic IFDS/IDE solver. Soot[19] is a framework that provides intermediate representations in different forms upon which different analysis can be performed. The representation used for interprocedural analysis in the paper[9] is jimple, Hence we discuss about the jimple representation in detail as follows.

**Jimple:**

Jimple[19] representation is 3 address statement based intermediate representation. Translation of bytecode to jimple is done by soot. Each statement only references at most 3 local variables or constants and redundant code is cleaned up. In jimple there are only 15 type of statements like Nop, AssignStmt, InvokeStmt etc. By passing the option $-fJ$ in command line , the class file is converted to jimple file. The variables starting with $ denotes stack position whereas those without $ represents real local variables. Soot initially provided results for intraprocedural analysis only.

In the paper [9] interprocedural IFDS/IDE implementation is build over soot such that multiple analysis based on IFDS/IDE solver can be integrated easily. Lets consider following program as example and construct its exploded supergraph for reaching definition analysis.
Listing 4.1: code 1

```c
void main()
{
    int f=2;
    f=foo(f);
    return(f);
}
```

Listing 4.2: code 2

```c
int foo(int f)
{
    f=f*20;
    return f;
}
```

Corresponding exploded supergraph of above listings for reaching definition analysis is as shown:

![Exploded Supergraph](image)

There are four kinds of edges that users need to define:

- **Call edge**: It connects the call site to start of called procedure passing the arguments.
- **Return edge**: values for called procedure are passed back to callee.
- **Call to return edge**: Pass information directly from call site to successors of call site from same procedure.
- **Normal edge**: all other statements such as assignment statement, etc.

IDE framework is an extension of IFDS framework by the same authors [17] that implements value computation as well along with reachability. To define IFDS problem, IFDSTabulation interface has to be implemented, implementing four types of flow functions as shown in figure and provide initial seeds depending upon the entry points (`scene.getEntryPoints()`). The parameters N, D, M represents node (unit), dataflow facts and methods respectively. Initial seeds returns map of facts and their corresponding zero values. Developers need to customize only this class in order to customize the code for other analysis. Flow functions returns flow function for individual edges and compute targets to which source node connects. For IDE solver interface, method `edgefunctions`, `joinlattice` has to be defined. Edge functions denote $V \rightarrow V$ type functions that
compute V type facts along side D facts for each edge. Joinlattice returns the join function or merge operation to be performed on Value lattice. The original algorithm[17] stores the path edges, whereas in this implementation[9] they store it in index data structure that allows $O(1)$ access. We customized this implementation[9] of interprocedural dataflow analysis to detect energy bugs for android applications and discuss it in further chapters.

```java
1 interface FlowFunctions<N, D, M> {
2     public FlowFunction<D>
3         getNormalFlowFunction( N curr, N succ);
4     public FlowFunction<D>
5         getCallFlowFunction( N callStmt, M destinationMethod);
6     public FlowFunction<D>
7         getReturnFlowFunction( N callSite, M calleeMethod, N exitStmt, N returnSite);
8     public FlowFunction<D>
9         getCallToReturnFlowFunction( N callSite, N returnSite); }
```

Figure 4.4: interface for defining flow functions[9]
Chapter 5

Identification of Energy Bugs

The analysis starts from the entrypoints specified and flows according to the generated callgraph. As discussed in previous chapter, to extend the generic IFDS/IDE solver\cite{9}, the four flow functions i.e., normal, call, return, and call-to-return have to be defined. According to reaching definition analysis for each assignment, new definitions are created. However, in our case, we are concerned with variables of wakelock instances only. Hence our analysis acts as identity function for other variables. An empty fact has to be defined to each statement that is valid throughout the analysis i.e., zeroValue. It is used to generate the fact unconditionally. In our analysis, we precompute all the wakelock variables declared and create the tuple for zeroValue as follows:

\[
\langle \langle \text{wakelock1}, [\text{nop} = 0] \rangle, \langle \text{wakelock2}, [\text{nop} = 0] \rangle, \ldots \rangle
\]

Here acq represents whether wakelock is acquired or not. Similarly the dataflow facts i.e., each wakelock along with its referencing variables are represented in format as follows:

\[
\langle \langle \text{wakelock1}, [\var{var1}], [\var{var2}], \ldots, [\text{nop} = 0] \rangle, \langle \text{wakelock2}, [\var{var3}], [\var{var4}], \ldots, [\text{nop} = 1] \rangle, \ldots \rangle
\]

We considered such a format, instead of individual tuple for each wakelock, to take care of aliases and to achieve fixed point. Also in this way in/out set has only single tuple besides zero value, since single tuple represents all the required information. Size of the tuple is also not an issue as there are very few wakelock definitions in an application.

5.1 Flowfunctions

Every flow function is simply function object returning target fact, given source fact and current node. According to this dataflow we now define the four flow functions as:

5.1.1 Normal flow function

They are applied to all statements except call, return. We apply analysis only to definition statement involving wakelock variables. We keep track of aliases of wakelock variables through this flow function. As we described above, our format of dataflow fact, which contains single tuple containing all wakelock variables and its aliases, we remove left operand of the assignment statement from source wakelock variable and add it to the right operand’s wakelock variable. The operands of statement can be either stack variable starting from $ or real variables. We store the variables in tuple as classname : methodname : variablename, so as to keep track of
all possible aliases without ambiguity. Let \( a = b \) be the assignment statement \( s \in Stmt \). The flow function can be represented as:

\[
T \xrightarrow{a} \begin{cases} 
T \setminus \{W_i.a\} & (b \notin W_i) \land (a \in W_i) \land (W_i \in T) \\
T \cup \{W_i.a\} & (b \in W_i) \land (a \notin W_i) \land (W_i \in T) \\
T & \text{otherwise}
\end{cases}
\]

If the right operand is alias of particular wakelock and left operand is not alias of that wakelock, then add left operand to its alias list. On the other hand if left operand is alias of some wakelock variable but right is not then its definition is killed and it is removed from list. In all other cases no change is done in input tuple.

### 5.1.2 Call flow function

Call flow functions deal with tuples passing to all the callees of the callsite statement. Hence it is applied to all callsites of program. We keep all the information of aliases i.e., classname, methodname etc. in the tuples. Since the tuples are only modified for aliases or acquire/release statements, and for all other statements no change is performed on the source fact. We pass the input tuple to all callees, which may or may not be modified by the callee.

\[
T_{\text{callee}} \xrightarrow{a} \cup T_{\text{caller}}
\]

### 5.1.3 Return flow function

They are applied to regular return edges as well as exceptional edges. Since all the modification in tuple of wakelocks should be passed back to caller, source tuple is passed as it is.

\[
T_{\text{caller}} \xrightarrow{a} \cup T_{\text{callee}}
\]

### 5.1.4 Call-to-return flow function

For every callsite there also exists an intraprocedural edge which is independent of callee. In our case, wakelock functions such as newWakelock, acquire, release, etc are the function calls that determines result of analysis. These are added to our analysis as library functions. Hence these methods are not analysed interprocedurally and only intraprocedural edge is added for them. Hence we wrote flowfunctions considering these three methods, which are very essential for analysis.

- **newWakeLock(int, String):** This function creates a newWakelock with specified flags and tag and its instance is assigned to a wakelock variable. Typically statement \( s \in S \) is like \( a=\text{PowerManager.newWakeLock(int,"String")} \). Right operand PowerManager. newWakelock(int,"String") is stored as wakelock value in tuple. Lets refer it as ‘b’. Hence we can define flowfunction as:

\[
T \xrightarrow{a} \begin{cases} 
T \setminus \{C.a\} & (C \in T) \land (C.value \neq b) \land (a \in C) \\
T \cup \{C.a\} & (C \in T) \land (C.value = b) \land (a \notin C) \\
T & \text{otherwise}
\end{cases}
\]

If the wakelock variable value in \( T \) and wakelock variable in \( s \) is same then the left operand should be present in its alias list. Hence if not present then we add it to the list. If the value of wakelocks is not same and the left operand is present then we remove it as its definition will be killed and now it will be added to current wakelock variable.
• **acquire()**: This function acquires the wakelock and keeps the particular device on. If a wakelock is acquired, it should be released else it creates energy bugs, as discussed in previous chapters. Typically statement \( s \in S \) looks like \( a.acquire() \), which acquires the wakelock referred by \( a \). Hence we find the wakelock its referring to, by checking its alias list. If it is present, acq value of wakelock is set to 1, indicating it is acquired. We define its flow function as:

\[
T \xrightarrow{s} \begin{cases} 
Value(W_i) = 1 & (W_i \in T) \land (a \in W_i) \\
T & \text{otherwise}
\end{cases}
\]

• **release()**: This function releases wakelock, so that the device can go back to sleep state. Statement \( s \in S \) \( a.release() \) releases the wakelock referred to by \( a \). Hence we try to find the wakelock \( a \) is referring to, by checking its alias list. If found we set the acq value back to 0. Flow function is defined as:

\[
T \xrightarrow{s} \begin{cases} 
Value(W_i) = 0 & (W_i \in T) \land (a \in W_i) \\
T & \text{otherwise}
\end{cases}
\]

Thus at the end of each component lifecycle, we can check if wakelocks’s value reached is 0/1, which might indicate it is released or it is acquired but not released at some path. There may be some extra tuples present for same wakelocks which indicate flow from different path. If single path also has wakelock value 1, the app is said to have wakelock bug. We define edge functions to calculate the wakelock acquire/ release value and flow functions to keep track of its aliases.
Chapter 6

Implementation Details

We used soot framework\[19\] which generates precise call graph, to be used by interprocedural dataflow analysis. Also, as we discussed in energy bugs chapter, we customize the generic implementation of IFDS/IDE solver\[9\] and create appropriate flow functions as discussed in wakelock bug detection chapter. The android application’s dex files are converted to jimple format\[19\] by plugin called dexpler\[8\], which is added to soot. Dexpler allows soot to convert not only java but also dex files to jimple representation. To construct callgraph, entrypoints of the program are to specified, since android applications do not have main method but different components with their own lifecycle. Each component has its own lifecycle and accordingly have fix method calls for specific events. We use the lifecycle implementation of android components by flowdroid\[12\].

Android lifecycle implementation: In flowdroid\[12\], the entrypoints are added by creating a dummy main method which links lifecycle of all components( activities, services, broadcast receivers, etc) and joins them at exit node. The example CFG of dummy main method created by flowdroid is shown in figure 6.1. We used this dummy main method created by flowdroid

![Figure 6.1: Dummymain method\[12\]](image)

29
to create entrypoints of our analysis. Currently intent based communication edges are not in- cluded in dummy main method or callgraph. Intent based edges can be added conservatively by mapping the respective actions from manifest file to intent calls and thus adding all possible edges. However, this is left as a part of future work presently. Flowdroid also includes the callback methods defined by all the components of the input application. It identifies all the callbacks overridden by the application, then creates customized dummy main method for each component involving callbacks.

Hence through dexpler .dex files are converted to jimple format and entry points are created through flowdroid. These jimple files and entry points are used by soot to create precise callgraph upon which our implementation of interprocedural wakelock bug detection analysis over heros[9] operates.
Chapter 7

Results

Path of the apk file to be analyzed and android library jar are given as input to our analysis. This apk is converted to jimple format to perform analysis.

7.0.5 Test Case

We consider an example of apk file to conduct Quiz. It acquires wakelock to keep screen on, in onCreate event of activity and releases in callback onKeyDown. The jimple format of code is like:

```java
Listing 7.1: code 3
13 Class Quiz{
14 PowerManager$Wakelock wake;
15 onKeyDown()
16 {
17 PowerManager$Wakelock $r2;
18 $r0=<@this: Quiz>;
19 $r2=$r0.<Quiz: wake>;
20 $r1.release();
21 }
22 onCreate(android.os.bundle)
23 {
24 Quiz $r0;
25 PowerManager$Wakelock $r7;
26 $r7=<android.os.PowerManager$Wakelock: newWakeLock(6,"MyLock")>;
27 $r0.<Quiz: wake>=$r7;
28 $r7=$r0.<Quiz: wake>;
29 $r7.acquire();
30 }
31 }
```

In this example, wakelock is acquired in onCreate and released in onKeyDown, however, the application may terminate abruptly due to less memory or due to other causes. Hence the wakelock remains acquired, in other words the component switched on by this wakelock stays on, draining huge amount of energy. For this example, our analysis generates following tuple at the end: At the end, the wakelock tuple is as shown:
[Pair \$r7 = \text{virtualinvoke} \$r6. android.os.PowerManager: android.os.PowerManager$\text{WakeLock} \text{newWakeLock}(\text{int, java.lang.String}) (6, ‘MyLock’), \[w = \text{com.iitb.QuizPageActivity: onCreate:}\$r7\ , w = \text{com.iitb.QuizPageActivity: wakeLock} , w = \text{com.iitb.QuizPageActivity: onKeyDown:}\$r2\ , \text{nop} = 0]]

The tuple represents single wakelock declaration and all its aliases. And corresponding value received by edge functions for wakelock is [1], indicating wakelock is left acquired through some path.
Chapter 8

Conclusion

With the tremendous increase in usage of smartphones and not only for basic purposes such as call, message etc, but mainly now a days for applications like multimedia, games, etc with new applications releasing everyday, constraint on energy consumption has become huge bottleneck. Hence we studied various techniques discussed above to improve the battery life concentrating on application level so that developer of each application can build the application keeping in mind all the techniques available to him for reducing energy consumption of his app. We studied detection of energy bugs, also different methods to model energy consumption without using external power meter. We also studied various optimizations that can be done in the major consumers of energy such as Network Components, LCD and CPU and proposed further enhancements that be done in available techniques making them more efficient. We created analysis tool for analyzing simple applications (without intents and threads), which produces context sensitive result for finding wakelock bugs in the application.

8.1 Future Work

This analysis tool can be easily extended to analyze applications involving intents and threads also, by mapping all the intent actions registered in manifest file to intent calls and following conservative approach.
Acknowledgements

I would like to express my deepest gratitude to my guide Prof. D. B. Phatak, for his patience and guidance throughout the project. I would also like to thank Nagesh Karmali for his continuous inputs for my work. I would also like to thank every one who supported me in this work.
Bibliography


