GREENDROID

Diagnosis of Efficient Energy Usage for Smartphone Applications

Vishal Kulkarni[1], Amitkumar Meena[2], Simran Gupta[3], Prof. Dhanashree Kanade[4]
[1-3] Students, [4]Assistant Professor
Department Of Computer Engineering
K.C. College of Engineering & Management studies & Research,
Kopri,Thane (E)-400 603, India.

Abstract -Nowadays the main problem or the major issue we face while using a smart phone is the rigorous and fast exhausting battery of the mobile. This causes a lot of problems for the people who utilise their phone for long durations. This paper aims to address this problem. We propose an idea to address this problem by our Android application Greendroid. Today the mobile application developers do not pay special attention to the energy consumed by the application thus resulting in the poor performance of the application beacause of vast comsumption of energy. Greendroid proposes to monitor the energy consumed and give a real time diagnosis of battery usage.

Keywords – Sensors, wake locks, Java Path finder(JPF), Energy Inefficient.

I. INTRODUCTION

There are tons of mobile applications readily available for various purposes like location determination or chatting applications. The internet based mobile applications connect the physical and cyber environments and together they provide smart services to the user. However unlike our desktop computers and other counterparts the mobile phones have resource constrained platforms. Due to these constraints in smart phones even the smallest of inefficiencies can lead to a bad user experience. Mobile developers need to ensure a satisfying experience for the users especially because the smart phone applications have requirements like seamless communication, frequent sensing of the environment and infinite computation. Even a small amount of ignorance of any of one these three factors can lead to very high energy consumption by the mobile application.

Mobile developers nowadays rarely have sufficient time or resources to carefully optimize the energy consumption and this is the main reason why so many applications suffer from energy and performance bugs. Energy bugs waste a lot of battery power and performance bugs significantly reduce the smart phones' responsiveness. Locating of energy bug is difficult in android applications. We have to rigorously test the application on various platforms and it is time consuming.

Energy inefficiency related problems can vary according to the different applications and the main issues in majority of the problems are related to wake lock deactivation and sensory data underutilization.

Wake lock Deactivation: When the phone is idle the android device quickly falls asleep but when making certain computations or performing certain activities like watching a movie or playing video games the user acquires a wake lock from the android OS. Now this wake lock should be unregistered as soon as the computation completes. Failing to do so results in a huge loss of energy.

Sensory Data Underutilization: The android sensor framework lets you access various types of in-built sensors that are capable of measuring motion, orientation, and various environmental conditions. Some examples of sensors are gyroscope, accelerometer, temperature sensor, proximity sensor etc.. The sensors are always enabled in the background. It is important to make sure that the sensors are always disabled when they are not in use especially when the activity is paused. Sensors judge their surroundings and collect data from the same. This data is obtained at high energy cost and hence this data should be utilized effectively. If the data is utilized poorly then it is equivalent to a wasting a huge amount of energy.

We tend to overcome this problem by analysing the performance of the application stage by stage by executing it using Java Path Finder (JPF) (Fig. 1). Java Path Finder (JPF) acts like a virtual machine which finds out all the possible ways to execute a program and checks the

possibility for any deadlocks or unhandled exceptions along all valid approaches. Also using Java Path Finder we can monitor whether the application is using the sensors and wake locks efficiently and they are getting released or unregistered or not.

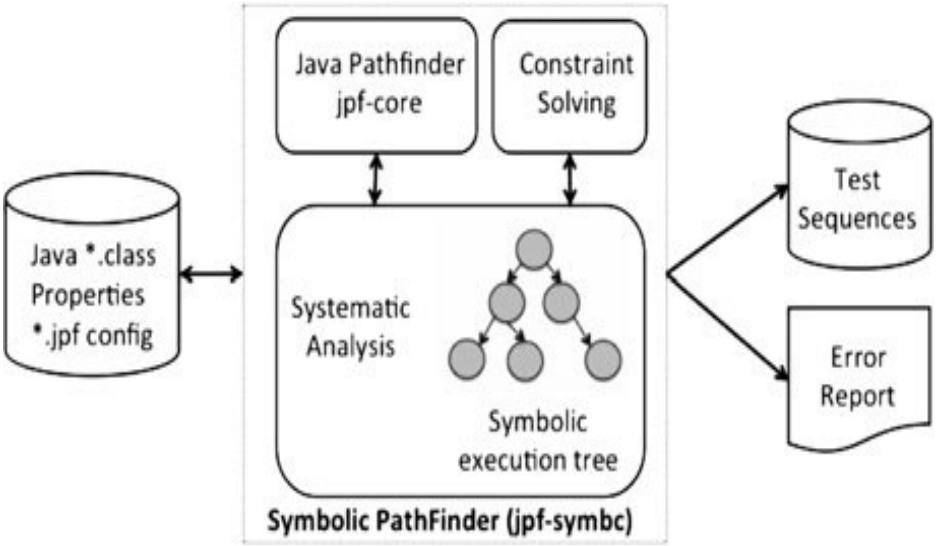

Fig. 1. Java Path Finder (JPF)

II. ACTIVITY LIFE CYCLE

Since our main aim is to analyse each and every application we need to get familiarized with the basic life cycle of each of its components. Hence, the activity life cycle study is a must.

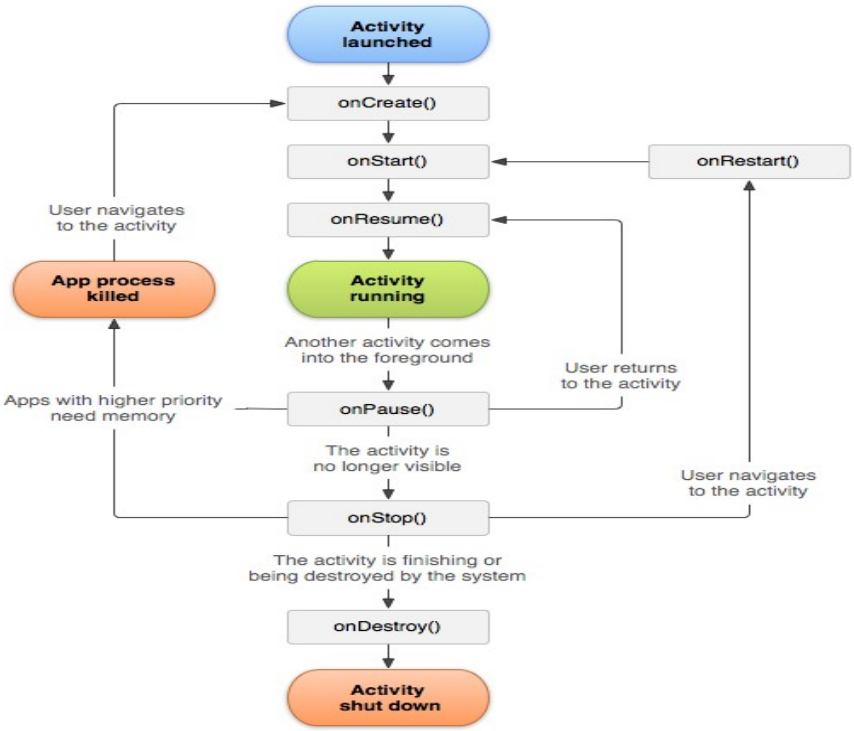

Fig. 2 Activity Life cycle

Fig2. shows the life cycle of any activity. The life cycle of any activity starts from the onCreate() function and ends on the onDestroy() function. The function onCreate() is called when the activity is first created. The activity always works on the foreground. When the activity is becoming visible to the user onStart() function is called. When it moves to the background and becomes invisible to the user then the onStop() function is called. When the user moves back to the paused or stopped activity then the activity's onRestart() or onResume() function is called. Amidst the running of the application an activity can be resumed, paused or killed in order to release memory for the activities with higher priorities.

III. ENERGY EFFICIENCY DIAGNOSIS

In this section we will look towards our approach to solve this problem and diagnose the energy inefficient applications.

A general approach to solving this problem is by using the Java byte code and configuration files of the Android application which define its logic. Now this java byte code is executed in Java Path Finders' virtual machine and it is diagnosed or analysed step by step by exploring all its states. During this exploration we monitor the sensor registrations and wake lock acquisitions. We also monitor at which state the sensors are disabled or unregistered and the wake locks are released. We then analyse how these sensory data is utilized at various stages of execution of the application. At the end of this process a report is prepared which compares the actual use of sensory data across all the stages of application processing and it is determined if the sensory data is underutilized or not. We also check if the wake locks are forgotten to be released or the sensors have been forgotten to be unregistered.

i. APPLICATION EXECUTION AND STATE EXPLORATION

An Android activity starts with a main activity and ends when all its components are destroyed. During the execution of the application it keeps receiving the user interaction events and system events by calling the registered Android application handlers. Every event changes the state of the application by changing the program data. To explore the state space we need to generate user interaction events and schedule the corresponding event handlers.

Generating User Interaction Events: Analysis of the configuration of the application can be done using the GUI layouts of the activities. We map the GUI components to a possible set of user activities. While execution the runtime controller uses the execution history and current state and generates all possible events associated with each activity component.

ii. MISSING SENSOR OR WAKE LOCK DEACTIVATION:

Missing sensor or wake lock deactivation is the major issue which results in huge amount of energy inefficiency. According to the Android process management policy the sensors or wake locks are not automatically deactivated even if the application components using them are destroyed. This causes huge performance degradation. In short we diagnose the execution states and make sure that the following two policies are never violated:

- *Sensor Management Policy:* A sensor listener l should be unregistered before any activity of the application dies.
- *Wake Lock management policy:* A wake lock wl should be released before the activity of the running application dies.

iii. SENSORY DATA UTILIZATION ANALYSIS

During the execution of an application some sensory data is collected and the same data is utilized by different application activities by performing modifications. We determine if the data is used in

an energy efficient manner. We use the concept of Dynamic Tainting which includes three phases: (1) Tainting each sensory data with a unique mark; (2) Communicating these taint marks through various levels of application component stages; (3) Analyzing the utilization of sensory data at different stages. We will view these stages in detail one by one.

Tainting Sensory Data

In this stage we use mock sensory data from an existing data pool. It is then fed to the application activity after the event handlers are called. The reference to the object of each sensory data is initialized with a unique taint mark before feeding it to the application. This data along with the taint mark will propagate through the application activity for further analysis.

Propagating Taint Marks

While the application is running the tainted sensory data is transformed into various forms by performing several arithmetic and relational operations. Here we track the taint marks and identify which application component depends on what sensory data. This technique tracks the Java byte code along with the taint marks in the Java Path Finders' virtual machine. A key advantage is that all the processing is done on the byte code level hence it does not require any application specific program instrumentation.

The taint propagation terminates when the application utilizing the sensory event finishes. This generally happens in two situations. Firstly, if the sensor event handler does not start any operation to handle further received sensor events the communication of taint marks stops at the end of this handler. Otherwise the propagation terminates only if all the operations of the application components are terminated. Hence, in this way taint propagation identifies which part of the activity is dependent on the collected sensory data and how efficiently they are put to use.

Theoretically, in multi-threading operations the taint propagation will only terminate when all the worker threads end. However, practically this may take a toll on the device usability since the taint propagation may take a huge amount of time and hence fails to generate an analysis report. A solution to this problem is setting a timeout value for large duration of taint propagations.

Analysing Sensory Data Utilization

After the propagation of sensory data with taint marks, the program data is tainted and now it is easy to analyse whether this data is utilized in energy efficient way or not.

Now we calculate the Data Utilization Coefficient (DUC) which is defined as the ratio between the usage of sensory data *d* at activity state *s* and the maximal usage of sensory data from our data pool *D* at any state S_b where b is the length of the user interaction event sequences.

$$DUC(s,d) = \frac{usage(s,d)}{Max_{s' \in S_b, d' \in D}(usage(s',d'))}.$$

Using this formula we can compare the usage of sensory data across different stages. Lower value of DUC value indicates lower consumption of sensory data at that state.

The usage of the sensory data can be calculated using the following equation:

$$usage(s,d) = \sum_{i \in API\_Call(s,d)} eTest(i,d,s) \times noInst(i).$$

In the above mentioned equation *API_Call(s,d)* calculates the number of times API calls are executed since the sensory data it fed to the application at states till data handling is finished. The

function *eTest(i,d,s)* is for effectiveness check and it returns a binary value. It returns 1 if the following two conditions hold (1) The API call *i* uses program data dependant on the sensory data. (2) The *API's* execution holds benefits to the user. If any one of the condition does not satisfy then it returns 0. Function *noInst(i)* returns the number of byte code instructions executed by the *API* call. The main idea behind calculating the usage factor is to determine how many times and to what extent the sensory data is used by an application to benefit the users.

Now we check how *eTest(i,d,s)* checks for effectiveness. For first condition we check if the *API* called has the same data as the sensory data. For second condition we take an outcome based strategy. According to this strategy if the API stores any data to file systems, database or network or updates the phones' status or passes any message for inter or intra application communication then the API passes the test. Under this condition we make an assumption that the data stored in the above mentioned ways will be beneficial to the user in one or other way.

IV. CONCLUSION

In this paper we presented the necessity of energy efficiency in smart phones and identified two common phenomena that commonly affect the mobile battery and cause energy waste: missing sensor or wake lock deactivation and sensory data underutilization. Based on these causes we propose an approach for automated diagnosis of energy utilization in Android applications. Our proposal examines the various stages an application component goes through and its usage of sensory data and monitoring of wake locks. Using this it helps user to locate if the data is utilized effectively and whether the sensors were deactivated, and wake locks were released.

V. ACKNOWLEDGEMENTS

We express our sincere gratitude to our co-guideProf. Kaushiki Upadhyaya and guide Prof. Dhanashri Kanade whose supervision, inspiration and valuable guidance helped us a lot to complete our work. Their guidance proved to be the most valuable to overcome all the hurdles in the preparation of this paper work. Also we are thankful to all those who have helped us in the completion of paper work.

VI. REFERENCES

[1] Yepang Liu, Chang Xu, S.C. Cheung, and Jian L, "GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications"
Available:http://sccpu2.cse.ust.hk/andrewust/files/TSE2014.pdf
[2] Android Development Website [Online].
Available: https://developer.android.com/training/scheduling/wakelock.html
[3] Sensors Overview [Online].
Available: https://developer. android.com/guide/topics/sensors/sensors_overview.html
[4] Sensor Managers [Online].
Available: https://developer. android.com/guide/topics/sensors/sensors_overview.html
[5] Java Path Finder from NASA JavaPathFinder project [Online].
Available: https://nebelwelt.net/teaching/15510 -SE/slides/12-jpf-1.pdf
[6] Android Life Cycle [Online]. Available: https://www. javatpoint .com/android-life-cycle-of-activity
[7] A.Maheswari, G.Muppidathi, R.Nandhini, G.Santhiya, "Automated Energy Problem Diagnosis In Android Applications" [Online]. Available: http://www.ijetie.org/ articles/IJETIE_201513038.pdf
[8] Jue Wang, Yepang Liu, Chang Xu, Xiaoxing Ma and Jian Lu, "E-GreenDroid: Effective Energy Inefficiency Analysis for Android Applications" [Online].
Available: http://sccpu2.cse.ust.hk/andrewust/files/Internetware2016.pdf

[9] James Newsome, Dawn Song, “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software” [Online].
Available: http://valgrind.org/docs/newsome 2005.pdf