Université de Lorraine
Erasmus Mundus Master's Programme in Pervasive
Computing & Communications for sustainable Development
PERCCOM

Mohammed Mustaqim Rahman

Analysing API Calls to Reduce Energy Consumption of Apps in Idle States

31 May 2017

Supervisors: *Professor Eric Rondeau (Université de Lorraine)*
Associate Professor Jean-Philippe Georges (Université de Lorraine)
Thierry Leboucq (Greenspector)

Examiners: *Professor Eric Rondeau (Université de Lorraine)*
Professor Jari Porras (Lappeenranta University of Technology)
Associate Professor Karl Andersson (Luleå University of Technology)

This Thesis is prepared as part of an European Erasmus Musdus programme PERCCOM - Pervasive Computing and Communication for Sustainable Development.

Co-funded by the
Erasmus+ Programme
of the European Union

This thesis has been accepted by partner institutions of the consortium (cf. UDL-DAJ, n°1524, 2012 PERCCOM agreement).

Successful defence of this thesis is obligatory for graduation with the following national diplomas:

- Master in Complex Systems Engineering (Université de Lorraine)
- Master of Science in Technology (Lappeenranta University of Technology)
- Master of Science - Major; Computer Science and Engineering, Specialisation; Pervasive Computing and Communications for Sustainable Development (Luleå University of Technology)

Abstract

Université de Lorraine
PERCCOM Master Programme

Mohammed Mustaqim Rahman

Analysing API Calls to Reduce Energy Consumption of Apps in Idle States

Master's Thesis - 2017
74 pages, 51 figures, 25 tables, 2 appendices
Keywords: API usage pattern, Energy Consumption, Testbed, Green ICT

Mobile devices are an inseparable part of modern era. Smartphones are used for various purposes in our daily life, such as- making calls, sending and receiving emails, listening to music, taking photos, reading books, booking tickets, etc. Hundreds of apps, countless tasks and lots of processing power with improved user experience is a common trend for every smartphone. But the issue of battery life is struggling to keep up with the improvement of processor, memory, storage and other hardwares. However, excessive energy consumption of apps in mobile devices due to bad programming practices, inefficient API usage patterns or energy greedy hardware can significantly reduce battery life. Moreover multitasking capability of smartphones have opened up a new dimension to its users. Frequent switching between apps often cause previously used apps to run in idle mode either in foreground or in background. In this thesis an analysis has been done to measure the energy cost of API methods called by an application during execution in idle states. The experiments were conducted on ten android apps from different domains. For energy consumption measurements of these apps on the device Greenspector software tool is used, as this thesis work is conducted collaboratively between Université de Lorraine and Greenspector. Based on the collected data from application's energy consumption measurements, an analysis has been done on API calls invoked by the application during execution and its impact on the device to find anomalies in order to reduce platform discharge. Finally, the work is summarised by providing a reference ranking to developers regarding energy consumption of certain categories of Android APIs.

Acknowledgements

At first, I wish to express my sincere gratitude to PERCCOM (Pervasive Computing and Communications for Sustainable Development) [33] selection committee for giving me the opportunity to join with excellent academic stuffs, colleagues and friends to acquire knowledge from the broad field of Science and Technology, especially from eco-friendly, green and sustainable Information and Communication Technology.

I want to thank my thesis supervisor Prof. Eric Rondeau, Jean Philippe Georges and Thierry Leboucq for their guidance and support during my thesis project. I also want to thank Pierre-Alexander Voye for all the support.

I am also thankful to all my teachers and supervisors from all host institutions for their cooperation and support. I whole heartedly thank my classmates who helped me a lot during two years of my master study.

I would like to thank my wife and parents who supported me throughout my life. My cordial appreciation goes to all of my family members, relatives and friends, without whose support none of my success would be possible.

Mohammed Mustaqim Rahman
31 May, 2017

Contents

1 Introduction **2**
- 1.1 Background . . . 3
- 1.2 Motivation . . . 5
- 1.3 Problem Definition . . . 5
  - 1.3.1 Research Questions . . . 5
  - 1.3.2 Research Contributions . . . 6
- 1.4 Delimitations . . . 6
- 1.5 Thesis Structure . . . 6

2 Related Works **8**
- 2.1 The GREENSOFT Model . . . 8
- 2.2 Taxonomy of Energy Profiling Schemes for Application . . . 10
- 2.3 State-of-the-Art Application Energy Profiling Schemes . . . 12
  - 2.3.1 Hardware Based Application Energy Profiling Schemes . . . 12
  - 2.3.2 Software Based Application Energy Profiling Schemes . . . 13
- 2.4 Comparison Between State-of-the-Art and Adopted Research Method . . . 15

3 Test-bed Design Theory and Implementation **17**
- 3.1 Units of Measurement . . . 17
- 3.2 Characteristics of Lithium-ion Battery . . . 18
- 3.3 Application Program Interface(API) . . . 19
- 3.4 Required Tools . . . 20
  - 3.4.1 Android Studio . . . 20
  - 3.4.2 Android Debug Bridge (ADB) . . . 20
  - 3.4.3 Traceview and Dmtracedump . . . 21
  - 3.4.4 Weka . . . 21
  - 3.4.5 Greenspector TestRunner . . . 22
- 3.5 Profiling Application . . . 23
  - 3.5.1 App selection and Enabling Debug Mode . . . 25
  - 3.5.2 Test-bed Design . . . 25
  - 3.5.3 Execution Scenario . . . 26
  - 3.5.4 Data Extraction Process . . . 27
  - 3.5.5 Data Analysis . . . 28
- 3.6 Summary . . . 29

4 Application Profiling Results **30**
4.1 Benchmarking Application by API Behaviour Analysis 30
4.1.1 Wikipedia 31
4.1.2 2048 33
4.1.3 CycleStreets 35
4.1.4 Hubble Gallery 37
4.1.5 Memory Game 39
4.1.6 MapBox 41
4.1.7 Flym News Reader 43
4.1.8 Omni Notes 45
4.1.9 Easy Sound Recorder 47
4.1.10 NIST Connect 48
4.2 Summary 50

5 Discussion **51**
5.1 Outlining Reference Energy Measurement Statistics 51
5.2 Comparison Between Benchmarked Applications 52
5.2.1 Analysing API Method Calls 52
5.2.2 Comparison Between Applications from Same Category 53
5.3 Impact of the Apps Over Internet 56
5.4 Takeaway for Developers 57
5.5 Contribution to Sustainability 58

6 Conclusion and Future Work **59**

A Appendix **61**
A.1 Dataset 61
A.2 Code 61
A.2.1 API cost calculator code 61
A.2.2 Tracefile to CSV Data Parser 64

List of Figures

1.1 Web interface of Greenspector 4

2.1 The GREENSOFT model. A reference model for green sustainable software 8
2.2 Cradle-to-grave inspired product life cycle for software products. [38] 10
2.3 Hardware based vs software based energy profiling 11
2.4 Taxonomy of mobile application energy profiling scheme[12] 11

3.1 Workflow of Tesbed for obtaining energy consumption and execution trace of an application 24
3.2 Analyse Execution Trace and Measurement Data 28

4.1 Platform discharge speed of Wikipedia app in different execution states . . . 31
4.2 Platform discharge of Wikipedia app in different execution states 31
4.3 Energy consumption of Wikipedia application over time 32
4.4 Distribution of API energy consumption of Wikipedia application over time . 32
4.5 Platform discharge speed of 2048 app in different execution states 34
4.6 Platform discharge of 2048 app in different execution states 34
4.7 Energy consumption of 2048 application over time 34
4.8 Distribution of API energy consumption of 2048 application over time 35
4.9 Platform discharge speed of CycleStreets app in different execution states . . 36
4.10 Platform discharge of CycleStreets app in different execution states 36
4.11 Energy consumption of CycleStreets application over time 37
4.12 Distribution of API energy consumption of CycleStreets application over times 37
4.13 Platform discharge speed of Hubble Gallery app 38
4.14 Platform discharge of Hubble Gallery app in different execution states 38
4.15 Energy consumption of Hubble Gallery application over time 39
4.16 Distribution of API energy consumption of Hubble Gallery application over times 39
4.17 Platform discharge speed of Memory Game 40
4.18 Platform discharge of Memory Game app in different execution states 40
4.19 Energy consumption of Memory Game application over time 40
4.20 Distribution of API energy consumption of Memory Game application over times 40
4.21 Platform discharge speed of Map Box app 42
4.22 Platform discharge of Map Box app in different execution states 42
4.23 Energy consumption of Map Box application over time 42

4.24 Distribution of API energy consumption of Map Box application over times . 42
4.25 Platform discharge speed of Flym app 44
4.26 Platform discharge of Flym app in different execution states 44
4.27 Energy consumption of Flym application over time 44
4.28 Distribution of API energy consumption of Flym application over times . . . 44
4.29 Platform discharge speed of Omni Notes app 46
4.30 Platform discharge of Omni Notes app in different execution states 46
4.31 Energy consumption of Omni Notes application over time 46
4.32 Distribution of API energy consumption of Omni Notes application over times 46
4.33 Platform discharge speed of Easy Sound Recorder Notes app 47
4.34 Platform discharge of Easy Sound Recorder app in different execution states 47
4.35 Energy consumption of Easy Sound Recorder application over time 48
4.36 Distribution of API energy consumption of Easy Sound Recorder application over times 48
4.37 Platform discharge speed of NIST Connect Notes app 49
4.38 Platform discharge of NIST Connect app in different execution states 49
4.39 Energy consumption of NIST Connect application over time 50
4.40 Distribution of API energy consumption of NIST Connect application over times 50

5.1 List of total number of API methods called by apps during execution 53
5.2 Energy Consumption Comparison Between Hubble Gallery and Memory Game 54
5.3 Energy Consumption Comparison Between Flym News Reader and NIST Connect 55

List of Tables

2.1 Impacts of ICT on the Environment 9
2.2 Related energy profiling schemes 16

3.1 List of Application 25

4.1 List of Benchmarked Application 30
4.2 Average platform discharge of Wikipedia app in different execution states . . 31
4.3 Distribution of API methods called by Wikipedia application 33
4.4 Average platform discharge of 2048 app in different execution states 33
4.5 Distribution of API methods called by 2048 application 35
4.6 Average platform discharge of CycleStreets app in different execution states . 36
4.7 Distribution of API methods called by CycleStreets application 37
4.8 Average platform discharge of Hubble Gallery app in different execution states 38
4.9 Distribution of API methods called by Hubble Gallery application 39
4.10 Average platform discharge of Memory Game app in different execution states 40
4.11 Distribution of API methods called by Memory Game application 41
4.12 Average platform discharge of MapBox app in different execution states . . . 42
4.13 Distribution of API methods called by Map Box application 43
4.14 Average platform discharge of Flym News Reader app in different execution states 43
4.15 Distribution of API methods called by Flym News Reader application 44
4.16 Average platform discharge of Omni Notes app in different execution states . 45
4.17 Distribution of API methods called by Omni Notes application 46
4.18 Average platform discharge of Easy Sound Recorder app in different execution states 47
4.19 Distribution of API methods called by Easy Sound Recorder application . . 48
4.20 Average platform discharge of NIST Connect app in different execution states 49
4.21 Distribution of API methods called by NIST Connect application 50

5.1 Reference platform discharge and energy consumption of the device 52
5.2 Network usage by the apps in idle states 57

List of Symbols and Abbreviations

AAPT	Android Asset Packaging Tool
ACC	API Cost Calculator
ACPI	Advanced Configuration and Power Interface
ADB	Android Debug Bridge
AMP	Activity Manager Profiler
API	Application Programming Interface
APK	Android Application Package
CLI	Command Line Interface
CPU	Central Processing Unit
DDMS	Dalvik Debug Monitor Server
DuT	Device Under Test
DAQ	Data Acquisition
GA	Generic Algorithm
GPU	Graphics Processing Unit
GPS	Global Positioning System
GUI	Graphical User Interface
HBP	Hardware Based Profiling
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IoT	Internet of Things
JS	Java Script
LCD	Liquid Cristal Display
OS	Operating System
PC	Personal Computer
PC	Program Counter
PID	Pattern Identifier and Distributor
PID	Process Identifier
PHP	Hypertext Preprocessor
SBP	Software Based Profiling
URL	Uniform Resource Locator
USB	Universal Serial Bus
VNI	Visual Networking Index
Wi-Fi	Wireless Fidelity

Chapter 1

Introduction

Number of cell phone users all over the world has been quintupled since the first camera phone was introduced. According to the latest release of Cisco's Mobile Visual Networking Index (VNI) there will be 5.5 billion mobile phone users in the world by the year 2020 [21], which is 70 percent of the world's total population [37]. Due to this enormous demand, a rapid evolution of mobile devices in terms of both hardware and software is being observed. Today, apps can provide virtually the same features as their equivalent desktop applications. To satisfy user demands countless features are being added to these devices. However, such evolution comes at a high cost. Nowadays, quad core processors, high performance GPUs, integration of various sensors, powerful cameras and trend of having larger displays are making the devices more energy demanding as ever. Moreover, Apps that fully exploits the available hardware can drain the devices' battery at an unsatisfactory rate.

These energy greedy apps leads us to an apposite problem. Using such apps can leave the battery empty, preventing a user form even making a phone call. Nevertheless, use of such apps will require frequent battery recharges. It leads to another problem. This is because modern lithium ion batteries have a limited re-charging life cycle. Present studies show that generally these batteries can perform 300-500 re-charging cycles before the performance starts to decrease [17].

In order to improve battery life, a method to highlight API misuses which can cause high energy consumption in mobile apps is shown in this thesis. Inefficient use of API calls are often referred to as energy bugs [10]. The goal of this thesis is to categorise API methods invoked by an application and rank them based on energy consumption. Thus providing a guideline to the developed to find energy bugs from apps by measuring the energy cost of API calls. API usage patterns of an application is analysed by measuring energy consumption and mapping it with execution trace. To conduct this thesis ten Android apps from different domain categories have been selected. In order to collect the energy consumption values for each API call, a software based approach is used. For collecting energy measurement data, a test bed has been created jointly with Greenspector company. A tool called TestRunner [9] provided by Greenspector is used to collect energy consumption data of an app. Every app is launched with a common scenario using Greenspector Testrunner. Once all the energy measurement data is collected, those values are then aligned with execution traces generated

simultaneously during execution of the application.

Using this proposed measurement framework, the energy consumption of individual APIs are analysed as well as their impact over the network. These API methods are then categorised according to the component it belongs to. Finally this API energy consumption patterns are given as an actionable knowledge to the developers. Overall this work will be in the trend of Green ICT by improving software energy efficiency of mobile and ubiquitous computing.

1.1 Background

Greenspector has developed a software tool during last 4 years conducting several Research & Development (R&D) projects (Code Vert, WebenergyArchive, etc.). It is able to detect dirty patterns in source code in several languages based on unitary measured rules and it is also able to measure resource consumption based on automatic tests. But regarding automatic tests it can only determine the resource consumption of a particular application given a particular execution scenario. It has no means to relate this resource consumption to the application behaviour. To have a better understanding of the application behaviour in runtime Greenspector proposed to develop a relation between resource (such as - memory, CPU, energy) consumption with API method calls invoked by the application.

Greenspector [9] is a tool to help the developers to analyse their applications, softwares and websites with static and dynamic analysis. Static analysis refers to a method of computer program debugging that is done by examining the code without executing the program, whereas dynamic analysis refers to the testing and evaluation of a program by executing data in real-time. Greenspector manages to detect power-draining patterns through static analysis of source code of many languages such as Android, Java, C, HTML, PHP, JS and more. The software also offers a continuous dynamic measurement module, with probes available on smartphones, PCs (Windows, Linux) and servers. These probes measure power and energy consumption, CPU, memory, network data, etc. Static and dynamic analysis are based on more than 150 rules which are the results of continuous research and development. Greenspector suggests corrections to improve the level of software eco-design with limitations in some cases and helps to prioritise the corrections with an action plan that provides estimated gains for each correction.

Greenspector provides necessary information to obtain resource utilisation of an application. It also provides a set of rules for energy oriented code analysis. Greenspector can be integrated in different IDE including Eclipse and Intelij. Moreover it also gives the opportunity to the developers to do energy consumption measurement tests remotely and locally using their Testbench and Testrunner tools respectively. The following features are offered by Greenspector:

1. **Web Interfaces:** Aggregates test data about all the application and analysis of a user and provides a graphical representation of the data in order to define an action plan to improve eco-score.

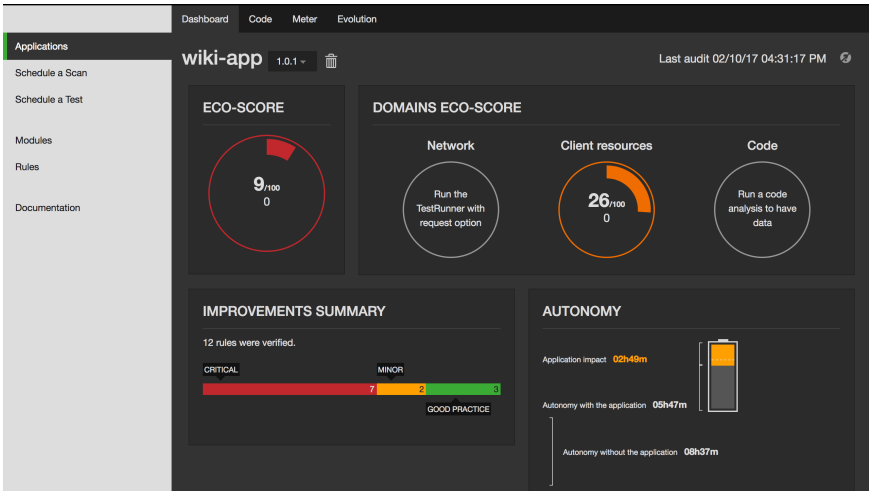

Figure 1.1: Web interface of Greenspector

2. **TestBench Cloud:** With the TestBench Cloud, a user can easily schedule standard automated test scenarios of their website or mobile app. Those test scenarios are executed in Greenspector Cloud on the computers and mobiles phones.

3. **Command Line Interface(CLI):** Greenspector command line is a tool based on Greenspector API. This tool helps to use most of the Greenspector features such as Code Analysis or Meter.

4. **Meter:** Meter tool is used for measures to get information on softwares about energy, CPU, performance and data. It then sends the results to Greenspector Server (automatically or manually using the command line interface) and analyse them with the web interface.

   Three probes are available, depending on the platform onto which a user wants to launch measures, such as PowerAPI for computer and server;EasyVirt Probe for virtual machine only and Android Probe for smartphones and tablets. Greenspector also developed different API and tools to run consumption measures in different languages:

   - Meter Java API
   - Meter Standalone Meter
   - JavaScript API
   - Meter Android API
   - TestRunnner

   In this thesis we have used the TestRunner(version 1.3.9 & 2.1.1) tool to launch automated tests. A detail description of the tool is given in chapter 3 section 3.4.5.

5. **Plugin:** Greemspector can be integrated in JMeter, Inntellij, Eclipse and SonarQube using the plugin tool provided for the mentioned IDE.

1.2 Motivation

The motivation behind this thesis is to develop a method that can be integrated to improve Greenspector tool which will help the developers to reduce time of correction and maximise gain. On other hand, the work explained in this thesis allows Greenspector to assess the quality of resources and energy consumption not only in unit measurement of rules but also in application contexts.

One of the major motivational aspects of this work is to reduce the energy consumption of applications on mobile devices. That will increase the duration of battery life by making recharging less frequent [14]. Thus increases the chances of reducing electronic wastes by increasing the duration of a mobile life span. Long lasting batteries will reduce the amount of waste which will eventually reduce recycling operations. At a larger context this will reduce the extraction of Earth's resources such as lithium and other materials required for production of batteries. Nevertheless, at global scale this research will contribute to reduce CO2e of mobile devices.

1.3 Problem Definition

A very common use case or usage scenario of smartphone is multitasking. Modern day smartphones contains hundreds of applications. Ability to do multitasking has given smartphones a new dimension. It is very common for the users to scroll down through any application's newsfeed such as Facebook, BBC, Wikipedia, etc. and at some moment of use often interesting links come across. Users click on these links and that opens up a new application. Meanwhile another application may pop up with a notification of interest. Single tap on that notification will open the application to show the entire information.

The scenario described above is common in everyday life. Usage of lots of application at the same time is normal in modern day mobile devices, thanks to the multitasking capability of OSs and quad core processors. Leaving the applications to run in the background is also a very common user behaviour. An application running idle or active in the background or idle in foreground seems to have no impact on the currently running application. But this is not entirely true. Due to high processing power the currently running application may not face any interruption in its performance, but multiple application running in idle or active mode definitely increase the platform discharge and CPU usage percentage. This leads to a research problem that how applications behave when they are idle? and what is the impact of these application on mobile's battery life? Considering the research scope of this problem, to give it a constructive form this thesis will mostly deal with following research questions.

1.3.1 Research Questions

1. Which group of API methods consumes most of the energy during execution in idle states?

This particular research question focuses on API method calls that occurs during execution of idle states of an application. The goal is to identify the energy cost of an API method and categorise it by components to which it belongs.

2. What behavioural difference exists between loading, idle foreground and idle background states of execution of the application?

   This research question focuses on the energy consumption of the application during loading, idle foreground and idle background. The objective is to draw a comparison analysis between these execution states and observe the similarities and differences.

1.3.2 Research Contributions

The contributions of this thesis are as following:

1. An approach for identifying API calls grouped by components or modules consuming high energy. Note that focus is to observe the behaviour of an application when it is running in idle mode.
2. API usage patterns in idle states.
3. Quantitative ranking of high energy demanding APIs for the developers to reduce energy consumption in certain execution states.
4. Comparison between states mentioned above based on their energy consumption.

1.4 Delimitations

The research is limited to the basic execution scenario (loading, foreground idle, background idle) of mobile application. This restricts from observing the behaviour of an application in its actual functional mode where it is performing desired tasks. Moreover for measuring energy consumption Greenspector TestRunner tool is used which records energy consumption at a time interval which is higher than the state-of-the-art. This may be a threat to the accuracy of the results obtained if a fair quantity of test data is not generated by reiterating the experiment a moderate number of times on the same application. .

1.5 Thesis Structure

This thesis is structured as follows.

Chapter 2: Related Works
In this chapter the broad concept of green software development is discussed and a brief description of the GREENSOFT model is given. Then a taxonomy of energy profiling schemes has been developed and based on that state-of-the-art schemes are discussed. Finally, a feasibility analysis based comparison of these schemes in terms of industrial standards of

software eco efficiency has also been done.

Chapter 3: Testbed Design Theory and Implementation
In this chapter the scientific theories related to the work and methodology adopted has been presented. Units of measurement, API, required tools, application selection, testbed setup, data extraction and analysis technique are the sections of this chapter.

Chapter 4: Application Profiling Results
This chapter discusses the outcome of the experiments. An application wise detailed behavioural analysis has been shown in this chapter. In other words this chapter contains API oriented benchmarking outcomes of the applications selected in the previous chapter.

Chapter 5: Discussion
In this chapter an analysis of the obtained results from the benchmarked applications are given to draw an outline of the similarities and differences.A comparison between applications belongs to the same category is also given. At the end of the chapter an overview of immediate afterwork has also been discussed which is related to the global scale impact of an application on the network.

Chapter 6: Conclusion and Future Works
A conclusion of the work done in this thesis and future works that can be done based on the existing work is discussed in this chapter.

Chapter 2

Related Works

In this chapter the broad concept of green software development is discussed and a brief description of the GREENSOFT model is given. Then a taxonomy of energy profiling schemes has been developed and based on that state-of-the-art schemes are discussed. Finally, a feasibility analysis based comparison of these schemes in terms of industrial standards of software eco efficiency has also been done.

2.1 The GREENSOFT Model

The GREENSOFT Model [38] is a conceptual model. It can also be explained as a reference model for developing "Green and Sustainable Software". The objective of this model proposed by S.Naumann *et. al.* is to support software developers, administrators, and software users in creating, maintaining, and using software in a more sustainable way.

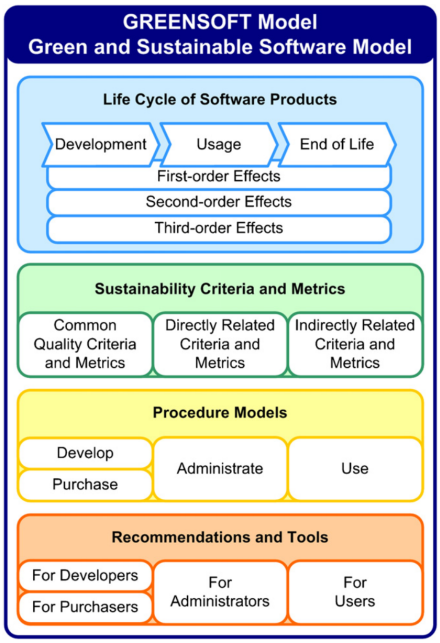

Figure 2.1: The GREENSOFT model. A reference model for green sustainable software

The model (see figure:2.1) consists four major parts:

1. **Life cycle of software products:** It is a "cradle to grave" life cycle module which enables the stakeholders to assess impacts of sustainable development(SD). The model comprises to three different orders of impacts which can also be referred to as effects of information and communication technology (ICT).This three orders of impact are:

Table 2.1: Impacts of ICT on the Environment

First-order Impact (effects of ICT supply)	Environmental effects that result from production and use of ICT, i.e. resource use and pollution from mining, hardware production, power consumption during usage, and disposal of electronic equipment waste.
Second-order Impact (effects of ICT usage)	Effects that result indirectly from using ICT, like energy and resource conservation by process optimization (dema- terialization & effects), or resource conservation by substitution of material products with their immaterial counterparts (substitution effects).
Third-order Impact (systemic effects of ICT)	long term indirect effects on the environment that result from ICT usage, like changing life styles that promote faster economic growth and, at worst, outweigh the formerly achieved savings (rebound effects).

   As we can see from figure 2.1 that "life cycle of software products" deals with three phases: development, usage and "end of life". Moreover, all the three phases has first-order, second-order and third-order impacts (see table 2.1). A detail distribution of different level of impacts at a different phase of the life cycle is shown in figure 2.2. So, following this model enables stakeholders while working in any of the phases to be aware of considering all order of impacts, because naturally it is easier to focus only on the first order impacts. Nevertheless, first order impact is the most important part of the life cycle of software product, because second and third order impacts will only occur if first-order impact exists.

2. **Sustainability criteria and metrics:** The model represents three categories of sustainability criteria and metrics for software product: Common Quality Criteria and Metrics, Directly Related Criteria and Metrics, and Indirectly Related Criteria and Metrics. A detail explanation of the categories can be found in [38].

3. **Procedure models:** It consists four sub-procedure models: develop, purchase, administrate and use. This part of the GREENSOFT model mainly deals with the circumstances under which a green and sustainable software is being engineered.

4. **Recommendations and tools:** It is to support stakeholders in applying green and sustainable techniques with different skill levels. when developing, administrating or

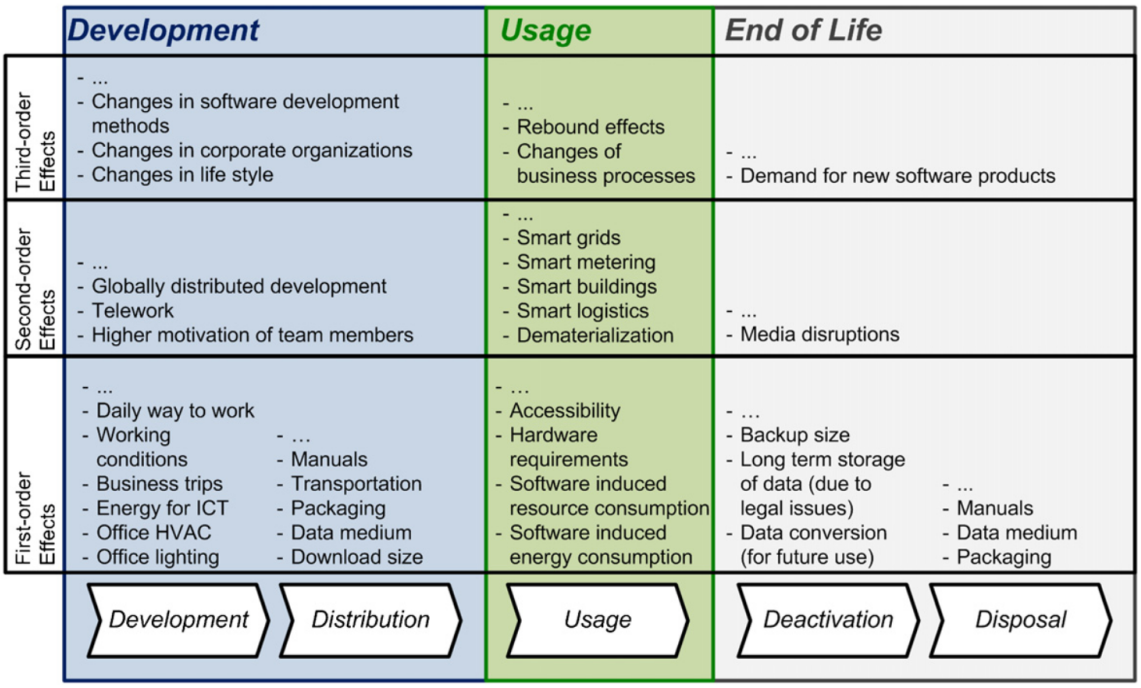

Figure 2.2: Cradle-to-grave inspired product life cycle for software products. [38]

using software products. Recommendations can be guidelines, checklists, best practice examples, implementation reports, etc. Tools can be software tools, but also any other tool, like paper-based data collection sheets. There

This thesis is focused mainly on the first-order impact of the usage phase of the Greensoft model which eventually improves the second order impact of a software product. The usage phase deals with resource and energy consumption. Our goal is to develop a model that guides the developer to improve the resource utilisation of the software in order to reduce energy consumption. The next section discusses about the taxonomy and state-of-art of application energy profiling schemes for mobile devices as this research is restricted to the applications running on mobile devices only.

2.2 Taxonomy of Energy Profiling Schemes for Application

This section contains a taxonomy for the classification of mobile application energy profiling schemes. Energy profiling schemes can be classified into two main categories based on their design pattern.They are: (1) Software Based Profiling and (2) Hardware Based Profiling.

Figure 2.3 demonstrates the design of hardware and software based energy profiling schemes. Hardware based energy profiling (HBP) is done using external hardware equipment, such as - volt meter, power miter, multimeter, etc. These devices are used to obtain voltage and current readings in order to estimate the power consumption of the mobile phone

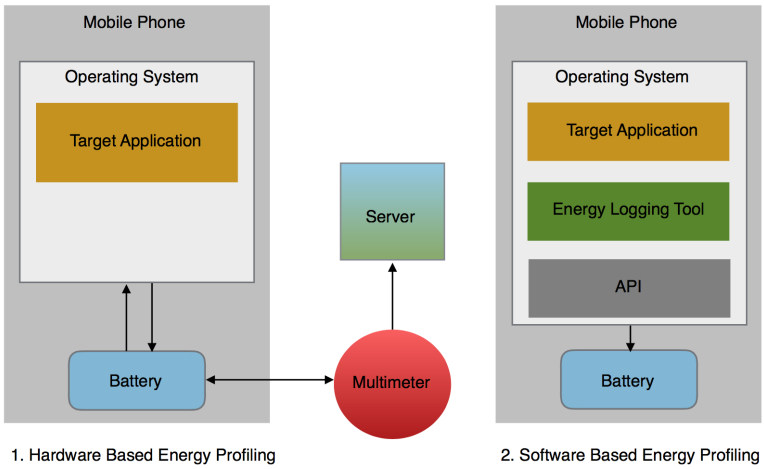

Figure 2.3: Hardware based vs software based energy profiling

against system activities. On the contrary, Software based energy profiling (SBP) exploits a software module (e.g., smart battery interface) to generate power models by collecting mobile components power usage statistics in order to estimate energy consumption of an app. Figure 2.3 shows the architecture of HBP and SBP.

HBP and SBP is further categorised based on the common characteristics among existing schemes (see figure 2.4). Common characteristics among software-based energy profiling schemes include, granularity, model flexibility, measurement source, and profiling type. Alternatively, the common characteristics among hardware-based energy profiling schemes include, granularity, power model design, measurement source, and execution environment [12].

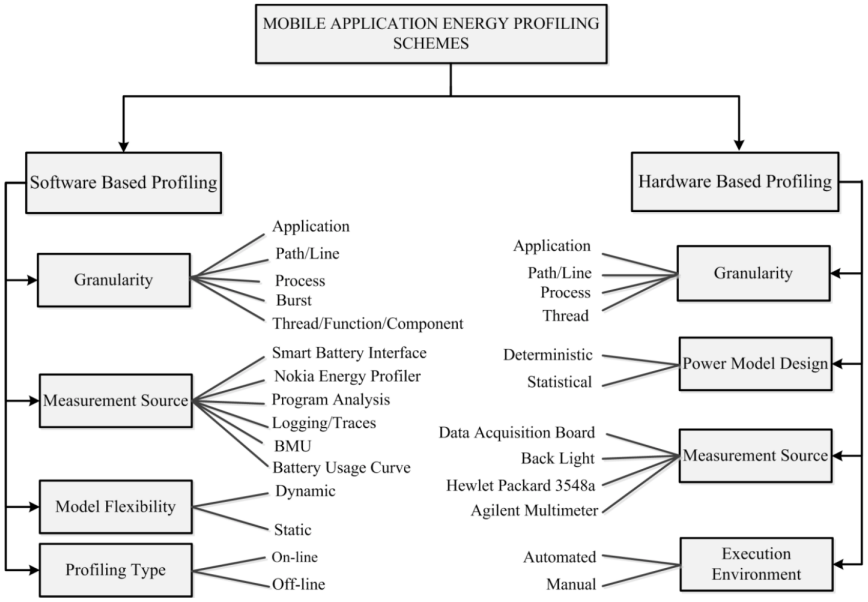

Figure 2.4: Taxonomy of mobile application energy profiling scheme[12]

In this thesis the adopted method focuses on SBP at the level of process granularity where the measurement source is smart battery interface, model flexibility is dynamic and profiling type is offline. The following section discusses the state-of-the-art of HBP and SBP schemes that are related to the work done in this thesis.

2.3 State-of-the-Art Application Energy Profiling Schemes

Over a decade, improving battery life has been one of the main concerns for the device manufacturers. At the same time software developers are practicing more and more energy efficient coding practices to make programs less power hungry. A lot of research approaches exist for measuring energy consumption in mobile applications. The advantages and limitations of these approaches are summarised here, as well as an outline of similarities and differences with the approach presented in this thesis are also given.

Energy profiling is a broad research combining operating systems, software engineering, networking and many other fields. But in this thesis works related to energy profiling of mobile applications are considered only. In particular the following dimensions while classifying related works were considered: number of apps used while evaluating a proposed technique; Scheme used for collecting and estimating energy measurements, i.e., Hardware based profiling (HBP), Software based profiling (SBP) before drawing conclusion and technique used for processing collected data to evaluate the behaviour of the tested applications.

2.3.1 Hardware Based Application Energy Profiling Schemes

1. **An Empirical Study [35]:** In this study hardware power monitor is used to measure the energy consumption of method calls in mobile apps form different domain during typical execution scenario. The aim of this study is to determine the most energy greedy application programming interface (API) methods. The empirical study design consists app execution scenario, data extraction process and data analysis method. App execution scenario for each of the 55 apps used is defined in the app execution scenario module. Then a three step process is employed by the data extraction process: a) collecting energy measurements and execution traces generated by the app at the same time using predefined execution scenarios, b) Aligning energy measurement data with execution traces and c) trace the API calls and patterns to the source code. In data analysis method, a quantitative analysis is made to report the distribution of the energy consumption for the executed method calls. Then a qualitative analysis in made based on the quantitative analysis to identify the methods and patterns that consume more energy. However, in this study energy consumed by APIs related to 3G/GSM and GPS were not measured, since these are already considered as energy greedy hardware components [19, 40].

2. **Power scope:** Power scope[24] maps energy consumption of a specific process during a certain time period. It captures current and voltage readings of the profiling computer through a digital multi-meter from an external power source. Then it samples the

data to obtain the current drawn by the profiling computer.It is referred to as energy monitor. The system monitor makes a small set of kernel modification. Then it records the value of program counter(PC) and process identifier(PID) of the currently running process. Energy monitor module is responsible for clock synchronisation. It generates a system interrupt on the profiling computer after the multi-meter takes each current sample. Lastly the energy analyser module which runs on an external server generates energy profiles for system activities. This was applied on a video player application. The initial study claims to have reduced the energy consumption of the video player application by 46%. However chances of having inaccurate profiling results can not be denied since the scheme depends of external triggering mechanism for clock synchronisation.

3. **DuT:** This scheme analyses battery power consumption when multiple mobile applications are executing simultaneously using various components of the device. Device under test(DuT) [19] and data acquisition(DAQ) are the two major components of the proposed methodology for power consumption behaviour analysis. For experimental setup to measure current, sense resistors are inserted on the power supply rails of the relevant components of the mobile device. Circuit schematics of the experimenting device must be provided by the manufacturer in order to insert sense resistors. On the other hand the software module which runs on a computer launches DAQ process to generate power model and energy consumption estimation. Then regular scenarios like video, audio playback, phone call, sms, emailing were executed to benchmark overall power consumption of different components of the device. The limitation of this method is that it must use a smartphone whose design files are available. Moreover the device used in the experiment is old and does not support 3G.

2.3.2 Software Based Application Energy Profiling Schemes

1. **Elens:** Elans [34] combines program analysis and per-instruction energy profiling to estimate energy consumption of an application. By recording the paths traversed during the execution of an application the program analysis module estimates energy consumption. It can measure energy consumption at different granularity level (see figure 2.2) without any intervention from the developer. Workload generator, Analyser and source code annotator are the modules of Elens's system design. Translating workloads into a set of paths, exploiting path information with software energy profile to measure energy consumption of an app and visualising energy usage to the application source code are the responsibility of the modules respectively. No OS kernel modification or Circuit schematics information is needed to implement this scheme. However, the assumption made by the scheme of being the per-instruction power profile always true is false because the instruction set may vary from architecture to architecture.

2. **Power-prof:** Power-Prof [32] uses genetic algorithm(GA) to estimate power models for mobile components (i.e. 3G, LCD, CPU, etc.) under certain context aware conditions. Smart battery interface is used to acquire time stamped power measurements. To improve accuracy or to reduce search space four different power states are considered

while analysing battery power draining rate. Overall this design is very efficient. This is beacause of its dedicated server for handling computational tasks. A disadvantage of Power-Prof can be the use of genetic algorithm(GA) which is time consuming and resource hungry.

3. **Power Booter:** PowerBooter [46], an automated technique for constructing power model, which uses builtin battery voltage sensors and knowledge of battery discharge behaviour to monitor power consumption. At the same time it explicitly controls the power management and activity states of individual components while generating power models. Power models for mobile components including CPU, LCD, Wi-Fi, cellular and GPS are constructed by applying a multi variable regression analysis on system utilisation traces and battery discharge curve. This technique has a resource rigorous and time consuming training phase where as it varies the states of the component of interest while keeping all the other components constant to avoid power estimation noise.Moreover, this method causes rapid battery discharge because it does not use any external server to perform expensive computational tasks.

4. **Eprof:** Eprof [39] is a fine-gained energy profiling tool for android apps. The study focuses on to resolve the asynchronous power behaviour issue which can be defined as the impact of an entity on the power consumption of the phone which persists long after the entity is completed. In order to resolve this issue Eprof adopted the last trigger accounting policy. Author in this paper identifies components tail power state, persistent power state wake-locks and exotic components within the smartphone as the key reasons for asynchronous power consumption behaviour. Where tail energy is defined as a routine which can trigger a component(e.g. disk, Wi-Fi, 3G, GPS) to enter into a high power state and remain in that state long beyond the end of the routine. Persistent state wake-lock occurs due to aggressive sleeping policies in smartphones OSs and apps exporting APIs which is needed to ensure the components stay awake, irrespective of user activities, in order to perform their intermittent activities in the background (e.g. network sync). Meanwhile, the exotic components such as GPS, camera, accelerometer and sensors consumes significant amount of battery power due to frequent activation and deactivation by separate entities. However, the advantage of this scheme can be referred to its ability to map the energy consumption of mobile components at different granularity(e.g. process, sub-routines, thread and system call) and the limitation of Eprof can be referred to as overlooking the energy consumption due to OS policy and poor software design.

5. **Sesame:** Sesame[23] constructs accurate and fine-grain energy models using battery interfaces on modern mobile devices. It also contributes in characterisation of battery interfaces, their limitation and a set of rules to overcome the limitation. Data collector, model constructor and model manager are the three major components of Sesame. Data collector method uses system statistics(e.g. CPU timing and memory usage) and Advanced Configuration and Power Interface (ACPI) to estimate the power consumption of system components. The model constructor employs two iterative techniques namely, model molding and predictor transformation. In model molding a high rate energy model is constructed by compressing low rate models to improve accuracy and

rate. Whereas the predictor transformation part Principal Component Analysis (PCA) to improve the accuracy of the molded model. Finally the model manager organises a hash table of multiple energy models where system configurations are used as a list of keys. However, Sesame only deliberates the components(e.g. CPU and memory) visible to the OS for power modelling while ignoring the foremost energy consuming components(e.g. Wi-Fi, GPS) in mobile devices.

6. **SEMO:** SEMO [22] profiles mobile application using battery usage information. SEMO consists of three major components, namely an inspector, a recorder and an analyzer. Inspector checks the information of the battery (e.g. percentage, health status voltage, temperature and total battery charge). Recorder records the information collected by the inspector and names of the applications which are running at that time. Recorder records data periodically with a record interval of one minute. Analyzer uses the data recorded by the recorder to rate the energy consumption of the applications and eventually rank them by their energy consumption rate. The advantage of this scheme is that it provides a ranking of the app to the developers to visualise the energy consumption behaviour of their app. However it is a low rate scheme compared with the other state of the art schemes because it records data every one minute.

7. **Wattson:** Wattson [36] is a power modelling and resource scaling based took that empowers developers to identify critical sections within mobile applications. It highlights the utmost energy-consuming entities by mapping application behaviour on a mobile platform. Wattson's design enables it to emulate processing speed and network characteristics without requiring any laboratory equipment. Energy consumption of an application is measured by applying utilisation counter method of the power modelling entity. and this is done by the scaling module, which emulates the mobile environment within a workstation. Wattson claim's that this scaling module is the solution to platform heterogeneity issues. The advantage of this scheme is that it emulates the power consumption of the display, CPU, and network components. This scheme is flexible because it does not require any external equipment and mobile phones to estimate power consumption of a mobile application. However, this flexibility can also be considered as a disadvantage of the scheme as emulations can not provide accurate energy consumption statistics since emulating device does not contain a lithium-ion battery. Moreover, display screens behaves and consumes differently in different brightness which can not be determined using an emulator. Furthermore, This tool does also ignores power consumption by GPU and GPS and profiling time is also high

2.4 Comparison Between State-of-the-Art and Adopted Research Method

The previous section discusses about a quite a few application energy profiling schemes. Some used HBP and some used SBP. Irrespective of the profiling technique used every scheme has its own advantage and limitations. The table 2.2 gives a brief comparison between the experimental approach of this thesis and other works.

Table 2.2: Related energy profiling schemes

	Tecnique	APP	Scheme
1	Power scope[24]	1	HBP
2	DuT[11]	6	HBP
3	Chung *et al.* [20]	4	HBP
4	Hao *et al.* [27]	5	SBP
5	Xu *et al.* [45]	6	SBP
6	Adopted research method in this thesis	10	SBP

Energy profiling is a broad research combining operating systems, software engineering, networking and many other fields. But in this thesis works related to energy profiling of mobile applications are discussed only. In particular the following dimensions are considered while classifying related works(see table 2.2): Number of Apps (APP) used while evaluating a proposed technique; Scheme used for collecting and estimating energy measurements, i.e., Hardware based profiling (HBP), Software based profiling (SBP) before drawing conclusion.

Some of the existing techniques shown in the table 2.2 are related to HBP and SBP. Efforts related to SBP in smartphones are also found in Blunck *et al.* [32], Huang *et al.* [30], Dong and Zhong [23], Gurun and Krintz [26], Ding *et al* [22] and Qian *et al* [42]. In comparison with the research method accepted in this thesis none of the existing approaches record application centric energy consumption measures at process granularity using SBP. Moreover, a quantitative analysis has been done to obtain distribution of energy greedy API methods within an application. It is also important to mention that results obtained in this thesis can be reproduced by following the approach mentioned in the next chapter, having access to Greenspector and using the code and data given in the appendix.

Chapter 3

Test-bed Design Theory and Implementation

Previous chapter discussed about the state-of-the-art application energy profiling schemes. In this chapter the scientific theories related to the work and methodology adopted has been presented. Units of measurement, required tools, application selection, testbed setup, data extraction and analysis technique has been discussed in the following sections.

3.1 Units of Measurement

To better understand the work done in the thesis it is important to clarify the units of measurements. A brief description of electrical power, energy and charge is given below.

Electrical Power: In physics, power is the rate of doing work. It is the amount of energy consumed per unit time.

$$P = W/t$$

In the SI system, the unit of power is the joule per second (J/s), known as the *watt*. But, The instantaneous electrical power P delivered to a component is given by,

$$P(t) = V(t) * I(t)$$

where,
P(t) is the instantaneous power, measured in *watts* (joules per second)
V(t) is the potential difference (or voltage drop) across the component, measured in *volts*
I(t) is the current through it, measured in *amperes*

Electrical Energy: Energy is the ability to do work, where work is done when a force moves an object. Whereas electrical energy is energy that's stored in charged particles within an electric field or in other terms we can define it as power used to move electric charge from one point to another over a time period. The unit of energy is *'joule'* but also referred to as *'watt-hour'* while measuring electrical devices.

$$E_{(Wh)} = P_{(W)} * t_{(h)}$$

In this thesis a mobile device has been used as a test test base and the capacity of the battery of the device is measured as *'milliampere hour'*. It is a unit for measuring electric power over time. *mAh* is commonly used to describe the total amount of energy a battery can store at one time. the relation between *'milliampere hour'* and *'watt-hours'* is as follows:

$$Q_{(mAh)} = 1000 * E_{(Wh)}/V_{(V)}$$

where,
The electric charge Q(mAh) in *milliamp-hours* (mAh) is equal to 1000 times the energy E(Wh) in *watt-hours* (Wh) divided by the voltage V(V) in *volts* (V).So,

$$milliamp - hours = 1000 * \frac{watt - hours}{volts}$$

or,

$$mAh = 1000 * \frac{Wh}{V}$$

so,

$$Wh = \frac{mAh * V}{1000}$$

3.2 Characteristics of Lithium-ion Battery

Lithium is the lightest of all metals, has the greatest electrochemical potential and provides the largest energy density for weight [28]. But attempts to make batteries have drastically failed to due to safety problems. So the research took a turn to non-metallic lithium battery using lithium ions. Sony Corporation was the first commercialised lithium ion batteries in 1991 by meeting certain precautionary requirements when charging and discharging despite being slightly lower in energy density than lithium metal due to safety purposes.[16]. Some of the advantages and limitations of lithium-ion batteries that were summarised in [16] are given below.
Advantages:

- Energy density is high.
- When new prolonged priming is not needed. Regular charge is all that's needed. Reasonable size can be produced economically.
- Self-discharge is relatively low- compare to nickel-based batteries self-discharge is less than half.
- Low Maintenance
- High current to applications can be provided by specialty cells.

Limitations:

- Protection circuit is required to maintain current and voltage within safe limits.

- Ageing of the battery is a factor, even if not in use - storage in a cool place at 40% charge reduces the ageing effect.[16]
- Restrictions to transportation- shipment of large quantities are subject to regulatory control due to security purposes. This restriction does not apply to personal carry-on batteries.
- High manufacturing cost - The manufacturing cost is 40 percent higher than nickel-cadmium.
- Not fully mature - metals and chemicals are changing on a continuing basis.

The nominal cell voltage of 3.60V as a power source is adhered by equipment manufacturers for most of the Li-ion systems.[31]. Though higher voltage can easily boost the *'watt-hours'* of the device on paper because we know, voltage multiplied by current equals to *watt*. Despite having the opportunity to increase the capacity of mobile batteries manufactures restrict themselves to 3.60V as cell voltage. Some manufactures use 3.70V as nominal cell voltage but even that can cause unfamiliar references when connecting more that one cell in series.[31].

To give a more practical idea let's consider a mobile device, which has a battery capacity of 3600 mAh and the voltage is 3.60V. So energy in watt-hours will be,

$$E = \frac{3600mAh * 3.60V}{1000}$$

$$E = 12.96Wh$$

Very roughly, a 3600 mAh battery can supply 3600 mA for one hour, or 360 mA for ten hours, or 36 mA for 100 hours, but this is a simplification since the capacity of the battery depends on the discharge rate. So a rating of 3600 mAh is actually an incomplete description of the battery's capacity unless we also know the corresponding discharge rate.

3.3 Application Program Interface(API)

In computer programming, A set of subroutine definitions, protocols, and tools for building application software is often defined as Application program interface(API) [13]. In general terms, application program interface is a set of methods of communication between various software components. Developing computer programs becomes easier with good APIs which provides all the building blocks, which are then put together by the programmer. An API may be for a web-based system, operating system, database system, computer hardware or software library. An API specification can take many forms, but often includes specifications for routines, data structures, object classes, variables or remote calls. POSIX, Microsoft Windows API, the C++ Standard Template Library and Java APIs are examples of different forms of APIs. Documentation for the API is usually provided by the creator to facilitate usage.

3.4 Required Tools

3.4.1 Android Studio

Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA.[2] In other words Android Studio in build on top of IntelliJ's powerful code editor and developer tools, which offers even more features that enhances the productivity when building Android apps, such as:

- A Gradle-based build system, which offers flexibility
- Integrated emulator, which is fast
- Single environment for all Android devices
- Building a new APK is not required to push changes to a running app.
- Code templates to help developers to build common app features and GitHub integration for importing sample code
- Extensive testing tools and frameworks
- To catch performance, usability, version compatibility, and other problems lint tools are avaiable
- C++ and NDK support
- Google Cloud Platform has built-in support, which makes it easy to integrate Google Cloud Messaging and App Engine

3.4.2 Android Debug Bridge (ADB)

Android Debug Bridge (ADB) is a command-line tool which is versatile and allows communication with a device (a connected Android device or an emulator).[1] A variety of device actions are facilitated by ADB commands, such as installing applications, debugging, devise status, network status, uninstalling apps etc.. Moreover, it also provides access to a Unix shell. That is very helpful when it is necessary to run a variety of commands on a device. ADB is a client-server based program that includes three components:

- A client, to send commands. The development machine hosts the client.
- A daemon (adbd), which is used for running commands on a device. The daemon runs on each device as a background process .
- A server, for managing communication between the client and the daemon. The server runs as a background process on the development machine.

3.4.3 Traceview and Dmtracedump

Traceview is a text and graphical viewer. It is used for viewing execution logs that is created by using the debug class on the code to log tracing information. Traceview helps to debug an application and profile its performance.[8]

Creating Trace Files

To use Traceview, log files are need to be generated, which contains the trace information for analysis.Trace logs can be generated in two ways:

One is to include the Debug class in the code. It calls methods such as startMethodTracing() and stopMethodTracing(). These methods are used to start and stop logging of trace information and stores to the disk. This option is very accurate and precise. This is because it specifies exactly where to start and stop logging trace data in code.

There is another method profiling feature. It is called DDMS which also generates trace logs. This option is less accurate because it does not require code modification. Although there is less control in this method on exactly where logging starts and stops, this option is useful when access to the source code of the application is not available.

Before start generating trace logs, being aware of the following restrictions is important: To use debug class the application must have permission to write to external storage.If using DDMS:Android 2.1 and earlier devices must have an SD card present and the application must have permission to write to the SD card. [8] An SD card is not needed for the devices equipped with android 2.2 or later. In the case of DDMS the trace log files are streamed directly to the development machine, whereas the trace log files are stored in the test device with debug class.

3.4.4 Weka

Weka contains data visualisation tools. It contains algorithms for data analysis. It also have predictive modelling features with graphical user interfaces for easy accessing.[44] The original version was primarily designed as a tool for analysing data from agricultural domains,[29][25] but In 1997 Java-based version (Weka 3) was launched, which is now used in many different application areas. It is largely used in particular for educational purposes and research. Weka has a lot advantages including:

- Weka is under GNU General Public License. So it is free.
- Runs on almost any platform, since it is implemented in Java
- An extensive collection of data preprocessing and modelling techniques.

Weka supports a wide range of standard data mining tasks, more specifically, data preprocessing, clustering, classification, regression, visualisation, and feature selection. All of Weka's techniques are predicated on the assumption that the data is available as one flat file

(.arff) or relation, where each data point is described by a fixed number of attributes which similar to csv file format.

3.4.5 Greenspector TestRunner

TestRunner is a tool offering to the developer the ability to easily launch standard automated tests. Using TestRunner on Android platforms it is possible to test an APK or a web site.

- The APK standard test scenarios include testing and energy measuring of APK launch, APK running idle in foreground and in background.
- The website standard test scenarios include testing and energy measuring of website launch, website idle in foreground, scrolling and website idle in background.

Prerequisites for launching TestRunner:

In order to measure URL or APK on Android platform the following environment configuration is needed.

1. A PC equipped with an Operating System (Mac OS, Windows, Linux)
2. Java version 7+ installed in the PC
3. Android SDK with Android Debug Bridge (ADB) and Android Asset Packaging Tool (AAPT) configured. AAPT is provided by the build-tools installed in Android-SDK. It is recommended to install build-tools 21.1.2 and ADB 1.0.32 to ensure maximum compatibility.
4. Git Bash to be installed (For windows only)
5. An Android smartphone which has already been connected to the computer via ADB Wi-Fi with Android version 4.3 (API Level 18) or higher. On Windows, sometimes device may not appear from the ADB devices command, in this case, a specific USB driver is needed.
6. In order to have the Platform Discharge metric, a phone from the supported devices list given below is needed.

   Supported Device list

   The probe can be installed on any Android Environment. However, energy probes (software metrics such as CPU, memory, platform discharge, etc.) will only be available for the following smartphones:

   - HTC Desire 820
   - Huawei Ascend P7
   - Huawei Mate 8

- LG Optimus 5 II
- Motorola Moto G (XT1039)
- Nexus 5
- Nexus 6
- Nexus 6P
- Samsung Galaxy S6
- Samsung Galaxy S7 Edge

Run TestRunner

1. Verify the prerequisites mentioned in the previous section, especially version installed of adb, aapt and Java on your computer.
2. Connect the smartphone to the computer via USB and type:
   adb tcpip 5555
3. Disconnect the USB cable and search the IP of your mobile in the settings(e.g. settings>About device>Status).
4. Establish connection with the mobile using ADB:
   adb connect mobile_IP
5. Download TestRunner from Greenspector and save it in a directory of your choice.
6. Verify that TestRunner is executable. If not make it executable by changing permission:
   chmod +x testrunner
7. Launch TestRunner and 2 new files(config.yml and jobs.yml) should be created.
   ./testrunner
8. Edit config.yml and jobs.yml according to the need.
9. Again run TestRunner
10. Wait for the test to be completed
11. Visit the web interface (see section 1.1.1) to see the results for the application tested. One can choose to store the results locally by changing the boolean value of the "online" tag in jobs.yml file.

3.5 Profiling Application

In this study API energy consumption by an application is analysed. To do so open source applications have been used. Ten apps from different domains are selected (see section 3.5.1). To measure the cost of API calls Greenspector Testrunner is used. It is a software that can measure the energy consumption of mobile apps in different states and execution scenarios

with a 200ms of recording time interval. As for hardware a brand-new, unlocked Samsung Galaxy S7 edge is chosen. It is also included in the supported device list of section 3.4.5. Samsung has two versions of the same device. Here European version of the device (SM-G935F) is used, which has a Exynos 8890 Octa chipset, Octa-core (4x2.3 GHz Mongoose and 4x1.6 GHz Cortex-A53) CPU, 3600 mAh battery and equipped with Android version 6.0.1 (Kernel version- 3.18.14-9689434) [5].

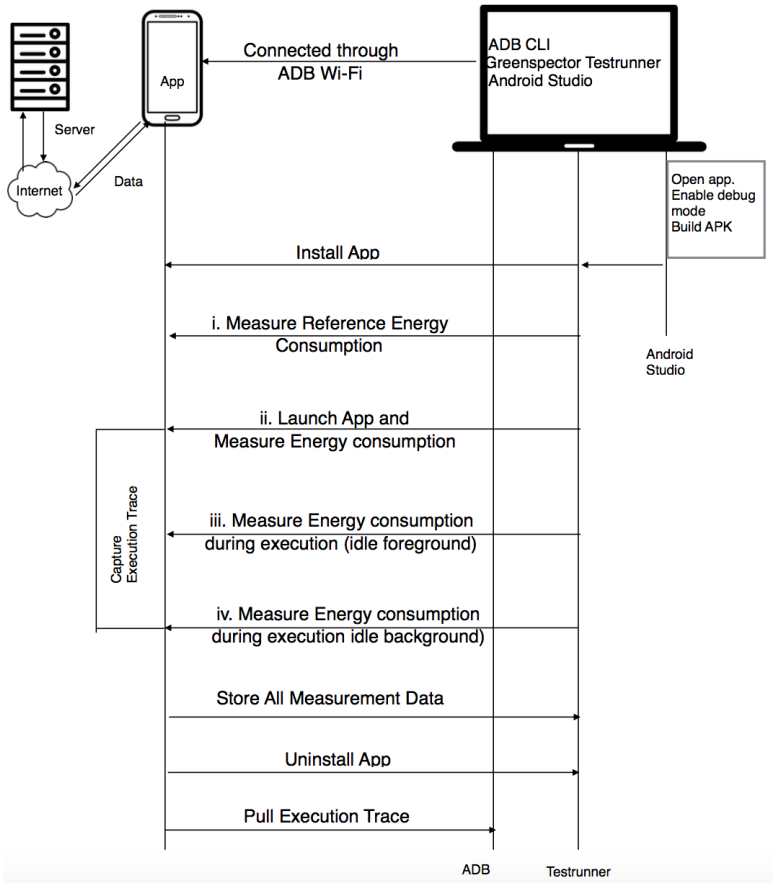

Figure 3.1: Workflow of Tesbed for obtaining energy consumption and execution trace of an application

This research is limited to android devices and the decision of choosing android device for testing was not random. It is evident that Android is the most used platform in mobile devices. Android holds the largest number of installed-base devices, with 1.9 billion in use in 2014, compared with 682 million iOS/Mac OS installed-base devices [43]. Moreover, android phones are generally more flexible and test can be engineered and monitored from a laptop for debugging and profiling purposes. In this thesis all the commands are deployed from a Laptop equipped with MacOS Sierra.

The diagram shown in figure 3.1 is the working methodology of this experiment. The diagram shows the workflow from the beginning of the experiment such as connecting development machine (laptop) with test device (mobile phone) with ADB over Wi-Fi to the

end where the execution trace data have been extracted from the test device. A detailed explanation of this diagram is given in the following sections.

3.5.1 App selection and Enabling Debug Mode

Ten apps have been selected. The reason for selecting ten application was not random. Some of the works that were discussed in the related works chapter uses different number of applications based on the demand of the research work. By analysing state-of-the-art schemes it was evident that ten apps can be considered as a moderate number of entity to conduct the experiment and outlining a summery. Minimum one and maximum two apps from one category have been selected for this experiment. A list of application with its category is shown in the table 3.1.

Table 3.1: List of Application

Category	Application Name	Version
Books and Reference	Wikipedia	2.5.192
Education	Hubble Gallery	1.5.1
Education	Memory Game	1.0.04
Libraries & Demo	MapBox	Debug
Games (Puzzle)	2048	2.08
Music & Audio	Easy Sound Recorder	1.2.5
News & Magazines	Flym News Reader	1.9.7
News & Magazines	NIST Connect	2.5
Productivity	Omni Notes	5.3.0 Beta
Travel & Local	CycleStreets	3.5

These categories have been selected based on the popularity ranking [6], compatibility with testbed designed for the experiment such as compatible with API level 23 [7] and availability as open source manner. It was possible to select either open source apps or closed source apps from google play for experiment. A decision was made to use open source apps considering a future work plan where access to applications' code will be necessary. After apps were selected and launched with Android studio a slight code modification has been done. To create the execution trace files (shown in figure 3.1), the debug class has been included and the startMethodTracing() method is called [8] to the main class. To build an Android Package Kit (APK) from android studio a new run configuration was build with gradle[41]. With debug mode enable it is not needed to sign the application as it is done automatically by android studio. Running such app generates a log file with (.)trace extension containing API method calls and its execution times.

3.5.2 Test-bed Design

To obtain measurement data using Greenspector TestRunner access to their server was needed. After authenticating an account with Greenspector a design plan of the testbed

has been introduced as shown in figure 3.1. Some strict measures are followed to avoid any noise in the test data. The mobile device used was without sim-card. It has been done to avoid any cellular network interference as it could produce noise which may lead us to false results. To perform tests a connection between test device and laptop using ADB was established. Connecting the devices over Wi-Fi is preferred rather than USB connection to avoid USB charging. Although it is possible to disable USB charging but in that case the test device must be rooted. It is also taken into consideration that Wi-Fi connection may cause extra consumption on the mobile device. To avoid this, measurement of the reference energy consumption of the device took place after the establishment of Wi-Fi connection. It was done to ensure that no extra energy consumption is added to the application while it is loading or running. During experiment internet connection is kept enable in the device since most of the application requires internet to establish a connection with its server.

3.5.3 Execution Scenario

Greenspector Testrunner can perform different execution scenarios and measure the application energy consumption at the same time. It uses the dumpsys tool [4] provided by android for CPU, RAM, Battery and storage stats to check how an application affects the overall device. Objective of this thesis was not maximising the code coverage. So a simple execution scenario was designed which is common in every app. Figure 3.1 provides a detailed view of the execution scenario. The steps of the execution scenario are as follows:

- Connecting test device to the laptop with ADB over Wi-Fi
- Clear existing energy measurement data files and execution trace files if any
- Install the application to the test device
- Measure platform discharge
- Launch the application
- Measure the platform discharge
- Execute the application in idle foreground
- Measure the platform discharge
- Execute application in idle background
- Measure the platform dischaerge
- Transfer measurement data files to the laptop
- Uninstall the application
- Transfer execution trace file to the laptop
- Disconnect test dive and the laptop

Battery discharge rate referred as reference platform discharge was measured when the device is idle and during three different execution states of the application. All the four platform discharge measurement states are as follows.

- When the device is idle and nothing happening other then routine tasks referred as the reference discharge rate.
- When the app is loading.
- When the app is idle but running in foreground.
- When the app is idle but running in background.

Before every measurement all the other application including sensors like NFC, Bluetooth, etc were closed. But location access was enable because some apps may use it in regular basis. The entire test scenario was automated and every state is measured for default 20 seconds. Keeping in mind that every application has different loading time to stay on the safe side it was decided to make the measurement duration fixed for simplicity. To avoid any sort of false reading every application was measured for 20 times on fully charged device.

The design of the execution scenario is simple. It is obvious that this execution scenario does not cover all the features of the application or more precisely almost no features. As already explained before that the objective of this thesis is not to improve code coverage rather improve energy footprints in idle execution states.

3.5.4 Data Extraction Process

Energy consumption measurement data obtained by Greenspector Testrunner is generally stored in the test operating machine directly. But it is also possible to send the data to Greenspector server for visualising profiling charts. It provides separate measuring data file for every step as explained in execution scenario.

On the other hand, The execution traces of the application are also generated simultaneously because the APK used was enabled for debugging. Note that is it also a requirement for running the Android Activity Manager Profiler (AMP). Generally, the execution trace file is in binary format and stored in the sdcard directory of the mobile device by default. Execution trace file has to be pulled from the mobile device to the laptop after completion of every test iteration. To view information from the execution trace android traceview [8] tool was used. It is only a visualiser and is not used for exporting data to text format. Dmtracedump [8] tool is used to export data from execution trace files. But in this thesis traceview is used to view and store the the execution trace file in text format. Then a custom parser is used to extract only the valuable data needed for measuring the energy consumption of the APIs.

Calculation Formula

Before the application is launched in the device we measure the standby discharge rate and energy consumption of the device referred to as reference platform discharge speed and reference platform discharge respectively. Reference platform discharge speed is calculated by using the following formula.

$$P_{(reference)} = \frac{StandbyStateBatteryDischarge + BatteryDischargebyGreenspectorTestRunner}{ExecutionTime}$$

To measure the platform discharge caused by the application the difference between platform discharge while the application is running and reference platform discharge is calculated. It is the impact of the application on the device. The following formula is used to measure platform discharge caused by the application where E is denoted as platform discharge or energy consumption.

$$E_{application} = E_{device} - E_{reference}$$

The impact of the application no the device is basically the summation of 3 different execution states as described in the previous section.

$$E_{application} = \sum_{i=loading,idleForeground,idleBackground} (E_i - E_{reference})$$

After measuring platform discharge caused by each step the next step is to match execution trace data with obtained platform discharge speed data.

3.5.5 Data Analysis

After both the data files are gathered as shown in figure 3.2, The API Cost Calculator (ACC) module was launched to match the platform discharge speed measurement data files with the parsed execution trace file in order to find the cost of API method. The calculation formula measuring energy consumption of API methods is given at the end of this section.

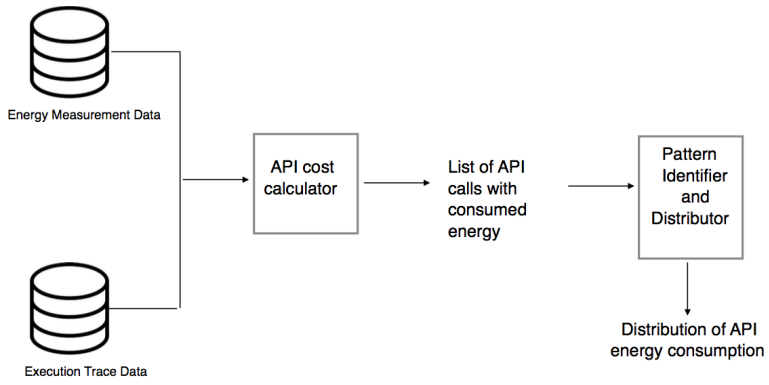

Figure 3.2: Analyse Execution Trace and Measurement Data

Once each API method was tagged with its energy consumption, the Pattern Identifier and Distributor (PID) module was launched to generate a raking of categorised distribution of API methods based of the energy consumption for the application.

Calculation Formula

Time interval for recording data of GREENSPECTOR TestRunner is 200ms. More than one API methods are called during this period of time. During execution of one API method other methods are also called and starts execution. So at one given point of time there can be more than one API method running.

$$P_{application} = \sum_{i=loading,idleForeground,idleBackground} (P_i - P_{reference})$$

In execution trace file every API method is tagged with the initiation time and execution time. So, the difference between initiation time and termination time is the execution time of an API method and is denoted as time delta in the following equation.

$$\Delta t = t_{apiCallTerminate} - t_{apiCallInitiate}$$

This time delta is then matched with platform discharge speed data times using the initiation time of API method.

$$\Delta t_{api} = \Delta t_{application}$$

Overlaps of API call initiations with common time delta are matched and filtered for a more accurate result.

$$E_{APImethod} = E_{\Delta t} - \sum E_{\Delta t_{OtherAPI}}$$

Here the energy cost of API method is derived from the platform discharge of the device at the matched time delta and subtracting the platform discharge of other API methods at the matched time delta from it.

3.6 Summary

This chapter discussed about the important theories and also the implementation of the experiment. Since the energy measurement time interval is much higher than the API method execution time mean vales were used to obtain consumption values at any given time. If more than one API method were executing at any given time, the platform discharge of that specific moment by the application was divided by the number of active api calls to obtain individual values.

Chapter 4

Application Profiling Results

This chapter discusses the outcome of the experiments. An application wise detailed behavioural analysis has been shown in the next section. It is very important to note that, here energy consumption has been denoted as platform discharge. Where platform refers to the device used for experiment and platform discharge rate/speed refers to the device's battery discharge per second.

4.1 Benchmarking Application by API Behaviour Analysis

The list of application that have been benchmarked is given in table 4.1. It also provides the version number and category of the app.

Table 4.1: List of Benchmarked Application

Application Name	Version	Category
Wikipedia	2.5.192	Books and Reference
2048	2.08	Games (Puzzle)
CycleStreets	3.5	Travel & Local
Hubble Gallery	1.5.1	Education
Memory Game	1.0.04	Education
MapBox	Debug	Libraries & Demo
Flym News Reader	1.9.7	News & Magazines
Omni Notes	5.3.0 Beta	Productivity
Easy Sound Recorder	1.2.5	Music & Audio
NIST Connect	2.5	News & Magazines

Every application has been measured 20 times at a constant environment before finalising the outcomes. The following sections explains the energy consumption statistics of the tested applications, energy consumption of API calls by categorisation and inefficient usage of API methods during execution if any.

4.1.1 Wikipedia

The average reference platform discharge rate of the device at the time of the experiment was 92.96 µAh/s with a standard deviation of 1.968%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 8 hours and 57 minutes. At the time of loading the application average platform discharge rate rises up to 131.28 µAh/s with a standard deviation of 3.92%. In idle foreground and background states the average platform discharge rate is 130.34 µAh/s (standard deviation 4.90%) and 97.29 µAh/s (standard deviation 5.779%) respectively. According to the measurements application's impact on the device is 2 hours which means the battery will last for 6 hours and 57 minutes. Table 4.2 shows the average platform discharge speed and energy consumption (platform discharge) of Wikipedia application in different execution states over 20 test iterations.

Table 4.2: Average platform discharge of Wikipedia app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	92.96	1.968%	1.70	1.65%
Loading	131.28	3.92%	2.39	3.86%
Idle Foreground	130.34	4.90%	2.35	4.66%
Idle Background	97.29	5.779%	1.76	6.21%

Figure 4.1 and 4.2 are graphical representations of table 4.2. As shown in the figure 4.1 that the platform discharge rate is almost same in loading and idle foreground state. Whereas, the platform discharge rate significantly falls in idle background state. The grey line in the diagram represents the reference platform discharge.

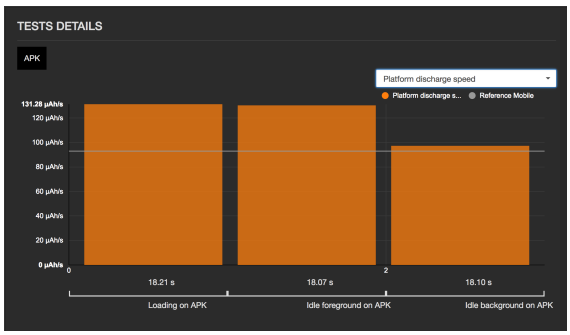

Figure 4.1: Platform discharge speed of Wikipedia app in different execution states

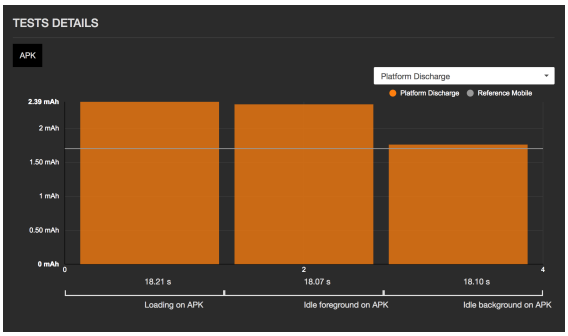

Figure 4.2: Platform discharge of Wikipedia app in different execution states

Figure 4.3 represents the energy consumed in micro Ampere hour (µAh) by the Wikipedia application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle

background state of the application respectively. It was interesting to find that the average energy consumption between loading and idle foreground states of the application has a difference of only around 1%.

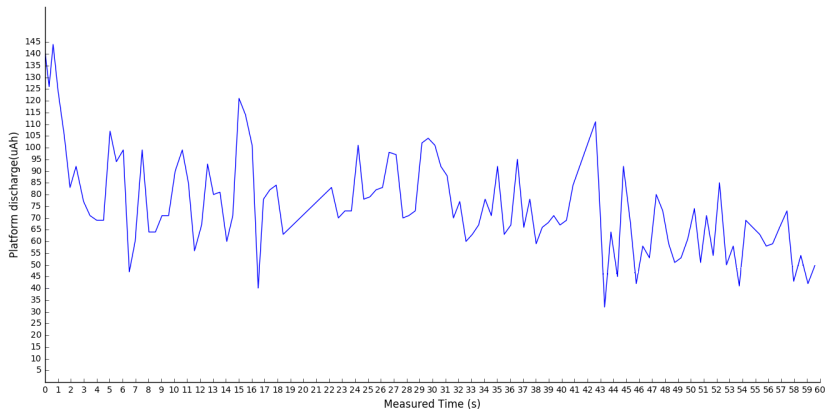

Figure 4.3: Energy consumption of Wikipedia application over time

Furthermore, the energy consumed by the API methods called by Wikipedia app were plotted in a graph over time. Figure 4.4 represents energy consumption of all the API methods that were invoked by Wikipedia application. Here, every dot represents an API method. The categorisation of API usage pattern was obtained using the Pattern Identifier and Distributor module described in methodology.

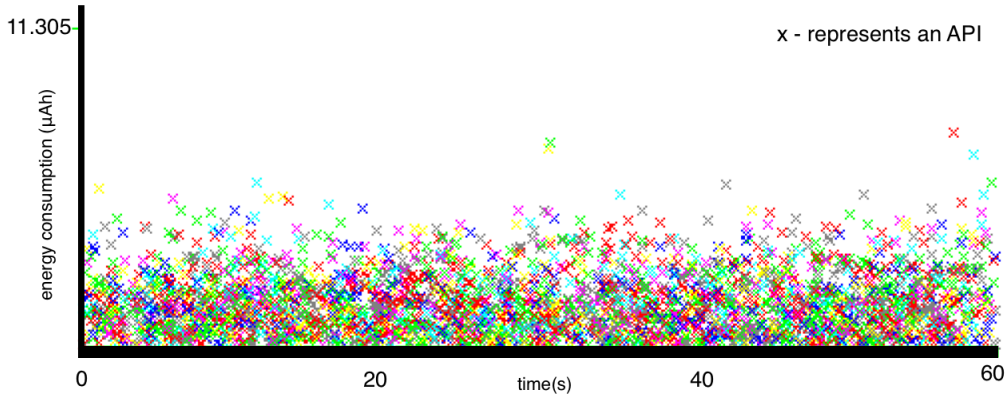

Figure 4.4: Distribution of API energy consumption of Wikipedia application over time

Based on quantitative analysis on the API usage pattern a categorisation of the API methods as shown in table 4.3.OS related calls are maximum then other components and the the energy consumption. It is also important to mention that this table does not represent the method invocations that may occur while exploiting every features of the application.

It was interesting to notice that the Wikipedia application continuously initiates API calls related to Facebook which has a significant impact on the apps energy consumption.

Table 4.3: Distribution of API methods called by Wikipedia application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	188 (6.4%)	5.622 (3.78%)
Network	159 (5.373%)	0.89545 (0.6%)
Database	68 (2.29%)	3.240 (2.155%)
Hardware	13 (0.439%)	0.117 (0.078%)
Display	45 (1.52%)	0.325 (0.218%)
Graphics	5 (0.168%)	0.008 (0.005%)
Facebook	321 (10.84%)	5.037 (3.376%)
Rest of the calls	2160 (72.992%)	133.989(89.784%)

It consumes almost 3.5% of the total energy consumption of the application and initiates around 321 calls per execution.

4.1.2 2048

The average reference platform discharge rate of the device at the time of the experiment was 102.47 µAh/s with a standard deviation of 7.62%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 8 hours and 07 minutes. At the time of loading the application average platform discharge rate rises up to 145.71 µAh/s with a standard deviation of 1.84%. In idle foreground and background states the average platform discharge rate is 128.22 µAh/s (standard deviation 6.08%) and 105.4 µAh/s (standard deviation 10.609%). According to the measurements the application's impact on the device is 1 hour and 32 minutes which means the battery will last for 6 hours and 35 minutes. Table 4.4 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of 2048 application in different execution states over 20 test iterations.

Table 4.4: Average platform discharge of 2048 app in different execution states

State of measurement	Platform Discharge Speed (µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	102.47	7.62%	1.87	7.48%
Loading	145.71	1.84%	2.66	2.17%
Idle Foreground	128.22	6.08%	2.32	5.68%
Idle Background	105.4	10.609%	1.89	10.41%

Figure 4.5 and figure 4.6 are graphical representations of table 4.4. As shown in the

diagrams the platform discharge rate has noticeable difference in loading and idle states. The energy consumption is high during loading and then continues to drop during idle foreground and background state. The grey line in the diagram represents the reference platform discharge which is almost equal to the consumption values of idle background state.

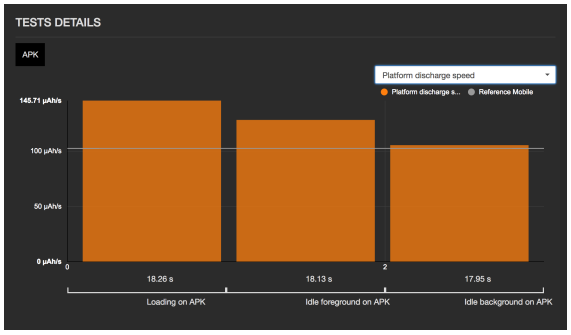

Figure 4.5: Platform discharge speed of 2048 app in different execution states

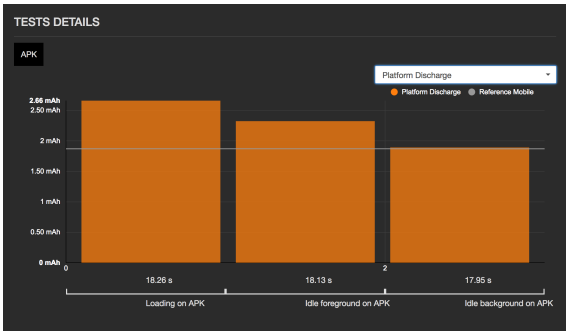

Figure 4.6: Platform discharge of 2048 app in different execution states

Figure 4.7 represents the energy consumed in micro Ampere hour (µAh) by the 2048 application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.

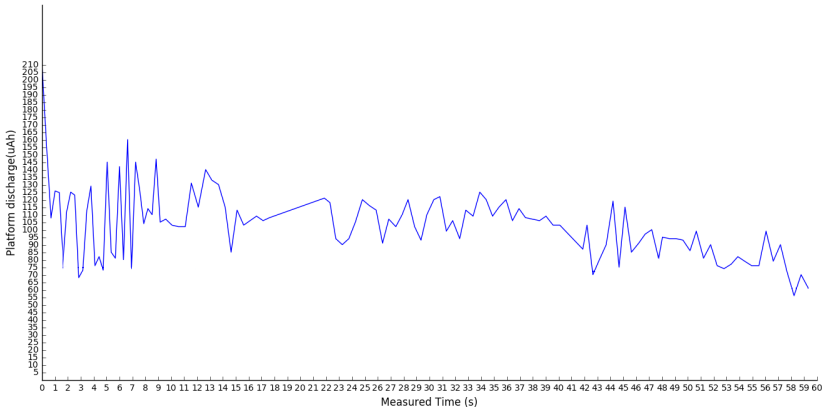

Figure 4.7: Energy consumption of 2048 application over time

Furthermore, the energy consumed by the API methods called by the 2048 application over time were plotted in ta graph. Figure 4.8 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method.

Based on a quantitative analysis on the API usage pattern, a categorisation of the API methods as shown in table 4.5. OS related calls are higher in number than other components as we have executed the app only in idle modes. This table does not represent the method invocations that may occur while exploiting every features of the application.

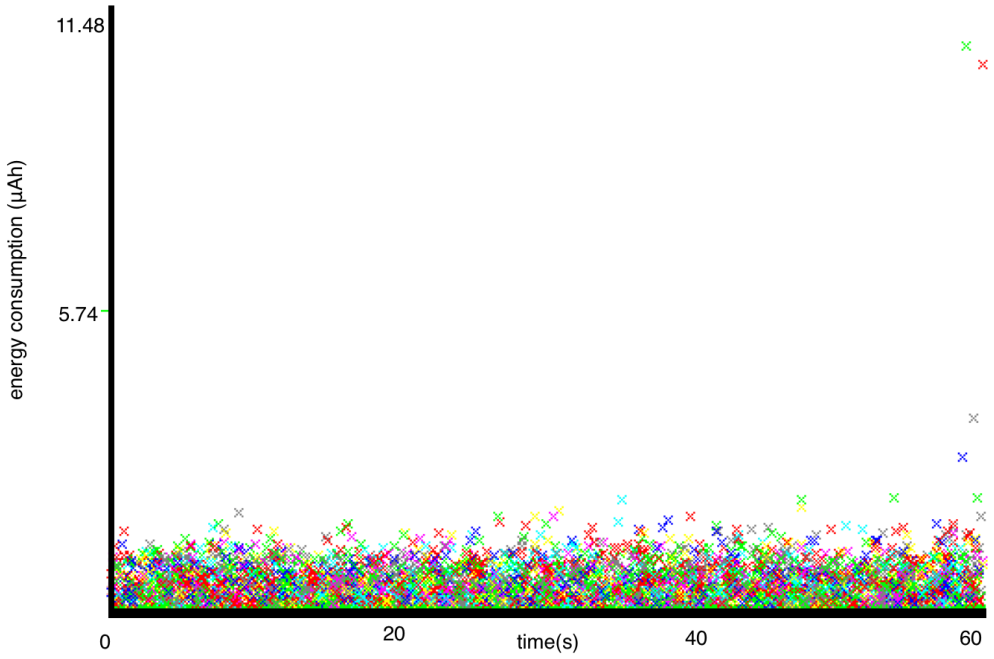

Figure 4.8: Distribution of API energy consumption of 2048 application over time

Table 4.5: Distribution of API methods called by 2048 application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	190 (4.852%)	7.722 (7.111%)
Network	33 (0.841%)	0.257 (0.237%)
Database	21 (0.53%)	0.122 (0.112%)
Hardware	33 (0.841%)	0.28 (0.258%)
Display	402 (10.256%)	4.416 (4.406%)
Graphics	352 (8.981%)	2.144 (1.974%)
Rest of the calls	2888 (73.6895%)	93,656(86.239%)

From the experiments no such abnormal calling pattern were observed. But it does not mean that the application does not contain api calls that have high energy consumption. From the graph 4.7 high consumption pick points are noticed and API methods with high consumptions are also visible form the figure 4.8

4.1.3 CycleStreets

The average reference platform discharge rate of the device at the time of the experiment was 93.21 µAh/s with a standard deviation of 8.96%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 8 hours and 56 minutes. At the time of loading the application average platform discharge rate rises to 132.67 µAh/s with a standard deviation

of 18.75%. In idle foreground and background states the average platform discharge rate is 117.09 µAh/s (standard deviation 13.47%) and 96.35 µAh/s (standard deviation 7.25%). According to the measurements the application's impact on the device is 1 hour and 43 minutes which means the battery will last for 7 hours and 13 minutes. Table 4.6 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of CycleStreets application in different execution states over 20 test iterations.

Table 4.6: Average platform discharge of CycleStreets app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	93.21	8.96%	1.71	9.12%
Loading	132.67	18.75%	2.44	18.93%
Idle Foreground	117.09	13.47%	2.12	12.69%
Idle Background	96.35	7.25%	1.76	7.7%

Figure 4.9 and 4.10 are graphical representations of table 4.6. From figure 4.9 it is seen that the platform discharge rate is high in loading and continues to fall in idle states. The grey line in the diagram represents the reference platform discharge. In idle background state the applications' consumption values are almost equal to the reference value.

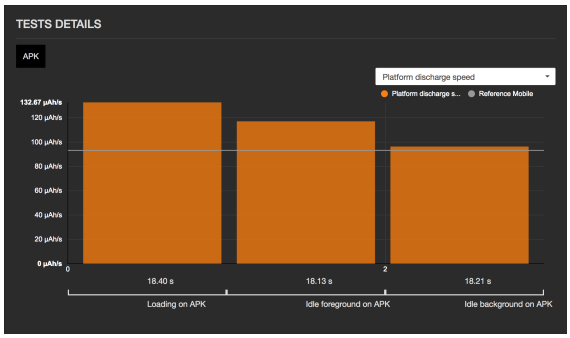

Figure 4.9: Platform discharge speed of CycleStreets app in different execution states

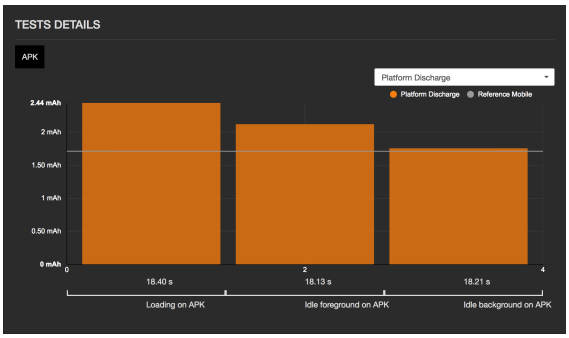

Figure 4.10: Platform discharge of CycleStreets app in different execution states

Figure 4.11 represents the energy consumed in micro Ampere hour (µAh) by the CycleStreets application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.

Furthermore, the energy consumption by the API methods called by the application over time is plotted in a graph. Figure 4.12 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method.

Based on a quantitative analysis on the API usage pattern, the categorisation of the API methods as shown in table 4.7. OS related calls are high in number but display and graphics

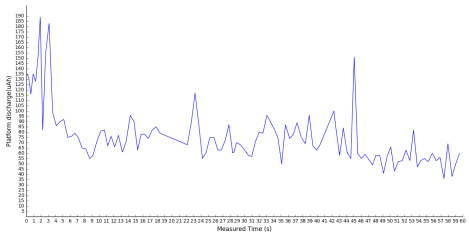

Figure 4.11: Energy consumption of CycleStreets application over time

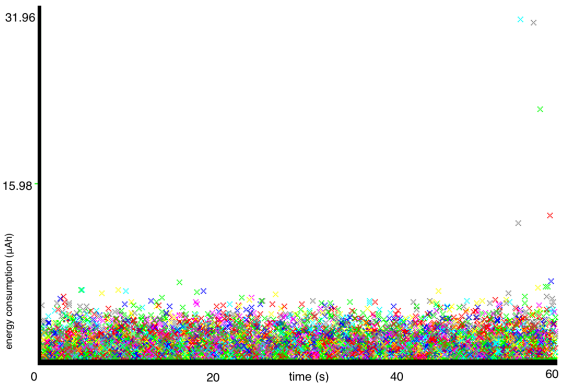

Figure 4.12: Distribution of API energy consumption of CycleStreets application over times

Table 4.7: Distribution of API methods called by CycleStreets application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	261 (4.877%)	13.164 (4.134%)
Network	197 (3.613%)	1.41 (0.445%)
Database	58 (1.08%)	0.44 (0.141%)
Hardware	33 (0.601%)	0.427 (0.0.134%)
Display	411 (7.673%)	7.385 (2.3%)
Graphics	323 (6.094%)	6.674 (2.106%)
Rest of the calls	4131 (76.05%)	287.641 (90.73%)

calls were higher than OS. We noticed that number of calls are not proportional to energy consumption. Despite OS having less number of calls than display and network, the energy consumption was almost equal to the cumulative value of them.

4.1.4 Hubble Gallery

The average reference platform discharge rate of the device at the time of the experiment was 91.22 µAh/s with a standard deviation of 1.69%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 9 hours and 08 minutes. At the time of loading the application average platform discharge rate is 97.38 µAh/s with a standard deviation of 13.08%. In idle foreground and background states the average platform discharge rate is 97.11 µAh/s (standard deviation 14.71%) and 99.19 µAh/s (standard deviation 6.87%). According to the measurements the application's impact on the device is 37 minutes 21 seconds which means the battery will last for 8 hours and 30 minutes. Table 4.8 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of Hubble Gallery application in different execution states over 20 test iterations.

Table 4.8: Average platform discharge of Hubble Gallery app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	91.22	1.69%	1.67	1.96%
Loading	97.38	13.08%	1.78	13.32%
Idle Foreground	97.11	14.71%	1.75	15.82%
Idle Background	99.19	6.87%	1.80	6.81%

Figure 4.13 and 4.14 are a graphical representation of table 4.8. The figure 4.13 reflects that the platform discharge rate is almost same for all the state. The grey line in the diagram represents the reference platform discharge. It was interesting to notice that idle background state has slightly higher energy consumption than other measured states.

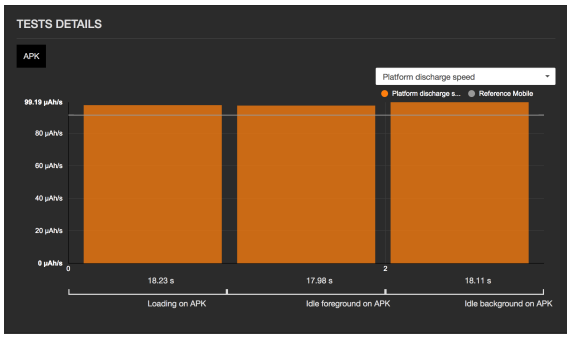

Figure 4.13: Platform discharge speed of Hubble Gallery app

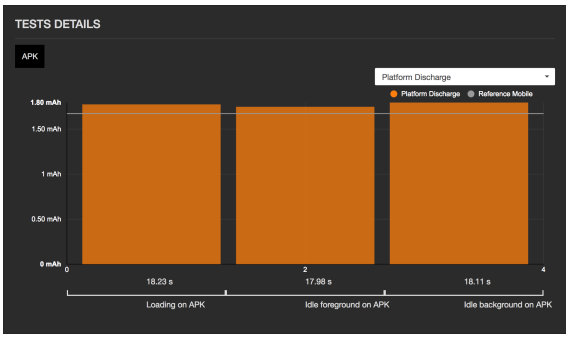

Figure 4.14: Platform discharge of Hubble Gallery app in different execution states

Figure 4.15 represents the energy consumption behaviour in micro Ampere hour (µAh) by the application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.

Furthermore, the energy consumption by the API methods called by the application over time were plotted in a graph. Figure 4.16 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method.

Based on a quantitative analysis on the API usage pattern, a categorisation of the API methods is shown in table 4.9. OS related calls are high in number being the only category to have a noticeable impact. Most of the energy consumed by the application falls in the rest of the call category which we considered as language calls essential for the app to perform.

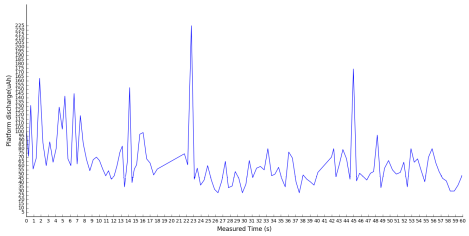

Figure 4.15: Energy consumption of Hubble Gallery application over time

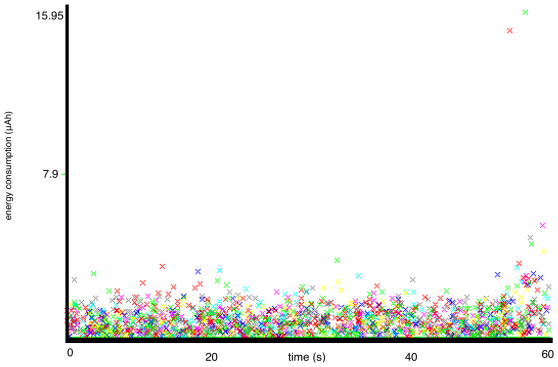

Figure 4.16: Distribution of API energy consumption of Hubble Gallery application over times

Table 4.9: Distribution of API methods called by Hubble Gallery application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	164 (9.218%)	2.665 (1.707%)
Network	17 (0.939%)	0.049 (0.031%)
Database	5 (0.277%)	0.019 (0.011%)
Display	3 (0.189%)	0.004 (0.002%)
Graphics	7 (0.396%)	0.028 (0.018%)
Rest of the calls	1572 (88.97%)	153.37 (98.226%)

4.1.5 Memory Game

The average reference platform discharge rate of the device at the time of the experiment was 97.14 µAh/s with a standard deviation of 5.89%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 8 hours and 34 minutes. At the time of loading the application average platform discharge rate is 179.14 µAh/s with a standard deviation of 5.73%. In idle foreground and background states the average platform discharge rate is 143.89 µAh/s (standard deviation 4.12%) and 118.71 µAh/s (standard deviation 3.25%). According to the measurements the application's impact on the device is 2 hours and 37 minutes which means the battery will last for 5 hours and 49 minutes. Table 4.10 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of Memory Game application in different execution states over 20 test iterations.

Figure 4.17 and 4.18 are a graphical representation of table 4.8. The figure 4.17 shows that the platform discharge rate is very high when the application is loading and then decreases for all the state. The grey line in the diagram represents the reference platform discharge. It was interesting to notice that idle background state has the most energy consumption than other measured states.

Figure 4.19 represents the energy consumption behaviour in micro Ampere hour (µAh)

Table 4.10: Average platform discharge of Memory Game app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	97.14	5.89%	1.78	5.69%
Loading	179.14	5.73%	3.29	6.16%
Idle Foreground	143.89	4.12%	2.61	4.78%
Idle Background	118.71	3.25%	3.33	3.42%

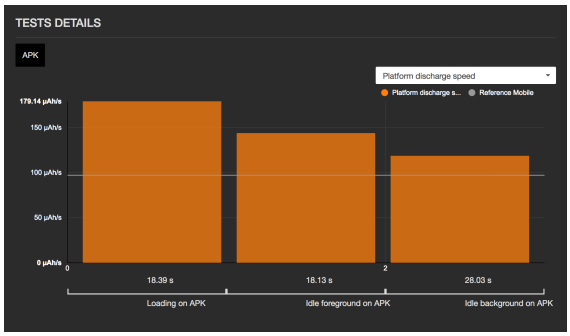

Figure 4.17: Platform discharge speed of Memory Game

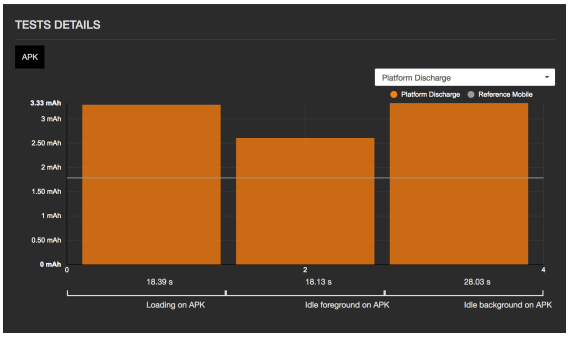

Figure 4.18: Platform discharge of Memory Game app in different execution states

by the application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.The behaviour of the application during idle background state was abnormal. It is noticeable from the graph that the platform discharge falls dramatically during last 20 seconds but in figure 4.18 the energy consumption is maximum during this specific time delta.

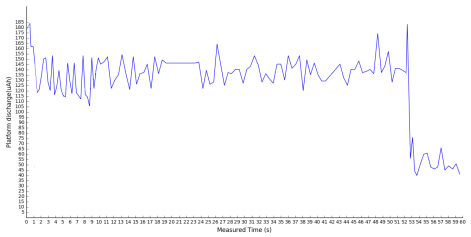

Figure 4.19: Energy consumption of Memory Game application over time

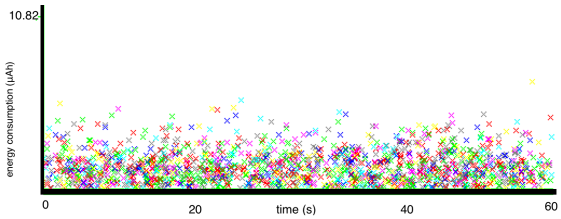

Figure 4.20: Distribution of API energy consumption of Memory Game application over times

Furthermore, the energy consumption of the API methods called by the application over time is plotted in a graph. Figure 4.20 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method. From dis

diagram we can notice that the density of API calls in the last 20 seconds of the measurement was higher which relates to the figure 4.18.

Table 4.11: Distribution of API methods called by Memory Game application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	90 (5.6%)	1.473 (2.949%)
Network	8 (0.49%)	0.0122 (0.024%)
Database	1 (0.621%)	0.00074 (0.0014%)
Display	293 (18.21%)	3.614 (7.238%)
Graphics	310 (19.266%)	25.28 (50.622%)
Hardware	3 (0.186%)	0.0282 (0.056%)
Rest of the calls	904 (56.18%)	19.529 (39.1%)

Based on a quantitative analysis on the API usage pattern, a categorisation of the API methods is shown in table 4.11. OS related calls are high in number but not more than display and graphics. Graphics being the category to have the maximum impact. Most of the energy consumed by the application falls in this category. Our observation shows that this category only consists 19% of the total number of calls but consumes more than 50% energy. A in-depth analysis of graphics related calls is needed to find out the inefficiency.

4.1.6 MapBox

The average reference platform discharge rate of the device at the time of the experiment was 106.18 µAh/s with a standard deviation of 10.91%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 7 hours and 50 minutes. At the time of loading the application average platform discharge rate is 139.15 µAh/s with a standard deviation of 11.01%. In idle foreground and background states the average platform discharge rate is 138.13 µAh/s (standard deviation 9.00%) and 109.66 µAh/s (standard deviation 6.66%). According to the measurements the application's impact on the device is 1 hours and 22 minutes which means the battery will last for 6 hours and 28 minutes. Table 4.12 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of Map Box application in different execution states over 20 test iterations.

Figure 4.21 and 4.22 are a graphical representation of table 4.12. The figure 4.21 shows that the platform discharge rate is very high when the application is loading and in idle foreground then decreases in the idle background state. The grey line in the diagram represents the reference platform discharge.

Figure 4.23 represents the energy consumption behaviour in micro Ampere hour (µAh) by the application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.The behaviour of the application during idle foreground state was similar to the loading state. The graph shows a balanced energy consumption behaviour of the application.

Table 4.12: Average platform discharge of MapBox app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	106.18	10.91%	1.94	10.76%
Loading	139.15	11.01%	2.52	11.21%
Idle Foreground	138.13	9.00%	2.43	8.66%
Idle Background	109.16	6.66%	1.98	6.31%

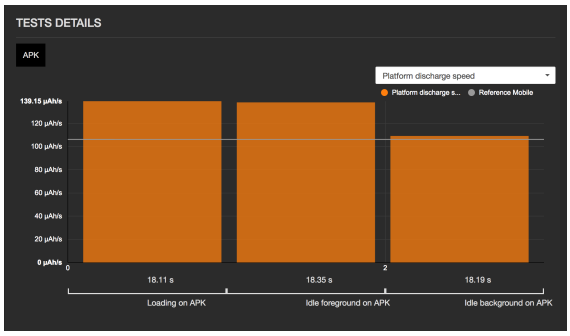

Figure 4.21: Platform discharge speed of Map Box app

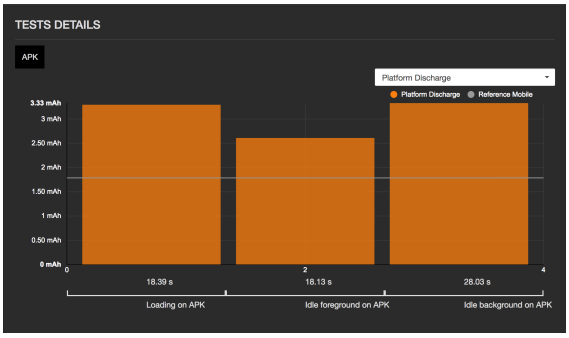

Figure 4.22: Platform discharge of Map Box app in different execution states

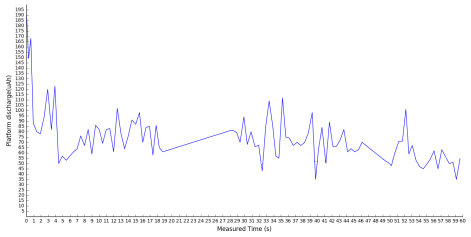

Figure 4.23: Energy consumption of Map Box application over time

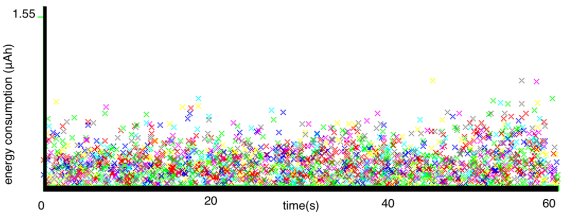

Figure 4.24: Distribution of API energy consumption of Map Box application over times

Furthermore, the energy consumption of the API methods called by the application over time is plotted in a graph. Figure 4.24 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method.

Based on a quantitative analysis on the API usage pattern, a categorisation the API methods as shown in table 4.13. OS related calls are high in number but not more than display and graphics. Display is the category to have the maximum impact. Most of the energy consumed by the application falls in this category if rest of the call section is excluded. Observation shows that this category only consists 20% of the total number of calls but consumes more than 10% energy which can be considered as good behaviour

Table 4.13: Distribution of API methods called by Map Box application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	86 (4.1727%)	1.431 (3.244%)
Network	8 (0.388%)	0.0069 (0.0158%)
Database	3 (0.14%)	0.0040 (0.009%)
Display	413 (20.038%)	4.545 (10.305%)
Graphics	274 (13.294%)	1.994 (4.519%)
Hardware	3 (0.145%)	0.0101 (0.023%)
Rest of the calls	1274 (61.81%)	36.134 (81.883%)

4.1.7 Flym News Reader

The average reference platform discharge rate of the device at the time of the experiment was 102.82 µAh/s with a standard deviation of 4.86%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 8 hours and 06 minutes. At the time of loading the application average platform discharge rate is 107.92 µAh/s with a standard deviation of 19.77%. In idle foreground and background states the average platform discharge rate is 112.27 µAh/s (standard deviation 27.08%) and 107.08 µAh/s (standard deviation 8.646%). According to the measurements the application's impact on the device is 27 minutes 47 seconds which means the battery will last for 7 hours and 38 minutes. Table 4.14 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of Flym News Reader application in different execution states over 20 test iterations.

Table 4.14: Average platform discharge of Flym News Reader app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	102.82	4.86%	1.87	4.24%
Loading	107.92	19.77%	2.00	20.01%
Idle Foreground	112.27	27.08%	2.02	27.06%
Idle Background	107.08	8.46%	1.95	7.79%

Figure 4.25 and 4.26 are a graphical representation of table 4.14. The figure 4.25 shows that the platform discharge rate is almost unchanged during the entire experiment. The grey line in the diagram represents the reference platform discharge which is almost in parallel with the orange bars in the diagram.

Figure 4.27 represents the energy consumption behaviour in micro Ampere hour (µAh) by the application over time in seconds (s). The time frame of 60 seconds in x-axis is a

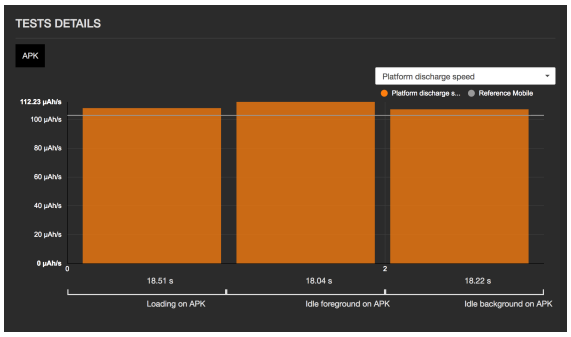

Figure 4.25: Platform discharge speed of Flym app

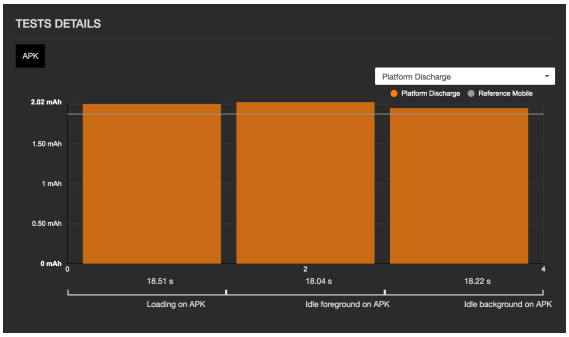

Figure 4.26: Platform discharge of Flym app in different execution states

combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.The energy consumption of the application during idle background state was a little bit downwards than the rest of the state. Which is a visual violation of the values received from the **??**.

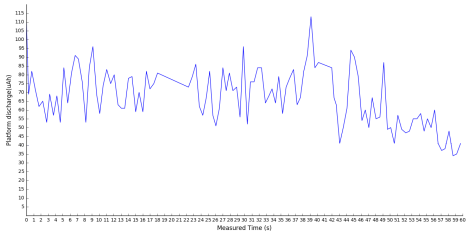

Figure 4.27: Energy consumption of Flym application over time

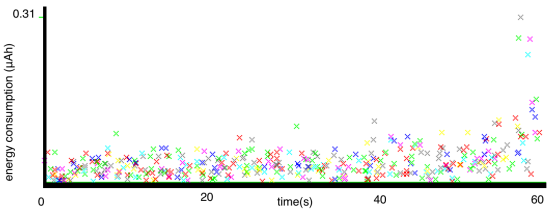

Figure 4.28: Distribution of API energy consumption of Flym application over times

Table 4.15: Distribution of API methods called by Flym News Reader application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	16 (3.19%)	0.017 (1.59%)
Network	8 (1.59%)	0.023 (2.09%)
Database	5 (1.00%)	0.011 (0.01%)
Display	67 (13.35%)	0.231 (21.077%)
Graphics	7 (1.72%)	0.021 (2.00%)
Hardware	3 (0.60%)	0.034 (0.31%)
Rest of the calls	369 (78.88%)	0.788 (71.92%)

Furthermore, the energy consumed by the API methods called by the application over time in plotted in a graph. Figure 4.28 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method. The

diagram shows that the API calls that were initiated during the last 20 seconds of has high energy consumption rate and that represents the justification of the unfamiliar mismatch between figure 4.25 and 4.27.

Based on a quantitative analysis on the API usage pattern, a categorisation the API methods is shown in table 4.13. Most of the energy consumed by the application falls in the rest of the call section. Nevertheless the energy consumption of the app is very low and almost equal to the reference value.

4.1.8 Omni Notes

The average reference platform discharge rate of the device at the time of the experiment was 106.67 µAh/s with a standard deviation of 11.01%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 7 hours and 28 minutes. At the time of loading the application average platform discharge rate is 172.20 µAh/s with a standard deviation of 17.49%. In idle foreground and background states the average platform discharge rate is 164.89 µAh/s (standard deviation 6.45%) and 109.74 µAh/s (standard deviation 11.48%). According to the measurements the application's impact on the device is 2 hours and 13 minutes which means the battery will last for 5 hours and 35 minutes. Table 4.16 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of Omni Notes application in different execution states over 20 test iterations.

Table 4.16: Average platform discharge of Omni Notes app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	106.67	11.01%	1.91	11.97%
Loading	172.20	17.49%	3.20	16.8%
Idle Foreground	164.89	6.45%	3.00	5.84%
Idle Background	109.74	11.48%	2.00	11.88%

Figure 4.29 and 4.30 are a graphical representation of table 4.16. The figure 4.29 shows that the platform discharge rate is very high when the application is loading and in idle foreground then decreases in the idle background state. The grey line in the diagram represents the reference platform discharge.

Figure 4.31 represents the energy consumption behaviour in micro Ampere hour (µAh) by the application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.The behaviour of the application during all states were similar. As we can see from the graph that the behaviour of the application is balanced but it was expected to see a down slope towards the end of the graph timeline since the platform discharge falls in idle background state.

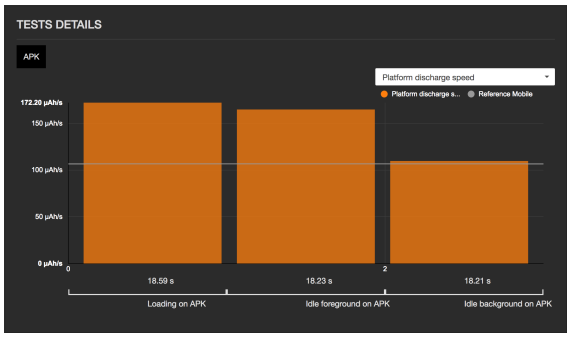

Figure 4.29: Platform discharge speed of Omni Notes app

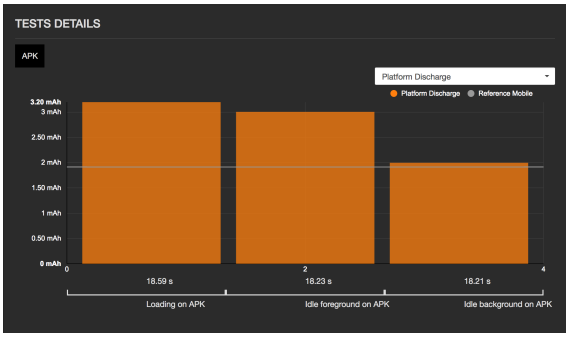

Figure 4.30: Platform discharge of Omni Notes app in different execution states

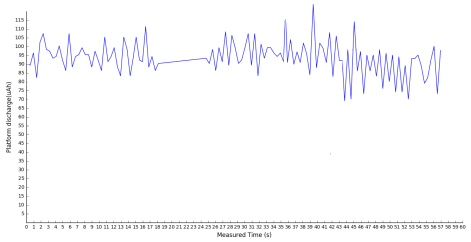

Figure 4.31: Energy consumption of Omni Notes application over time

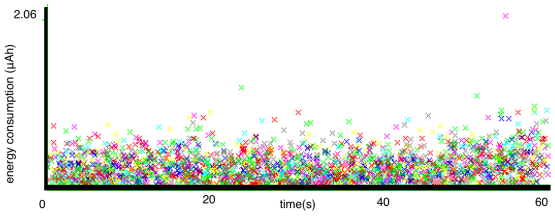

Figure 4.32: Distribution of API energy consumption of Omni Notes application over times

Furthermore, the energy consumed by the API methods called by the application over time is plotted in a graph. Figure 4.32 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method.

Table 4.17: Distribution of API methods called by Omni Notes application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	123 (5.27%)	2.075 (3.21%)
Network	8 (0.34%)	0.01 (0.015%)
Database	1 (0.042%)	0.002 (0.003%)
Display	388 (16.65%)	5.787 (8.97%)
Graphics	368 (15.79%)	4.26 (6.6%)
Hardware	5 (0.21%)	0.021 (0.033%)
Rest of the calls	1437 (61.67%)	52.34 (81.15%)

Based on a quantitative analysis on the API usage pattern, a categorisation the API methods is shown in table 4.17. OS related calls are high in number but not more than display and graphics. Display is the category to have the maximum impact. Most of the energy consumed by the application falls in this category if the rest of the calls are excluded. Our observation shows that this category only consists 16% of the total number of calls but consumes only around 8% energy which can be considered as good behaviour.

4.1.9 Easy Sound Recorder

The average reference platform discharge rate of the device at the time of the experiment was 102.22 µAh/s with a standard deviation of 08.39%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 8 hours and 09 minutes. At the time of loading the application average platform discharge rate is 147.36 µAh/s with a standard deviation of 4.78%. In idle foreground and background states the average platform discharge rate is 145.84 µAh/s (standard deviation 6.45%) and 103.42 µAh/s (standard deviation 5.56%). According to the measurements the application's impact on the device is 1 hours and 51 minutes which means the battery will last for 6 hours and 17 minutes. Table 4.18 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of Easy Sound Recorder application in different execution states over 20 test iterations.

Table 4.18: Average platform discharge of Easy Sound Recorder app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	102.22	8.39%	1.87	8.5%
Loading	147.36	4.78%	2.70	4.64%
Idle Foreground	145.84	6.45%	2.65	5.06%
Idle Background	103.42	5.56%	1.88	6.15%

Figure 4.33 and 4.34 are a graphical representation of table 4.18. Figure 4.33 shows that the platform discharge rate is very high when the application is loading and in idle foreground then decreases in the idle background state. The grey line in the diagram represents the reference platform discharge.

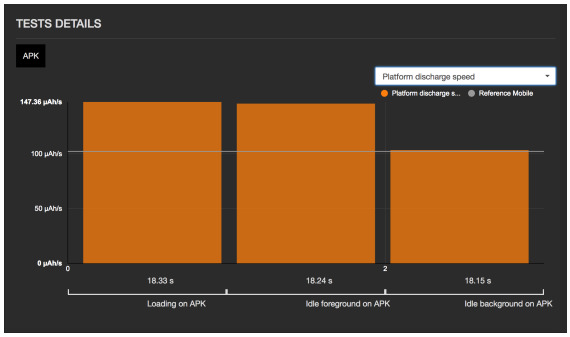

Figure 4.33: Platform discharge speed of Easy Sound Recorder Notes app

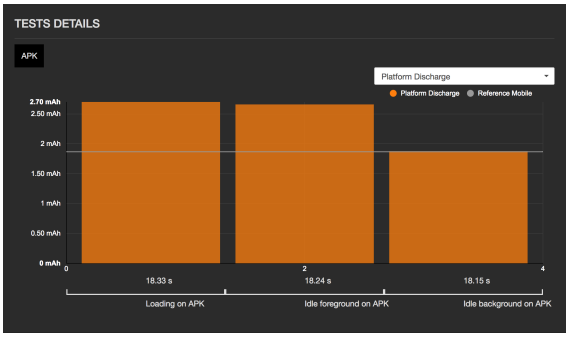

Figure 4.34: Platform discharge of Easy Sound Recorder app in different execution states

Figure 4.35 represents the energy consumption behaviour in micro Ampere hour (µAh) by the application over time in seconds (s). The time frame of 60 seconds in x-axis is a

combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.The behaviour of the application during all states were steady except for the moment when the application shifts from the loading state to the idle foreground state.

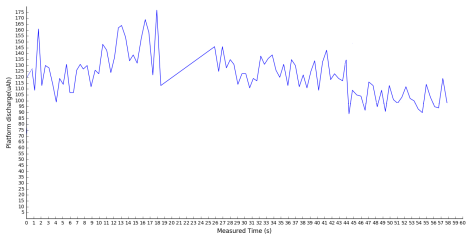

Figure 4.35: Energy consumption of Easy Sound Recorder application over time

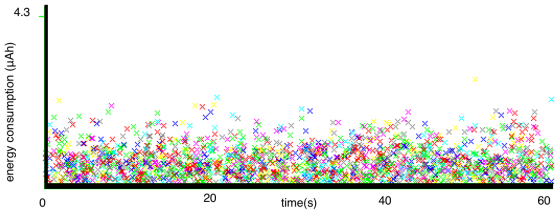

Figure 4.36: Distribution of API energy consumption of Easy Sound Recorder application over times

Furthermore, the energy consumed by the API methods called by the application over time is plotted in a graph. Figure 4.36 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method.

Table 4.19: Distribution of API methods called by Easy Sound Recorder application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	96 (5.24%)	0.62 (1.905%)
Network	2 (0.087%)	0.0046 (0.014%)
Database	9 (0.492%)	0.018 (0.055%)
Display	351 (19.3%)	2.703 (8.2%)
Graphics	272 (14.93%)	1.13 (3.44%)
Rest of the calls	1091 (59.92%)	28.61 (86.38%)

Based on a quantitative analysis on the API usage pattern, a categorisation of the API methods is shown in table 4.19. OS related calls are high in number but not more than display and graphics. Display is the category to have the maximum impact. Most of the energy consumed by the application falls in this category if we exclude the rest of the calls. Our observation shows that this category only consists 19% of the total number of calls but consumes only around 8% energy which can be considered as good behaviour.

4.1.10 NIST Connect

The average reference platform discharge rate of the device at the time of the experiment was 100.47 µAh/s with a standard deviation of 07.94%. At the time of the measurement the device had active display with maximum brightness. At this reference discharge rate the device's battery is suppose last for 8 hours and 17 minutes. At the time of loading the application average platform discharge rate is 183.90 µAh/s with a standard deviation

of 6.90%. In idle foreground and background states the average platform discharge rate is 183.40 µAh/s (standard deviation 4.56%) and 105.73 µAh/s (standard deviation 5.61%). According to the measurements the application's impact on the device is 3 hours which means the battery will last for 5 hours and 17 minutes. Table 4.20 shows the average platform discharge speed and also energy consumption (platform discharge (mAh)) of NIST Connect application in different execution states over 20 test iterations.

Table 4.20: Average platform discharge of NIST Connect app in different execution states

State of measurement	Platform discharge speed(µAh/s)	Standard Deviation	Platform Discharge (mAh)	Standard Deviation
Reference	100.47	7.94%	1.83	7.99%
Loading	183.90	6.90%	3.38	6.96%
Idle Foreground	183.40	4.56%	3.30	4.49%
Idle Background	105.73	5.61%	1.91	5.67%

Figure 4.37 and 4.38 are a graphical representation of table 4.20. Figure 4.37 shows that the platform discharge rate is very high when the application is loading and in idle foreground then decreases in the idle background state. The grey line in the diagram represents the reference platform discharge.

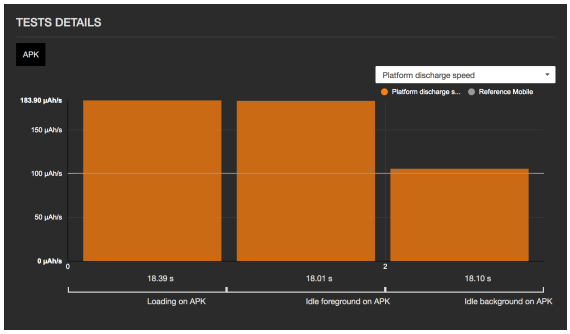

Figure 4.37: Platform discharge speed of NIST Connect Notes app

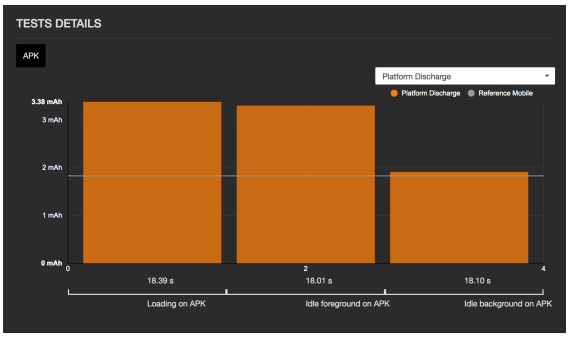

Figure 4.38: Platform discharge of NIST Connect app in different execution states

Figure 4.39 represents the energy consumption behaviour in micro Ampere hour (µAh) by the application over time in seconds (s). The time frame of 60 seconds in x-axis is a combination of three separate frames of 20 seconds each representing loading, idle foreground and idle background state of the application respectively.

Furthermore, the energy consumed by the API methods called by the application over time is plotted in a graph. Figure 4.40 represents energy consumption of all the API methods that were invoked by the application. Here, every dot represents an API method.

Based on a quantitative analysis on the API usage pattern, a categorisation of the API methods is shown in table 4.21. OS related calls are high in number but not more than

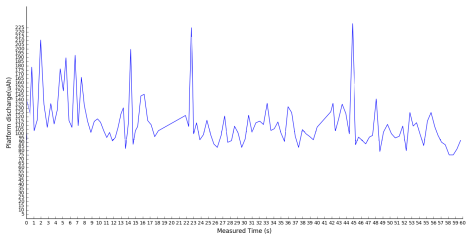

Figure 4.39: Energy consumption of NIST Connect application over time

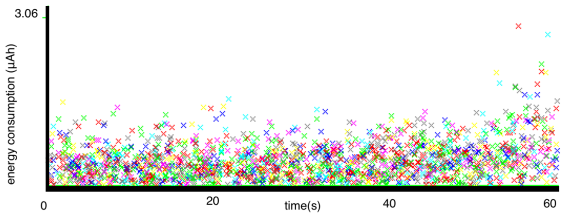

Figure 4.40: Distribution of API energy consumption of NIST Connect application over times

Table 4.21: Distribution of API methods called by NIST Connect application

Category of API method calls	Number of calls	Energy consumed (µAh)
OS	88 (5.024%)	0.516 (3.125%)
Network	1 (0.056%)	0.0005 (0.003%)
Database	5 (0.28%)	0.004 (0.025%)
Display	286 (16.3%)	2.15 (13.046%)
Graphics	334 (19.03%)	1.76 (10.64%)
Rest of the calls	1035 (59.014%)	12.077 (73.033%)

display and graphics. Display is the category to have the maximum impact despite having less number of calls than Graphics category. Most of the energy consumed by the application falls in this category if we exclude the rest of the calls. Our observation shows that this category only consists 16% of the total number of calls but consumes only around 13% energy which indicates that there can be some inefficient cals that can be removed or replaced to improve the energy consumption behaviour.

4.2 Summary

The applications benchmarked in this chapter were selected based on the popularity of the category, availability as open source and compatibility with the designed testbed. For future work purposes it was necessary to use open source applications. Otherwise closed source applications can be benchmarked using the technique described in the previous chapter.Nevertheless, a discussion of the results obtained from the experiments is elaborately discussed in the following chapter.

Chapter 5

Discussion

In the results chapter outcome of the experiment of 10 different applications were presented. Out of these applications each of them belongs to a different category. Exception was made in 'Education' and 'News & Magazines' category, where 2 applications were experimented from each category. Design of the protected testbed and extensive analysis helped to generate satisfactory results without interference of noise. In this chapter an analysis of the obtained results from the benchmarked applications are given to draw an outline of the similarities and differences. Which will give a better view of behavioural inconsistency if any. A comparison between applications belongs to the same category is also given later in this chapter. At the end of the chapter an overview of immediate afterwork has also been discussed which is related to the global scale impact of these application on the network and internet rather individual impact.

5.1 Outlining Reference Energy Measurement Statistics

A specific test scenario has been repeated 20 times on each of the 10 applications. So overall 200 iterations of the test scenario were executed, where I have measured the reference platform discharge of the device. It was not hard to notice from the results that the reference platform discharge is almost consistent.

From the table 5.2 we can see that the platform discharge of the device varies in between 91µAh/s to 106µAh/s. Similarly the reference energy consumption of the device lies in between 1.67mAh to 1.91mAh. Since the standard deviation of these measurements are less than 10% except Mapbox and Omni Notes, We can consider that the noice reduction technique applied in the testbed was successful. We have also noticed that at this energy consumption rate in standby mode with the display on having maximum brightness the devices' (Samsung Galaxy S7 Edge) battery will last for up to 9 hours. It is important to note that the device used for testing contains a brand new 3600 mAh battery. Often it is considered that it takes 10 to 15 discharge cycles for the lithium-ion battery to achieve its full power. So at the end we can say that the battery performance of our device was tested with minor noice interference considering the fact that it was unused. The performance may

Table 5.1: Reference platform discharge and energy consumption of the device

Application Name	Reference platform discharge speed(µAh/s)	Reference Energy Consumption (mAh)
Omni Notes	106.67	1.91
MapBox	106.18	1.94
Flyn News Reader	102.82	1.87
2048	102.47	1.87
Easy Sound Recorder	102.22	1.87
NIST Connect	100.47	1.83
Memory Game	97.14	1.78
CycleStreets	93.21	1.71
Wikipedia	92.96	1.70
Hubble Gallery	91.22	1.67

vary with a battery which has already passed 100 or more discharge cycles.

5.2 Comparison Between Benchmarked Applications

Our testbed was designed in such a way so that we can measure an application and provide the developer a ranking of the API calls based on their energy consumption ratio grouped by the components it is referring to. Each application has its own unique features and performance behaviour. But from the experiments we have realised that despite every application being unique somehow they all follows a similar behaviour pattern. Applications tend to consume the maximum energy when it is loaded. While aligning execution trace files with the energy measurement data we found that the reason behind the rise of the energy consumption. In most of the cases during loading than idle state the application initiates calls to API methods that are necessary to launch the application. The average percentage of API method calls during the loading period is above 50%.But it is also important to remember that we have not utilised any features of the application, since our focus was to benchmark applications using API usage pattern in idle states only. A very different API usage pattern may occur while using any feature of any specific application.

5.2.1 Analysing API Method Calls

After we aligned the API call trace with energy measurement data using the API cost calculator module it generated a list of API calls during execution of the application. From that trace we kept a track on total number of calls made by the application. A list of benchmarked application with total number of API calls is shown in figure 5.1. We calculated the cost of every API method by calculating the time duration of its execution and energy consumption of the application during that specific time frame. A sorted list of API calls based on the energy consumption of individual apps was the major finding of our study. A

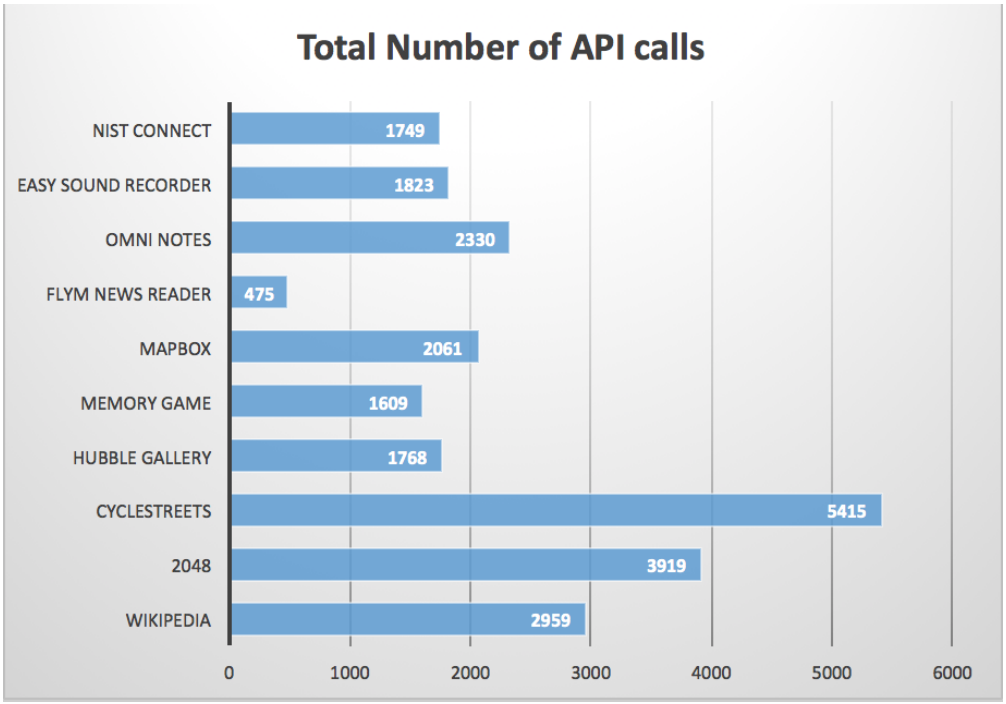

Figure 5.1: List of total number of API methods called by apps during execution

developer can look into the list and easily identify if there is any unwanted method calls. To make it simple we filtered calls related to the components of the device and summarised the total consumption of energy related to it.

One of the limitation of our experiment is the time interval between two energy measurement data. We recorded energy consumption of the device and the application every 200 millisecond but state-of-the-art energy measurement tool like monsoon meter records data every 10 millisecond. Moreover, a lot of method calls takes place between this period of time. Lots of methods are also called when few calls are already in execution phase. So there are certain chunks of time frame when multiple calls are running simultaneously and causing a rise in the energy consumption. Since the data recording interval is much longer than the frequency of method invocation there is a scope of improving accuracy. If we can reduce the data recording time interval of Greenspector Testrunner more precise cost of API calls can be obtained. All the data list of our experiment can be found in the appendix.

5.2.2 Comparison Between Applications from Same Category

Out of the 10 applications that were benchmarked, we considered 2 categories with more than one application to observe the behaviour. Our quest was to find out similarities or differences. Hubble Gallery and Memory Game belongs to Education. Flym News Reader and NIST Connect belongs to News & Magazines. A comprehensive analysis between the applications within a category is given in the following subsections.

Hubble Gallery vs Memory Game

Hubble Gallery is an educational application which contains images of space captured by Hubble's telescope. Any one who is interested can look into the details of the images and have more knowledge about our galaxy and the space. It asks for storage access permission for downloading images. On the other hand we another application from the same category called Memory Game. Which is a puzzle game for kids to test their memory skills. Since it is a game it has more detail in graphics and needs more memory processing power than Hubble Gallery.

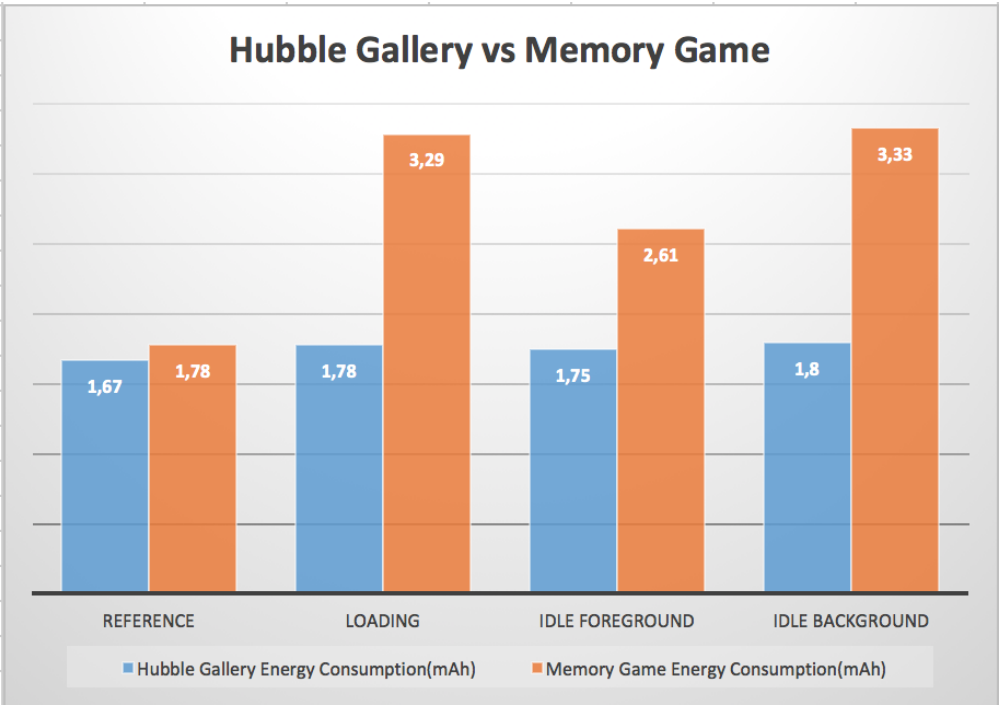

Figure 5.2: Energy Consumption Comparison Between Hubble Gallery and Memory Game

From energy measurement analysis (see figure 5.2)we say that Hubble Gallery has a very minor impact on the devices' energy consumption. It has a rise in energy consumption of 0.13 mAh from reference to idle background state. Whereas Memory game almost doubles the energy consumption of the device while loading. Energy consumption decreases in the idle foreground state but again rises in idle background state. Both applications consumes maximum energy in their idle background state. But difference between energy consumption behaviour of the applications is that Hubble Gallery has close to nothing impact but Memory Game has a significant impact of 1.55 mAh rise from reference to idle background state.

There is also a visible difference in the pattern of method calls between these applications. OS is the category with maximum number of calls for Hubble Gallery but display and graphics for Memory Game. For memory app 57% of total consumption is caused by Display and Graphics. Whereas 98% of the total consumption of Hubble Gallery application falls in the rest of the calls category.From the parallel comparison between these two apps we

can say that apps belongs to the same category can have completely different behavioural pattern.

Flym News Reader vs NIST Connect

Flym News Reader is a news application where one can select different news portal to feed recent affairs at the home page. It is a very useful application to stay up to date with global affaires. One can also check the authenticity of any news by comparing different news source using this application. On the other hand NIST Connect is a news application for the students of NIST. Where they can see recent announcements form course teacher, authority, etc. and also find their course materials.

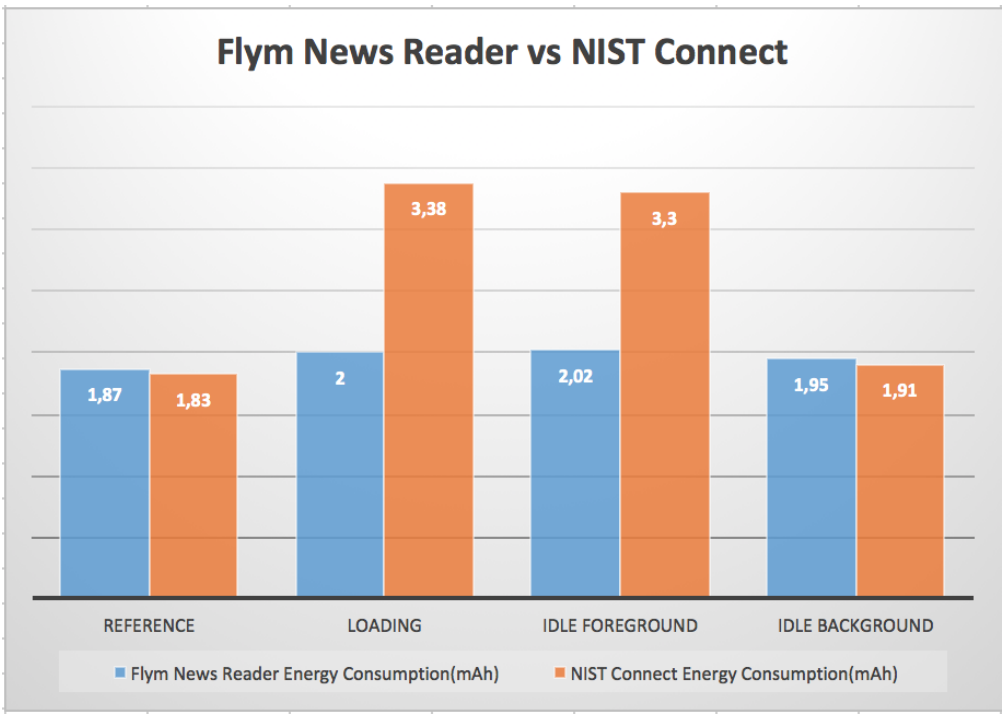

Figure 5.3: Energy Consumption Comparison Between Flym News Reader and NIST Connect

From the energy measurement analysis (see figure 5.3)we can say that Flym News Reader makes ignorable change on devices' energy consumption. Highest energy consumption by Flym News reader is only 0.15 mAh higher than reference value. On the other hand NIST connect almost doubles the reference consumption values while execution. Platform discharge of the device rises to 3.38 mAh while execution which is 1.52 mAh higher than reference value. Though there is a huge difference in loading and idle background states of the application's behaviour but they have a similar platform discharge value in idle background state.

Flym News Reader has not much activities in idle states. It makes only around 500 method calls during our 1 minute execution. Maximum energy in consumed by display

which is around 21% of total consumption. Again NIST Connect has a heavier impact on the device. It makes almost 1700 method calls during 1 minute execution. Display takes 13% of total platform discharge by the application. One interesting thing that we observed is that despite having a large difference in total number of calls both apps has a similar idle background state platform discharge.

5.3 Impact of the Apps Over Internet

The network traffic generated by an app can have a significant impact on the battery life of the device. In order to optimize that traffic it is important to both measure it and identify its source. Network requests can come directly from a user action, requests from the app code, or from a server communicating with the app. The Network Traffic tool provided by android provides the opportunity to view how and when an app transfers data over a network.

Apps use the networking hardware on a device for various reasons. In order to properly optimize an app's use of networking resources, it is obligatory to understand how frequently an app is using the network and for what reasons. For performance analysis purposes, a break down use of network hardware into these categories are considered as standard by android developer community [3] :

1. User-initiated network requests - Requests initiated by the user, such as a user request for an updated articles list in a news app.

2. App-initiated network requests - Requests initiated within Android app code that are not used to immediately satisfy a user action, such as an app request to cache the text of unread articles in a news app.

3. Server-initiated network requests - Requests initiated by a server to an app that are not used to immediately satisfy a user action, such as notification of a newly available article in a news app.

To perform network tests on the apps that are used in this thesis; these three different network usage probabilities are defined as three constants on the app's development projects (source code). Then following the manual provided in [3] a network test bed build type is constructed. The Network Traffic tool in Android Studio helps to see how an app uses network resources in real time, while it is running. To improve the repeatability of testing, it is always better to start with a known initial state for an app by clearing app data. The execution scenario developed in this thesis does not use any functional feature of the app. So the the designed testbed for network tests only analyses the usage during idle states. Moreover, the environment of the of the testbed was kept same as the energy consumption measurement testbed. A comparison of the selected apps network usage is shown in table 5.2.

The user initiated network requests column contains 0 KB because no user interaction has been made during execution of test scenario. Apps like Omni Notes, 2048, Easy Sound Recorder, Memory Game does not make any network request during the execution in idle states. Whereas, Mapbox, Flyn News Reader, NIST Connect, Cycle Streets, Wikipedia and

Table 5.2: Network usage by the apps in idle states

Application Name	User Initiated Network Requests (KB)	App Initiated Network Requests (KB)	Server Initiated Network Requests (KB)
Omni Notes	0	0	0
MapBox	0	11	443
Flyn News Reader	0	13	65
2048	0	0	0
Easy Sound Recorder	0	0	0
NIST Connect	0	33	32
Memory Game	0	0	0
CycleStreets	0	276	765
Wikipedia	0	345	112
Hubble Gallery	0	18	131

Hubble Gallery are the apps that sends packets to the internet for initialisation. Every time the app is launched some packets are transferred to the server. As a result this action also causes server initiated network requests as well, such as - loading news, showing popups, providing a suggestion. Since all the apps were tested in idle states and it was shown in the previous chapter that the energy consumption of the apps due to network components were very low, such outcomes of network traffic were expected. Nevertheless this step of network usage verification confirms the good practice in coding by the developers of the tested apps related to network.

5.4 Takeaway for Developers

Aim of this research is to provide a guideline to the developer. One can test their app in development phase to observe the energy consumption and API call behaviour. This will give a clear vision in which way they must move forward for a better optimised release version. Summarising the results after the above discussion following points the the key observations for the developers.

1. Energy consumption of the application is approximately twice as the reference energy consumption of the device.

2. The energy consumption difference between idle foreground state and loading is not more than 10%

3. Ideal energy consumption of the application in idle background state is 10% higher than the reference value.

4. API calls are mostly initiated during the loading state.

5. Display and graphics related API method calls consumes more than network related calls.

6. Cost of an API in not proportional to the number of calls.

7. Cost of an API is proportional to its execution time.

Comparing with existing works in this field this research makes noble contribution on idle state behaviour of an application. No other research extensively focuses on the API usage pattern of an application while loading and in idle states to make relation with energy consumption.

5.5 Contribution to Sustainability

This thesis focuses on identifying the energy and network usage behaviour of mobile application. Such study can help the developers to create more eco friendly application with optimised resource utilisation. Fossil fuel extraction, CO2e, abandoning coal energy, promoting green energy; when such movements are working together to shape the future world, a study for improving the battery live of the most used every day device will certainly have its marks in the domain of sustainability. A better performing app which needs less memory to perform its tasks with minimum network interaction can improve the battery life. This work certainly will help the developers to benchmark their app by using the provided test method. Though lifespan of a mobile device heavily depends on human behaviour, this study will nonetheless help to improve the device performance over the course of time.

Chapter 6

Conclusion and Future Work

Smartphones are the most widely used invention of modern times. With every new release of smartphones number of users are rising. Availability of internet and the power of smartphones opened up new research areas like IoT and other smart technologies. But this rapid growth comes at a price. More and more devices are being produced. Users are trapped by the trendsetters to upgrade to the latest devices and OS versions almost every two years. So the life span of mobile devices are much less today and this leads to a problem. Short life cycle speeds up the production cycle and both causes CO2 emission to the environment. Considering a phone remains active for about 2 years the footprint of a device is over 94kg CO2e [15]. Even though the footprint of a device is tiny almost equivalent of 6kg of beef [18] but if some one is addicted like most of us and uses it for an hour every day the add up can be as much as 1 tonne CO2e per year. Considering the number of users all over the world cumulative carbon footprint of smartphones including its usage is almost a quarter of global emission.

This research work in this thesis is a small contribution to improve CO2 footprint of these devices by improving energy consumption of applications used in it. To improve battery life it is important to follow green patterns and apply green rules while designing applications. To do so Greensoft model can be used as a reference model. But applications that already exists and running in millions of devices are causing impact on the device and also on the environment. This research was to look into the API call pattern of applications, how API calls evolve from loading to idle states and what is the total energy consumption of the application during this period of time. In search of these answers 10 applications from different domains were benchmarked. Every application has its own features and different purposes. But our target was to observe application behaviour during idle states only. So every featured activity of these applications were overlooked since none was used.

To identify API usage pattern of an application a testbed was designed with a simple execution scenario which installs the application, launches it, keeps it open on the display for 20 seconds, keeps it running in the background for 20 seconds, closes the application and finally uninstall it. Execution trace of the application and energy footprints of the device and the application were recorded during this execution scenario. Since both data were generated in the test device no clock synchronisation was needed. Then these data files were exported

to another machine for measuring energy cost of API calls. As an outcome a ranking of categorised API calls were obtained.

This research makes a noble contribution on software based application profiling at process granularity. It gives a in depth analysis to the developer how his/her application is performing in idle states, where the application is consuming most of the energy and which method calls are responsible for that. So the developer can trace it back in the source code to remove it. It that is a mandatory call then replacing the method with more efficient methods can also improve the applications energy footprint.

Future Work

This thesis measures the energy cost of API calls during execution of an application. A categorised ranking of API calls are also given based of energy consumption. But all experiments were limited to simple execution scenario from launching the application to idle foreground followed by idle background.

Aiming the same objective the future work of this thesis is to observe the behaviour of these applications with more advance execution scenario. Tests were conducted on a Samsung Galaxy S7 edge equipped with android Marshmallow (6.0.1). A future objective is to observe the behaviour of an application on different devices and android versions. A performance comparison of an application can be drown from this tests. Moreover, all experiments in this thesis were done with fully charged battery. So, it will be interesting to reiterate the experiments in different charged stage of battery and observe any changes in API energy cost behaviour. Last but not the list, in this thesis open source applications were used for the experiments. One of the reason behind was to trace the energy greedy API calls back to the source code. An immediate future work will be to implement a method that can trace back the origin of the API call in the source code. So a developer can straight away identify the rise in energy consumption and reduce it with best practices.

Appendix A

Appendix

A.1 Dataset

All datasets used in this thesis to obtain the results are available in the following link.

Link: https://1drv.ms/f/s!AlqXsUskVRSsemNfxlRYHhhtXfU.

A.2 Code

For updated version of the code and instruction on "how to execute"
visit github - https://github.com/mustaqimRahman/PERCCOM_Masters_Thesis

A.2.1 API cost calculator code

```
from sys import argv
import json
import datetime
import matplotlib.pyplot as plt
import csv
import operator

'''
dataPersing function opens the json file obtained from greenspector containing
measured data. then it retrieves thetimestamp and platform discharge(AH_PL)
values from the data and returns the following lists of values:
1.measuredTime 2.batteryStat
'''
class energyConsumption:
    def dataPersing(self, filename):
        # in open filename must contain the gsptfile as input.
        global timedata
```

```
    with open(filename) as json_data:
        data_list = json.load(json_data)

        # list for storing timestamps
        timelist = []
        #list for storing platform discharge values (AH_PL)
        batteryStat = []

        #insert timestamps into the timelist
        for data in data_list:
            for msr_data in data["measures"]:
                timelist.append(datetime.datetime.fromtimestamp
                (msr_data["time"] / 1e3))
                #insert AH_PL values into batastatlist
                if "AH_PL" in msr_data["values"]:
                    batteryStat.append(int(msr_data["values"]
                    ["AH_PL"]) / 1000)

    #declaring variable for storing time intervals
    measurementTimeIntervals = timelist[0] - timelist[0]
    #list for storing time intervals
    measuredTimes = []

    #insert values into measuredTimes list
    for index,item in enumerate(timelist):
        if index < len(timelist)-1:
            x = timelist[index+1]-timelist[index]
            measurementTimeIntervals = x + measurementTimeIntervals
            measuredTimes.append(measurementTimeIntervals)

    measuredTimes.insert(0, (measurementTimeIntervals -
    measurementTimeIntervals))
    timeDelta = str(measuredTimes[len(measuredTimes)-1]).split(':')
    timedata = int(timeDelta[1]) * 60 + float(timeDelta[2])
    meanConsumption=(sum(batteryStat)/timedata)/1e6
    return measuredTimes, batteryStat, meanConsumption

def plotData(self, x):
    time=x[0]
    value=x[1]
    timeData = []
    for data in time:
        temp =str(data).split(':')
        timeData.append(float(temp[2]))
```

```
        # Common sizes: (10, 7.5) and (12, 9)
        plt.figure(figsize=(16, 7.5))

        # Remove the plot frame lines.
        ax = plt.subplot(111)
        ax.spines["top"].set_visible(False)
        ax.spines["right"].set_visible(False)

        # Ensure that the axis ticks only show up on the bottom and left of the plot.
        ax.get_xaxis().tick_bottom()
        ax.get_yaxis().tick_left()
        ax.set_axis_bgcolor('white')
        plt.title("Platform Discharge of a Device due to an App over Time",
        fontsize=16)

        plt.plot(timeData, value)
        plt.xticks(range(0, 61, 1), [str(x) for x in range(0, 61, 1)], fontsize=10)
        plt.yticks(range(5, max(value)+5, 5), [str(x) for x in range(5,
        max(value)+5, 5)], fontsize=10)
        plt.ylim(ymin=0)
        plt.gca()
        plt.grid(True)
        plt.ylabel("Platform discharge(microAh)", fontsize=12)
        plt.xlabel("Measured Time (s)", fontsize=12)
        plt.savefig('graph.png')
        #plt.show()

class APIcost:
    def measureCost(self, tracefile, measureConsumption):
        rawfile= []
        tstart=[]
        excl=[]
        APIname=[]
        APIcost=[]
        APIcost_file = open(tracefile, 'r')
        reader = csv.reader(APIcost_file)
        for row in reader:
            rawfile.append(row)
            tstart.append(int(row[1]))
            excl.append(int(row[5]))
            APIname.append(row[7])
        for data in excl:
            APIcost.append(int(data)*measureConsumption)
        # store the API method name with its consumption
        map_APInameAndCost = zip(APIname, APIcost)
```

```
        # reduce the mapAPInameandCost list by adding the values of same
        #API method calls and make a dictionary
        reduce_APInameAndCost = dict()
        for each in map_APInameAndCost:
            key, val = each
            if key in reduce_APInameAndCost:
                reduce_APInameAndCost[key] += val
            else:
                reduce_APInameAndCost[key] = val
        # sort the reduced dictionary
        sorted_reducedList = sorted(reduce_APInameAndCost.items(),
        key=operator.itemgetter(1))
        with open('APIConssumptionMeasures.csv', 'wb') as out:
            csv_out = csv.writer(out)
            csv_out.writerow(['methodName', 'value'])
            for data in sorted_reducedList:
                csv_out.writerow(data)

script, gsptfile, tracefile = argv
gsptObj=energyConsumption()
costAPI=APIcost()
x=gsptObj.dataPersing(gsptfile)
gsptObj.plotData(x)
y=costAPI.measureCost(tracefile, x[2])
```

A.2.2 Tracefile to CSV Data Parser

```
#!/usr/bin/env ruby
@values = Array.new

STDIN.read.split("\n").each do |a|
   if a =~ /(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+([\w+.$]+)/
 id = $1
 tstart = $2
 tend  = $3
 gstart = $4
 gend = $5
 excl = $6
 incl = $7
 meth = $8
 @values << "#{id},#{tstart},#{tend},#{gstart},#{gend},#{excl},#{incl},#{meth}"
 end
end

puts "#{@values.join("\n")};"
```

Bibliography

[1] Android debug bridge. [Online]. Available: https://developer.android.com/studio/command-line/adb.html

[2] Android studio. [Online]. Available: https://developer.android.com/studio/index.html

[3] "Collecting network traffic data." [Online]. Available: https://developer.android.com/topic/performance/power/network/gather-data.html

[4] Dumpsys system diagnostics. [Online]. Available: https://source.android.com/devices/tech/debug/dumpsys.html

[5] Galaxy s7 edge and galaxy s7. [Online]. Available: http://www.samsung.com/us/explore/galaxy-s7-features-and-specs/

[6] Most popular google play categories. [Online]. Available: https://www.appbrain.com/stats/android-market-app-categories

[7] "Platform codenames, versions, api levels, and ndk releases," Android, Tech. Rep.

[8] Profiling with traceview and dmtracedump. [Online]. Available: https://developer.android.com/studio/profile/traceview.html

[9] "Greenspector," 2017. [Online]. Available: https://greenspector.com/

[10] Y. H. a. M. Z. A. Pathak, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," 2011.

[11] A.Carroll and G.Heiser, "An analysis of power consumption in a smartphone," *USENIX ATC'10*, 2010.

[12] R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. Shiraz, "A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues," *Journal of Network and Computer Applications*, 2015.

[13] V. Beal, "Api - application program interface," Webopedia, Tech. Rep., 2016.

[14] D. Beres. 10 secrets to better smartphone battery life. [Online]. Available: http://www.rd.com/advice/saving-money/secrets-to-better-smartphone-battery-life/

[15] M. Berners-Lee, "What's the carbon footprint of using a mobile phone?" The Guardian, Tech. Rep., 2010.

[16] I. Buchman, "Is lithium-ion the ideal battery," Battery University, Tech. Rep., 2017.

[17] I. Buchmann, "How to define battery life," Battery University, Tech. Rep., 2016.

[18] D. Carrington, "Giving up beef will reduce carbon footprint more than cars, says expert," The Guardian, Tech. Rep., 2014.

[19] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *USENIX annual technical conference*, 2010.

[20] Y.-F. Chung, C.-Y. Lin, and C.-T. King, "Aneprof: Energy profiling for android java virtual machine and applications," *ICPADS'11*, pp. 372–379, 2011.

[21] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update, 2016–2021," Cisco, Tech. Rep., February 2017. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.pdf

[22] F. Ding, F. Xia, W. Zhang, X. Zhao, and C. Ma, "Monitoring energy consumption of smartphones," in *IEEE 4th International Conference on Cyber, Physical and Social Computing, and Internet of Things (iThings/CPSCom)*, 2011, pp. 610–3.

[23] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems mian," in *MobiSys '11 Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011.

[24] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," *WMCSA'99*, pp. 1–9, 1999.

[25] S. R. Garner, S. J. Cunningham, G. Holmes, C. G. Nevill-Manning, and I. H. Witten, "Applying a machine learning workbench: Experience with agricultural databases."

[26] S. Gurun and C. Krintz, "A run-time, feedback-based energy estimation model for embedded devices," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis: ACM*, 2006, pp. 28–33.

[27] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," *GREENS'12*, pp. 1–7, 2012.

[28] A. M. Helmenstine, "What is the lightest metal?:metals that floats on water," ThoughtCo., Tech. Rep., 2017.

[29] G. Holmes, A. Donkin, and I. H. Witten, "Weka: a machine learning workbench," in *Intelligent Information Systems,1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, 1994.

[30] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl, "Anatomizing application performance differences on smartphones," in *Proceedings of the 8th international conference on Mobile systems, applications, and services: ACM*, 2010, pp. 165–78.

[31] I.Buchmann, "Confusion with voltages," Battery University, Tech. Rep., 2017.

[32] M. B. Kjærgaard and H. Blunck, "Unsupervised power profiling for mobile devices," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services: Springer*, 2012, pp. 138–49.

[33] A. Klimova, E. Rondeau, K. Andersson, J. Porras, A. Rybin, and A. Zaslavsky, "An international master's program in green ict as a contribution to sustainable development," *Journal of Cleaner Production*, vol. 135, pp. 223–239.

[34] D. Li, W. G. J. Halfond, R. Govindan, and S. Hao, "Estimating mobile application energy consumption using program analysis," in *IEEE 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 92–101.

[35] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[36] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," in *Proceedings of the 18th annual international conference on Mobile computing and networking - Mobicom '12*, 2012.

[37] U. Nation, "World population prospects: The 2015 revision," 2015.

[38] S. Naumann, M. Dick, E. Kern, and T. Johann, "The greensoft model: A reference model for green and sustainable software and its engineering," in *Sustainable Computing: Informatics and Systems 1*, 2011, pp. 294– 304.

[39] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof," in *EuroSys*, 2012, pp. 29–42.

[40] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Sixth Conference on Computer Systems (EuroSys'11)*, 2011, pp. 153–68.

[41] B. Poiss, "Build apk file with android studio," Cyberhouse GmbH, Tech. Rep., 2013. [Online]. Available: https://stackoverflow.com/questions/16709848/build-unsigned-apk-file-with-android-studio

[42] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," in *Proceedings of the 9th international conference on Mobile systems, applications, and services: ACM*, 2011, pp. 321–34.

[43] Telecoms. (2014, January) Android domination to continue in 2014; iphone loses ground. [Online]. Available: http://telecoms.com/210391/android-domination-to-continue-in-2014-iphone-loses-ground/

[44] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011.

[45] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics," *NSDI'13*, pp. 43–56, 2013.

[46] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2010.