

Universidade do Minho

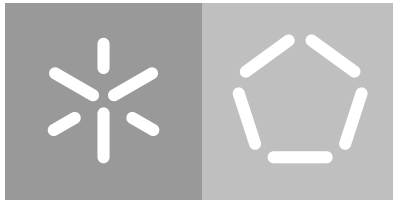
Escola de Engenharia

Departamento de Informática

Rui Antonio Ramada Rua

**GreenSource - Repository tailored
for Green Software Analysis**

July 2018



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Rui Antonio Ramada Rua

GreenSource - Repository tailored for Green Software Analysis

Master dissertation

Master Degree in Computer Science

Dissertation supervised by

João Batista Vieira Saraiva

July 2018

ACKNOWLEDGEMENTS

All stories have their end. The delivery of the master's thesis marks the end of a history of five long years. These were years of hard work and sacrifice, both for me and for so many others who accompanied me. At the end of this story, I can only show my gratitude to all those who have supported and inspired me to conclude this cycle.

First, I have to give my family credit for all the efforts and sacrifices they made, as well as all the support they gave me. Secondly, I have to thank my mentors of my dissertation, João Saraiva and Marco Couto, for all the help, support and advice they have always made available to me whenever I needed it. Without their passion for the Green Computing area, I would not have the motivation to develop this work. A warm thanks is also extended to the members of GreenLab, whose criticisms, advice and work developed together allowed me to develop several competences.

Then I have to thank those whose contagious joy always encouraged me during this journey. A special thanks to the UC's, the B. Master's and Proxys, whose account of countless "peripécias" would result in a document of greater extension than this dissertation.

Finally, special thanks to FC Porto, President Pinto da Costa, Sérgio Conceição, Marega, Alex Telles and Herrera. Their commitment, willpower and passion applied to their work represents a source of inspiration for my work.

ABSTRACT

Both energy consumption analysis and energy-aware development have gained the attention of both developers and researchers over the past years. The interest is more notorious due to the proliferation of mobile devices, where energy is a key concern.

There is a gap identified in terms of tools and information to detect and identify anomalous energy consumption in Android applications. A large part of the existing tools are based on external hardware (costly solutions in terms of setup-time), through predictive models (requiring previous hardware calibration) or static code analysis methods. We could not identify so far a tool capable of monitor all relevant system resources and components that an application uses and appoint its energy consumption, while being easily integrated with the application and/or with its development environment. Due to the lack of a tool capable of gathering all this information, a natural consequence is the lack of information about the energy consumption of applications and factors that can influence it.

This dissertation aims to carry out a study on the energy consumption of applications and mobile devices in the Android platform, having developed in this scope the GreenSource infrastructure, a repository containing the source code, representative metadata and metrics relatively to a large number of applications (and respective execution in physical devices).

In order to gather the results, an auxiliary tool has been developed to automatize the process of testing and collect the respective results for each one of the applications. This tool is a software-based solution, allowing to obtain results in terms of consumption through executions made directly on a physical device running the Android platform.

The developed framework, the AnaDroid, has the capability to perform static and dynamic analysis of an application, being able to monitor power consumption and usage of resources for each application through tests execution. This is done following a white-box testing approach, in order to test applications at source code level. It invokes calls to the TrepnLib library at strategic locations of the application code (through instrumentation techniques) to gain control over relevant portions of the source code, like methods and unit tests. In this way the programmer can have results about the use, state and consumption of resources such as energy, CPU, GPU, memory, sensor usage and complexity of developed test cases.

The information gathered through the use of the AnaDroid over a large set of applications was stored in GreenSource backend. With the collected results, we expect to be able to characterize and classify applications, as well the tests developed for it. It is intended that this will be made publicly available and serve as a reference for future works and studies.

RESUMO

Quer a análise do consumo de energia, quer o desenvolvimento de aplicações com consciência neste sentido têm vindo a cativar a atenção de desenvolvedores e investigadores nos últimos anos. O interesse é mais notório devido à proliferação de dispositivos móveis, onde a energia é uma preocupação fundamental mas ainda pouco explorada. Como tal, existem lacunas identificadas em termos de ferramentas e informações para detectar e identificar o consumo anómalo de energia em aplicações Android.

Grande parte das ferramentas existentes são baseadas em hardware externo (soluções dispendiosas em termos de tempo de setup), através de modelos preditivos (que exigem calibração prévia) ou métodos de análise estática de código. Não conseguimos identificar até ao momento uma ferramenta capaz de monitorizar de forma precisa todos os recursos e componentes relevantes do sistema usados por uma aplicação, bem como de determinar o seu consumo energético. Esta lacuna tem como consequência natural a falta de informação sobre o consumo de energia de aplicações e fatores que podem influenciá-lo.

Esta dissertação tem como objetivo realizar um estudo sobre o consumo de energia na plataforma Android, tendo sido desenvolvido neste âmbito a infraestrutura GreenSource. Esta contém um repositório que engloba o código fonte, resultados e métricas relativas a um grande número de aplicações.

A fim de obter resultados ilustrativos para um grande número de aplicações, foi desenvolvida uma ferramenta para automatizar o processo de teste e reunir os respectivos resultados. A ferramenta desenvolvida é baseada em software, permitindo obter resultados em termos de consumo através de execuções realizadas diretamente num dispositivo físico Android.

Esta *framework*, denominada AnaDroid, possui a capacidade de analisar aplicações de forma estática e dinâmica, bem como de monitorizar o consumo e uso de recursos durante a sua execução. Para este efeito, são efetuadas invocações a uma biblioteca denominada TrepnLib, em locais estratégicos do código da aplicação para obter controlo sobre partes relevantes deste. Desta forma obtêm-se resultados sobre o uso, estado e consumo de recursos, tais como consumo energético, CPU, GPU, memória, sensores.

As informações reunidas através da execução do AnaDroid foram armazenadas na base de dados do GreenSource. Com todos os resultados coletados, pretende-se caracterizar e classificar energeticamente aplicações e testes desenvolvidos para estas. Pretende-se disponibilizar abertamente estes resultados, para que possam servir como referência para futuros trabalhos, análises e estudos.

sobre porções relevantes do *source code* desta. Desta forma o programador pode ter obter resultados acerca do uso e consumo de recursos como energia, CPU, GPU, memória, uso de sensores e complexidade dos casos de testes desenvolvidos. A informação recolhida através da utilização da ferramenta sobre o repositório de aplicações permite servir para caracterizar e classificar aplicações e os testes para ela desenvolvidos. Pretende-se que esta venha a ser disponibilizada abertamente e a servir de referência para trabalhos e comparações futuras.

ACRONYMS

| | |
|------|--|
| APK | Android Package |
| ART | Android Runtime |
| API | Application Programming Interface |
| CC | Cyclomatic Complexity |
| CPU | Central Processing Unit |
| CSV | Comma-Separated Values |
| DBM | decibels (dB) with reference to one milliwatt (mW) |
| DRAM | Dynamic Random Access Memory |
| GPS | Global Positioning System |
| GPU | Graphics Processing Unit |
| HTTP | Hypertext Transfer Protoco |
| J | Joule |
| MHZ | Megahertz |
| MS | milliseconds |
| MW | milliwat |
| ORM | Object-Relational-Mapping |
| OS | Operating System |
| REST | Representational State Transfer |
| RSSI | Received Signal Strength Indicator |
| SD | Secure Digital |
| SDK | Software Development Kit |
| SFL | Spectrum-Based Fault Localization |
| SLOC | Source Lines of Code |
| SQL | Structured Query Language |

XML eXtensible Markup Language

CONTENTS

| | | |
|----------|--|-----------|
| 1 | INTRODUCTION | 1 |
| 1.1 | Motivation | 4 |
| 1.2 | Objectives | 5 |
| 1.3 | Document structure | 6 |
| 1.4 | Contributions | 7 |
| 2 | STATE OF THE ART | 8 |
| 2.1 | Software monitoring | 8 |
| 2.2 | Energy Profiling | 12 |
| 2.3 | Software metrics | 13 |
| 2.4 | Energy bugs | 15 |
| 2.5 | Energy consumption analysis in mobile devices | 17 |
| 2.5.1 | Hardware-based solutions | 18 |
| 2.5.2 | Model-Based Solutions | 19 |
| 2.5.3 | Software-based solutions | 22 |
| 2.6 | GreenDroid | 24 |
| 2.7 | Software Metrics Repositories | 26 |
| 3 | THE PROBLEM AND ITS CHALLENGES | 29 |
| 4 | POWER PROFILER - TREPn PROFILER AND TREPnLIB | 31 |
| 4.1 | Trepn Profiler | 31 |
| 4.2 | TrepnLib: the Trepn profiler as a Java API | 33 |
| 4.2.1 | Trepnlib | 33 |
| 4.2.2 | Instrumentation Types | 36 |
| 5 | ANADROID FRAMEWORK | 39 |
| 5.1 | jInst - an automatic instrumentation tool | 40 |
| 5.2 | Project building | 42 |
| 5.3 | Test execution | 42 |
| 5.4 | Results analysis | 44 |
| 6 | GREENSOURCE - A REPOSITORY TAILORED FOR GREEN SOFTWARE ANALYSIS | 48 |
| 6.1 | Data provenance | 49 |
| 6.2 | GreenSource database | 50 |
| 6.3 | GreenSource's Backend | 53 |
| 6.4 | GreenSource Workflow | 55 |

| | | |
|-----|-----------------------------|----|
| 7 | RESULTS | 56 |
| 8 | CONCLUSIONS AND FUTURE WORK | 63 |
| 8.1 | Achievements | 63 |
| 8.2 | Future Work | 64 |
| A | SUPPORT MATERIAL | 71 |
| B | DETAILS OF RESULTS | 73 |
| C | TOOLING | 78 |
| C.1 | Trepp Profiler | 78 |
| C.2 | Android Debug Bridge (ADB) | 78 |
| C.3 | Simiasque | 79 |
| C.4 | Java Parser | 79 |
| C.5 | Django framework | 80 |
| C.6 | Postgres | 80 |
| C.7 | Exerciser Monkey | 80 |

LIST OF FIGURES

| | | |
|--------------|--|----|
| Figure 1.0.1 | Worldwide Device Shipments Prediction (in millions), carried out by Gartner ¹ | 2 |
| Figure 2.1.1 | An example of a gdb ² dump | 9 |
| Figure 2.1.2 | Netbeans Profiler for Java | 10 |
| Figure 2.1.3 | Application Exerciser Monkey execution for Android | 11 |
| Figure 2.4.1 | types of energy bugs and respective frequency of occurrence[12] | 16 |
| Figure 2.5.1 | PowerScope's data collection process | 18 |
| Figure 2.5.2 | Monsoon Power Monitor | 19 |
| Figure 2.5.3 | Parameters used in the energy consumption forecast model in [37] | 20 |
| Figure 2.5.4 | State machines obtained for 3 tested devices in [38] | 21 |
| Figure 2.5.5 | screenshots of Power tutor application | 22 |
| Figure 2.5.6 | Qualcomm SoC market share in 2017 | 23 |
| Figure 2.5.7 | Treppn Profiler | 23 |
| Figure 2.5.8 | Petra[19] workflow | 24 |
| Figure 2.6.1 | JInst Workflow | 25 |
| Figure 2.7.1 | Schema of the Sourcerer database | 28 |
| Figure 4.1.1 | Power values comparison between Treppn and Monsoon | 32 |
| Figure 4.2.1 | TreppnLib usage workflow | 35 |
| Figure 5.0.1 | AnaDroid workflow | 39 |
| Figure 5.1.1 | Abstract Factory pattern implementation | 40 |
| Figure 5.2.1 | Android Project Building workflow | 42 |
| Figure 5.3.1 | Test Execution Diagram with Exerciser Monkey | 44 |
| Figure 6.0.1 | GreenSource infrastructure overview | 49 |
| Figure 6.2.1 | GreenSource database schema | 53 |
| Figure 6.3.1 | Django architecture | 54 |
| Figure 6.3.2 | An example of a Django model mapping | 54 |
| Figure 6.4.1 | GreenSource workflow | 55 |
| Figure 7.0.1 | Coverage of processed applications | 57 |
| Figure 7.0.2 | Comparative graphs for time and energy | 60 |
| Figure 7.0.3 | Comparison between PkTest and Material Library | 61 |
| Figure 7.0.4 | Evolution of consumed energy and method coverage throughout tests for Material Library | 62 |

| | | |
|--------------|---|----|
| Figure B.0.1 | Comparison between an specific app (Material Library) and the average values for the main results | 77 |
| Figure C.3.1 | Simiasque application | 79 |

LIST OF TABLES

| | | |
|-------------|--|----|
| Table 2.2.1 | results obtained after execution of the program <i>fasta</i> | 13 |
| Table 5.4.1 | Results presented in TestResults.csv file | 46 |
| Table 5.4.2 | Content of AppResults.csv, for CountryFlagsDemo app | 47 |
| Table 7.0.1 | Time spent by AnaDroid for processing the Material Library | 56 |
| Table 7.0.2 | Test Results of Android DisplayingBitmaps | 59 |
| Table 7.0.3 | App Ranking | 60 |
| Table A.0.1 | Specifications of the used Android Device | 71 |
| Table A.0.2 | Seeds used for tests execution with Application Exerciser Monkey | 72 |
| Table B.0.1 | Metrics resume of invoked methods during tests | 73 |
| Table B.0.2 | Lowest, average and highest energy consumption value obtained during app testing | 76 |

LIST OF LISTINGS

| | | |
|-----|--|----|
| 2.1 | <i>Profiling</i> of method <code>doWork()</code> using JRAPL | 12 |
| 2.2 | Java Code Snippet with diverse CC values | 15 |
| 2.3 | An example of a wakelock in Android [35] | 16 |
| 2.4 | Instrumentation of a method with Estimator calls | 25 |
| 4.1 | TrepnLib interface | 34 |
| 4.2 | Context setup in TrepnLib via Reflection | 34 |
| 4.3 | bar method instrumentation | 36 |
| 4.4 | JUnit4 test instrumentation | 37 |
| A.1 | command used for running monkey tests | 71 |

INTRODUCTION

With the advancement of the technological age, the software engineering community has been focusing on how software is developed and continually progressed in this direction. Focusing on productivity in the development process [1, 2] (in the ease and the time with which programs are developed) and in the performance [3] (memory and CPU usage, essentially) of the product to be developed, innumerable methodologies and tools have been created in this scope.

Efforts in this regard have been made at various levels, from the hardware level [4] to controllers [5], compilers, programming languages [6], middleware, frameworks [6], and IDEs. While the lower-level approaches are more focused on program performance, the higher-level approaches like model-driven [7], event-driven or IDEs [2] models are intended to increase the productivity of software development, abstracting inherent development processes (such as compilation design and deployment) and allowing developers to focus on the most essential functional aspects of their software product.

All efforts to improve software development performance and productivity are primarily intended to meet the needs of the users of the product. Both the good performance and the speed of delivery of a product are factors that can be crucial to its success. In addition to these factors, there is another aspect that has proved to be vital for computing devices, and that has had a relevant impact on the way software is developed for these devices. This factor is the portability/mobility of the devices, which led to the emergence of a new segment in the market, the segment of mobile devices.

In the last decade we have witnessed a revolution in the portability of computer systems. The factor of portability became much valued by users, making it so for the the manufactures of these. Since tablets, smartphones and laptops, the users interest in this type of device has been growing steadily, and is expected to remain at the same level in the years to come.

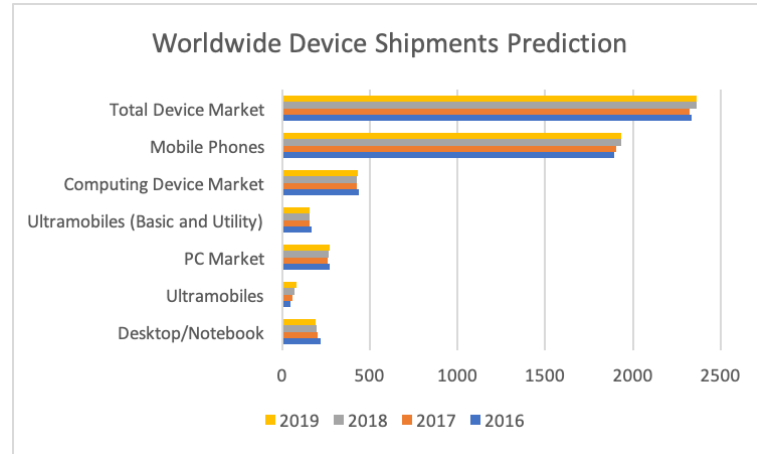


Figure 1.0.1.: Worldwide Device Shipments Prediction (in millions), carried out by Gartner¹

As figure 1 indicates, the sale of mobile computing devices represents a large part of the sales of its market, and is expected to continue in the next few years. The producers of these devices (and software developers for mobile platforms) continue to focus on increasing functionality and simplicity of use in order to dominate market segments and captivate the attention of masses for their products. In this way, with the enormous number of functionalities that a mobile device enables its users, these devices become more and more essential in their day-to-day life given the enormous number of contexts and situations in which they can be useful.

In order for a device to be considered "mobile", the main characteristics that it must possess for this must be the ease of transport and the autonomy of the device. However, in order to increase the portability of these devices, it is necessary to optimize all their components, so that they can be reduced in size without critically compromising performance.

Consequently, given the reduced size (and consequent limited capacity) of the battery of such devices, the optimization of energy consumption for these has proved to be a crucial aspect for manufacturers, as well as for developers for these platforms [8, 9]. Efforts in this direction began with focus on hardware components, developing and upgrading controllers[5], increasing battery capacity, and optimizing components performance[4]. These have been extended to virtual machines [10] and to compiling approaches (Dalvik's virtual machine switch to ART² in Android is a relevant example) until it reached software.

As far as software is concerned, to perform optimizations at this level, it is necessary to locate the critical points of source code consumption, in order to make changes and consequently improve its performance. For resources such as CPU, memory or disk, several tools have emerged, allowing to locate these points and associate them with portions of software. However, as far as energy consumption is concerned, the main platforms still lack

¹ More information about the study in: <https://www.gartner.com/newsroom/id/3754963>

² More information at: <https://source.android.com/devices/tech/dalvik/>

reliable tools that can make this association, although in recent years there are several efforts to this end[11]. However, identify critical points of energy consumption in software is not enough. It is also necessary to know what reasons lead to anomalous consumptions in this points and what changes can be made at the code level in order to improve performance. Also in this context efforts have been made to identify energy-inefficient programming patterns[12], as well as libraries[13] and other factors (such as the use of sensors, wi-fi, among others[14]). However, the information that exists in this sense is not sufficient. It is not yet significant, and cannot be considered characteristic of the Android platform.

What this thesis aims to achieve is the development of a new measurement tool (AnaDroid), capable of gathering all kinds of metrics and data related to the Android application code and its execution. Based on previous work developed in this sense, it is intended to apply some methodologies and techniques of these to the development of a tool that has the capacity to perform monitoring, static and dynamic analysis of the applications and the execution environment. In this way we intend to gather data on all the factors that we consider that may have an influence on the energy performance of an Android device and relate these to source code.

In order to be able to offer relevant information, characterizing the platform and current paradigm of application development, it was intended to gather a significant set of Android (functional) projects. By running the AnaDroid tool on its apps on multiple devices, it is intended to cluster a large number of data on them so as to achieve significant relevance.

By making this information available to the community in an open way so that it can be consulted and open to contributions, it can be re-used for further work and analysis to draw relevant conclusions about the energy performance of the Android paradigm. In this sense, another objective of this dissertation is to offer a repository of metrics and data with all these capacities.

In this way, we created the GreenSource infrastructure, a repository of metrics and data obtained from Android applications dynamic and static analysis. This infrastructure is composed of a backend, which communicates with a database composed by 21 tables, establishing relationships between application code entities and obtained data/metrics.

Until the writing of this dissertation, the AnaDroid framework was executed over 352 applications, having been successful in its execution to 61.6% (217) of these. For each of these applications, a minimum of 20 tests were performed using the Exerciser Monkey framework. These data were stored in GreenSource, which resulted in more than 6000 database table rows. We intend to continue to increase these numbers in order to grow the repository, as well as to test different devices and versions of Android, so that comparisons can be made regarding these factors.

1.1 MOTIVATION

This section clarifies the main reasons that led to this work, as well as the reasons that make it a relevant topic nowadays. We will present some of the problems identified regarding energy consumption in software and in the Android platform.

As a consequence of the increasing computational power and functionalities of these devices, a clearly evident problem arises: the consumption of battery. Given the limited power capacity of the batteries of mobile devices, with this increase the lifetime of these components naturally decreases, reducing the operability time of these devices and making them less "mobile".

The use of sensors, network, graphics and other functionalities at the software level are recognized as energy-greedy consuming tasks[14]. However, the impact of different ways of using these features at the energy level is still somewhat uncertain. At the CPU level, it is intuitive to assume that changes in the code that decrease the computation time/effort might be reflected positively in the energy consumption of this component and even to the application/program itself[15]. However, since these changes may imply greater use of other system resources (memory, network, sensors, storage accesses, etc.), changes of this kind may even have a negative impact on the energy consumption of the application.

In order to allow a programmer to observe the impact that changes in the code can have at global and local level (low level of granularity, and if possible at the test or method level), it would be necessary to have a tool capable of relate and locate energy consumption of an application source code. Beside that, this tool had to be capable of gather factors and metrics related to the execution of code that might influence its energy performance.

The reduced set of information that exists regarding the energy consumption of applications on the Android platform is dispersed by several efforts and works [13, 16?], having been obtained from the analysis of a reduced set of applications (tens or hundreds). These facts contribute to this information can not be considered significant and characteristic of the Android platform. It would be clearly essential to be able to devise a tool that would be able to analyze a large number of applications energetically, so that a relevant knowledge base about the energy consumption of applications and their platform could be gathered.

Being possible to have access to a tool of this nature, it would be advantageous if its results could be organized in an open and structured way. In this way, these could be reused for further studies and easily consulted by the scientific community.

1.2 OBJECTIVES

In this section we will present the goals we want to achieve with this work and which questions we want to give an answer to. These objectives can be summed up in 3 research questions, presented in the final of the section.

As said in the previous section, there are several factors and data that can influence the energy consumption of an application and respective code. To allow a programmer to observe the impact that changes in the code can have, one of the contributions of this dissertation is the development of a framework, the AnaDroid. This tool is purely software-based and allows to relate and locate energy consumption of an application source code, as well to obtain metrics related to these and the respective execution.

To achieve such goal, we had to find the largest set we could gather of factors and metrics related to the execution of code that might influence the energy performance that could be extracted from the static and dynamic analysis of an Android project.

The best way we found for this effect was to reuse and upgrade the concept of GreenDroid, a tool developed to locate energy consumption of an application source code. This tool has the ability to implement an Android project, build its APKs and run on a device the unit tests developed by the developer. This execution was monitored with a profiler based on models, the Power Tutor, and after this, the code invoked during the tests was related to the consumption obtained.

However, in order to get as much information about an application's code execution as possible, as well as relevant information about its coding (in addition to energy consumption), the use of the Power Tutor is not adequate. For this GreenDroid needed to be reworked to interact with another, more powerful and accurate tool, the Trepn Profiler. To achieve this goal, a generic solution was defined to integrate this or other possible profilers in the framework and consequently, in the code. Many other reformulations have been done, such as obtaining metrics through static analysis, supporting new testing methodologies, among others, so that the new framework offers more functionalities. Given the diversity of changes that were made to this tool, the final artifact also had its name changed to AnaDroid.

As a form of providing and share the results obtained, as well to prove and take advantage of the power of the tool developed, an open repository was developed. It contains hundreds of Android applications and respective results and metrics obtained with the execution of them (or portions) in a physical device. These results were obtained by the AnaDroid, since it was developed to have the capacity to process and execute several applications.

By agglomerating a high number (thousands, so far) of results, we pretend to obtain a set of information characterizing the Android development paradigm, which will allow to

relate consumption with resource use, to energetically compare different applications and devices and to obtain quality metrics of tests and software. In addition, it is hoped that the information retrieved from this repository may be (re)used in further works and researches.

In this way, the main questions to which the work developed in the scope of this thesis intends to answer are:

- **Q1:** How can GreenDroid be extended to be more expandable and powerful in its application analysis?
- **Q2:** What metrics and data can be gathered that can impact and justify the energy performance of an application (and its code)?
- **Q3:** In what way can we automate the process of running AnaDroid on a large set of applications and centralize the results in a repository?

1.3 DOCUMENT STRUCTURE

This section describes the organization of this document. All the chapters present in it begin with a brief introduction of the subject to address in its course. This dissertation is organized as follows: the current chapter starts by introducing the area of research, and then describes the motivation underlying this work, as well as the goals intended to be reached. The next chapter, Chapter 2, is destined to present the state of the art. In this chapter are described previous works, techniques and approaches used in this area of research. These were studied in order to be taken into account in the implementation of the work carried out in this dissertation.

In Chapter 3, the problem we want to solve is clarified and the biggest problems and challenges that were faced during the development phase are identified. The corresponding solutions chosen when we designed and implemented our methodologies, as well as when developing the AnaDroid framework and GreenSource infrastructure are also mentioned.

In the next chapter, Chapter 4, is the beginning point for the core of this dissertation. Here is described how the Trepn Profiler works and the approach used to introduce this profiler into applications and development environment Section 4.2.1 is where we describe the profiler itself: what it is, how can be used and its main features/limitations. In section 4.2 we start by explaining the approach followed to easily manage the profiling process, from the types of instrumentation used to the library developed for that purpose.

Chapter 5 describes the operation of AnaDroid framework, from its workflow to implementation aspects. Each phase of its operation is described in detail, from the instrumentation phase of the code to the analysis phase of the obtained results. Chapter 6 describes the GreenSource infrastructure. Every task is described, from the Android projects gathered, to test cases execution and storage and presentation of results.

In Chapter 7 is where we show the results and some conclusions we can take from them. We compare different results for different applications analyzed by our tool and present some comparisons between them, in the form of tables and graphs. Finally, in Chapter 8 we start by identifying the contributions and applications of our work, explaining how the results gathered can be reused in several ways. Then we discuss about the conclusions we came to, as well as the future work we want to do.

1.4 CONTRIBUTIONS

This section lists the main contributions in terms of tools and scientific knowledge that the work carried out within this dissertation has provided. The contributions are as follows:

- **AnaDroid:** Tool that resulted from the evolution of GreenDroid, which consisted of the starting point to carry out the energy consumption analysis of applications. Several changes were made to how GreenDroid performed the instrumentation, exercised and analyzed the code and energy consumption of applications. Its concept has been extended to be able to interact with more testing frameworks, as well as new energy profilers, such as Trepn Profiler. With the inclusion of these new tools and with changes made to its workflow and how it analyzed the application code statically, it was possible to extract more information that can be associated and justify the energy performance of the applications.
- **GreenSource and its content:** In order to demonstrate the power of the new framework developed, the AnaDroid, it was executed over hundreds of Android projects. Having access to a large number of applications and a tool with this capability of analysis, it was decided to build an infrastructure capable of storing the information that resulted from this and subsequent executions. As such, GreenSource was built, a repository containing data and metrics related to the code and performance of applications that can be related to the energy consumption of their source code. The information that this repository contains is openly available for consultation, intending to offer a relevant scientific contribution that can be significant and characterizing the Android platform, being able to be reused in later studies. In order to achieve these objectives, we intend to continue to populate this repository with information on more applications, tests and devices.

STATE OF THE ART

This chapter will describe the current state of art of the scientific field where this dissertation is inserted. We will present the techniques, methodologies and tools that are currently being used. Many of these have served as inspiration or have characteristics that were taken into account in the implementation of the work developed in the scope of this dissertation.

The first section 2.1 begins by describing, in a general way, what is being done in terms of software monitoring. During the description, several examples and motivations are presented that have made use of this practice essential for software development. The section 2.2 states the motivations and several works focused on energy profiling. The next section 2.3 focuses on software metrics, giving special focus to metrics relevant to the elaboration of this dissertation. The fourth section 2.4 aims to give an overview of coding practices that previous work has proven to be detrimental to the software's power consumption.

Finally, the last sections of this chapter focus more on the concrete work developed within this thesis. The 2.5 section describes the methodologies and tools used in mobile platforms in terms of energy monitoring. The penultimate section 2.6 describes the main functionalities, features and limitations of the GreenDroid tool, which preceded the developed analysis tool, AnaDroid. Finally, in the last section 2.7 we describe the results of later work that focused on the creation of software repositories, whose implementation and characteristics inspired our work.

2.1 SOFTWARE MONITORING

Since the beginning of software development, techniques have begun to emerge to monitor software and to help find errors and abnormal behaviors. Increasingly, society relies on software to manage their possessions, lifestyle, among other essential pillars, increasingly influencing its activity on a day-to-day basis. This makes each software product more and more refined in order to satisfy the preferences and needs of its users. These factors make program testing and monitoring progressively an essential process in software development. With the constant automation and computerization of all types of systems, there is a wide range of scenarios and contexts of use that have to be foreseen in the development phase, in

order to make the use of these adequate and functional. In addition, as a consequence of the growth of software engineering, the natural increase of competition of different products in several sectors arises. For each problem found that a software product can solve, several solutions emerge quickly, whose success is often determined by its performance and good functioning.

As such, developers use a variety of software monitoring and evaluation tools during the development phase to anticipate critical usage scenarios and see how their applications behave. The information they derive from this process may be useful for making changes in advance so as to prevent future errors and problems that may harm the end user (directly or indirectly) or other systems that depend on it, ensuring that it will function as intended. From the most rudimentary techniques (for example, the simple printing of variables and state in the course of the code) to the most robust ones that can be found today, all programmers at some point resort to debugging/problem-solving techniques during the development process.

The first robust tools that existed for this purpose consisted of command line tools for the most used high level languages at that time (in the decade of 1980, where the most used were C, PASCAL, FORTRAN and COBOL) and created by the developers of compilers, that allowed to associate variables and code instructions with their registers and memory addresses. Figure 2.1.1 shows an example of use of one of these tools.

```

R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
-----code-----
0x5555555476a <main+138>: call 0x55555554590
0x5555555476f <main+143>: mov  eax,0x1
0x55555554774 <main+148>: leave
=> 0x55555554775 <main+149>: ret
0x55555554776: nop  WORD PTR cs:[rax+rax*1+0x0]
0x55555554780 <_libc_csu_init>: push r15
0x55555554782 <_libc_csu_init+2>: push r14
0x55555554784 <_libc_csu_init+4>: mov  r15d,edi
-----stack-----
0000| 0x7fffffffed18 --> 0xa7616161616161b4
0008| 0x7fffffffed20 --> 0xa7a7a7a7a7a7a7a7
0016| 0x7fffffffed28 --> 0xb4a7a7a7a7a7a7a7
0024| 0x7fffffffed30 --> 0xa7a7a795966f
0032| 0x7fffffffed38 ("333 ", '3' <repeats 14 times>, "\247\247\247\247")
0040| 0x7fffffffed40 ("3333333333\247\247\247\247")
0048| 0x7fffffffed48 --> 0xa7a7a7a73333
0056| 0x7fffffffed50 --> 0x72747f6f7274e803
-----
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x000055555554775 in main ()
0xR14: 0x0

```

Figure 2.1.1.: An example of a gdb¹dump

This way, developers could insert breakpoints during the execution of the code, and could check the content of the variables in order to evaluate the program execution and detect errors. Monitoring and debugging techniques have been evolving continuously, providing more and more relevant information to programmers about the execution of their programs.

¹ GNU Project Debugger <https://www.gnu.org/software/gdb/>

Currently there are a number of profilers, analysis and monitoring tools that support developers, giving information on the consumption of resources consumed by the application (memory, CPU, runtime, bandwidth, among others) as well as suggestions through static² and dynamic source code analysis. These tools are often embedded (directly or through plugins) in IDEs, making the use of this type of tasks more complete and accessible, as can be seen in figure 2.1.2.

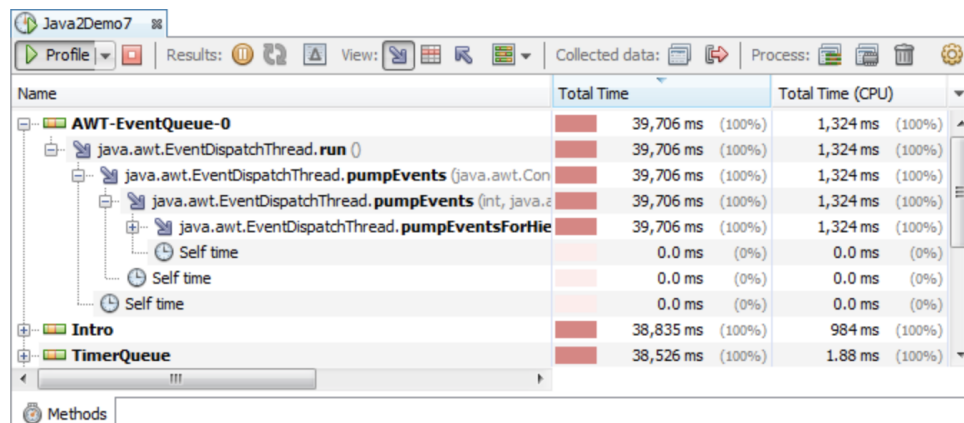


Figure 2.1.2.: Netbeans Profiler for Java

However, even given the wide variety of these solutions, the task of monitoring and preventing/correcting mistakes is still seen as painful nowadays. Brian Kernighan[17] argued that the debugging task was twice as difficult as writing a program. Also, a study³ concluded that the debugging task was the process that consumed the most time during application development. A study⁴ more recently confirms that this task is still painful for programmers. It included members of DevOps⁵ teams who followed agile methodologies, and concluded that 23% of those covered still spend more than 25% of its time to monitor and correct system bugs.

A methodology that has proved essential for discovering errors in the development phase is the elaboration of unit tests. These allow to test logical portions of software (modules, procedures/functions) in order to evaluate their operation. Conducting test suites for programs under development helps to verify the correct operation of the program as well as to identify errors and potential critical points in the program.

The continuous elaboration of tests during the development phase brings clear benefits to the final product and the programmer. It allows developers to more easily identify and control portions of code with errors and encourage them to write source code in a more modular, debuggable and testable way.

² An example is the PMD:<https://pmd.github.io>

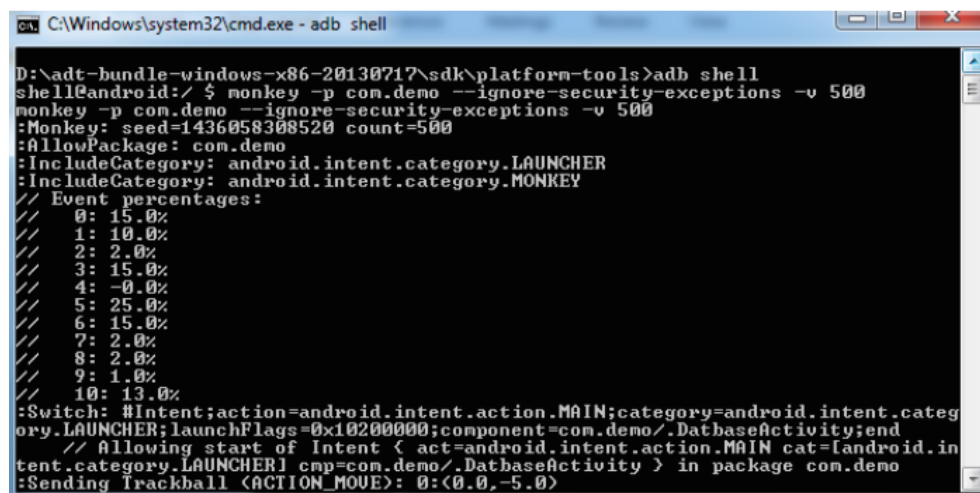
³ More information at: <https://goo.gl/AGDw95>

⁴ As described in: <https://goo.gl/5tEr4A>

⁵ Definition and more information: <http://dev2ops.org/2010/02/what-is-devops/>

For this purpose, there are several tools for a wide range of programming languages, in order to perform different types of tests. These allow a relatively quick and modular way of including and executing tests, while covering a large part of the input domain of the tested code fragments. For the Java language, an interesting tool in this context is GZoltar[18]. This tool is easily integrated into the Eclipse⁶ development environment and through SFL techniques (Spectrum-based fault localization), using heuristics to determine which program locations are most prone to fail, assists the programmer in the tests and debugging. This helps to minimize the set of developed tests and to identify points of failure of the program, assigning probability of failure to instructions, methods, classes and packages, through the results of the tests developed.

A type of test that is often used in the development process is stress testing, that tests a software part beyond the limits of normal operation, in order to evaluate the performance, robustness or availability. This type of test can be performed over all kind of parts, like (User)interfaces, web servers or databases. For this purpose, there are a wide variety of tools. Tools like the Application Exerciser Monkey ⁷ (this one for the Android platform) allow the simulation of user interface and I/O events. This tool is capable of generating (pseudo-)random inputs, simulating user and system events (such as touches, clicks, gestures) with multiple loads, to broadly cover input domains and simulate usage contexts. An example of use of this tool is shown in figure 2.1.3.



```

C:\Windows\system32\cmd.exe - adb shell
D:\adt-bundle-windows-x86-20130717\sdk\platform-tools>adb shell
shell@android:/ $ monkey -p com.demo --ignore-security-exceptions -v 500
monkey -p com.demo --ignore-security-exceptions -v 500
:Monkey: seed=1436058308520 count=500
:AllowPackage: com.demo
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
// 0: 15.0%
// 1: 10.0%
// 2: 2.0%
// 3: 15.0%
// 4: 0.0%
// 5: 25.0%
// 6: 15.0%
// 7: 2.0%
// 8: 2.0%
// 9: 1.0%
// 10: 13.0%
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x10200000;component=com.demo/.DatabaseActivity;end
// Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.demo/.DatabaseActivity } in package com.demo
:Sending Trackball (ACTION_MOVE): 0:(0.0,-5.0)

```

Figure 2.1.3.: Application Exerciser Monkey execution for Android

As unit tests help identify errors and critical points in software during the development phase, it is also possible to use techniques that (re)use tests to identify these errors and points in terms of energy. With the growing concern with the energy consumption of programs and applications, there are some tools already developed that were designed in

⁶ <https://www.eclipse.org/ide/>

⁷ Application Exerciser Monkey: <https://developer.android.com/studio/test/monkey.html>

this sense [19–21] and that through the execution of tests try to identify and locate abnormal energy consumptions in the software source code.

2.2 ENERGY PROFILING

Since the beginning of the technological age, the main concern of development teams has focused on aspects that make the use of their programs enjoyable to those who use them. Initially the focus was on aspects such as the usability, correct functioning, performance and safety of its products. In this way their productivity was increased, as well as the satisfaction of their users.

The technological advances that have occurred over the years, coupled with the increased concern with the consumption of environmental resources inherent in the operation of today's machines, mean that there are currently engineering areas focused on optimizing the efficiency and management of these resources. From data centers with a large number of machines to a simple smartphone, the reduction in energy consumption can bring great benefits to owners and users, since they make it possible to increase the period of operability, as well as the lifetime of the batteries and to reduce the ecological footprint, which consequently entails advantages in terms of monetary savings.

Energy optimizations, either on handheld devices or even on large machines, were focused from the outset at the hardware level, producing ever more efficient components and controllers. The impact that the elimination of energy bugs in the software could have on system performance was seen as of little relevance. However, as these advances are stagnating due to the approximation of the physical and scientific limits, the paradigm has been changing. It is possible to observe in recent times a growing interest in making energy-aware software optimizations.

A reference tool in the measurement of software energy consumption is RAPL[22]. This framework can be applied in Intel architectures to estimate the power consumption of the processor, GPU, L3 cache and DRAM. These estimates are accurate[23] and are performed through an energetic model that uses hardware performance counters and input/output models. There are several works and frameworks that have re-used RAPL to develop tools (such as JRAPL⁸ that allow invocations via Java source code with a reduced level of granularity(at function/method level). The following listing shows an example of code monitoring using JRALP.

```
double beginning = EnergyCheck.statCheck();
doWork();
double end = EnergyCheck.statCheck();
```

Listing 2.1: *Profiling* of method doWork() using JRAPL

⁸ JRAPL: <http://kliu20.github.io/jRAPL/>

Currently there is a field of computer science that focuses on this issue, called *Green Computing*, which studies and evaluates how to optimize the energy consumption of various computer components. Through source code static and dynamic analysis (in development and production environments) and its execution, this area evaluates the consumption impact of resources that different software components can have, from architectural components[24], libraries[12], languages[25, 26], algorithms[27], etc.

There were some assumptions inherent in being a recent and still unexplored area that have been demystified or proven with the results of recent investigations. For example, for many developers it was a given that decreasing the complexity of computational work combined with a consequent decrease in execution time would bring about proportional gains in terms of energy consumption. A recent study[25] carried out in this area compared the consumption of known programs written in 10 different languages and proved that programs with shorter execution times were often not the most energy efficient. This allows to conclude that there are other factors related to other resources internal and external to the program that directly influence the consumption of this program. An example of some of the results of this study are in table 2.2.1.

| binary-trees | | | |
|--------------|---------|-----------|-------|
| | Energy | Time | Ratio |
| C | 36.06 | 1124.67 | 0.032 |
| Fortran | 63.56 | 2112.17 | 0.030 |
| Ocaml | 84.63 | 3525.47 | 0.024 |
| Java | 96.09 | 3305.65 | 0.029 |
| Racket | 115.45 | 11260.66 | 0.010 |
| C# | 155.19 | 10797.15 | 0.014 |
| Go | 588.14 | 16291.66 | 0.036 |
| Jruby | 617.96 | 19276.14 | 0.032 |
| Lua | 1841.62 | 209217.00 | 0.009 |
| Perl | 3276.56 | 96097.28 | 0.034 |

Table 2.2.1.: results obtained after execution of the program *fasta*

This area has attempted to classify libraries and API's[12, 13] of different languages and technologies in terms of energy consumption, as well as try to identify and catalog various energy bug patterns. All to provide documentation and useful tools for developers to use good green computing practices and improve energy consumption of their programs.

2.3 SOFTWARE METRICS

With the continuous growth and evolution of computer systems and software engineering, there was a need to find manners to compare and classify programs and applications. From

complex systems (small embedded systems to large distributed systems) to small programs, a number of quantitative and qualitative measures emerged that could evaluate software and software features such as performance, security, cost of maintenance, or even computing effort. Different types of measures can be obtained from a software product, from its source code, through analysis techniques, its execution in controlled environments (through tests, for example) or through its performance as final product. This makes it clear that these measures can be useful at different stages of the software development cycle.

Software metrics can be divided essentially into two types which relate to how they are obtained:

- **Static metrics** are normally obtainable at the early phases of software development life cycle and deal with structural features of software. These are used to estimate the amount of effort needed to develop, develop and maintain the source code. To obtain such metrics, the tools with the capability to generate them usually resort to data structures and representations (like Control-Flow graphs, finite-state machines or Syntax Trees) to analyze and predict code portions behavior. Some of the most known and used metrics of this kind are Cyclomatic Complexity (CC) [28] and Lines of Code (LoC). The former was introduced by Tom McCabe in 1976 and uses a quantitative measure of the number of linearly independent paths through a program to describe its code complexity. The latter is simply the total lines of source code that a program contains, and is also used with the same purpose.
- **Dynamic metrics** are accessible at the late stage of the software development life cycle. These metrics capture the dynamic behavior of a program and/or system and are harder to obtain. These are usually obtained using monitoring and profiling tools that measure resources usage (time, CPU, disk, etc) during its execution. At a program/function/procedure level, the most commonly used are the execution time, the memory consumed and other hardware/sensors usage. At system level, there are a set of metrics that are used to express the performance, availability, and reliability[29], like the Mean Time Between Failure (MTBF) or Mean Time To Recover (MTTR).

The way these metrics are calculated differs between tools and approaches used. There are studies [30] that have tested several tools used to obtain metrics about source code that prove the differences that can be obtained in terms of values depending on the tool chosen. For example, for cyclomatic complexity (CC), its calculation is usually done by creating a Control-Flow graph (CFG) in which nodes (N) correspond to indivisible blocks of a program and the directed edges (E) connect nodes that are performed consecutively. Then the calculation of this metric (in the case of a method or subroutine) is done as follows:

$$CC = E - N + 2.$$

There are other approaches (such as ⁹ that applied a set of code heuristics in order to save the effort to obtain a CFG. The range of interpretations and annotations leads to the values of CC of 2 for the block of java code of the following snippet with the tool Eclipse Metrics Plugin ¹⁰, 4 with GMetrics ¹¹, and 5 with SonarQube ¹².

```
int foo (int a, int b) {
    if (a > 17 && b < 42 && a+b < 55)
        return 1;
    }
    return 2;
}
```

Listing 2.2: Java Code Snippet with diverse CC values

As well as exist metrics to evaluate the system and its portions of code, there are also metrics to classify the tests performed on them, so as to assign quality measures to them. The most used metric for this purpose is the Code Coverage. This is a dynamic metric, typically displayed in percentage, that reflects the amount of code that is tested during a test execution. A program with a high percentage of test coverage means that executed code has been tested more often and is therefore less likely to contain errors.

By comparing different software products and their metrics, it is sometimes possible to identify recurring practices that lead to good and poor performance and ratings by the software. These practices, typically known as software patterns, can then be properly identified and cataloged so that they can be replicated or avoided.

2.4 ENERGY BUGS

With the emergence of the green computing area, the concept of energy bug [31] emerged. These are bugs that cause reduced battery life in smartphones, and can happen at various levels, either at software and hardware level or external and unknown factors. Regarding software, through several intensive studies in this area [12], we have tried to identify and catalog software coding practices and standards that lead to excessive energy consumption by programs and applications. The most common are errors and bad programming practices, software patterns [32] and inappropriate use of APIs[12].

⁹ https://www.leepoint.net/principles_and_practices/complexity/complexity-java-method.html

¹⁰ <http://eclipse-metrics.sourceforge.net/>

¹¹ <http://gmetrics.sourceforge.net/>

¹² <https://www.sonarqube.org/>

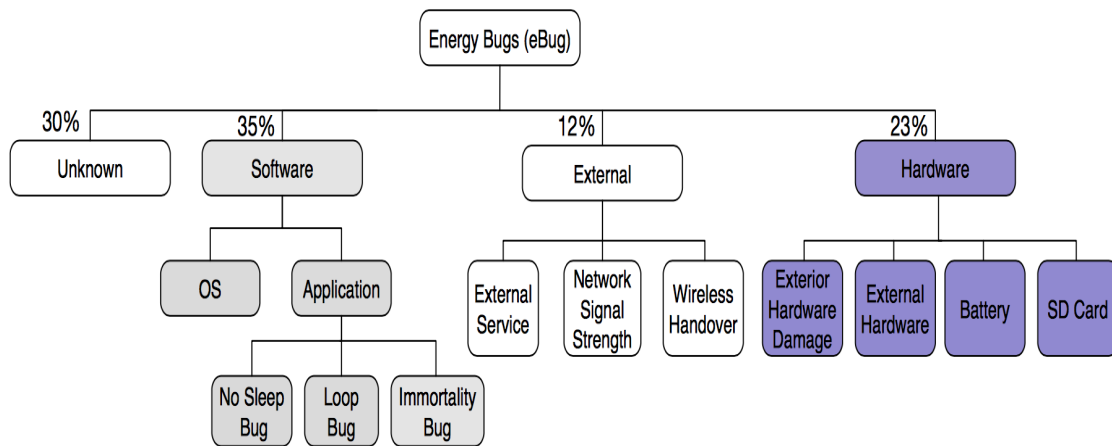


Figure 2.4.1.: types of energy bugs and respective frequency of occurrence[12]

Bad programming practices that increase the complexity of program algorithms may correspond to directly proportional energy consumption in terms of magnitude. However, there are factors that can condition this premise and compromise its veracity. A study [25] concluded that although the execution time reduction by decreasing the complexity of an algorithm implies a natural decrease of work done by the CPU, it could, for example, imply greater use of other resources (such as disk and memory), which in energy terms may even lead to higher consumption.

There are software practices and patterns that are seen as ideal for certain use contexts that produce excessive consumptions. One of the most recent studies on power consumption in Android platform has found that the use of the MVC pattern in applications with many views, as well as the principle of information hiding, encapsulation, or obfuscation[33], can significantly increase energy consumption.

Moreover, regarding energy bugs in mobile devices, it is common to find in several studies [12, 34] the reference to bad programming practices frequently encountered in source code related to wakelocks. This term defines the power management mechanisms that ensure that a mobile device does not go into a deep sleep state. The existence of this mechanism serves to ensure that applications have quick access to system resources (such as wi-fi, for example) and decrease application and system response time.

```

public void longMethod( Object o ) {
    while (someVariable != certainWakeLockValue ) {
        someVariable = PowerManager.WakeLock;
    }
    doSomething();
}
  
```

Listing 2.3: An example of a wakelock in Android [35]

2.5 ENERGY CONSUMPTION ANALYSIS IN MOBILE DEVICES

As referred to in section 2.2, the technological advances of recent years have brought additional concerns in terms of the resources consumed by its infrastructures and operability. For mobile devices, great progress has been made in the last decade. These allowed the decrease in size and consequent increase in the portability of devices and their components, including the power supply, such as their batteries. With the increasing computational power of these devices and components, there is an increasingly essential need to optimize its operation to reduce battery consumption and consequently its operability time.

For the Android platform, the most globally used on mobile devices¹³, there are a number of recent applications that make it hard-boiled for long periods of time. One of such applications is Pokemon GO, one of the most popular applications of 2016. This application has a high energy impact because it uses GPS location services intensively, produces sounds, interacts with the camera and renders 3D graphics. A recent study¹⁴ made with this application has proven that the use of this application completely drains the battery of an Android device when used continuously for 2 hours and 40 minutes.

Some of the most used applications¹⁵(Facebook, Google Maps, Instagram, among others) also consume a lot of battery. Applications with this impact tend to constantly add new features in order to improve User Experience and encourage users to continue to use the application, which implies an increased load on the system. In this sense, the Android community has tried find ways to control the costs associated with using these applications. One of the most relevant efforts came in 2014 with Android 5.0 and the *Project Volta*¹⁶. In this update the focus of the improvements introduced focused on energy consumption. A virtual machine switch was made (from Dalvik to ART¹⁷) for a more efficient one, that compiles application after installation. This leads to a decrease in the opening time of an application (as it was previously done in runtime) and improves *garbage collection*, bringing about long-term improvements in energy consumption. In addition, Android 5.0 introduced the JobScheduler¹⁸ which allows the operating system to cluster and reschedule applications tasks (such as when the device is charging or connected to the Internet, for example). It also provided an API that allowed access to system data (such as instantaneous power consumption) giving developers the ability to estimate and monitor such data through software.

However, the Android platform still lacks tools that allow developers to quickly and reliably monitor power consumption, as well locate energy hotspots in their code. The fact

13 Market share in 2017: <https://www.idc.com/promo/smartphone-market-share/os>

14 As described in: <https://dzone.com/articles/how-to-efficiently-test-your-mobile-app-for-batter>

15 Most used applications in 2017: <https://blog.sagi1.com/most-used-apps/>

16 <https://developer.android.com/about/versions/android-5.0.html#Power>

17 More information at: <https://source.android.com/devices/tech/dalvik/>

18 Reference page at: <https://developer.android.com/reference/android/app/job/JobScheduler.html>

that it is an unstable, constantly changing architectural and functional environment, both in the system and in the way it codifies and structures applications leads to the increasing challenge of creating a tool that achieves cover all these variations.

In this sense, during the investigation for this dissertation we identified several attempts made to construct tools of this type, through essentially three types of methodologies: hardware-based, model-based and software-based.

2.5.1 Hardware-based solutions

Hardware-based solutions consist of solutions that require auxiliary power measurement tools external to the target device of the application. These solutions are in theory the most costly solutions in terms of setup time and monetary cost (since it involves purchasing auxiliary devices).

One of the first of such solutions is the PowerScope [36]. It dates back to 1999 and was one of the first attempts to estimate the energy consumption of mobile computing systems. This system required two systems (the one running the application to monitor and one auxiliary) connected to a multimeter that controlled the consumption of the first system. This architecture is due to the fact that we want to reduce the overhead of the monitoring, dividing the task of accumulating and obtaining values with the auxiliary system.

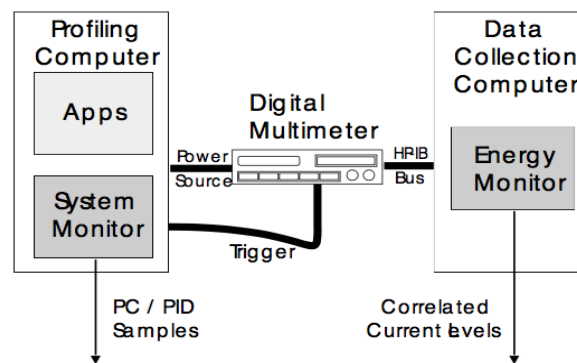


Figure 2.5.1.: PowerScope's data collection process

In addition, the task of analyzing and generating results was performed after monitoring, also for the same reason. In spite of presenting relevant results and having even made it possible to map states of hardware components to their energy consumption, the applicability of this solution in nowadays systems would be difficult, either by the necessary architecture or by the set of assumptions that it required for its operation (access to the open-source kernel and modifications at its level).

However, the most common and relevant example, which is always used as a benchmark for comparison in terms of the measurement accuracy of power consumption in scientific

publications, is the Monsoon Power Monitor¹⁹. It performs power measurement using an external instrument connected to the device's battery interface.

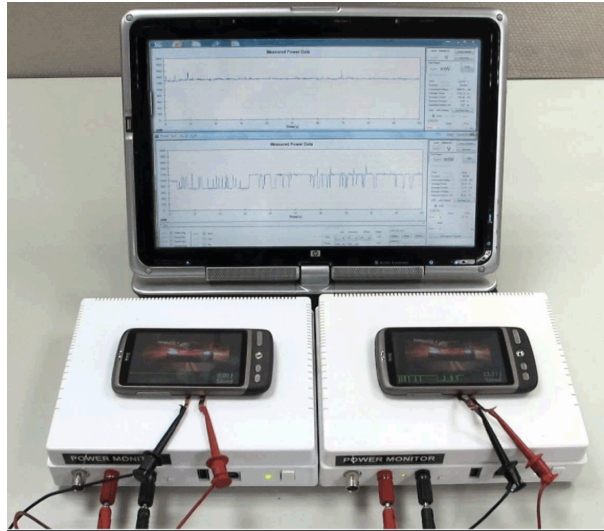


Figure 2.5.2.: Monsoon Power Monitor

2.5.2 Model-Based Solutions

These kinds of solutions have been popular and emerging since the beginning of this century. These may not depend on auxiliary external power measurement devices during the application, but are required for prior calibration and model adjustment. These are divided into essentially 3 types (although there may be solutions that expand or unify aspects of these types):

- Use-based: These models estimate the energy consumption of (sub) components by correlating the energy spent by them through their use (for example, in percentage terms). These models use mathematical models that assign weights that represent the estimated consumption of resource utilization that are used during the profiling phase.
- based on events: Makes linear associations between resources and power consumption through relevant system events (for e.g. *system call write* of 1 GB to disk).
- based on code analysis: This methodology consists of associating energy consumption with code statements. It is typically done in a static and context-independent way, which can sometimes imply inaccurate predictions.

¹⁹ Monsoon Power Monitor: <https://www.msoon.com>

Two known examples which follow this kind of methodologies are [37] and [38]. Both tried to predict the energy consumption of mobile applications through energy models, in which (estimated) consumption are mapped to states of hardware components. The first work, which carried out optimizations on the first smartphone running the Android platform, argued that the system load was defined by user usage (which is not so true today, since platforms currently run different services on *background*). The authors developed an application that acted as *logger* which periodically sent usage data to their servers. Assigning consumption values to parameters and states of the components (inferring through the logs) provide for the energy consumption. These values as exemplified in figure 2.5.3) were estimated by calibrations performed with the aid of an auxiliary energy metering apparatus.

| HW Unit | Parameter | Description | Range (of $I_{i,j}$) | Coefficient (c_j) units |
|---------|------------------|--|--------------------------|--------------------------------|
| CPU | hi_CPU_util | Average CPU utilization while operating at 384 MHz | 0-100 | 3.97 $mW/\%$ |
| | med_CPU_util | Average CPU utilization while operating at 246 MHz | 0-100 | 2.79 $mW/\%$ |
| Screen | screen_on | Fraction of the time interval with the screen on | 0-1 | 150.31 mW |
| | brightness | Screen brightness | 0-255 | 2.07 $mW/(\text{step})$ |
| Call | call_ringing | Fraction of the time interval where the phone is ringing | 0-1 | 761.70 mW |
| | call_off_hook | Fraction of time interval during a phone call | 0-1 | 389.97 mW |
| EDGE | edge_has_traffic | Fraction of time interval where there is EDGE traffic | 0-1 | 522.67 mW |
| | edge_traffic | Number of bytes transferred with the EDGE network during time interval | $\neq 0$ | 3.47 $mW/byte$ |
| Wifi | wifi_on | Fraction of time interval Wifi connection is on | 0-1 | 1.77 mW |
| | wifi_has_traffic | Fraction of time interval where there is Wifi traffic | 0-1 | 658.93 mW |
| | wifi_traffic | Count of bytes transferred with Wifi during interval | $\neq 0$ | 0.518 $mW/byte$ |
| SD Card | sdcard_traffic | Number of sectors transferred to/from Micro SD card | $\neq 0$ | 0.0324 $mW/sector$ |
| DSP | music_on | Fraction of time interval music is on | 0-1 | 275.65 mW |
| System | system_on | Fraction of time interval phone is not idle | 0-1 | 169.08 mW |

Figure 2.5.3.: Parameters used in the energy consumption forecast model in [37]

The second one is an event-based approach, also based on the use-based approach. In this work, a more precise model was created, in which they intended to cover some failures of previous models of use-based solutions. making changes at the kernel level to trace system calls. According to these, some of the failures of these models consisted of the following:

- Several hardware components have "tail" states: Components such as SD cards, network interface controllers, and GPS remain in high power states even after intensive I/O activities for some time. Given this, the use-based models do not anticipate these consumption and may have a relevant impact on their accuracy.
- Low-level optimizations in drivers/component drivers can "break" the power model. During the investigation inherent to this work, situations were identified where sys-

tem calls that did not involve the use of components (such as opening and closing files on the SD card).

- The fact that some components do not have quantitative use (camera, for example) leads to their consumption can only be associated with binary (active/ inactive) states of use. This fact coupled with the varying sampling period of use of these components leads to the achievement of relevant error margins during the profiling period.

Given these premises, the authors then developed a model based on finite state machines, which included "tail" states. However, each device requires a different state machine, which results in the dependence between the machine and the device, implying the need to study a device and consumption of its components in order to create its state machine.

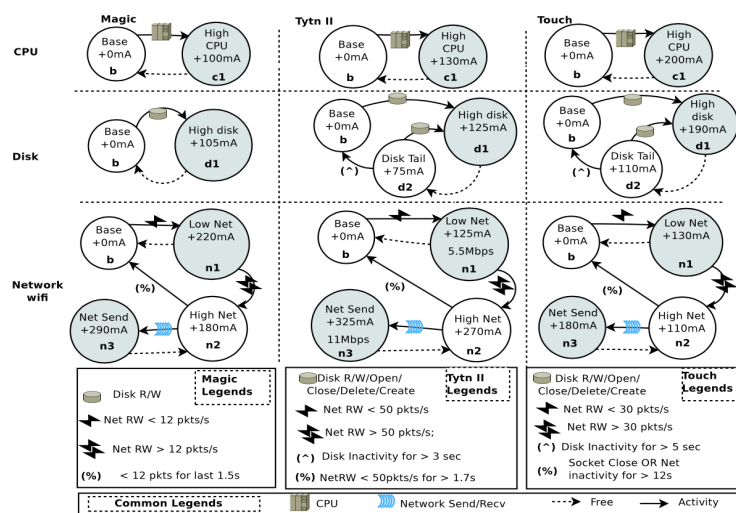


Figure 2.5.4.: State machines obtained for 3 tested devices in [38]

Another model-based tool that has high relevance in this dissertation context is the Power Tutor ²⁰. This tool is one of the most robust of this genre and is similar to those previously mentioned. It is an application for Google phones that displays the power consumed by major system components such as CPU, network interface, display, and GPS receiver and different applications. It uses a power consumption model built by direct measurements during control of device power management states. A configurable display for power consumption history is provided. It also provides users with a text-file based output containing detailed results.

²⁰ Power tutor: <https://goo.gl/sKeLb3>

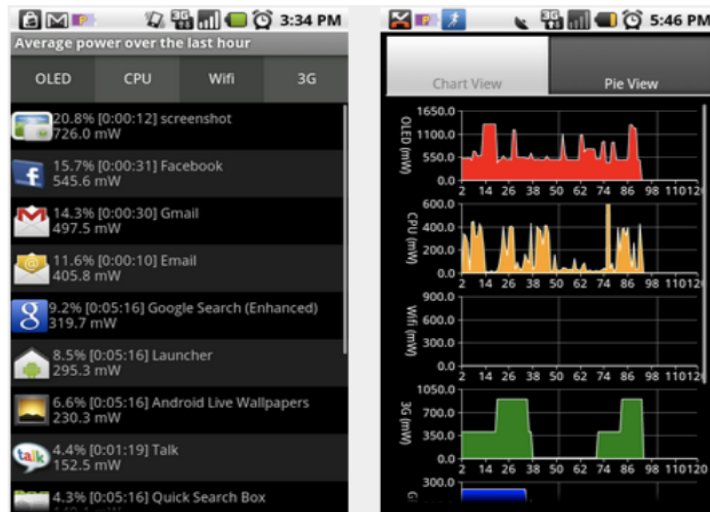


Figure 2.5.5.: screenshots of Power tutor application

2.5.3 Software-based solutions

More recent studies have allowed the development of more versatile tools that can estimate energy consumption in a more direct way. These don't require the use of models and consequently calibrations, which involves the use of external measurement equipment to the target apparatus of an application. Beside that, these are more independent of the device, having no need to model power states for each device.

One relevant example of software-based solution is Treppn Profiler²¹. It is a tool developed by Qualcomm that works on devices with Snapdragon chipset-based Android devices. Is a diagnostic tool designed for expert consumers like Android developers. It can be used to profile hardware usage (like GPS, WiFi and others), resources usage (memory, CPU) and power consumption of the system or even standalone Android applications. This tool does not need external (hardware) tools, as it gets its power readings from the power management Integrated Circuit (PMIC) and the battery fuel gauge software. The main limitation of this profiler is that only gets accurate battery power readings from chipsets developed by Qualcomm. However, this company rules the smartphone SoC (System on a Chip) market due to date²².

²¹ Treppn Profiler: <https://developer.qualcomm.com/software/treppn-power-profiler>

²² <https://goo.gl/SavDDw>

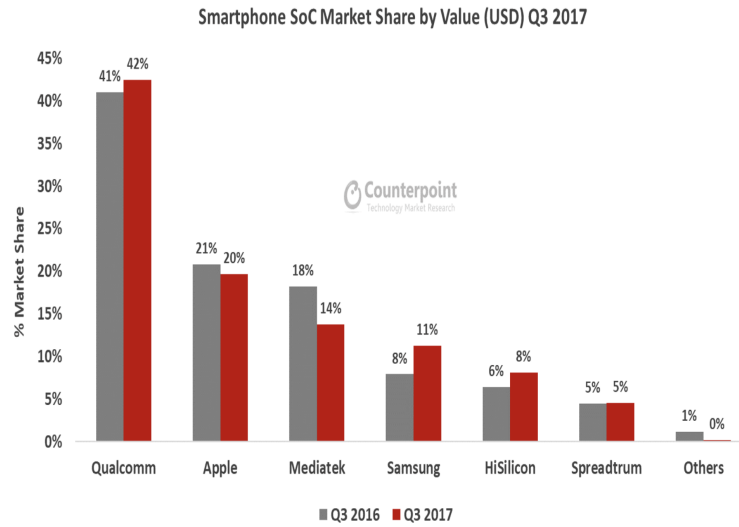


Figure 2.5.6.: Qualcomm SoC market share in 2017

Studies performed (such as [11]) with this profiler demonstrate that it obtains precise power values, virtually identical to those obtained by hardware-based solutions (such as Moonsoon). The fact that it works as an application and allows it to be invoked through code (java) and command line (abd), as well as acting as *service* (similar to a Unix daemon in Android) the versatile and easily integrable tool with applications of this platform.

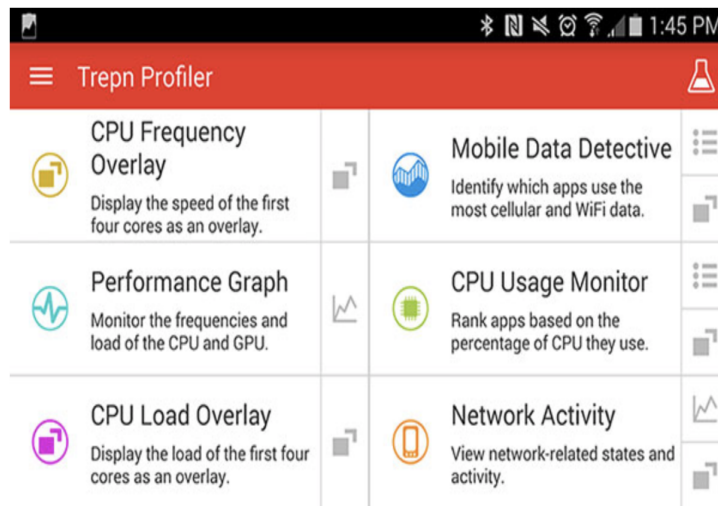


Figure 2.5.7.: Trepan Profiler

More recently, in 2017, a tool called PETRA [19] has emerged with an approach using the API available in the *Project Volta* referenced in the 2.5 section. This tool can estimate the energy consumption of the source code of an Android application with a low granularity level at the method level, giving accurate results [39]. These results have a margin of error

in the order of 2-5 % when compared to those obtained with hardware-based tools such as Monsoon.

To calculate these consumptions, PETRA uses values from the Power profile file²³ available from each manufacturer of Android devices in specific storage location of the device. This file gives approximations of how much components of the devices consume against the level of a certain utilization level (for example, how many Amperes the CPU consumes at a given frequency). Using tools like dmtracedump²⁴, Batterystats²⁵ and Systrace²⁶, it performs logging of method inputs and outputs, as well as changing states of use of components of hardware and power to be consumed by the battery.

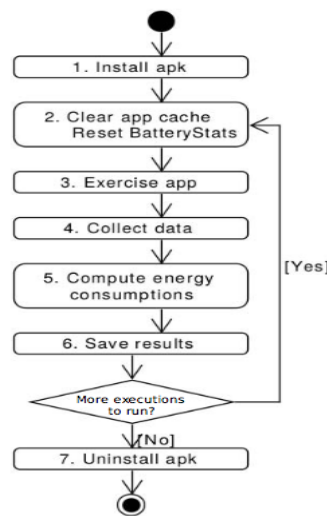


Figure 2.5.8.: Petra[19] workflow

As PETRA relies on different sources of information, so imprecisions in those sources could affect the quality of the estimations. For example, the values contained in the power profile file, provided by the device manufacturer, define just an approximation of the battery drain caused by a component in a second. Moreover, PETRA does not consider the consumption of sensors and GPU usage. For this reason, PETRA is not a reliable alternative for perform energy monitoring of apps that strongly stress this kind of hardware.

2.6 GREENDROID

This section describes the main features, workflow and limitations of the GreenDroid[20] tool. This consists of a tool capable to analyze the power consumption of Android applications and detect possible power leaks in the source code. The tool focuses on providing

²³ More information at: <https://source.android.com/devices/tech/power/values>

²⁴ <https://developer.android.com/studio/profile/traceview.html>

²⁵ <https://developer.android.com/studio/profile/battery-historian.html>

²⁶ <https://developer.android.com/studio/profile/systrace-commandline.html>

to the developers several representations of the analysis made to the energy efficiency of Android applications.

This tool performs energy profiling, interacting with the Power Tutor, the tool already referenced in the previous section 2.5.2, reusing its energy model and creating a Java API called Estimator. This API allows the communication with the Power Tutor through invocations in the application source code. Although the limitations of this tool (namely the *sampling* rate, which is 1 second), the GreenDroid tried to locate energy faults of an application and associate them with code fragments methods, classes and packages, in this case).

```
public void dummyMethod() {
    Estimator.traceMethod("dummyMethod", "Draw" , Estimator.BEGIN);
    // ... CODE ...
    Estimator.traceMethod("dummyMethod", "Draw" , Estimator.END);
}
```

Listing 2.4: Instrumentation of a method with Estimator calls

The GreenDroid wanted to offer an easy-to-integrate tool with Android applications to save developers time and effort. In this way a tool was developed that instrumented the code of the application automatically, denominated JInst. This tool uses a Java framework named Javaparser²⁷ that allows the transformation of the Java source code into an Abstract Syntax Tree [40] representation, as well as operations of crossing and manipulation of elements of this.

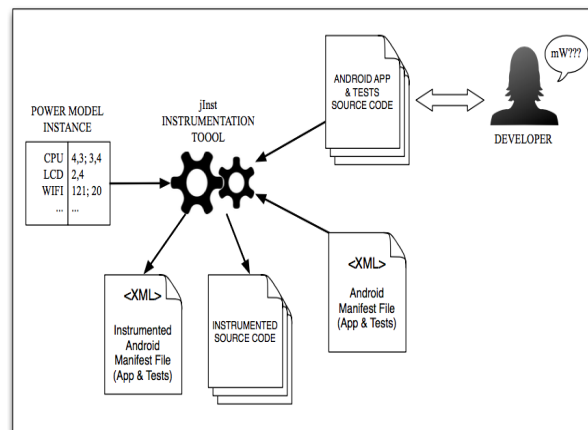


Figure 2.6.1.: JInst Workflow

In order to achieve greater accuracy of results, it was necessary to execute such fragments several times and in a manner which would then allow to agglomerate and associate their executions with the consumption recorded during the profiling phase. In order to achieve

²⁷ JavaParser: <https://github.com/javaparser/javaparser>

that goal, the platform was developed starting from the principle that the profiling process would start from the test cases developed for each Android application (assuming they were done with JUnit framework, which is one of the most used for this purpose). In order for a programmer to be able to estimate the energy consumption of fragments of his application, he should then develop tests that invoke the portions of code to monitor and instrument application methods and test cases.

At the end of the process of the GreenDroid execution process, an energetic classification of the invoked code blocks was presented. For that purpose, it was defined three possible categories for them, according to their relation with excessive consumption: Green, Yellow and Red. Each method is assigned one of these categories according to the frequency of occurrence in tests with anomalous energy consumption, where the Red classification is assigned to blocks with high frequency of occurrence in energy-greedy tests. In order to this methodology obtain reasonable accuracy, a large number of tests would have to be performed in order to reduce the classification error.

2.7 SOFTWARE METRICS REPOSITORIES

As software engineering evolved, powerful solutions were emerging whose interest in replicating and distributing was interesting. Given the complexity, the size of each of these software solutions and the large number of times it had to be replicated, infrastructures have emerged that are easily accessible and capable of hosting these solutions. The main solution found was the creation of software repositories.

Software repositories can be defined as a central place to keep resources that users can pull from when necessary. These serve the general purpose of promoting collaborative use by offering remote access to code modules and software packages. Currently, these are used in conjunction with version control systems, storing metadata for a set of files or directory structure.

The emergence of this type of infrastructures was accompanied by the open-source movement²⁸. Open-source software is based on shared information, allowing the use of technologies without the need of software license purchase, as the opposite of what is done by the proprietary system. In this way, any entity is free to examine or modify any tool available under such licenses.

Complex web-hosted open-source repository services like GitHub provide millions²⁹ of open-source and private projects, being able to provide statistics about projects, from individual files to project and management characteristics. The software solutions they

²⁸ <https://opensource.org/>

²⁹ <https://github.com/search?q=is:public>

contain are immensely diverse, ranging from the smallest program to large tools (such as Node.js ³⁰, developed by novices or large companies, from all sectors of the globe.

With easy access to software and respective source code that these complex web-based repositories offer, tools and studies emerged that had these as object of study. By analyzing the information contained in these infrastructures, it is possible to obtain characterizing results on the paradigm of software development.

A recent example of one of these works is DéjàVu [41]. This work collected 4.5 million projects (428 million files) taken from non-forked GitHub repositories, in order to determine the amount of code written in Java, C++, Python, and JavaScript that was copied or duplicated. They concluded that 70% of the code consists of clones of previously created files, where JavaScript has the highest rate of file duplication (only 6% of the files were distinct) and Java had the least duplication (60% distinct files). Lastly, a project-level analysis showed that between 9% and 31% of the projects contained at least 80% of files that can be found elsewhere. The results of this study, as well as the software artifacts developed in its context, are publicly available.

Results of studies of this nature, which involve processing and agglomeration of a large amount of information, are usually publicly available in a structured way so that it can be reused. Another relevant example of this is an infrastructure containing a repository containing Java projects and static metadata related to their code. This repository, called Sourcerer [42], contains over 70,000 Java projects, taken from several web-based open-source repositories. For the extraction of these projects, auxiliary tools were developed that automated the search and extraction process of the projects. These projects were selected from a larger set, only those whose code was considered as functional were selected. The Sourcerer also provides various representations of its content, from the source code, resources and dependencies and intermediate representations containing metrics and code metadata. It also includes a MySQL database, useful for querying the data in search for specific facts and patterns, including static analysis.

The way its database was structured focuses on the way in which Java project files and entities are structured and related. It was not designed to describe relationships between its content and entities outside of the project sources, such as devices, executions, and others.

³⁰ Node.js: <https://github.com/nodejs>

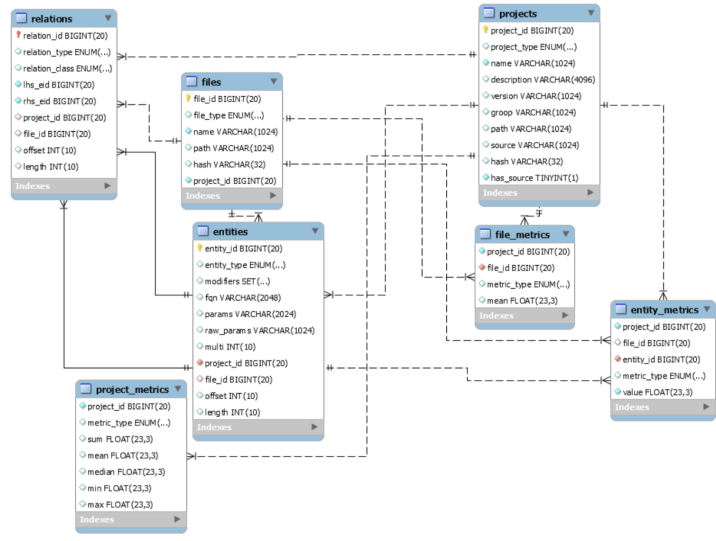


Figure 2.7.1.: Schema of the Sourcerer database

THE PROBLEM AND ITS CHALLENGES

In the work described in this thesis, we address the problem of relating power consumption with the source code of android applications, as well as to gather factors about its execution and the way in which it was developed, in order to determine the reasons that led to such consumption. We intended to develop a methodology to identify energy consumption of source code fragments, relate it to metrics, resources usage and other relative results.

In order to achieve the objective of relating the energy consumption of an application with the corresponding source code, some techniques used in GreenDroid have been reused. For this, it was necessary to understand its operation and reflect on how it could be reused. We intended to reuse GreenDroid concept and expand its capacities, so that it can be applied on a significant set of applications. To overcome this challenge, it had to be defined how it would be more natural to include a generic profiler of energy measurement, in addition to the one chosen for the AnaDroid, the Trepn Profiler. It was necessary to take into account this premise, in order to connect the tool to possible expansions to support the measurement with other energy profilers whose use may be considered pertinent and adequate.

As such, the most natural way to include a profiler in a project's development environment for the Android platform was to provide a way to include calls the profiler directly in several strategic points of the source code of this. This solution allows easy communication (for procedures like time and action logging, for example) between the application and the profiler. This was already the alternative used by the previous framework, but the way the inclusion was made had to be refined. It was necessary to define generic types of instrumentation that were considered relevant and feasible, in the sense in which its consideration may be useful for interacting with monitoring tools. This types of instrumentation were idealized given the capabilities of the tools we could use for this purpose and the limitations of these and from the Android platform itself.

To achieve the goal of trying to obtain all kinds of metrics and metadata that can be derived from the source code analysis and that may be pertinent to energy consumption analysis, the way the code was instrumented and analyzed previously was rethought.

Looking at other work and previous studies that attempted to find factors that had an impact on energy consumption, metrics were sought that could describe the complexity of

code elements that might help explain its energy performance. For studies related to this platform, it was verified that several referred to the use of certain platform APIs[12, 13] can have a negative impact on energy performance. Others tried to perceive what impact can have the usage of some hardware[14] (CPU, LCD, GPS) in power consumption. As such, the GreenDroid instrumentation tool would have to be refactored and improved to survey the API's used in the source code of its methods, so that it could also correlate its use with the proven results obtained by those studies. It also must provide other relevant data relative to the application execution, like the usage of hardware and sensors. All this changes made us led to the derivation of GreenDroid to a new tool called AnaDroid.

Finally, to offer an even more relevant scientific contribution and relevant and characteristic results on the platform's energy consumption, alternatives have been devised to centralize all the information collected about the execution of AnaDroid over several applications and devices. The chosen infrastructure, named GreenSource, had to be carefully thought out so that it could be easily filled, extended (to support more features) and consulted, so that it could be used for comparative purposes and later studies.

All the solution we came to, as long as the different ones we tried, are explained in the following chapters.

POWER PROFILER - TREP N PROFILER AND TREP NLIB

In this chapter, we discuss the approaches applied for monitoring the energy consumption of Android applications. All profilers that can be used for this purpose have several limitations that can not be ignored. The choice fell on the Trepn Profiler, due to its power and high number of features.

We start by discussing in Section 4.1 the manner how this profiler works. This is a free software-based profiler developed by Qualcomm. Later, in Section 4.2, we describe the Android Library developed in the context of this dissertation, that eases the process of interacting with Trepn and the instrumentation approaches taken into account in its design.

4.1 TREP N PROFILER

As explained in section 2.5.3, Trepn is a software-based solution for power profiling. This profiler works on Snapdragon chipset-based devices and is a powerful tool capable of access device hardware/sensors state and usage during the profiling timeline.

Trepn can be used as a normal application, or be used as a service¹ (a unix-like daemon in Android). When used as a service, it is possible to interact with it through invocations via Java source code or from the adb² command-line tool. This versatility makes this profiler easy to integrate in Android-based tools and applications, for the purpose of measuring and profiling an entire application or portions of it. It provides the capability of change application states while monitoring, which can be used to log and mark specific events during the profiling timeline.

The profiler also offers visual information while or after profiling, when used as a standalone application. It provides an overlay view of different charts so that the users can associate the performance of the profiler object with the resource usage and energy consumption. After other profiling phase, it is possible to export the profiled real-time data to a .csv format, making it easily to parse. The resulting file contains samples reflecting the hardware state and resources usage at 100 ms sample rate. Trepn is known as one of the

¹ Android Service: <https://developer.android.com/guide/components/services>

² <https://developer.android.com/studio/command-line/adb>

best tools[11] to measure power and energy consumption in Android ecosystem. There are scientific artifacts that proved that this profiler gives accurate power measures, comparable to those obtained by other reference tools like the Monsoon Power Monitor.

| Brightness | Monsoon | Trepn Profiler | Difference |
|------------|---------|----------------|------------|
| 20% | 1247 mW | 1245 mW | -0.2% |
| 30% | 1340 mW | 1321 mW | -1.4% |
| 40% | 1490 mW | 1408 mW | -5.5% |
| 50% | 1550 mW | 1517 mW | -2.1% |
| 60% | 1643 mW | 1627 mW | -1.0% |
| 70% | 1773 mW | 1729 mW | -2.5% |
| 80% | 1857 mW | 1838 mW | -1.0% |
| 90% | 2002 mW | 1951 mW | -2.5% |
| 100% | 2148 mW | 2088 mW | -2.8% |

Source: Qualcomm internal power measurements of a Nexus 6 running Trepn software at various screen brightness levels

Figure 4.1.1.: Power values comparison between Trepn and Monsoon

The device hardware/resources usage and statistics provided by Trepn Profiler that can be exported after the profiling phase and available to most of devices are the following:

- **Battery power:** Power the battery consumes, measured in mW.
- **Battery status:** Battery charging status. The different status are: 0-Not charging, 1-Charging(USB), 2-Charging(AC), 3-Charging(generic source).
- **CPU Load:** Load across all cores.
- **CPU Load Normalized:**Normalized CPU load across all cores.
- **CPU Frequency:** Clock speed for each CPU core.
- **GPU Load:** Usage for the 3D GPU core.
- **Bluetooth® State:** State of the Bluetooth radio. The different states are: 0-Off, 1-Turning On, 2-On,3-Turning Off.
- **Mobile Data State:** State of the cellular radio. The different states are: 0-Disconnected, 1-Connecting, 2-Connected (dormant), 3-Connected(no traffic), 4-Connected (sending traffic), 5-Connected (receiving traffic), 6-Connected (sending and receiving traffic).
- **Wi-Fi RSSI Level:** RSSI level for the Wi-Fi connection; values are expressed as dBm.

- **Wi-Fi State:** State of the Wi-Fi connection. The different states are: 0–Wi-Fi currently being disabled, 1–Wi-Fi disabled, 2–Wi-Fi currently being enabled, 3–Wi-Fi enabled, 4–Wi-Fi in unknown state; occurs during enabling or disabling errors.
- **GPS State:** Current state of the GPS system. The different states are: 0–GPS stopped, 1–GPS unknown state, 2–GPS running.
- **Memory usage:** Total system memory usage.
- **Percent battery remaining:** Remaining charge in the battery.
- **Screen brightness:** Brightness level of the screen’s backlight.
- **Screen State** – State of the screen (0–Off,1–On).
- **Application State:** Data point for developers to instrument their application/code, accompanied by the respective description (optional), allowing a more precise understanding of data points; value is a signed 32-bit integer.

However, not all of these metrics / statistics provided by Trepn Profiler are available for all devices. The ability to provide them varies between versions of chipset and Android platform.

4.2 TREPNLIB: THE TREPn PROFILER AS A JAVA API

Given the possibility of invoking Trepn via Java source code, we designed a solution to easily integrate the management of the profiling process in the applications. The best approach that was found to do so was to instrument the source code of the applications, with calls to functions abstracting the operation of the process of monitoring and communication with the Trepn service. For this purpose, we developed an Android Library, which design and functioning will be described in 4.2.1, providing an API that allows to isolate and profile code blocks (like methods, loops) of any Java class present in the application source code. The following section describes the types of instrumentation supported by the TrepnLib, and consequently by the Anadroid framework.

4.2.1 *Trepnlib*

The TrepnLib library makes it easier to manage the profiling process and the API provided makes it possible to integrate the tool into the source code and the development environment of the application. This was developed taking into account the use cases that were intended for its application, providing two different monitoring types.

By instrumenting the source code with the API provided by the TrepnLib, it is possible to determinate/estimate the power consumption and profile the isolated portion, as well as log other relevant events, like the start/end of code blocks, identify recursive calls or manage and distinguish data from different runs. To provide all this capabilities, TrepnLib was designed the taking into account 2 instrumentation types: Test-Oriented and Method-Oriented.

```
public interface TrepnProfileLibrary {

    void loadPreferences (Context ctx, String preferencesFilename);

    /// Method Oriented Profiling

    void startProfiling(Context ctx);

    void updateState(Context ctx, int state, String description);

    void stopProfiling(Context ctx);

    /// Test Oriented Profiling

    void startProfilingTest(Context ctx);

    void stopProfilingTest(Context ctx);

    void traceMethod(String methodName);

}
```

Listing 4.1: TrepnLib interface

One of the main features of the TrepnLib is the capability to call Trepn Profiler in a normal Java class that don't inherit the application Context from the Android SDK classes, that have access do this interface. In order to communicate to the Trepn service, it is necessary to send an Intent to the Trepn BroadCastReceiver. To achieve this task, it is required to have access to the core Messaging system running on Android, and inside an application, access to the Context of the application is required. The Context class gives an interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes and up-calls for application-level operations such as launching activities, receiving intents, etc. The TrepnLib provides a way to set the Context (passing it by argument) if can be accessible in a Java class, or gets it if need, through Java Reflection.

```
private static Application getApplicationUsingReflection() throws Exception {
    return (Application) Class.forName("android.app.ActivityThread")
        .getMethod("currentApplication").invoke(null, (Object[]) null);
}
```


}

Listing 4.2: Context setup in TrepnLib via Reflection

Another mechanism included was the ability of TrepnLib to work in Silent Mode (not interacting with Trepn service to perform profiling). In this way, the library is only used for tracing the invocation of application methods. It is also possible to perform energy measurement without performing the tracing operation (Measure mode). In this way it is possible to separate the execution of the application (or tests) in 2 different processes (with and without the tracing operation), in order to reduce the overhead added by the combined use of both options. This way the Hawthorne Effect[39] can be minimized, in which the measurements are affected by the measurement process. This mechanism is controlled by the developer, that only has to set an integer in a configuration file called GDFlag to choose the respective mode. This is only available for test-oriented instrumentation of an application, where the methods and respective consumption can be associated through non-direct approaches (as for example, as with GreenDroid, where the methods were classified energetically according to its frequency of occurrence in tests with anomalous consumption).

Furthermore, we provided functions to start and stop the profiling process, given the type of monitoring (method or test-oriented), that interacts with the Trepn Service, as well as creates auxiliary files that are used to manage several runs, states and contexts. Methods to trace usage of methods and log states/events (like the beginning or end of methods) are also provided.

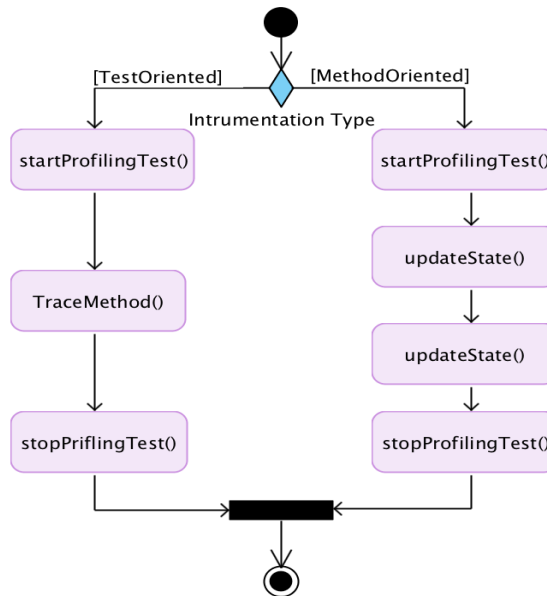


Figure 4.2.1.: TrepnLib usage workflow

4.2.2 Instrumentation Types

In order to instrument the source code of an Android application to monitor the energy consumption of portions of the application, it was necessary to decide at what level of granularity could be done, and how it could be controlled and monitored. For all the conceived alternatives, it were discussed and analyzed the pros and cons, as well as the possibility of being implemented. These alternatives are limited either by the architecture of the Android platform or by the sampling rate of the reliable tools^[43] available for this purpose. Each instrumentation type is determined by the type of monitoring which is intended to perform.

In order to perform software monitoring, it is necessary to define an interval in which this process occurs. In terms of source code monitoring, in the Java language, the level of granularity that can be considered goes from the execution of the simplest operation (a sum or assignment, for example) to the most complex program or application. The possibility of monitoring at the lowest level, or even at the instruction or at line of code level was provisionally set aside. The execution time must be large enough to significantly exceed the measurement granularity, in order to obtain accurate measures. Since the modern processors can execute thousands of MIPS (Millions of Instructions Per Second), the capability of obtain power consumption of Java code at that level is not reliable using Trepn Profiler (which sample rate is 100 ms) or any other available profiler. It was to be possible to delimit the execution of simple instructions (like `a=b;`), as well obtain samples of the battery drained during the execution. As such, higher levels of granularity had to be considered.

At the method level, for methods whose computation complexity is relevant, it is possible to obtain more accurate values/estimates of the power consumed due to its execution. As such, it was considered in the framework the possibility of performing instrumentation at this level, through the following form: Through instrumentation of the beginning and end of the method (i.e. before any point of return/exit of the method), in order to delimit the interval implementation. This type of instrumentation is controlled by the TrepnLib Android library, which controls the nesting level of the method, allowing to distinguish recursive calls and consecutive/mutual calls between methods of the application classes. At the end of the test run, it is only necessary to associate the run interval of the tests with the usage samples and state of the device resources, made available by the device and withdrawn during this interval.

```
package com.example.test;
public class Foo{
    public void bar(){
        TrepnLib.updateState(context, 1, "com.example.test.Foo<bar>");
        .....
        ... method body ...
    }
}
```

```

.....
TrepnLib.updateState(context, 0, "com.example.test.Foo<bar>");
return;
}
}

```

Listing 4.3: bar method instrumentation

Another type of instrumentation supported so far by Anadroid is test-oriented instrumentation. This type of instrumentation delimits the interval of time in which a test occurs, being able to meet through external mechanisms (explicit sending of states change to the Trepn service) or directly in the code, as happens, for example, for unit tests developed using the JUnit framework. For this case, we can define methods that are invoked at the beginning and end of each test (methods annotated with `@After` and `@Before` after JUnit4 or methods named `tearDown()` and `setUp()` in earlier versions), which are created or instrumented with calls to the library in order to delimit the execution time of the tests.

```

@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    @Before
    public void setUp(){
        startProfilingTest(InstrumentationRegistry.getTargetContext());
    }
    @After
    public void tearDown(){
        stopProfilingTest(InstrumentationRegistry.getTargetContext());
    }
    @Test
    public void simpleTest() throws Exception {
        Context ctx = InstrumentationRegistry.getTargetContext();
        assertEquals("com.example.testapp", ctx.getPackageName());
    }
}

```

Listing 4.4: JUnit4 test instrumentation

Through the use of this mechanism, together with the instrumentation of the beginning of the methods (only to record the use of the method and the respective execution time), they can then associate the methods with sampling of use and state of resources of the device, in order to to estimate their consumption by the methods used to perform the tests. Furthermore, a developer can include the profiling phase in the testing phase of his development phase, and choose what portions of the application to test and monitor.

Once the types of instrumentation have been defined, alternatives have been devised to make it possible in the future to easily include other profilers in the Anadroid whose

manipulation can be accomplished via code instrumentation. The solution achieved will be described in section 5.1, where the operation of the Anadroid automatic instrumentation tool is described.

ANADROID FRAMEWORK

As explained before, our work is mainly focused on monitoring Android applications' resource consumption (such as energy and time), relate it to source code metrics and with the obtained results. After gathering a relevant amount of information, we intended to build a large dataset of information referring to several usage scenarios of a considerable amount of applications. In order to automate this process, we developed the Anadroid framework. This tool comes as an evolution of the GreenDroid framework[20], making it more accurate and complete.

In the following sections it will be described each one of the stages of Anadroid workflow. In resume, what this tool does is: takes an Android project, creates an instrumented copy of this, with instrumented calls to the TrepnLib, builds and tests the application in a connected physical device. Then collects the running data and generates the results from test execution, sending them to the GreenSource's backend.

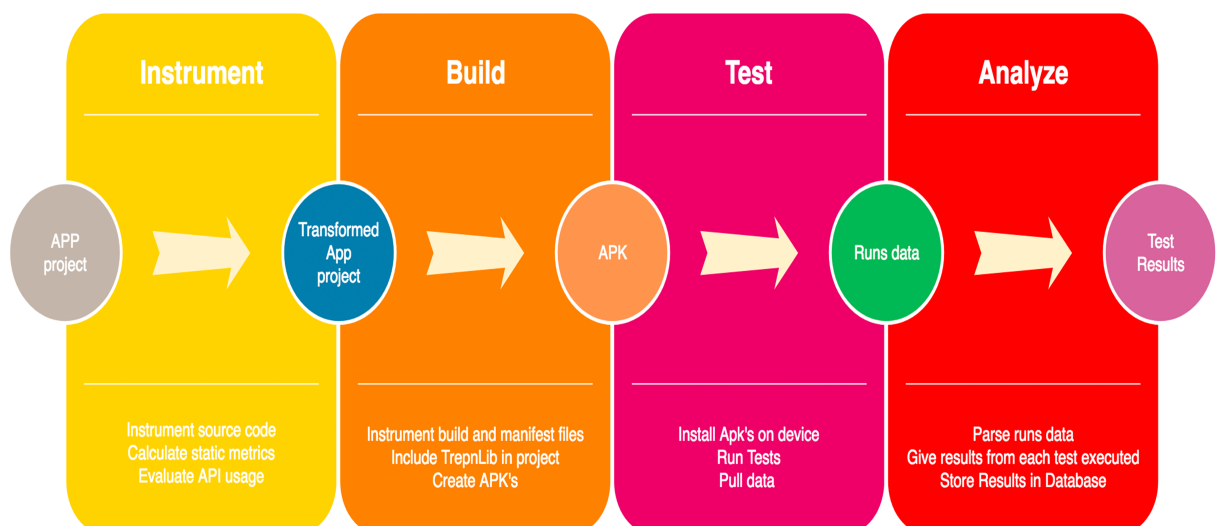


Figure 5.0.1.: AnaDroid workflow

5.1 JINST - AN AUTOMATIC INSTRUMENTATION TOOL

The first step of the AnaDroid workflow is the instrumentation of the Android project. This step is necessary to introduce the calls to the Profiler in the project code and include the library in it, so that power monitoring can be performed. For this purpose, we reused the jInst tool that was developed for the GreenDroid tool. This tool is capable of instrumenting the application’s source code and additional configuration files. In addition, it also collects static metrics and metadata about application methods and classes.

Taking in account the goal of extend the GreenDroid framework to integrate its operation with a new energy profiler, in a more modular and generic way, we applied the Abstract Factory software architectural pattern¹. In this way, in order to include a new profiler in the future, able to monitor tests and/or methods, a new Factory Class must be created, in order to instantiate a representation of the tool. Then the behavior for each profiling type must be defined, by implementing the interfaces defined by Profiler, MethodOrientedProfiler and TestOrientedProfiler classes.

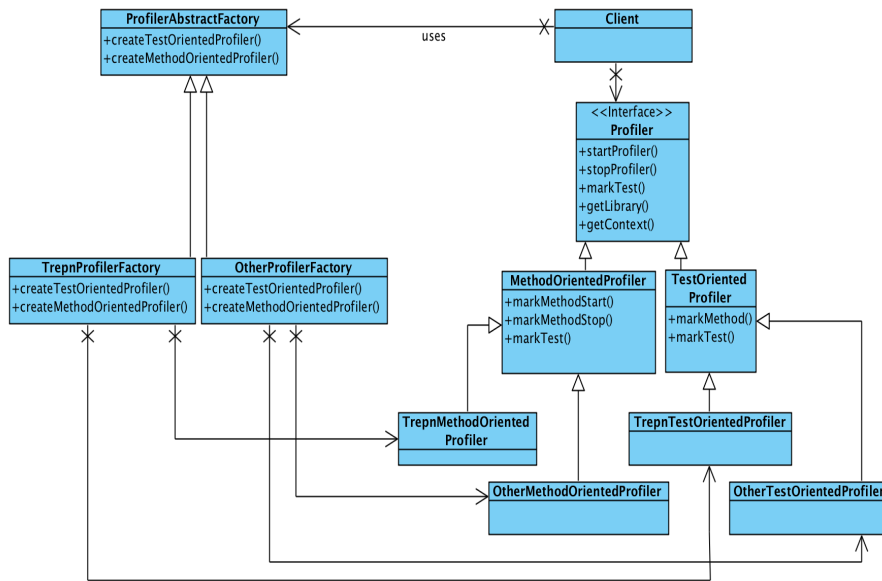


Figure 5.1.1.: Abstract Factory pattern implementation

The jInst tool starts by creating a parallel project to the original so that it does not run the risk of corrupting its files and negatively interfering with the development process. This approach is common in regression testing² techniques. In order to instrument the source code, each Java Class is mapped to an AST representation. This operation is possible with the help of the JavaParser³ library, which allows to obtain an AST of each one of the

1 <https://www.oodesign.com/abstract-factory-pattern.html>
 2 Regression Testing: <https://smarter.com/learn/automated-testing/what-is-regression-testing/>
 3 <https://github.com/javaparser/javaparser>

Java classes of the project, as well as crossing mechanisms to navigate through the tree. Then, according to the type of instrumentation, the representation of the respective portion of code in the AST is instrumented and the tree transformed in a Class again. Another project sources needs instrumentation as well, like the `AndroidManifest.xml` file. This file contains essential information about the application, like the necessary permissions that limit access to specific components or features of this or other applications. The read/write permissions on the device's external storage are needed in order to the `TrepnLib` be able to manage several runs and log important events, so the manifest file is instrumented (if needed) to include these permissions.

At the same time that the AST is analyzed and transformed, metrics and metadata about the code are obtained and calculated. The `jInst` tool generates, for each application analyzed, a file containing the signature of all the methods of each Class. This is necessary in order to easily calculate the methods coverage of the executed tests. At the end of the instrumentation process, the following information is obtained for each application method:

- **Code lines (SLoC):** metric that counts the number of method source lines. This metric is typically used to reflect the amount of computing effort, productivity, and code maintenance.
- **Cyclomatic Complexity:** This metric is used to calculate the complexity of a program. It can be defined as the number of linearly independent paths that can be traversed during the execution of a program.
- **Number of arguments:** number of arguments that the method receives.
- **Declared variables:** The number of declared variables allied to their type.
- **Static method:** Boolean indicating whether the method is static or not (i.e. if it is a class method and is resolved at compile time).
- **Android APIs used:** libraries belonging to Android Sdk and their methods used.
- **Java APIs used:** libraries belonging to the Java SDK and repeatable methods used.
- **External APIs used:** Libraries belonging to external libraries (from the project itself or other sources) and their methods used.
- **Declared permissions:** Permissions that the application needs to run on the device, present in the `AndroidManifest.xml` file.

After passing through `jInst`, the new instrumented project is ready to be built in order to generate the application to be installed on the device.

5.2 PROJECT BUILDING

The next phase of the AnaDroid consists in build the project sources in order to generate the respective APKs, the package file format used by the Android OS for distribution and installation of mobile apps. In order to automate the construction for several projects and consequent generation of the applications and/or test applications (generated from the test modules present in the project itself), mechanisms were developed to automate this process. Some instructions are included that adapt the building scripts. Some to adapt to the development tools present in the AnaDroid execution machine, as well as others that try to prevent compatibility errors between versions of libraries and platforms or even may interfere with the execution of the tests (such as the maximum memory heap size of the JVM). At the end of this phase, the project is built and its APKs are instrumented to be able to make calls to the energy profiler.

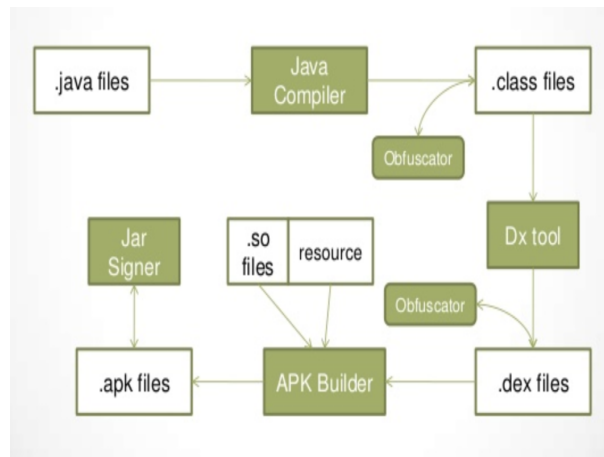


Figure 5.2.1.: Android Project Building workflow

5.3 TEST EXECUTION

This phase performs the process of installing the application and running tests on a physical device. In order to automate this process, tools have been developed that use the ADB tool (Android Debug Bridge), present in the Android SDK, allowing to perform and manage these processes from the development machine.

At the time, there are 2 testing frameworks supported by the AnaDroid framework: the Exerciser Monkey (which performs stress tests) and the JUnit testing framework, that can be used through the unit tests written by the application developer.

The JUnit testing framework is one of the most common frameworks used in the Java environment, and it provides a simple way to write unit tests to experiment program correctness and behavior. These tests are normally included in the project source, in testing

modules of the application. The GreenDroid tool was only capable of test applications using this framework. However, during several tests that were done on the applications that we gathered, we verified that a significant number of tests contained in the projects of the application were too simple. These covered small portions of the applications and rarely tested aspects related to the UI. This fact, coupled with the popularity of the Exerciser Monkey led to this also be considered by the AnaDroid.

The UI/Application Exerciser Monkey is a program that runs on a physical device (or emulator) and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. This tool is used to stress-test applications, in a random yet repeatable manner. However, unlike JUnit, these tests are performed without knowledge of the context, operation and implementation of the application, since the tests in JUnit are developed specifically for a particular application.

Nevertheless, the Exerciser Monkey tool has some parameters that can be manipulated in order to direct the generated events to different types of events (touches, movements, system-events, among others). Since we intend to perform tests for a large number of applications without having to know their context, a set of random seeds has been defined, resulting in different combinations of tests. When using the same seed value, re-running the Monkey will generate the same sequence of events. The seeds are used to perform test over all the applications, in order to execute the same actions for each one of these, establishing terms of comparison between different applications. The system key events (keys like Home, Back, Start Call, End Call, or Volume controls) were disabled for all executions, in order to avoid perform actions external to the app that can interfere with the performance and normal execution of both app and tests.

According to the testing framework chosen by the developer, the runtime (dangerous⁴) permissions necessary to run the application (and TrepnLib) are given, the tests are run on the device. After monitoring the execution of the tests. the resultant files and data related to the execution of each test are pulled from the device.

Before and after each test run, information about the status of the device is collected, which may interfere with the performance of the tests. The information collected is:

- **SDK version:** The Android SDK version that is running on the device;
- **API level:** Implementations and performance may vary between different API levels;
- **Number of processes running:** Total of processes running in the device;
- **Used CPU:** Percentage of used CPU before and after the execution of tests. If there are CPU intensive tasks running at least one of this moments, these may interfere with the performance of the tests;

⁴ Dangerous permissions overview: <https://developer.android.com/guide/topics/permissions/overview#normal-dangerous>

- **Free memory:** available memory at the device.

Between executions, the testing application is stopped and the respective cache is cleaned, in order to previous executions don't interfere with the performance of the next executions. In addition, each test runs 2 times. As discussed in section 4.2.1, TrepnLib has the capability to run in Measure and Silent Mode. In this way an execution of each type is carried out, in order to reduce the overhead of tracing of methods at the same time that the profiling of system and energy occurs.

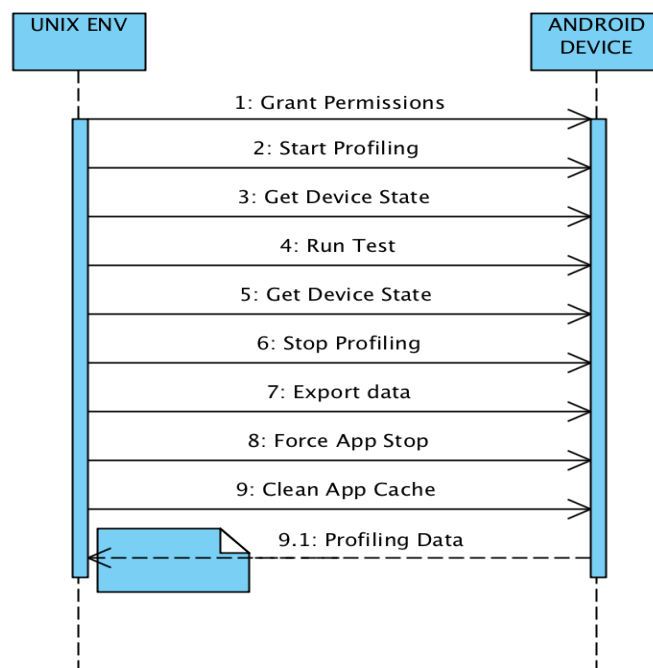


Figure 5.3.1.: Test Execution Diagram with Exerciser Monkey

Finally the app is uninstalled from the device and the data collected is analyzed by the Analyzer tool.

5.4 RESULTS ANALYSIS

The last phase of the AnaDroid workflow consists in analyzing and relating the results obtained from the execution of the previous phases. This function is performed by the Analyzer tool.

The Analyzer has the task of collecting the data related to the execution of the tests on the application, generating the results for each test and presenting them to the programmer. In addition, it sends implementation results to the GreenSource backend in order to centralize results and contribute to the growth of knowledge regarding the power consumption and

features of Android applications. The Analyzer begins by relating the various files pertaining to the execution of each test, which contain relevant information logs taken during its execution. Depending on the type of instrumentation and the framework chosen for the execution of the tests, the files that can be generated as soon as the tests are finished are the following:

- **GreenDroidResultTrace<ID>.csv:** A file generated for tests whose application has undergone test-driven instrumentation. This file contains resource usage values, gathered by Trepn Profiler during execution, associated with the timestamp in which they were collected. It also contains log of the start and end timestamp of the test run, in order to filter only the information collected during this interval.
- **TracedMethods<ID>.txt:** Contains the signature of the invoked methods during the execution of the tests, if the type of instrumentation for the application is test-oriented. The method name is associated with your package, class, and arguments, in order to identify it only in the application.
- **TracedTests.txt:** Like previous files, it is only generated for the method-oriented instrumentation type, if the JUnit framework is used. This file is useful for controlling the order of running the tests and for associating the identifier with its test name.
- **GreenDroidResult<ID>.csv:** This file is similar to the previous one, differing only in that it does not contain logs of the beginning and end of the tests, but of the beginning and end of the methods associated with a state and its timestamp . This state serves not only to identify the beginning and end of the methods, but also to infer the nesting level of the method (ie identify recursive calls and methods within other methods). This file is generated if the instrumentation method chosen is method-oriented.

After collecting the generated files, the calculations are made and inferred the use and consumption of resources consumed during the execution of the tests, being associated this information to the execution of the methods that were invoked during them. These methods are also classified energetically and according to the APIs used, if they have been classified as Energy-greedy (information based on previous studies[12]) In the end, the data and metrics presented to the programmer for each test unit (method or test) are in the following table:

The form on which this data and metrics are made available to the user is in the form of files. In addition to containing these data, it provides and relates other results related to the methods invoked during the tests, obtained through the static analysis performed in the instrumentation phase. The files displayed at the end of Analyzer execution differ from the type of instrumentation performed. These files are as follows:

| Metric | Unit | Description |
|------------------|------|---|
| Consumption | J | Test or method the total consumption |
| Time | ms | Test or method run time. |
| Method Coverage | % | For test-driven instrumentation, coverage at the method level is shown. |
| Wifi | 0-1 | If Wifi was used during the execution of the monitored block. |
| Mobile Data | 0-1 | If mobile data was used during the execution of the monitored block. |
| Screen State | 0-1 | If there was interaction with the screen. |
| Battery Charging | 0-1 | If the device was charging during execution. |
| Avg RSSI Level | dBm | average level of RSSI obtained. |
| Avg Memory Usage | B | Arithmetic mean of memory consumed. |
| Top Memory Usage | B | Peak of memory consumed |
| Bluetooth | 0-1 | If Bluetooth was used during the monitored block execution. |
| Avg GPU Load | % | Average percentage of GPU usage. |
| Avg CPU Load | % | Average percentage of CPU utilization. |
| Top CPU Load | % | Max percentage of CPU utilization. |
| GPS | 0-1 | If GPS was used during the execution of the monitored block. |

Table 5.4.1.: Results presented in TestResults.csv file

- **AppResults.csv:** Displays a summary of all methods invoked during tests, containing your signature, times it was invoked, and some metrics about them.
- **TestResults.csv:** Presents an overview of the execution of each test performed. For each test run, it provides the information contained in table 5.4.1.
- **MethodResults.csv:** Such as the previous file, presents an overview of the execution of each method invoked during the tests. For each method, it provides the resource state / usage obtained during its execution.
- **MethodsInvoked.json:** Contains for each method invoked during the tests, the APIS used and their classification. These are classified according to their provenance (Android SDK, Java SDK, External or Unknown).

For method-oriented instrumentation, only the last two are presented. As for the other supported instrumentation type, only MethodResults.csv is not displayed.

| Class | Method | Times invoked | CC | LoC | Android APIs | N args |
|--|-----------------------|---------------|----|-----|--------------|--------|
| FlagsActivity | onCreate | 198 | 2 | 6 | 2 | 1 |
| BaseFlagFragment | validate | 1071 | 6 | 18 | 0 | 0 |
| VerifyPhoneFragment | onCreateView | 198 | 1 | 4 | 5 | 3 |
| CustomPhoneNumberFormattingTextWatcher | hasSeparator | 781 | 4 | 8 | 0 | 3 |
| BaseFlagFragment | onPostExecute | 135 | 0 | 1 | 0 | 0 |
| FlagsActivity | onOptionsItemSelected | 19 | 3 | 6 | 4 | 1 |
| BaseFlagFragment | onPhoneChanged | 715 | 0 | 1 | 0 | 0 |
| BaseFlagFragment | initCodes | 198 | 1 | 2 | 1 | 1 |
| CustomPhoneNumberFormattingTextWatcher | reformat | 715 | 6 | 24 | 0 | 2 |
| CountryAdapter | getView | 4224 | 2 | 8 | 7 | 3 |
| VerifyPhoneFragment | onActivityCreated | 198 | 1 | 3 | 1 | 1 |
| Country | getCountryCode | 87757 | 1 | 2 | 0 | 0 |
| CustomPhoneNumberFormattingTextWatcher | stopFormatting | 67 | 1 | 3 | 0 | 0 |
| CustomPhoneNumberFormattingTextWatcher | onTextChanged | 2155 | 4 | 7 | 0 | 4 |
| BaseFlagFragment | initUI | 198 | 1 | 38 | 16 | 1 |
| BaseFlagFragment | hideKeyboard | 1071 | 1 | 3 | 9 | 1 |
| Country | getCountryCodeStr | 44 | 1 | 2 | 0 | 0 |
| FlagsActivity | onCreateOptionsMenu | 198 | 1 | 3 | 1 | 1 |
| BaseFlagFragment | onItemSelected | 44 | 0 | 1 | 0 | 0 |
| Country | getPriority | 61 | 1 | 2 | 0 | 0 |
| BaseFlagFragment | doInBackground | 198 | 0 | 1 | 0 | 0 |
| CustomPhoneNumberFormattingTextWatcher | afterTextChanged | 2155 | 11 | 28 | 10 | 1 |
| CustomPhoneNumberFormattingTextWatcher | beforeTextChanged | 2155 | 4 | 7 | 0 | 4 |
| Country | getResId | 4177 | 1 | 2 | 0 | 0 |
| CustomPhoneNumberFormattingTextWatcher | getFormattedNumber | 2700 | 1 | 2 | 0 | 2 |
| VerifyPhoneFragment | send | 1071 | 3 | 10 | 4 | 0 |

Table 5.4.2.: Content of AppResults.csv, for CountryFlagsDemo app

GREENSOURCE - A REPOSITORY TAILORED FOR GREEN SOFTWARE ANALYSIS

This section describes the process of creating and developing the GreenSource infrastructure, from the communication between each one of the parts that compose it, to its multiple features and implementation decisions. This infrastructure unifies the operation of 3 distinct systems:

- the GreenSource backend, with which it communicates through a RESTfull API, which allows communication with the GreenSource database through HTTP requests.
- An Unix-based development environment, with access to a set of Android applications to test, the Android SDK and AnaDroid framework installed.
- One or more Android devices (with Trepn Profiler installed), used to run applications and respective tests.

The section [6.1](#) enunciates the procedures followed to gather over 600 Android applications to be used as an initial study basis. Section [6.2](#) describes the aspects of implementation of GreenSource backend, in order to integrate the database on this infrastructure. We explain the database schema and the reasons that led to consider each one of the tables that compose it. The penultimate section [6.3](#) describes the process of Greensource backend creation, from the reasons that led to the use of the tools and methodologies used until the development process of each of its components. Finally, the section [6.4](#) describes the actual workflow of the GreenSource.

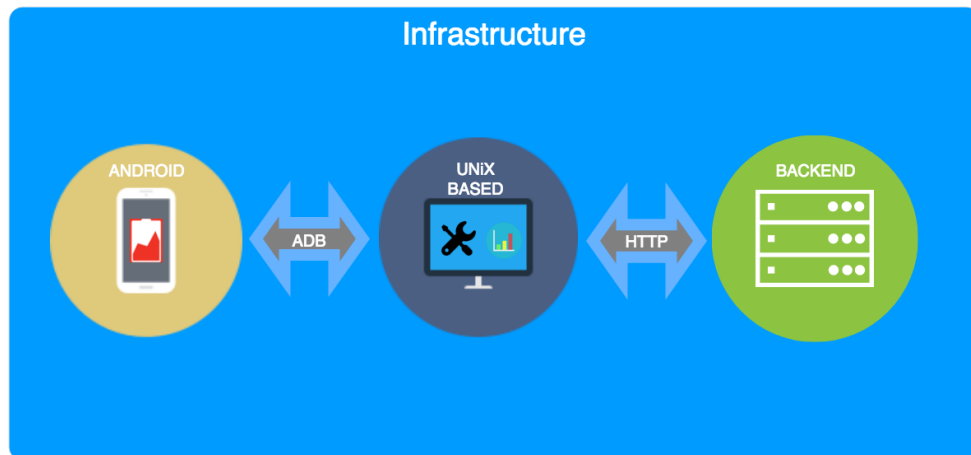


Figure 6.0.1.: GreenSource infrastructure overview

In order to give a greater purpose to the Anadroid framework, it has been integrated into the GreenSource repository. Anadroid is responsible for generating metadata and metrics about applications and running tests on them and then being sent to the GreenSource backend. This backend was developed with the intent of centralizing test execution information for various executions, applications, and devices (and their states). In addition to storing this information, it also makes it available in an open form for consultation, so that it can be used in later studies and contribute to the still meager knowledge about the energy consumption of applications on the Android platform.

The way the database was structured and developed, allows it to accompany the expansion of Anadroid, being easily extensible to support different test frameworks, devices and energy profilers.

6.1 DATA PROVENANCE

In order to obtain diverse and relevant material for the accomplishment of studies that could reach significant magnitude, it was necessary to collect a high number of Android applications on which the developed framework could act. The goal was to collect applications from a variety of sources in order to obtain a diverse set and whose projects/source code would be openly accessible. Excepting the alternative of developing a tool that analyzes open-source repositories, which identifies Android projects and extracts that content, we reused works that have the same goal.

So far there have been a lot of work with this motivation. We take advantage of the collection work done during the development of GreenDroid, whose goal was to extract an Android projects from the SourcererDB[42]. This is an open repository that contains thousands of Java projects, the leading development language for the Android platform. These projects were collected from other open repositories, such as Sourceforge, Apache

and Java.net. Among the thousands of projects contained in the repository, we selected those that we can identify as Android projects, by executing queries on the repository database.

Of all identified projects, we selected a subset that we identified as functional. Of the total collected, many cases were found that contained errors and/or lack of functional portions that prevented the projects from being compiled and constructed. Excluding these projects, we obtained a set containing more than 600 functional projects, which allow us to build applications that can be installed and run on Android devices. This set represents the starting point for the creation of the repository, already having a considerable size and minimally representative, and can be increased in future updates. Among the applications collected, most of the projects are builded with Gradle building automation, having also a few dozens with the (deprecated) Eclipse construction.

6.2 GREENSOURCE DATABASE

In order to be able to store all the information resulting from the analysis of the applications and the executions of tests on them, a repository has been created. This open repository is supported by a database, which has been carefully designed to be expandable for future refinements and expansions of the Anadroid tool.

This database is a relational database, implemented in PostgreSQL. The choice fell on this engine, due to the performance, ease of configuration, familiarity with relational models and their engine.

The database schema consists of 21 tables, which refer to the elements that compose the application, as well as metadata and metrics related to the execution and analysis of the ones made on them. The function of each of the tables is as follows:

- **AndroidProject:** This table is used to identify the Android project of each application. it contains information about the building tool used for generate the application APK file.
- **Application:** This table is used to identify and reference the application entity. This way is possible to search and compare applications and data related to them. The attributes in this table helps to better identify and describe each one of the applications.
- **AppMetric:** Table resulting from the normalization of data made after the first idealization of the relational model. It intends to identify only the relation (Many-to-Many) that involves an application and a certain metric. This table considers attributes that help to describe the nature of this relationship, including the value of the metric, coefficient and timestamp (in order to help identify each one of the relationships only and

make temporal research possible). There are similar tables to this one in this scheme, that intend to relate a certain entity with a metric relative to the same one.

- **AppHasPermission:** Identifies the relationship between an application and its permissions (also deriving from an Many-to-Many relationship).
- **AppPermission:** Identifies a permission ¹ required to run an application. The reason for considering this table is essentially the same as the AppBuildTool table.
- **Class:** It identifies solely one class of each application, in order to relate to relate each one of them with metrics that can be associated with application performance.
- **ClassMetric:** Contains metrics and metadata of Java Classes.
- **ImportClass:** Contains the Java packages imported by each Class. **Method:** It identifies one method of an application. The execution of certain methods in the application context is an elementary factor to consider, in what energy is concerned, and as such, it is essential to consider it in this model.
- **MethodMetric:** Table similar to the AppMetric table, which reflects the relationship (Many-to-Many) between a method and a certain metric. Contains reference to another table (MethodInvoked). This reference is a foreign key that is used to associate (invariably) invocations of methods in tests with the values of the metrics obtained by this same method.
- **MethodInvoked:** This table is used to identify a metric relative to a method, obtained as a result of the execution of a given test. As such, it contains the keys of the MethodMetric and TestResults tables as foreign keys.
- **Test:** It arises in order to relate the execution of a test on a particular application, with a particular test tool. As such, it serves to relate these 3 entities considered in the model.
- **Tool:** Testing tool/framework used to run a test over a particular application.
- **TestOrientation:** Approach of instrumentation used on the application, so as to include the energy monitoring capacity on the latter (According to the types of instrumentation referred in 4.2).
- **TestResults:** It identifies the execution of a test over an application. Since there can be several executions of the same test (with the same type of instrumentation, same profiler, same testing framework) on an application, this table is necessary to consider and differentiate these executions, complementing the Test table. It has as attributes

¹ Android permissions: <https://developer.android.com/guide/topics/permissions/overview>

descriptive information about the execution of the test, as well as reference to other entities of the model that have an impact on the way the test was executed (device, its initial and final state and profiler used).

- **Profiler:** Identifies the profiler used to monitor test.
- **Device:** Identifies the (physical) device on which the test was run.
- **DeviceState:** Describes the state of a device (in terms of resource utilization) at a given time.
- **TestMetric:** Table similar to AppMetric and MethodMetric. It relates the execution of a test with metrics resulting from this, which are related to the test and not to other elements of this (method, application).
- **Metric:** Identifies a particular metric. This table helps to avoid the redundancy of considering this table as an attribute in several tables that refer to metrics. The metrics are separated into categories (an Enum type), and there are 3 categories considered: static, dynamic or hybrid.
- **Study:** This table serves to be able to associate the consideration of certain metrics with previous studies that relate to classifications, analysis and monitoring of energy consumption, such as [12].

In this way, it is guaranteed the tool's expandability in several ways, being easy to include new metrics, instrumentation approaches, devices, profilers and testing frameworks.

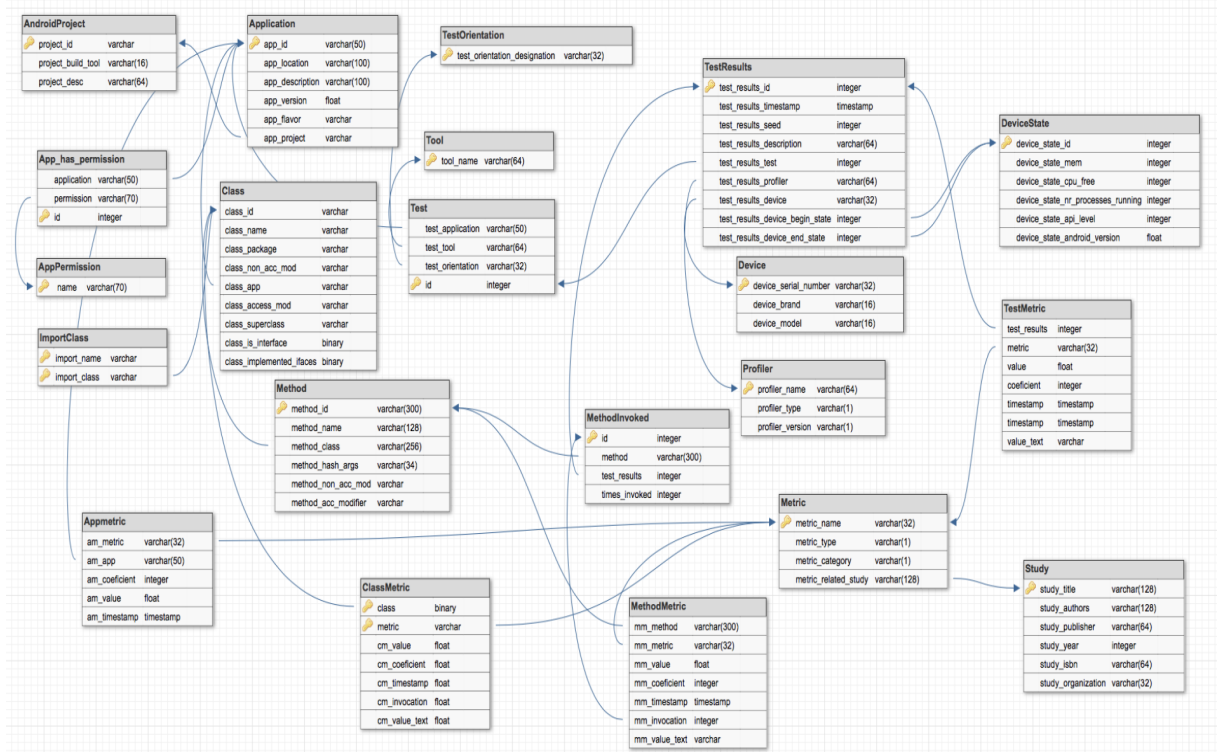


Figure 6.2.1.: GreenSource database schema

6.3 GREENSOURCE'S BACKEND

The main function of GreenSource's backend is to store and manage the information gathered through AnaDroid tool runs on Android applications. As such, the GreenSource database is contained within this subsystem, having the function of providing a uniform way of communicating with it. In this way, mechanisms can be put in place that help the management, validation and manipulation of data at a higher architectural level, obtaining an abstraction level independent of the database engine used. The greenSource backend was developed in the Python² language, with the help of the Django³ framework, following the MVC⁴. The use of this framework abstracts the implementation of mechanisms such caching, session management, among others, that help to manage and better scale this subsystem.

² <https://www.python.org/>

³ <https://www.djangoproject.com/>

⁴ Model-View-Controller pattern, an architectural software standard commonly used in web applications

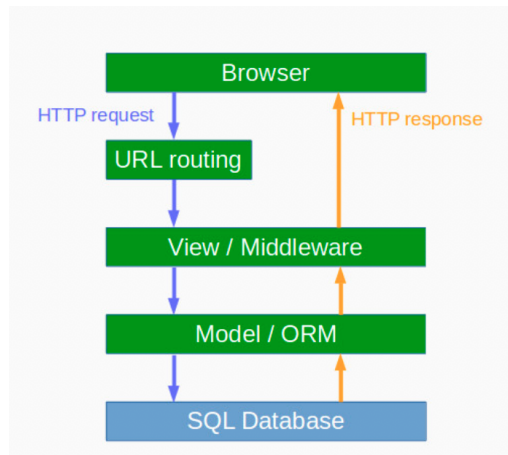


Figure 6.3.1.: Django architecture

One of the objectives of this dissertation was to offer an easy way to consult the metadata and metrics, in an open, simple and queryable form. In this way, the communication interface chosen rests on a RESTful API, which enables a uniform form of communication that provides the ability to consult, insert, change and delete data through HTTP requests. The data contained in these requests in JSON, a standardized, parsable, human-readable format that makes the data easily manipulated. Through the use of this interface, it is possible to design other applications (web, desktop, mobile) completely independent of the application, that use this communication interface, being able to manipulate the information contained in the backend in the way that suits to them, be it to present, insert or reuse data for other operations.

As such, for the implementation of this subsystem, the data model (and its relationships) began to be developed. These have been developed to be mapped through the Django mechanism of ORM, into the database described in 6.2.

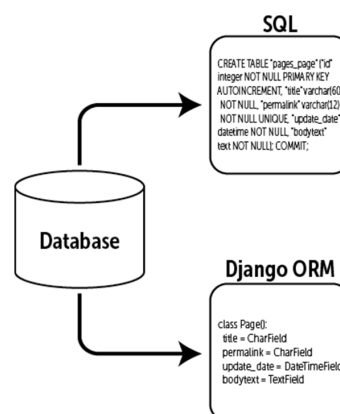


Figure 6.3.2.: An example of a Django model mapping

In order to define the URLs to communicate with the backend, the routes of the RESTful interface were designed so that its use was intuitive. Then the views were built, which in Django correspond to the controllers. The view concept in Django is distinct from others MVC frameworks, since are used to define the data that is visible, not the way it is presented. The views were then associated with their routes and several options were defined to serialize the models to the JSON format thus making the system fully functional.

6.4 GREENSOURCE WORKFLOW

This section describes the workflow of monitoring an application execution and store the respective results. The following image 6.4 summarizes the procedures followed for every monitored application. For each Android project, we instrument the source code and built the respective(s) APKs. Then we install these in a physical device and prepare the profiler. Then we execute the test(s) over the application using one of the supported testing frameworks. Between tests, the application is stopped and the respective cache is cleaned. Then we uninstall the application and the results are pulled from the device. Finally, the results are analyzed, presented and sended to the GreenSource backend, where are validated and stored in the infrastructure database.

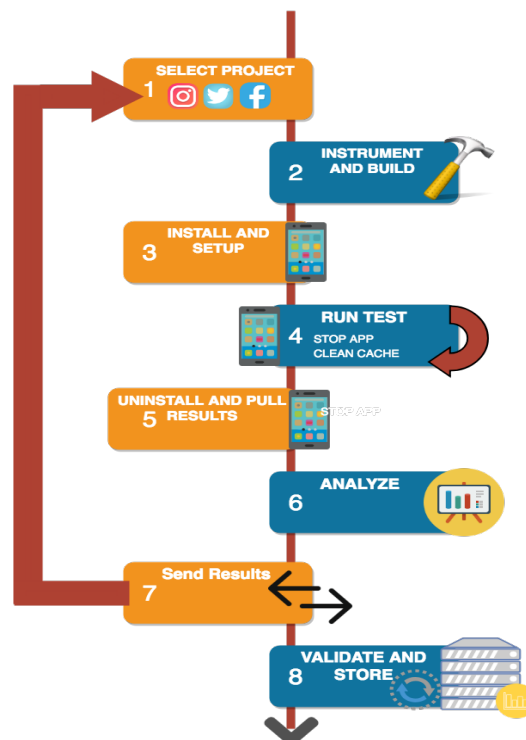


Figure 6.4.1.: GreenSource workflow

RESULTS

This chapter demonstrates some results obtained with the help of the AnaDroid tool, which are then stored in the database of the GreenSource backend. Several types of results were selected for the execution of application tests with the UI/Application Exerciser Monkey test framework. These results allow to compare tests, applications and portions of these, as well its executions.

The process of running AnaDroid on a wide range of applications is an extremely costly process over time. This time is influenced by both the performance of the development machine and the Android device on which the tests are performed. To process the set of gathered Androids projects (as referenced in 6.1) it takes approximately 2 weeks. An exemple for a particular application is shown in table 7.0.1.

| Task | Time (s) |
|---------------------------------|----------|
| Building app | 23,12 |
| Instrument | 3,2 |
| Elapsed time of 20 tests | 857,2 |
| Total time of warm-up/cool-down | 380 |
| Analyze Results | 6.83 |
| Send to GreenSource Backend | 5,37 |
| Recharge battery | 60 |
| Total: | 1 335,72 |

Table 7.0.1.: Time spent by AnaDroid for processing the Material Library application¹

In order to refine the final results of the AnaDroid tool, as well as its execution process, this process was repeated dozens of times. This was performed until the probability of occurrence of errors was reduced and the consistency and accuracy of the results increased. Until the writing of this dissertation, the framework was successfully executed on a total of 352 Android projects. The features and specifications of the device in which the applications and tests were executed are described in A. This has been done with test-oriented monitoring, and we tried to run tests until we reached a relevant method coverage, approxi-

¹ <https://github.com/Micnubinub/MaterialLibrary>

mately equal to or greater than 60%. These were done using the framework UI Application Exerciser Monkey, since its tests reach much higher values of method coverage than those obtained with the JUnit unit tests present in each project. In order to reach this level of method coverage, 20 equal tests (generated from the same seeds) were carried out for each one of these applications. If this level of method coverage was not reached after 20 tests, the process would continue for more 30 tests. These tests were executed using the same seeds, in order to generate the same sequence of events for every application. Each one of the tests was run twice, in Measure and Silent Mode, so that the tracing operation does not add more overhead to the monitoring task.

In order to prevent the pseudo-random events generated by the Exerciser Monkey from turning on/off system resources or invoking other applications, some precautions had to be taken. The first consisted of using an auxiliary application called Simiasque², which hides status bar under an overlay mask, preventing monkey tests from clicking it. The second was to prevent Exerciser Monkey from generating system-events (pressing the Home, Back, Start Call, End Call, or Volume buttons) in order to prevent generated events from being made outside the running application interface or prevent the phone from rebooting.

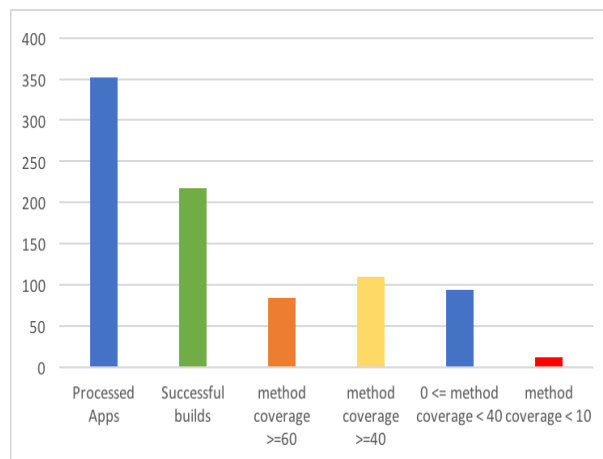


Figure 7.0.1.: Coverage of processed applications

From the set of projects analyzed, 217 gave rise to successful builds, resulting in applications (.apk files). Since the tests performed with the Exerciser framework were not developed specifically for each application, there are tests that did not significantly invoked the application code present in your project. Thus, in order to demonstrate the nature of the results that AnaDroid allows to obtain, we selected applications whose method coverage was equal to or greater than 40%, to draw conclusions about tests whose results are essentially related to the execution of application code. From the execution of applications during the tests, a coverage greater than 60% in 85 applications and greater than 40% in 110 applications was obtained. Only 13 applications had coverage below 10%.

² <https://github.com/Orange-OpenSource/simiasque>

Given the high number of metrics/results for each application, as well as the number of applications and tests performed, the object of analysis in this section has been reduced. In order to demonstrate the nature of the values obtained and proceed to their analysis and discussion, a subset of applications was selected for each aspect discussed. Otherwise it would be necessary to include large charts and tables (such as table B.0.2) that would make it difficult to interpret and analyze.

After running Anadroid, the final results are displayed in .csv files that summarize the execution of tests performed for each application. For example, for the Android application `DisplayingBitmaps`³, the application that invoked more methods in the total of the tests (754424) the results related to the tests are the in the table 7.0.2. The results obtained for each tests are values obtained at system-level, assuming that there are no other applications running at the same time. In order to warrant this assumption, we have done factory-reset on the device and did not provide a Google account credentials, in order to avoid Google services to constantly check for updates for its applications. In addition to the results presented in table 7.0.2, others that have been omitted for each test are presented because they are irrelevant in this context since their values are the same for all tests and all applications (such as mobile data, wifi, GPS, Bluetooth or screen state(table 5.4.1)). The device was used without a SIM card inserted, and the screen was kept unlocked. All sensors were turned on. For each method invoked during the tests, its respective metrics are presented in the way as they appear in the table 5.4.2.

By obtaining this type of results for a large set of applications, it is possible to compare applications for each one of the obtained metrics, in order to be able to correlate them with the (energy) performance of these. To illustrate examples of the comparisons that can be made in the future, when a significant number of applications are reached, in order to be considered representative of the Android platform, we have selected the following applications:

- Android `DisplayingBitmaps`: due to being the application with more methods invoked during the execution of the tests.
- Compass ⁴: Due to being the application that obtained the test with greater energy consumption and greater execution time for its best test (test with lower value for the these metrics).
- Hacker Live Wallpaper ⁵: Due to being the application that for its best test, it obtained a higher average percentage of CPU usage.

³ <https://github.com/googlesamples/android-DisplayingBitmaps>

⁴ <https://github.com/iutinvg/compass/tree/master/app>

⁵ <https://play.google.com/store/apps/details?id=com.gulshansingh.hackerlivewallpaper>

- Basic Networking⁶: Due to being the application with the highest average of memory consumed while running its best test.
- Vin Scanner⁷: Application with higher percentage of GPU usage during their best test.
- PkTest⁸: Application that has achieved considerable test runtime and uses above average amount of sensors/hardware usage.
- Material Library: It obtained total execution time of tests quite similar to the PkTest application.

| Test Number | Consumption (J) | Time (ms) | Coverage (%) | Avg RSSI Level | Avg Mem Usage | Top Mem Usage | Avg GPU Load | Avg CPU Load | Max CPU Load |
|----------------|-----------------|-----------|--------------|----------------|---------------|---------------|--------------|--------------|--------------|
| 89160419 | 74.919 | 33515 | 69.69 | -50.158 | 836987.535 | 864272 | 0.88 | 61.789 | 100.0 |
| 11 | 48.04278 | 21551 | 40.40 | -47.181 | 843599.578 | 876748 | 5.912 | 53.960 | 100.0 |
| 435986 | 90.456 | 24204 | 69.69 | -44.041 | 822303.591 | 867128 | 7.407 | 56.781 | 100.0 |
| 40201 | 71.834 | 29017 | 69.69 | -48.761 | 825611.297 | 856576 | 4.766 | 56.546 | 100.0 |
| 16 | 76.4522 | 27988 | 69.19 | -44.029 | 808640.368 | 840004 | 4.689 | 54.929 | 100.0 |
| 231251 | 51.927 | 18337 | 69.69 | -46.129 | 820401.516 | 843000 | 3.446 | 50.508 | 100.0 |
| 927139 | 58.230 | 26049 | 69.69 | -47.1288 | 815680.168 | 848352 | 5.056 | 59.879 | 100.0 |
| 123456789 | 60.152 | 25338 | 69.69 | -47.35 | 826982.464 | 864620 | 5.305 | 55.934 | 100.0 |
| 256773292 | 59.510 | 23190 | 69.69 | -42.847 | 827095.791 | 855452 | 5.625 | 54.002 | 100.0 |
| 330101 | 98.010 | 35165 | 69.69 | -45.801 | 827265.815 | 868648 | 5.118 | 61.424 | 100.0 |
| 12131145 | 50.336 | 24695 | 69.69 | -46.141 | 833746.1977 | 873272 | 1.9199 | 53.0411 | 100.0 |
| 1986 | 69.578 | 30640 | 63.63 | -45.511 | 811824.113 | 855756 | 3.877 | 56.746 | 100.0 |
| 2018 | 49.380 | 22700 | 69.69 | -46.224 | 814691.094 | 853128 | 2.554 | 52.995 | 100.0 |
| 1893 | 60.794 | 29015 | 62.12 | -46.1268 | 847369.156 | 880256 | 3.937 | 57.120 | 100.0 |
| 8913489 | 79.76 | 28309 | 69.69 | -47.545 | 830391.543 | 867140 | 6.125 | 55.557 | 100.0 |
| 72929123 | 58.72 | 25635 | 69.69 | -45.861 | 821123.176 | 847348 | 3.570 | 54.953 | 100.0 |
| 236236 | 68.39 | 25603 | 72.22 | -47.446 | 838037.96 | 870228 | 4.640 | 56.984 | 100.0 |
| 37666 | 39.376 | 19059 | 69.69 | -46.162 | 827088.545 | 860412 | 5.640 | 53.141 | 100.0 |
| 8894018411 | 57.80 | 24832 | 69.69 | -46.671 | 820214.612 | 848360 | 3.927 | 53.182 | 100.0 |
| 5637 | 53.105 | 27954 | 69.69 | -46.277 | 820144.307 | 864340 | 5.6738 | 56.569 | 100.0 |
| Total coverage | | | 72.22 | | | | | | |

Table 7.0.2.: Test Results of Android DisplayingBitmaps

For the first 5 applications, the tool was re-executed on it and a comparison was made according to the average value of metrics (usually associated to energy consumption[25]) obtained during its execution.

As can be seen in the table 7.0.3, looking only at the ranking of the applications in relation to the metrics value, it is concluded that the metric that is most directly related to the energy

⁶ <https://github.com/googlesamples/android-BasicNetworking>

⁷ https://github.com/bees4honey/mobile_vin_scanner

⁸ <https://github.com/zubietaroberto/AndroidKeyStoreTest>

| RANKING | | | | | | | |
|-----------------------|-----------|-----------|------------|-------------|--------------|-------------|--------------|
| App | #Methods | Time [ms] | Energy [J] | Avg CPU [%] | Avg Mem [MB] | Avg GPU [%] | Coverage [%] |
| DisplayingBitmaps | 754424(1) | 18337(2) | 39,38(2) | 50,51(2) | 808640,37(3) | 0,88(2) | 72,22 |
| Compass | 405817(2) | 24491(1) | 42,77(1) | 40,49(4) | 820870,42(2) | 0(5) | 60 |
| Hacker live wallpaper | 69596(4) | 6257(4) | 12,79(4) | 53,36(1) | 806042,34(4) | 0(5) | 51,68 |
| Basic Networking | 4532(5) | 3778(5) | 5,472(5) | 28,99(5) | 829371,23(1) | 0,45(3) | 71,88 |
| Vin Scanner | 141961(3) | 15830(3) | 37,22(3) | 41,99(3) | 799428,65(5) | 8,55(1) | 60,78 |

Table 7.0.3.: App Ranking

consumption is the execution time, since the 5 applications have the same ranking for both energy and time.

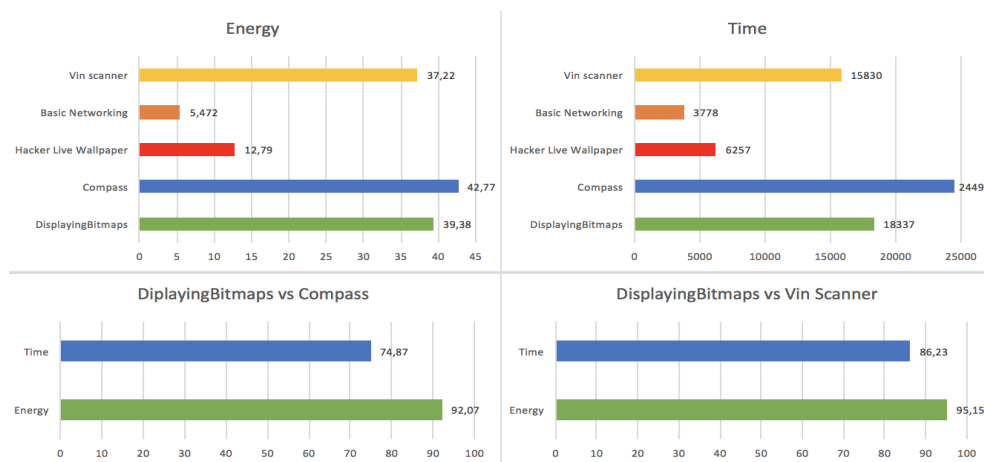


Figure 7.0.2.: Comparative graphs for time and energy

However, just by analyzing the lower graphs in the figure 7.0.2 we can conclude that this relation is not linear. The DisplayBitmaps and Compass applications have approximate energy values (DisplayingBitmaps consumed 92% of the Compass consumed energy. However, in terms of runtime, the difference between the values obtained is more pronounced, since the ApplicationBitmaps application consumed only 74% of the Compass time. To corroborate this conclusion, we present another example, comparing the PkTest and Material Library applications, which have a similar total execution time of all tests (with a slightly difference of 1.66%).

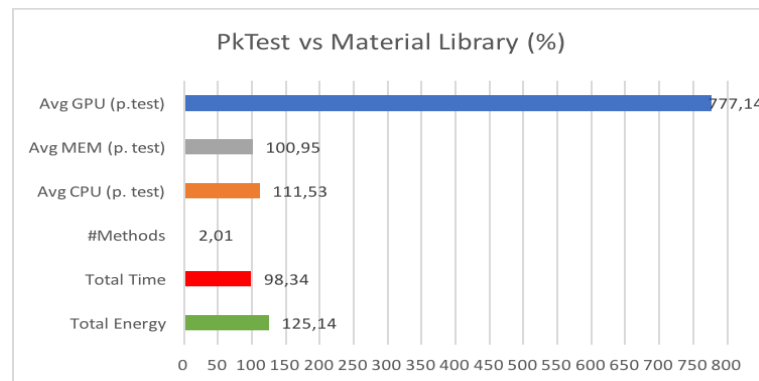


Figure 7.0.3.: Comparison between PkTest and Material Library

Through the analysis of the previous graph, we can conclude that although the PkTest application has a lower (but similar) execution time performance than the Material Library application, it has a higher energy consumption, even invoking only 6280 methods during the execution of the test, well below the 311,236 invoked by the latter. However, the hardware usage values (CPU, GPU, Memory) are higher for PkTest, and for GPU, the average usage percentage value (5.44 %) of this feature is 777.24% higher than the registered for the Material Library application. In this way we can conclude that the use of this type of hardware also has a considerable impact on the energy performance of an application.

During the execution of AnaDroid on the applications, an interesting aspect was also observed regarding the coverage of methods throughout the tests. For practically all the applications, and most notably for the applications with a high number of methods, it was possible to verify that the coverage was increasing during the first tests, being kept practically constant during the rest of the executions. This fact becomes even more curious due to the fact that the app cache is cleaned between test runs. However, for other metrics such as CPU, power, or runtime, it was not possible to identify behavior of a similar nature during runs.

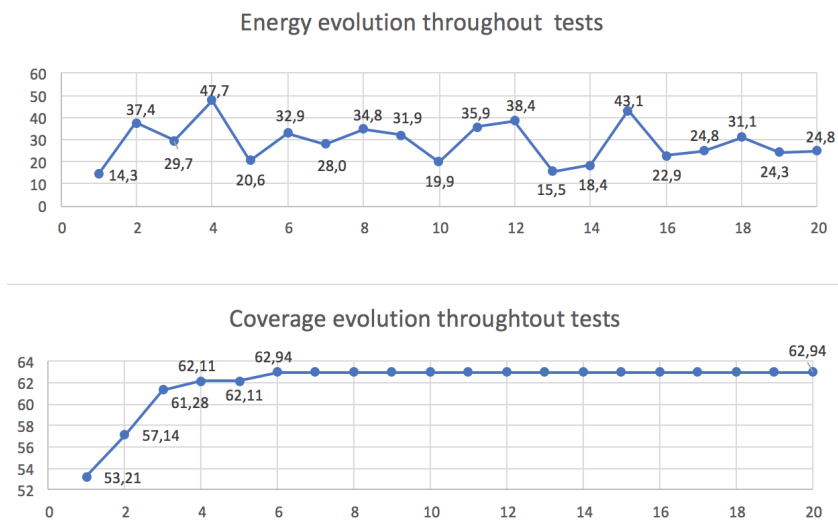


Figure 7.0.4.: Evolution of consumed energy and method coverage throughout tests for Material Library

CONCLUSIONS AND FUTURE WORK

In this chapter are presented the main contributions of the work developed in the scope of this dissertation. For this purpose, we will describe the manner in which the research questions listed in the 1.1 section are answered. Next, critical analyzes are done on the work done, and the conclusions to be drawn are presented. Finally, potential improvements and further work will be described in order to improve and extend the contribution of this thesis.

8.1 ACHIEVEMENTS

The contributions of this thesis go from a tool capable of gathering relevant metrics and metadata relevant to justify the consumption of code blocks of Android applications, to the development of an infrastructure capable of automating and gathering executions of this tool.

We successfully implemented our methodology, resulting in a global infrastructure containing more than 500 Android applications and results from over 6 000 tests executed over some of these. For the research questions we identified before, we answer to them all successfully. The answers to them are the following:

Q3: In what way can we automate the process of running AnaDroid on a large set of applications and centralize the results in a repository?

- **Question 1:** *How can GreenDroid be extended to be more expandable and powerful in its application analysis?* We started by expanding this tool to easily support the inclusion of new energy profilers, so that we can interact with the Trepn profiler. The better approach we found for this purpose was applying an architectural software pattern, the AbstractFactory. With the integration of new powerful profiler tools and an improved static analysis approach done over application source code, we were able to significantly increase the amount of information obtained for each application.

- **Question 2:** *What metrics and data can be gathered that can impact and justify the energy performance of an application (and its code)?* For answering that question, have reviewed several works related to software metrics and power consumption in Android. In the end, we gathered as many relevant metrics and data as we could get, static and dynamic, through the tools we had at our disposal. Some of these results were related to other results verified in studies on the Android platform, in order to unify the knowledge that exists in this sense. All metrics and metadata provided by the AnaDroid are detailed in chapter 5.
- **Question 3:** *In what way can we automate the process of running AnaDroid on a large set of applications and centralize the results in a repository?* We refined tools that automatized the process of running AnaDroid over hundreds of applications: We developed the Analyzer tool, that is capable of analyze the results and send them to the GreenSource backend, where all the metrics and metadata are stored in database tables.

8.2 FUTURE WORK

Energy performance is becoming more important nowadays. This applies to all types of systems, including software, especially for applications running in mobile or IoT devices. Since the Android platform continues to be the most used platform, Our effort focused on this.

With this work we have been able to develop an infrastructure capable of automating the process of execution, analysis and collection of results for a significant number of Android applications. This involved using various technologies of different types, from database engines, monitoring tools, Android framework, web and command line tools, etc.

However, there are still many future improvements to be made in order to extend, improve and validate the power of the features offered by this work. One of the main ones, which would bring more value and relevance to the developed work, would be the validation of the accuracy of the results obtained. This validation could be done by comparing these with the results obtained under the same conditions using other profilers, such as Monsoon. Since Trepn Profiler has proven its accuracy[11], this procedure could also help to understand the overhead caused by monitoring the system in the results obtained.

Still relative to AnaDroid, it would be interesting to make this tool openly available to the Android developer community. This could be a reference tool to assist the development of applications, and can be used to test, monitor the application, giving the possibility to anticipate the energetic and general performance of an application. Given this possibility, it would also be interesting to continue to extend this tool to be used with other profilers, as

well as with other testing frameworks used for this platform (such as Calabash¹, Robotium², among others).

In order to make the contribution given by the work developed even more relevant, it is essential to increase the matter upon which it affects. In order to achieve this goal, several tasks can be performed:

- **Test on several devices:** Given the large number of versions of the Android platform and the different hardware combinations present in these devices, conducting a large number of tests on a large number of different devices could allow comparison between them. This way we could draw conclusions about the performance of each of them and/or the Android versions that runs on these.
- **Increase the number of applications contained in the repository:** With the increase of the study material, the relevance and significance of the material collected on these could become the platform's characteristics.
- **Correlate the data/metrics obtained:** By reaching a significant number of results, these can be subjected to analyzes and studies (like data mining, for example) that allow correlating factors that have a significant impact on energy consumption. In this way, it could be possible to answer many questions, such as: What impact has the use of a sensor (e.g GPS) on energy consumption? What impact in power consumption has the use of two similar API's ?

In terms of infrastructure improvements, a significant improvement that would significantly increase User Experience would be the creation of a frontend. This subsystem would have the task of communicating with the backend, in order to present in a more intuitive and user-friendly way the contained information. This could be presented on the graphical form, containing tables, statistics and dashboards with the information provided by the backend.

By making these proposals for future work, the possibility of reaching a reference work increases significantly. The work is intended to help get critical assumptions about Android applications and their development to help developers develop more energy-efficient applications. This thesis proposes work that can be used as a starting point for mobile devices to have more battery time and consequently provide to their users more actual mobile usage time.

¹ <https://calaba.sh/>

² <https://github.com/RobotiumTech>

BIBLIOGRAPHY

- [1] Y. Kashiwagi, Y. Tawara, H. Chaki, K. Yamada, M. Kainaga, and T. Isobe, "An optimizing c compiler for the gmicro/500 microprocessor," in *Proceedings [1992] The Ninth TRON Project Symposium*, pp. 63–69, Dec 1992.
- [2] M. Erwig, R. F. Paige, and E. V. Wyk, eds., *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, vol. 8225 of *Lecture Notes in Computer Science*, Springer, 2013.
- [3] B. Wang, Y. Wu, and W. Zheng, "Task optimization based on cpu pipeline technique in multicore system," in *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 143–150, June 2011.
- [4] J. W. Yoo and K. H. Park, "A cooperative clustering protocol for energy saving of mobile devices with wlan and bluetooth interfaces," *IEEE Transactions on Mobile Computing*, vol. 10, pp. 491–504, April 2011.
- [5] M. Pedram, "Power minimization in ic design: Principles and applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, pp. 3–56, Jan. 1996.
- [6] *WOLFHPC '14: Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, (Piscataway, NJ, USA), IEEE Press, 2014.
- [7] J. Bézivin, *Model Driven Engineering: An Emerging Technical Space*, pp. 36–64. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.
- [8] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, (New York, NY, USA), pp. 22–31, ACM, 2014.
- [9] A. Hindle, "Green mining: A methodology of relating software change to power consumption," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 78–87, June 2012.
- [10] W. Lu, Y. Wang, Y. Wu, and G. Zhang, "An improved strategy to eliminate redundant compilation based on dalvik jit," in *2015 Eighth International Conference on Internet Computing for Science and Engineering (ICICSE)*, pp. 237–240, Nov 2015.

- [11] “Comparing energy profilers for android,” in *Proceedings of 21st Twente student conference on IT, Enschede, The Netherlands*.
- [12] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di 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, MSR 2014*, (New York, NY, USA), pp. 2–11, ACM, 2014.
- [13] L. Zhang, C. Stover, A. Lins, C. Buckley, and P. Mohapatra, “Characterizing mobile open apis in smartphone apps,” in *2014 IFIP Networking Conference*, pp. 1–9, June 2014.
- [14] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIX-ATC’10*, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2010.
- [15] M. Rashid, L. Ardito, and M. Torchiano, “Energy consumption analysis of algorithms implementations,” in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–4, Oct 2015.
- [16] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, “Detecting energy bugs and hotspots in mobile apps,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, (New York, NY, USA), pp. 588–598, ACM, 2014.
- [17] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*. New York, NY, USA: McGraw-Hill, Inc., 2nd ed., 1982.
- [18] J. Campos, A. Ribeiro, A. Perez, and R. Abreu, “Gzoltar: An eclipse plug-in for testing and debugging,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, (New York, NY, USA), pp. 378–381, ACM, 2012.
- [19] D. D. Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, “Petra: A software-based tool for estimating the energy profile of android applications,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 3–6, May 2017.
- [20] M. Couto, J. Cunha, J. P. Fernandes, R. Pereira, and J. Saraiva, “Greendroid: A tool for analysing power consumption in the android ecosystem,” in *2015 IEEE 13th International Scientific Conference on Informatics*, pp. 73–78, Nov 2015.
- [21] T. Carção, “Measuring and visualizing energy consumption within software code,” in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 181–182, July 2014.

- [22] H. David, E. Gorbato, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 189–194, Aug 2010.
- [23] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 194–204, April 2013.
- [24] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, and S. U. Khan, "Survey of techniques and architectures for designing energy-efficient data centers," *IEEE Systems Journal*, vol. 10, pp. 507–519, June 2016.
- [25] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. a. Saraiva, "Towards a green ranking for programming languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP 2017*, (New York, NY, USA), pp. 7:1–7:8, ACM, 2017.
- [26] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017*, (New York, NY, USA), pp. 256–267, ACM, 2017.
- [27] C. Jing, T. Gu, and L. Chang, "Escal: An energy-saving clustering algorithm based on leach," in *2008 IEEE International Symposium on Knowledge Acquisition and Modeling Workshop*, pp. 403–406, Dec 2008.
- [28] T. J. McCabe, "A complexity measure," in *Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76*, (Los Alamitos, CA, USA), pp. 407–, IEEE Computer Society Press, 1976.
- [29] S. Tripathi, Q. Abbas, and R. Beg, "Availability metrics: under controlled environments for web services," vol. 2, 09 2011.
- [30] E. Alikhashashneh, R. Raje, and J. Hill, "Using software engineering metrics to evaluate the quality of static code analysis tools," in *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pp. 65–72, April 2018.
- [31] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks, HotNets-X*, (New York, NY, USA), pp. 5:1–5:6, ACM, 2011.
- [32] L. Cruz and R. Abreu, "Using automatic refactoring to improve energy efficiency of android apps," *CoRR*, vol. abs/1803.05889, 2018.

- [33] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause, "How does code obfuscation impact energy usage?," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 131–140, Sept 2014.
- [34] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "Energypatch: Repairing resource leaks to improve energy-efficiency of android apps," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.
- [35] J. Zhang, A. Musa, and W. Le, "A comparison of energy bugs for smartphone platforms," in *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*, pp. 25–30, May 2013.
- [36] J. Flinn and M. Satyanarayanan, "Powerscope: a tool for profiling the energy usage of mobile applications," in *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, pp. 2–10, Feb 1999.
- [37] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 168–178, Dec 2009.
- [38] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, (New York, NY, USA)*, pp. 153–168, ACM, 2011.
- [39] D. D. Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 103–114, Feb 2017.
- [40] B. Fluri, M. Wuersch, M. Plnzger, and H. Gall, "Change distilling: tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725–743, Nov 2007.
- [41] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on github," *Proc. ACM Program. Lang.*, vol. 1, pp. 84:1–84:28, Oct. 2017.
- [42] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241 – 259, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

- [43] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 105–114, Oct 2010.

A

SUPPORT MATERIAL

```
timeout -s 9 300 adb shell monkey -s <monkey_seed> -p <app_package> -v --pct-  
syskeys 0 --ignore-crashes --ignore-security-exceptions --throttle 10 500
```

Listing A.1: command used for running monkey tests

| Feature | Details |
|-----------|-------------------------------------|
| Chipset | Snapdragon 400 Qualcomm MSM8226 |
| CPU | 1.2 GHz Quad Core |
| GPU | Adreno 305 |
| RAM | 1 GB |
| Mem | 8 GB |
| Screen | IPS LCD 720 x 1280 pixel 16M colors |
| Wifi | 802.11b/g/n |
| Bluetooth | 4.0 com A2DP/LE |
| GPS | A-GPS/GLONASS |
| Battery | 2070 mAh |

Table A.0.1.: Specifications of the used Android Device

| Monkey Seeds used | | | |
|-------------------|-------------|-----------------|-------------|
| Execution Order | Monkey Seed | Execution Order | Monkey Seed |
| 1 | 11 | 26 | 234235 |
| 2 | 1893 | 27 | 2136643 |
| 3 | 1986 | 28 | 6443456 |
| 4 | 16 | 29 | 673456 |
| 5 | 2018 | 30 | 834567 |
| 6 | 123456789 | 31 | 23569 |
| 7 | 89160419 | 32 | 79436 |
| 8 | 927139 | 33 | 577779043 |
| 9 | 330101 | 34 | 1178253 |
| 10 | 40201 | 35 | 464657 |
| 11 | 8913489 | 36 | 85632 |
| 12 | 435986 | 37 | 22 |
| 13 | 8894018411 | 38 | 8723733 |
| 14 | 72929123 | 39 | 455678 |
| 15 | 231251 | 40 | 4227799 |
| 16 | 5637 | 41 | 11099507 |
| 17 | 37666 | 42 | 321241566 |
| 18 | 256773292 | 43 | 32477 |
| 19 | 12131145 | 44 | 63737 |
| 20 | 236236 | 45 | 346346 |
| 21 | 454 | 46 | 26233 |
| 22 | 5321 | 47 | 20101001 |
| 23 | 253456314 | 48 | 3452356 |
| 24 | 3146346 | 49 | 3522366 |
| 25 | 134534567 | 50 | 46019191 |

Table A.o.2.: Seeds used for tests execution with Application Exerciser Monkey

B

DETAILS OF RESULTS

| Method Metric | Value | |
|--------------------|-------|-----|
| | Avg | Top |
| LoC | 1,822 | 168 |
| CC | 1,49 | 21 |
| Nr. Args | 0,22 | 4 |
| Nr. Declared Vars | 0,24 | 40 |
| Nr. Android API's | 0,54 | 25 |
| Nr. Java API's | 0,1 | 9 |
| Nr. External API's | 0,46 | 37 |

Table B.o.1.: Metrics resume of invoked methods during tests

RawFeature=+tnum;-onum

| App | Energy of test | | |
|---|----------------|------|---------|
| | Lowest | Avg | Highest |
| com.github.lassana.continuous_audiorecorder | 6,67 | 11,5 | 29,30 |
| demo.apprate.enrique.com.appratedemo | 5,74 | 6,7 | 8,34 |
| com.thbs.skycons | 10,06 | 52,2 | 72,90 |
| com.bees4honey.vinscanner | 37,22 | 52,9 | 82,10 |
| org.michaelevans.etsyblur | 5,95 | 10,2 | 16,04 |
| com.psaravan.flexiimageview.demo | 4,18 | 8,5 | 15,83 |
| com.example.android.basicsyncadapter | 6,17 | 9,6 | 25,19 |
| com.android.bryan.omgandroid | 6,87 | 10,2 | 14,60 |
| org.segin.ttleeditor | 11,99 | 24,1 | 51,96 |
| mmbialas.pl.navigationdrawersi | 4,93 | 9,0 | 15,62 |
| com.yelinaung.programmerexcuses | 11,35 | 17,5 | 24,32 |
| com.example.android.interpolator | 7,42 | 10,8 | 15,57 |
| com.androidbootcamp.sunshine | 10,03 | 15,4 | 24,38 |
| com.example.android.revealeffectbasic | 5,73 | 8,2 | 11,10 |

| | Energy of test | | |
|---|----------------|------|--------|
| com.example.android.displayingbitmaps | 39,38 | 63,8 | 98,01 |
| com.ironic.cryptosign | 6,43 | 9,9 | 15,16 |
| bettycc.com.appcompatv21demo | 8,20 | 15,2 | 26,16 |
| com.example.dawang.utakatik | 5,91 | 8,8 | 11,16 |
| com.example.hakan.aesdeneme2 | 5,61 | 13,4 | 27,44 |
| yuki312.android.imageryheader | 7,68 | 11,1 | 14,06 |
| com.mycompany.neuerversuch | 5,78 | 7,4 | 9,75 |
| pl.polak.flipview.demo | 5,36 | 8,4 | 12,00 |
| com.mongmx.androidbarcodeexample | 15,76 | 23,4 | 40,28 |
| pa.com.poroto.pktest | 21,77 | 36,4 | 56,20 |
| com.example.android.floatingactionbuttonbasic | 5,79 | 8,0 | 11,21 |
| com.example.zafir.foodsaver | 5,19 | 14,7 | 31,22 |
| com.oliveira.airon.myfirstapplication | 5,81 | 9,2 | 16,09 |
| com.area29games.sort | 5,50 | 12,2 | 22,71 |
| com.micnubinub.materiallibrary | 14,30 | 29,1 | 47,66 |
| com.pwitchen.weathericonview | 2,24 | 4,1 | 7,77 |
| com.alexkang.bluechat | 4,65 | 22,4 | 44,84 |
| com.tehmou.rxmaps | 21,16 | 72,1 | 142,38 |
| com.example.android.camera2basic | 11,41 | 20,8 | 31,94 |
| com.example.russell.simpleuserinterface | 5,16 | 8,5 | 12,72 |
| our.memo | 2,61 | 5,4 | 10,96 |
| com.lsjwzh.loadingeverywhere.sample | 2,95 | 7,7 | 17,46 |
| com.example.android.actionbarcompat.styled | 5,89 | 7,9 | 11,95 |
| com.example.android.basicnetworking | 5,47 | 7,2 | 10,89 |
| me.avacariu.bisect | 8,00 | 20,7 | 55,30 |
| com.sevencrayons.compass | 42,77 | 50,7 | 56,38 |
| com.zekunyan.linktextview.sample | 12,52 | 19,8 | 25,23 |
| com.plusgaurav.spotifystreamer | 5,14 | 7,1 | 10,69 |
| com.savinoordine.menuanimation | 4,21 | 7,7 | 10,02 |
| com.peoplr.shannoncox.hellorequests | 9,75 | 14,7 | 21,64 |
| com.kaiinui.android_qr_sample | 11,98 | 16,9 | 26,38 |
| softwareinclude.ro.portforwardandroid | 9,54 | 13,6 | 19,54 |
| com.example.android.donebar | 7,99 | 11,3 | 22,22 |
| com.example.android.slidingtabscolors | 9,59 | 17,6 | 28,26 |
| com.ecloud.pulltozoomview.demo | 11,27 | 17,9 | 23,17 |
| com.pixelimpressions.wwwparsingexample | 5,72 | 8,2 | 14,43 |

| | Energy of test | | |
|---|----------------|------|-------|
| com.example.rx | 8,51 | 11,4 | 15,58 |
| com.example.android.inotifications | 7,42 | 11,2 | 16,43 |
| me.mattlogan.parallaxlistview | 7,85 | 19,0 | 28,64 |
| com.nabi.nf70.bmi | 5,45 | 7,3 | 11,11 |
| advertiser.ble.read.on.ly.bleadvertiser | 4,60 | 6,9 | 9,65 |
| im.mdp.gsmtracker | 5,56 | 7,2 | 11,20 |
| nu.jixa.hitta | 8,37 | 11,1 | 14,32 |
| edu.barry.euclid.mobile.crypto | 10,93 | 29,1 | 67,23 |
| com.example.android.activityscenetransitionbasic | 7,87 | 11,6 | 15,98 |
| com.norsemen.broadcastlongrunningservice | 8,98 | 12,6 | 23,12 |
| miniproject.barcodescanner | 6,32 | 8,9 | 14,37 |
| com.bmpak.anagramsolver | 4,09 | 7,1 | 8,47 |
| com.example.android.clippingbasic | 9,84 | 15,9 | 37,80 |
| com.example.paintapplication | 10,06 | 28,0 | 67,31 |
| org.jast.userinterface | 5,81 | 8,3 | 14,90 |
| com.javatechig.alarmservice | 5,78 | 8,3 | 15,33 |
| ntpclock.kamalan.com.app | 5,79 | 7,9 | 10,75 |
| com.ryanharter.android.tooltips.sample | 10,44 | 29,1 | 55,22 |
| io.x8.kontaktbeacon | 4,34 | 8,3 | 10,23 |
| com.example.android.elevationdrag | 8,30 | 11,3 | 16,06 |
| com.kawakawaplanning.floattest | 5,22 | 6,3 | 8,98 |
| com.dantelab.rippleeffectview | 10,38 | 20,0 | 29,37 |
| com.example.android.immersivemode | 6,89 | 9,0 | 17,74 |
| com.example.android.actionbarcompat.listpopupmenu | 7,88 | 13,3 | 31,13 |
| com.lorz88.modernartui | 7,12 | 15,3 | 23,47 |
| com.example.android.actionbarcompat.basic | 5,66 | 7,0 | 9,72 |
| com.example.android.basicnotifications | 6,00 | 8,7 | 19,19 |
| com.sharathp.userinterface | 5,21 | 7,3 | 9,79 |
| net.macdidi.myandroidtutorial | 12,46 | 25,0 | 50,14 |
| com.example.nirzvi.userinterface | 10,16 | 15,4 | 20,55 |
| com.pipirrsolutions.aeseztargetcontroller | 8,93 | 16,8 | 32,42 |
| com.example.android.elevationbasic | 5,54 | 8,1 | 10,17 |
| net.qiujuer.imageblurring | 11,42 | 20,7 | 29,97 |
| com.jvacx.sunshine | 6,41 | 10,5 | 15,11 |
| com.gitonway.lee.niftynotification | 14,75 | 30,4 | 54,80 |
| nsu.ccfi.ru.uilimageplayer | 9,27 | 17,7 | 25,88 |

| | Energy of test | | |
|---|----------------|------|-------|
| | | | |
| com.app.lcs.android.lcstracker | 5,81 | 7,3 | 9,56 |
| com.github.fernandodev.androidproperties.sample | 4,94 | 6,8 | 8,22 |
| tumblr.shareproject | 7,76 | 12,5 | 24,68 |
| com.norsemen.longrunningservice | 6,87 | 12,1 | 14,23 |
| com.example.android.actionbarcompat.shareactionprovider | 24,21 | 33,4 | 48,33 |
| bixolon.android.fewlaps.com.fewlapslovesbixolon | 4,59 | 7,1 | 9,54 |
| com.homelab.labinstrument | 5,78 | 7,6 | 12,12 |
| com.example.android.documentcentricapps | 11,53 | 22,0 | 49,55 |
| com.livejournal.karino2.pdf2jpegzip | 5,73 | 7,2 | 10,52 |
| com.example.android.cardreader | 6,51 | 10,1 | 14,51 |
| com.ashokslsk.moodscanner | 5,39 | 7,3 | 9,05 |
| com.gulshansingh.hackerlivewallpaper | 12,79 | 21,0 | 47,47 |
| tada.com.example.bruce.androidtada | 6,91 | 10,5 | 13,20 |
| com.alhazmy13.myapp | 5,89 | 10,9 | 25,68 |
| org.intracode.sortvisualizer | 8,48 | 24,2 | 44,76 |
| com.nopeet.mediascannerconnector | 6,63 | 10,3 | 15,29 |
| de.florian.processlimit | 5,05 | 9,5 | 22,79 |
| com.lugeek.encryption | 6,94 | 12,8 | 19,75 |
| org.thanthoai.securesms | 11,65 | 19,7 | 37,73 |
| com.example.android.slidingtabsbasic | 15,05 | 31,2 | 49,15 |
| com.lee2384.jonathan.lcsfantasytracker | 7,13 | 16,9 | 25,49 |
| com.example.android.cardemulation | 5,46 | 9,9 | 26,79 |
| com.example.i306851.androidapparser | 10,17 | 13,4 | 22,43 |
| com.example.android.storageclient | 3,62 | 7,1 | 10,64 |

Table B.o.2.: Lowest, average and highest energy consumption value obtained during app testing

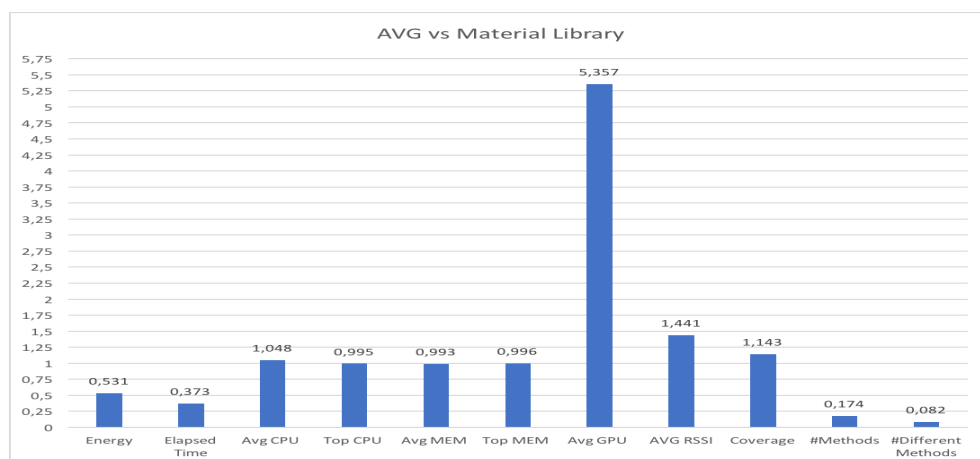


Figure B.0.1.: Comparison between an specific app (Material Library) and the average values for the main results



TOOLING

C.1 TREP N PROFILER

Trepn Profiler is a tool for profiling the hardware state/usage and energy consumption in Android Platform. Trepn is developed by the Qualcomm community and works on devices with Snapdragon chipset-based devices. It can be used as a normal application or as a service. It can profile the usage and power consumption of CPU, GPU, Wi-Fi, wakelocks, memory, SD card, Audio and also the run-time energy consumption of the whole device. For the usage statistics of different hardware components, Trepn depends on the /proc and other system files. Trepn samples information after every 100ms, but this rate can be adapted to be performed at longer intervals, in order to decrease the overhead of the sampling.

Trepn offers different modes of information visualization, providing an overlay view of different graphs and charts in the foreground. In this way, the developers can associate the performance of applications with the resource utilization and energy consumption at run time. It also allows exporting the real time profiled data for offline analysis, like a .csv format. Finally, Trepn can be controlled from an external application or command-line tools, facilitating the process of automated profiling.

C.2 ANDROID DEBUG BRIDGE (ADB)

ADB is a tool that provides access to a Unix shell that can be used to execute commands over an Android device or Android virtual device (AVD) through a command-line tool interface. This tool is part of the Android SDK, and is inclusively used by Android Studio. Besides the communication between computer and Android device, the ADB also allows to install and execute applications, copy information between devices or even record the device screen.

C.3 SIMIASQUE

Simiasque is an android application designed for Android developers. It allows to manually hide the status bar under an overlay mask, preventing stress tests (like monkey tests) from clicking it, in order to prevent this framework to invoke other resources and interfere with the testing process. Without hiding the status bar, Monkey tests frequently pulls the bar down, browsing among notifications and settings rather than the application UI. This app can be a simple solution for this common problem. It has a single Activity, providing a simple UI with a switch that allows to turn on/off the blue overlay mask. It also provides a service that allows to perform the same action through the adbC.2 tool.

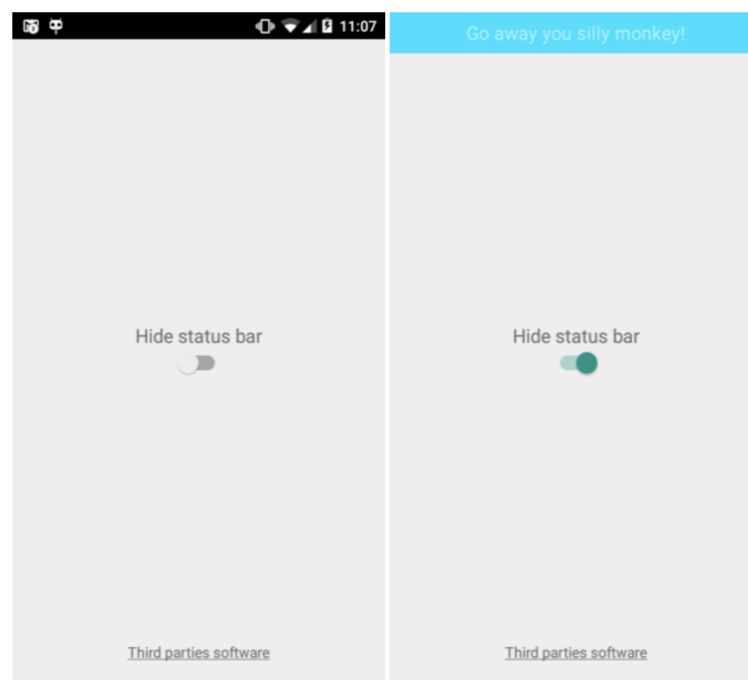


Figure C.3.1.: Simiasque application

C.4 JAVA PARSER

Java Parser is an open-source framework written in Java, that provides mechanisms to parse, analyze, transform and generate an Abstract Syntax Tree (AST) from Java code. The AST is a structure representing the Java code in a way that its easier to manage, instrument and transform. This code abstract representation is useful to perform code refactoring or obtain code metrics.

C.5 DJANGO FRAMEWORK

Django is a free and open source framework for creating web applications, written in Python. It is a web framework that follows the MVC (Model-View-Controller) pattern, providing a set of components that provides an easier and faster way to develop websites. The use of this framework abstracts the implementation of mechanisms such ORM, caching, authentication or session management, enabling the development process to be fast and effective.

C.6 POSTGRES

PostgreSQL is an object-relational database management system (ORDBMS), which was developed at the University of California at Berkeley Computer Science Department. It is an open-source system that runs on all major operating systems, using and extending the SQL language combined with several features that safely store and scale data workloads

C.7 EXERCISER MONKEY

The UI/Application Exerciser Monkey is a program that runs on a physical or virtual device. It generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. It generates a specified number of events without any user interaction, which is helpful for perform stress testing. This tool generates the random events from set of commands and collect the crashes or memory reports.

This work is funded by FEDER - European Regional Development Fund and by FCT - Foundation for Science and Technology within the project FCOMP- 01 0124 FEDER - 020484 and grant ref. BI1 - 2017_PTDC/EEIESS/ 5341/ 2014_UMINHO.

FCT Fundação
para a Ciência
e a Tecnologia



European Union
European Regional
Development Fund