# Multi-Layer Tracing of Android Applications for Energy-Consumption Analysis

by

Meysam Feghhi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Meysam Feghhi, 2017

# Abstract

The continuous increase in the use of mobile devices has been driving research in the improvement of the energy consumption of these devices and the applications running on them. In this thesis, we present a tool that helps Android developers understand the implications of their changes to the application's energy profile throughout its evolutionary development. The presented tool is an extension over *GreenAdvisor*, an already existing tool that predicts energy changes based on the changes in the application's system-call profile and then looks for the responsible code using a set of predefined relevant keywords [1]. Towards improving *GreenAdvisor*, *GreenAdvisor2.0* instruments the application source code and collects, in addition to system-call counts, system-call and method-call timing information and uses this evidence to locate the methods responsible for changes in the energy profile. In order to evaluate our work, we synthetically produced conditions in which the decisions of *GreenAdvisor2.0* can be marked as correct or incorrect. Using this information, we then quantified the accuracy and effectiveness of *GreenAdvisor2.0* and compared them to that of the original *GreenAdvisor* and random guess. We found that *GreenAdvisor2.0* made sensibly more correct decisions than the other two competitor approaches in cases where the system-call profile was impacted significantly by a re-factoring commit which synchronously consumed more energy.

*To Haniyeh*

*For standing beside me although two continents away and supporting me when I needed it the most*

# Acknowledgements

I would like to thank my supervisor, Prof. Eleni Stroulia for her great supervision. This thesis would not have been possible without her patience and quality guidance.

I would also like to thank Dr. Abram Hindle for helping us understand the statistical analysis used in this thesis.

I am also grateful to Karan Agarwal for walking us through the steps to make GreenAdvisor work.

# Table of Contents


**1 Introduction** **1**
- 1.1 Thesis Overview . . . . . 4
- 1.2 Contributions . . . . . 5
- 1.3 Thesis Organization . . . . . 5

**2 Related Work** **6**
- 2.1 Energy Measurement and Modeling . . . . . 6
- 2.2 Empirical Studies Analyzing Patterns of Energy-Consumption 9
- 2.3 Chapter Summary . . . . . 12

**3 Methodology** **13**
- 3.1 The Original GreenAdvisor . . . . . 13
- 3.2 GreenAdvisor2.0 . . . . . 17
  - 3.2.1 Workflow . . . . . 17
  - 3.2.2 Architecture . . . . . 25
- 3.3 Chapter Summary . . . . . 26

**4 Evaluation** **28**
  - 4.0.1 Results and Analysis . . . . . 31
- 4.1 Chapter Summary . . . . . 35

**5 Threats to Validity** **36**

**6 Conclusion and Future Work** **38**

**Bibliography** **40**

# List of Tables

4.1 Calculated Measures for Experiment I . . . . . . . . . . . . . . . 32
4.2 Calculated Measures for Experiment II . . . . . . . . . . . . . . 33
4.3 Calculated Measures for Experiment III . . . . . . . . . . . . . . 34
4.4 Calculated Measures for Experiment IV . . . . . . . . . . . . . . 34

# List of Figures


3.1 Interaction of applications, C libraries, and kernel through system-calls 14
3.2 GreenAdvisor2.0 Workflow . . . . . . . . . . . . . . . . . . . . . . 17
3.3 An Example Data File in $sys_call_{v1}$ and $sys_call_{v2}$ . . . . . . . . . 18
3.4 An Example Data File in $sys_call_{v1.i}$ and $sys_call_{v2.i}$ . . . . . . . . 20
3.5 An Example Data File in $method_call_{v1.i}$ and $method_call_{v2.i}$ . . . 21
3.6 An Example of Instrumentation in a Method . . . . . . . . . . . . 23
3.7 GreenAdvisor2.0 Components . . . . . . . . . . . . . . . . . . . . 25
3.8 An Example output of *GreenAdvisor2.0* . . . . . . . . . . . . . . . 27

# List of Symbols

| | |
|---|---|
| $src_{v1}$ | Application source, version 1 |
| $apk_{v1}$ | Compiled APK of application source, version 1 |
| $src_{v2}$ | Application source, version 2 |
| $apk_{v2}$ | Compiled APK of application source, version 2 |
| $src_{v1.i}$ | Instrumented application source, version 1 |
| $apk_{v1.i}$ | Compiled APK of instrumented source, version 1 |
| $src_{v2.i}$ | Instrumented application source, version 2 |
| $apk_{v2.i}$ | Compiled APK of instrumented source, version 2 |
| $system_call_{v1}$ | Set of 5 system-call counts profile obtained from runs of version 1 |
| $system_call_{v2}$ | Set of 5 system-call counts profile obtained from runs of version 2 |
| $system_call_{v1.i}$ | Set of 5 system-call times profile obtained from runs of instrumented version 1 |
| $system_call_{v2.i}$ | Set of 5 system-call times profile obtained from runs of instrumented version 2 |
| $method_call_{v1.i}$ | Set of 5 method-call times profile obtained from runs of instrumented version 1 |
| $method_call_{v2.i}$ | Set of 5 method-call times profile obtained from runs of instrumented version 2 |
| $T_{s.v1}$ | Tuple consisting number of times $s$ is invoked in runs of version 1 |
| $T_{s.v2}$ | Tuple consisting number of times $s$ is invoked in runs of version 2 |
| $T_{s.v1.i}$ | Tuple consisting number of times $s$ is invoked in runs of instrumented version 1 |
| $T_{s.v2.i}$ | Tuple consisting number of times $s$ is invoked in runs of instrumented version 2 |
| $I_{s.v1}$ | Tuple indicating the impact of instrumentation on system-call s in runs of version 1 |
| $I_{s.v2}$ | Tuple indicating the impact of instrumentation on system-call s in runs of version 2 |

| | |
|---|---|
| $H_{s.m.v1}$ | Tuple indicating number of times system-call s is invoked in method m in runs of version 1 |
| $H_{s.m.v2}$ | Tuple indicating number of times system-call s is invoked in method m in runs of version 2 |
| $\theta$ | Set of significantly changed system-calls, obtained by comparing counts of system-calls in runs of version 1 with that of runs of version 2 |
| $\lambda$ | Set of undesirably affected system-calls by instrumentation, obtained by comparing the impact of instrumentation on counts of system-calls in runs of version 1 with that of runs of version 2 |
| $\delta$ | Set of system-calls whose count were significantly changed but were not undesirably affected by instrumentation, obtained by subtracting $\lambda$ from |

# Chapter 1

# Introduction

Mobile devices (e.g. smartphones and tablets) have now become the most popular kind of general-purpose computers. Despite phenomenal improvements in the processing power of these devices, their battery is still considered to be a limiting resource.

Mobile processor manufacturers have always been trying to optimize the power-consumption behavior of their products. For example, the Apple A8 processor, which was first introduced in September 2014 with iPhone 6 and iPhone 6 plus, was claimed to have 25% more CPU performance and 50% more graphics performance while drawing only 50% of the power compared to its predecessor, the Apple A7.

Another class of studies have been dedicated to the improvement of energy consumption at the level of operating systems. For example, Android Marshmallow, released by Google in October 2015, was claimed to had become smarter than its predecessors in terms of battery usage by (a) putting the device into a sleep state when it is at rest, and (b) limiting the impact of seldom-used apps on battery life.

Still, with all these improvements, an energy-hungry application can drain your phone's battery and leave you phone-less in the middle of nowhere, which highlights the need for improvement of energy consumption at the application level. If, however, application developers are meant to consider energy consumption as a software quality, new tools are necessary to support this task.

In order to make a tool for developers that helps with energy-aware devel-

opment, energy leaks should first be studied. One way to find energy leaks is to compare two different implementation of a single feature. In real world, when a developer pushes a re-factoring commit to the application's source-code repository, it is likely that they create a new implementation of an already existing feature. If the developer has access to a tool which shows how energy is impacted with the changes being committed, they will have a better chance of making energy-optimized apps.

Energy consumption can be measured both physically and logically. In physical measurement, the application under test is usually run on an instrumented device where the current drawn by hardware components is physically monitored and measured [8]. In logical measurement however, the energy consumption is estimated using different parameters and metrics such as number of system-calls made [1], or the power state of hardware components [7]. The process of collecting these metrics is called *profiling*. Profiling can be done in different levels of execution stack from hardware up to the operating system and application under test and often needs some kind of modification in that level which is called instrumentation. Instrumentation itself comes at a price: it is difficult and it leads to more energy consumption which complicates the process of estimating true energy consumption. Therefore, it is necessary to control and carefully assess the impact of instrumentation when it is required in an approach to measure energy consumption.

In this paper, we introduce *GreenAdvisor2.0* which is a major extension over *GreenAdvisor*, first introduced by Agarwal *et al.*[1]. The original *GreenAdvisor* predicts changes in the energy profile of an Android application by looking for the significant changes in the application's system-call profile when the application code undergoes a change/re-factor. If an increase in the energy profile is predicted, the tool finds keywords in the added source-code which are verbally or semantically relevant to the name of system-call whose count has increased and points to the place of these keywords as the source of blame. *GreenAdvisor2.0* replaces this blame assignment procedure by adopting a multi-layer tracing approach, which produces time-stamped system-call and method-call profiles. Finally, these profiles are used to localize the method-to-

blame for the energy-consuming change.

In assigning blame for a surge in the energy-consumption of an app to a specific piece of code, *GreenAdvisor2.0* recognizes two conditions: *Synchronous* and *Asynchronous*.

The energy is consumed synchronously in an application method when it is bound to the life time of the method. The energy may also be consumed as a result of a method-call long after the method is finished executing which is called asynchronous energy-consumption.

In our research, we explore the following questions:

**Q1:** Does the new approach of blame assignment work when the system-call profile is impacted by a re-factoring commit which synchronously consumes more energy?

**Q2:** Does the new approach of blame assignment work when the system-call profile is impacted by a re-factoring commit which asynchronously consumes more energy?

**Q3:** Does the new approach of blame assignment work when the system-call profile is impacted by a re-factoring commit which we may not be able to categorize as completely synchronous or asynchronous?

We evaluate our work by theoretically answering the above questions and conducting a series of four experiments to support our answers. Our experiments are all based on synthetic test-cases where we inject both system-call-producing and system-call-free code to a number of methods in the source-code of an Android application. In these experiments, we synthetically produced conditions in which the decisions of *GreenAdvisor2.0* can be marked as correct or incorrect. Using this information, we then quantified the accuracy and effectiveness of *GreenAdvisor2.0* and compared them to that of the original *GreenAdvisor* and random guess. The results of supporting experiments suggest that *GreenAdvisor2.0* made sensibly more correct decisions than the

other two competitor approaches in cases where the system-call profile was impacted significantly by a re-factoring commit which synchronously consumed more energy

## 1.1 Thesis Overview

- **The Original GreenAdvisor**: The original *GreenAdvisor* runs two versions of an Android application and looks for the changes in the system-call profile to predict changes in the energy profile. It then identifies the code responsible for this change by looking for verbal connections between the name of the increased system-call and the tokens in code.

- **GreenAdvisor2.0**: In order to ensure that the two Android applications are executed the same way, *GreenAdvisor2.0* uses manually written use-case test scripts instead of JUnit test suite used by the original *GreenAdvisor*. *GreenAdvisor2.0* refines the blame assignment procedure of its previous version, by (a) adding application-level instrumentation, and (b) performing a richer system-call profiling. As a result, three kinds of profile events are collected when the Android application executes: timestamped system-call, timestamped method-start and timestamped method-end. Composing all these events and using statistical tests, *GreenAdvisor2.0* determines which methods are to blame for changes in the application's system-call profile.

- **Evaluation and Conclusion**: First, the situations where *GreenAdvisor2.0* may and may not work effectively were explained and then the results of four supporting experiments were presented. In these experiments, conditions in which the decisions of *GreenAdvisor2.0* can be marked as correct or incorrect were synthetically produced. Using this information, accuracy and effectiveness of *GreenAdvisor2.0* were quantified and compared to that of the original *GreenAdvisor* and random guess. The results of supporting experiments suggest that *GreenAdvisor2.0* made sensibly more correct decisions than the other two com-

petitor approaches in cases where the system-call profile was impacted significantly by a re-factoring commit which synchronously consumed more energy.

## 1.2 Contributions

This thesis makes the following important contributions:

- We correct the approach taken by the original *GreenAdvisor* for testing Android applications.
- We propose a multi-layer tracing approach which is used to collect system-call and method-call profile information. We then compose the two to blame methods for significantly changing the system-call profile.
- We propose three actions that can be taken to minimize undesired effects of instrumentation on the energy-consumption profile.
- We propose a quantification evaluation approach which is based on synthetic test-case generation and is used to compare *GreenAdvisor2.0* with the original *GreenAdvisor* and random guess.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we present an overview of the related literature. In Chapter 3, we explain our methodology in details. In Chapter 4, we evaluate our blame-assignment procedure in theory and practice. In Chapter 5, we explain the potential flaws and threats to validity. Finally, Chapter 6 concludes this thesis.

# Chapter 2

# Related Work

As we studied the background of different steps of our method, we organized the literature in two main topics: energy measurement and modeling, and empirical studies analyzing patterns of energy-consumption.

## 2.1 Energy Measurement and Modeling

This class of studies have been conducted to measure and model energy. Energy measurement could be done both physically and logically. In logical approach the amount of energy consumed is often estimated using different metrics in application, OS, and/or hardware levels. After measurement at the physical level is collected, different approaches, such as regression or other statistical analysis, may be used to build an energy model in a specific granularity level.

Hindle *et al.* proposed and implemented GreenMiner, a dedicated hardware mining software repositories testbed composed of 4 galaxy nexus phones and a Raspberry Pi [8]. The Raspberry Pi starts the test and runs a sequence of interactions (swipes and taps) on an emulator, then collects system-call profile and physical energy measurement data and uploads it onto a centralized server.

Chowdhury *et al.* used a big-data approach in which they train a model based on a set of applications' energy measurements and profile information and use that model to estimate the energy consumption of a new application for a test run [6]. In this study, they used a feature selection algorithm called

elastic net and recursive elimination to select 15 features and build a model that can estimate energy-consumption mostly within 10% error.

Mittal *et al.* presented an energy-emulation tool that enables developers to estimate the energy use of their mobile apps from within their development environment by scaling the emulated resources including the processing speed and network characteristics to match the app behavior to that on a real mobile device [14].

The Eprof platform, developed by Pathak *et al.*, uses a finite state machine to model the energy consumption of an application with power states represented as nodes and system-calls represented as the state-transition edges [15]. Based on this model and the input parameters of the system-calls invoked by each method, they estimate the energy consumption of an application at the method level. This approach, however, requires the manual modification of the application framework, a potentially time-consuming and error-prone task. Hao *et al.* presented a similar approach, Elens, which records the execution trace and run-time information (e.g. state of hardware components during execution) of an Android application and then constructs a model to estimate the energy consumption of the application in three levels of granularity including specified path, method, and line by line [7]. Both *Eprof* and *Elens* suffer from the same shortcoming: if the cause of the energy consumption is not due to the use (and state) of hardware, neither model will recognize it.

Agarwal *et al.* presented *GreenAdvisor*, a tool for analyzing energy consumption and its changes as the application evolves, based on system-call traces [1]. This approach relies on their empirical studies, which validated the energy-consumption "Rule of Thumb": When the count of a system-call changes significantly from one version to the next, the application's energy consumption will also change [2]. *GreenAdvisor* uses `strace`, a system-call trace utility, to record the count of system-call invocations in each version. When it detects a change in the count of a system-call's invocations, it uses a bag-of-words method to locate the code that is likely responsible for this change (based on associations between method names and system-call names). However, this blame-assignment method is not very robust.

Schubert *et al.* designed and developed a software energy-profiler which makes it possible to calibrate a hardware platform once (using a power meter), and then use the calibration data to obtain energy profile of the software running on that platform, without requiring the use of a power meter [18]. This study relies upon a minimal instrumentation of the kernel code (tens of lines of code) and accounts for both synchronously and asynchronously consumed energy.

Pathak *et al.* study of *Eprof* also relies upon instrumentation for obtaining profiling information and use that to model energy consumption [15]. In this study, the source-code is first instrumented for method-call and system-call tracing. The instrumented binary is then run on an instrumented mobile-platform/OS to gather both detailed method-call and system-call traces at run-time.

Li *et al.* referred to [3] in their study of *vlens* as an efficient method of instrumentation [11]. Their main strategy was to minimize the placements of probes and claimed that this significantly reduces the possible instrumentation overhead. In another study, Lu *et al.* proposed a lightweight and automatic approach to estimate the method-level energy consumption for Android apps [13]. In this approach, probes are only placed at the beginning of methods which increases the number of byte-code instructions only by 2% and take only 3% longer to execute which they claimed to be a minimal insignificant overhead.

Our approach is based on logical measurement of energy and uses system-call counts as a metric for estimating the amount of energy consumed in two different versions. We also perform application level instrumentation in which we try to reduce the system-call overhead by keeping the profile data in memory and writing it on disk only once at the end of each use-case. Not only we try to perform a low-overhead instrumentation but we also control for any unintentional affect on energy by taking a number of conservative actions explained in Chapter 3.

## 2.2 Empirical Studies Analyzing Patterns of Energy-Consumption

Another class of literature we assessed have been conducted to analyze the patterns of energy-consumption. Two main objectives of studies in this class are to identify energy leaks and find out what areas the developers should focus on more for energy-consumption optimization.

According to an empirical study on the energy-consumption of Android applications, Li *et al.* found that only 39% of energy is consumed in non-idle state [10]. This means that the other 61% is consumed in idle state where no application code is under execution. This finding has an interesting implication: bad choice of color schema could waste energy way more than a badly written code does. They also classified the consumed non-idle energy and found that 85% of it is consumed on API calls (possibly single lines of code invoking services provided by libraries or services external to the project), 13% consumed on system events (e.g. context switches, garbage collection, etc.), and only 2% on the user code. Assessing API calls, they found that network is consuming the most energy among all other services. They also found that making a HTTP request consumes significantly higher energy compared to other network steps. They also assessed developer-written code and found that a HTTP request making loop consumes more energy than a loop invoking any other API; both these loops consume more energy than a loop without any API calls.

On a similar study, Linares-Vasquez *et al.* mined and analyzed 55 mobile apps for energy-greedy APIs and usage patterns [12]. They discussed the cases where either the anomalous energy consumption is unavoidable or where it is due to sub-optimal usage or choice of APIs and finally provided a recipe for Android developers to reduce energy-consumption while using certain categories of Android APIs and code patterns.

In another empirical study, Li *et al.* investigated the impact of different coding practices that are commonly suggested or proposed in the official Android developers web site on energy consumption [9]. They mainly focused

on three categories: network usage, memory consumption, and low-level programming practices and found some interesting results about each. In the first category, network usage, they found that the energy consumption of downloading 1000 bytes of data is roughly the same as downloading 1 byte; therefore, an energy-efficient programming practice would be to bundle several small HTTP requests into larger ones whenever it is possible. In the second category, memory consumption, they found that high memory usage consumes more allocation energy and may cause the application to become more energy consuming. On the flip side, insufficient memory may cause more energy consumption in other components of the smart phone, such as 3G or WiFi network. Thus, developers need to have guidelines to help them decide what is the appropriate amount of memory to allocate. Finally, in the third category, low-level programming practices, they found that avoiding references to array length in a loop reduced energy by 10%, static invocations consumed 15% less energy than virtual invocations, and direct field-accesses used 30-35% less energy than indirect getter and setter methods.

Mining Stack Overflow for energy related discussions, Pinto *et al.* created and analyzed a data-set of more than 300 questions and found that developers do not have the necessary tools and knowledge for energy-aware development [17]. Wilke *et al.* also mined user feedback on Google Play and found that energy inefficiency negatively impacts the user ratings of both free and paid applications, which suggests a general disregard for energy-efficiency driven development among Android developers [19]. They also identified the major causes of energy inefficiency of many Android applications to be background activities when the app is minimized, faulty GPS behaviour, unnecessary CPU and RAM activities, and synchronization attempt with the Internet when the device is not connected to any network.

Pathak *et al.* conducted a study that introduces and characterizes an energy bug called no-sleep bug. This bug arises from mishandling power control APIs. It keeps the components on during the active use of the app and results in significant and unexpected battery drainage [16]. They also introduced asynchronous energy behaviour as a challenge for energy modeling in

the granularity level of functions in another study [15]. This hard-to-track behaviour could happen in three cases:

*Tail Power State:* GPS, WiFi, SDCard, 3G components could enter in high power state during the execution of a routine and stay in that state long after the routine ends.

*Wakelocks:* During execution of a routine, a wakelock may be acquired. It can cause some components to enter a high power state and stay in that state until the lock is released in possibly another routine.

*Exotic Components:* Components such as camera which drain energy when they are switched on and they continue until they are switched off in possibly another routine.

Bao *et al.* performed an empirical study by mining power-management commits and found that for different kinds of Android application (e.g., Games, Connectivity, Navigation, Internet, Phone & SMS, Time, etc.), the dominant power-management activities differ [4]. This means that the few developers that focus on energy-efficiency while developing apps tend to focus on different areas depending on the type of app they are developing. For example, a developer who is creating a gaming app should focus more on the color schema and memory optimization, while a developer who is creating a messaging app should focus more on network usage optimization.

Chan *et al.* used real network and application measurements to comprehensively analyze the energy consumption of 12 common mobile applications by breaking down their total energy consumption into data and signaling (due to LTE signaling) energy components [5]. The results of this study show that signaling energy consumption may become a major concern for mobile carriers.

The aforementioned empirical studies suggest that bad programming practices such as careless use of APIs could vastly affect the application's energy consumption, the problem that our tool, the *GreenAdvisor2.0*, is aimed to solve.

## 2.3 Chapter Summary

In this chapter, literature related to this thesis was discussed. The literature is organized in three main topics: empirical studies analyzing patterns of energy-consumption, energy metrics and modeling, and instrumentation and profiling.

The literature reviewed in the first topic pursue two main objectives: identifying energy leaks, and finding out what areas the developers should focus on more for energy-consumption optimization.

The literature reviewed in the second topic suggest that energy measurement could be done both physically and logically and shine light on the difference between the two.

Finally, the literature in the third topic explain instrumentation-based profiling and the measures taken to control its undesired impact on the energy-consumption.

# Chapter 3

# Methodology

*GreenAdvisor2.0* extends the original *GreenAdvisor* in two dimensions: (a) it improves on its testing procedure, and (b) it improves the reliability and accuracy of its blame-assignment feature.

## 3.1 The Original GreenAdvisor

For the sake of security, consistency, code reuse, and developers' convenience, operating systems provide access to different services and underlying hardware components through a set of predefined functions called *system-calls*. Applications residing in user space use these system-calls to access resources such as hard-disk, memory, sensors, camera, and other periferals as well as services such as creating, executing, management, and transferring data between processes, receiving event notifications, etc. Figure 3.1 shows an overview of this interaction between user and kernel spaces.

During the evolution of an Android project, every commit that makes a change in the code of a use-case could also make a change in the system-call profile of that use-case. In a study that focuses on the system-call profile and its relationship with energy profile, Agarwal *et al.* prove that there is a correlation between the two [2]. For this purpose, system-call counts were measured for different versions of two Android applications, Firefox and Calculator. Then linear and regression models were built using these measurements to generate estimations about energy consumption which led into a classification of versions as being low/high energy consuming. Finally,

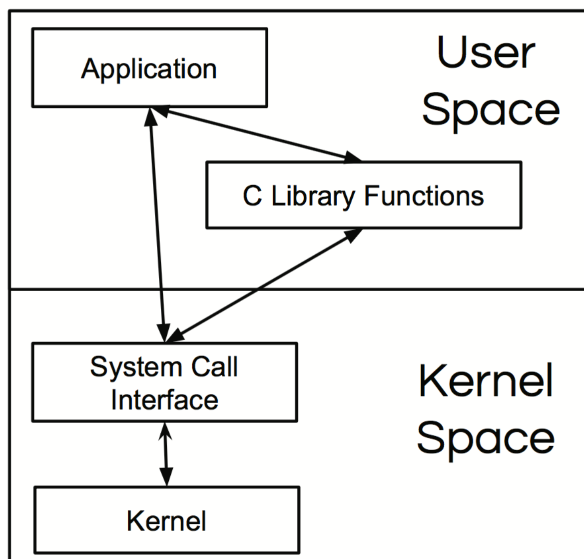


Figure 3.1: Interaction of applications, C libraries, and kernel through system-calls

by analyzing energy and system-call profiles, they introduced *The Rule of Thumb* which states the following:

*"If the system-call profile changes significantly from the previous version, it is probable that the application's energy consumption has changed as well."* [2]

To evaluate this rule, Agarwal *et al.* computed four different metrics: $Precision$, $Recall$, $Specificity$, and $F_1$:

$$Precision = \frac{SS}{SS + SN}$$

$$Recall = \frac{SS}{SS + NS}$$

$$Specifity = \frac{NN}{NN + SN}$$

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

In the above equations, $SS$ is the number of times that both the changes in energy consumption and system-call counts are significant; $NS$ is the number

of times the change in energy consumption is significant, while there is no significant change in counts of system-calls; $SN$ is the number of times the system-call profile changes significantly but the energy consumption does not; finally, $NN$ is the number of times that neither of the two change significantly.

A high $Precision$ indicates that a significant change in system-call profile leads to a significant change in energy profile. A high $Recall$ indicates that the cases where significant energy consumption changes were observed, co-occur with the cases where system-call profile significantly changes. A high $Specificity$ indicates that the *The Rule of Thumb* may produce a few false positives and finally $F_1$ is a measure of accuracy; the higher the $F_1$ the more balanced and accurate the model is.

Observing $Precision$, $Recall$, and $F_1$ being much higher than random guess and a quite high $Specificity$; Agarwal *et al.* concluded that the *The Rule of Thumb* should hold in majority of cases. They used this conclusion as the basis of *GreenAdvisor* which is primarily aimed to predict energy changes without having to rely upon hardware-based instrumentation.

*GreenAdvisor* runs $jUnit$ test suite of two versions of an application and looks for changes in the system-call profile. Since the $jUnit$ test suite of an application is supposed to evolve through time, *GreenAdvisor* would sometimes incorrectly attribute the change in system-call profile only to the evolution of source code, ignoring the fact that the change could have been because of a modification in the test suite.

Since *GreenAdvisor* requires system-call profiles to identify energy-consumption changes, it has to run the application versions under a specific test. It is important to note that if the testing procedure driving the earlier application version is different from that of the subsequent version (i.e. different set of application methods are called or some of the methods are called different number of times), the energy-consumption profile will change too. If such a change co-occurs with changes in the application's source code (e.g. a refactoring commit), the detection of code to blame becomes complicated because we can not tell with certainty which of the two changes (e.g. change in

testing procedure or change in the application's source code) have caused the change in the system-call invocation and eventually energy profile. Therefore, the assumption that the change in system-call profile can be attributed only to the evolution of source code would be a valid assumption only if the test script for driving both versions are exactly similar.

After *GreenAdvisor* finds the system-calls whose counts were significantly changed, it uses a bag of words, a dictionary containing 24 system-call names and their associated regular expressions, to identify the code responsible for changes in the system-call profile from the earlier application version to the subsequent. These regular expressions capture the invocation of methods whose name include particular keywords which are thought to be verbally or semantically connected to the name of the system-call to blame (i.e. the system-call whose count has been significantly changed).

This blame assignment methodology is argued to suffer several shortcomings:

***(A) The bag-of-word does not include the system-call to blame:*** Due to the fact that the bag-of-word is a predefined dictionary, it might not contain all the system-calls defined by the operating system. The associated regular expressions could also be incomplete or incorrect. Agarwal *et al.* also recognized this issue in their study [1].

***(B) The regular expressions might match undesired code segments:*** For example the word *Thread* is matched when the system-call to blame is *read* because the latter is a sub-string of the former.

***(C) The responsible code segment might not include a relevant keyword:*** It is possible that there is no verbal or semantic connection that can be traced to identify the code-to-blame.

## 3.2 GreenAdvisor2.0

The implementation of *GreenAdvisor2.0* relies upon the addition of two profiling functionalities to the original *GreenAdvisor*: *Timestamped System-Call Profile* and *Method Call Profile*.

### 3.2.1 Workflow

Figure 3.2, shows *GreenAdvisor2.0*'s workflow of detecting and localizing changes in the energy profile of an Android application which consists of four main steps:

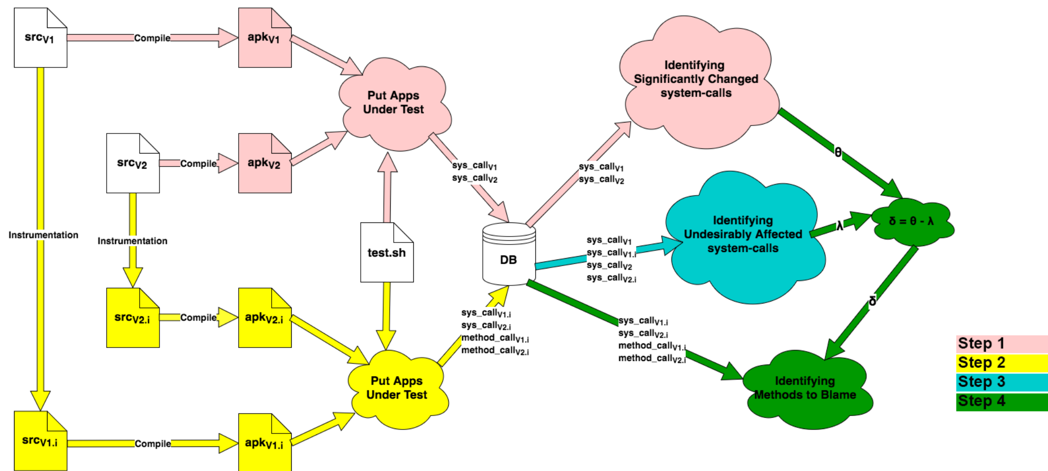


Figure 3.2: GreenAdvisor2.0 Workflow

***(Step 1) Finding Significantly Changed System-calls:*** Each of the two versions of the Android application under test, $src_{v1}$ (the version before refactoring commit) and $src_{v2}$ (the version after refactoring commit), are compiled and two `apk` files, $apk_{v1}$ and $apk_{v2}$, are generated. Then test cycles start with a call to `startTestCycle` service of the Core. The $apk_{v1}$ file, `strace` package files, and the test script are sent over to the Emulator. Any installed version of the application as well as old profile data (which may have been made during previous runs) are deleted from the Emulator. The `apk` file is installed. The `strace` utility is executed with appropriate option parameters to listen for the application under test. Then the test script is executed five times on version 1.

Each time, the profile data file produced which contains counts of system-calls invoked is downloaded and kept in the database. We call the set of these five profile data files $sys_call_{v1}$ throughout this thesis. Then the exact same steps are taken to produce and download profile data files for the second application version, $sys_call_{v2}$. Figure 3.3 shows an example data file in these sets.

```
1  {
2    "getsockname": {
3      "count": 24
4    },
5    "fchmodat": {
6      "count": 41
7    },
8    "getppid": {
9      "count": 245
10   },
11   ...
12 }
```

Figure 3.3: An Example Data File in $sys_call_{v1}$ and $sys_call_{v2}$

Retrieving $sys_call_{v1}$ and $sys_call_{v2}$ from database, for each system-call $s$ invoked we construct two tuples, $T_{s.v1}$ and $T_{s.v2}$. $T_{s.v1}$ consists of 5 counts each representing the number of times $s$ is invoked in an execution of version 1 and $T_{s.v2}$ consists of 5 counts each representing the number of times $s$ is invoked in an execution of version 2. In order to assess whether or not $T_{s.v2}$ is significantly different from $T_{s.v1}$, *GreenAdvisor2.0* uses *Student's T-Test* to find the *P-value* of change and the *Bonferroni* correction to determine its significance. Student's T-Test is a statistical hypothesis test which can be used to determine if two sets of data are significantly different from each other. Statistical hypothesis testing is based on rejecting the null hypothesis if the likelihood of the observed data under the null hypotheses is low. If multiple hypotheses are tested, the chance of a rare event increases, and therefore, the likelihood of incorrectly rejecting a null hypothesis increases. The Bonferroni correction compensates for that increase by testing each individual hypothesis at a significance level of $\alpha/m$, where $\alpha$ is the desired overall alpha level and $m$ is the number of hypotheses. For example, if a trial is testing $m = 20$

hypotheses with a desired $\alpha = 0.05$, then the Bonferroni correction would test each individual hypothesis at $\alpha = 0.05/20 = 0.0025$. By performing Bonferroni correction we are being more conservative in recognizing changes as being significant. We call the set of significantly changed system-calls obtained in this step $\theta$ throughout this thesis.

> $\theta$ is the set of significantly changed system-calls, obtained by comparing counts of system-calls in runs of version 1 with that of runs of version 2

***(Step 2) Instrumentation and Execution of Instrumented Versions:*** Both versions of the source code, $src_{v1}$ and $src_{v2}$, are instrumented and produce $src_{v1.i}$ and $src_{v2.i}$. As discussed earlier, only those methods that have changed between the two versions are instrumented at this step. The instrumented sources are then compiled into apks $apk_{v1.i}$ and $apk_{v2.i}$ which are executed five times each using the same test script as before. The execution of an instrumented apk produces a method-call profile in addition to a new system-call profile which includes timestamps for each system-call invocation. Both method-call profiles, $method_call_{v1.i}$ and $method_call_{v2.i}$, and both system-call profiles, $sys_call_{v1.i}$ and $sys_call_{v2.i}$, of the instrumented versions are also stored in the database. Figure 3.4 and 3.5 show example data files in these sets.

***(Step 3) Finding Undesirably Affected System-calls:*** Retrieving $sys_call_{v1}$, $sys_call_{v1.i}$, $sys_call_{v2}$, and $sys_call_{v2.i}$ from database, this time four tuples are constructed, $T_{s.v1}$, $T_{s.v1.i}$, $T_{s.v2}$, and $T_{s.v2.i}$ for each system-call $s$ invoked, similar to the tuples constructed in the first step. Then by performing a pairwise difference between $T_{s.v1}$ and $T_{s.v1.i}$ a new tuple $I_{s.v1}$ is obtained which indicates the impact of instrumentation on the invocation of $s$ in version 1:

$$I_{s.v1} = T_{s.v1.i} - T_{s.v1}$$

Similarly we can obtain $I_{s.v2}$ which indicates the impact of instrumentation on the invocation of $s$ in version 2:

```json
{
  "getsockname": {
    "invocation_timestamps": [
      "1489724535264800338",
      "1489724541050482301",
      "1489724551560480054"
    ],
    "count": 3
  },
  "rt_sigtimedwait": {
    "invocation_timestamps": [
      "1489724513783194509"
    ],
    "count": 1
  },
  ...
}
```

Figure 3.4: An Example Data File in $sys_call_{v1.i}$ and $sys_call_{v2.i}$

$$I_{s.v2} = T_{s.v2.i} - T_{s.v2}$$

As discussed earlier, if $I_{s.v1}$ and $I_{s.v2}$ are significantly different, then the timestamped system-call profile recorded for $s$ could not be trusted in the process of blame assignment. In this case, we claim that $s$ is undesirably affected by instrumentation and we call the set of undesirably affected system-calls $\lambda$ throughout this paper.

> $\lambda$ is the set of undesirably affected system-calls by instrumentation, obtained by comparing the impact of instrumentation on counts of system-calls in runs of version 1 with that of runs of version 2

***(Step 4) Blaming Methods:*** Subtracting set $\lambda$ from set $\theta$, we obtain a new set of system-calls whose counts in executions of version 1 were significantly different from that of version 2 but were not undesirably affected by instrumentation. We call this set $\delta$:

$$\delta = \theta - \lambda$$

```
1  [
2    {
3      "timestamp": "1490338868838096847",
4      "method_name": "de.danoeh.antennapod.adapter.NavListAdapter:getCount",
5      "type": "start"
6    },
7    {
8      "timestamp": "1490338869647096495",
9      "method_name": "de.danoeh.antennapod.adapter.NavListAdapter:getCount",
10     "type": "end"
11   },
12   {
13     "timestamp": "1490338869648437003",
14     "method_name": "de.danoeh.antennapod.activity.MainActivity:loadData",
15     "type": "start"
16   },
17   ...
18 ]
```

Figure 3.5: An Example Data File in $method_call_{v1.i}$ and $method_call_{v2.i}$

> $\delta$ is the set of system-calls whose count were significantly changed but were not undesirably affected by instrumentation, obtained by subtracting $\lambda$ from $\theta$

It is now time to identify methods which may have been responsible for the change in the count of system-calls in set $\delta$. For each system-call $s$ in set $\delta$ and method $m$ in the application's source code, a question should be answered:

*"Does method $m$ invoke system-call $s$ significantly more (or less) times during the executions of version 1 compared to the executions of version 2?"*

Retrieving $sys_call_{v1.i}$, $method_call_{v1.i}$, $sys_call_{v2.i}$, and $method_call_{v2.i}$ from database, we have three kinds of events: timestamped system-call invocations, timestamped method starts, and timestamped method ends. Since all these events are timestamped we could attribute the invocations of system-calls to the methods. Using all this information, for each system-call $s$ in set $\delta$ and method $m$ in the application's source code, two tuples are constructed, $H_{s.m.v1}$ and $H_{s.m.v2}$. $H_{s.m.v1}$ consists of 5 counts, each representing the number

of times $s$ is invoked by $m$ in an execution of version 1, and $H_{s.m.v2}$ consists of 5 counts, each representing the number of times $s$ is invoked by $m$ in an execution of version 2. Finally we use *Student's T-Test* and *Bonferroni* correction to determine if $H_{s.m.v1}$ is significantly different from $H_{s.m.v2}$. If they are significantly different, then *GreenAdvisor2.0* introduces $m$ as a method to blame for the change in counts of $s$.

While *GreenAdvisor* only uses counts of system-calls, *GreenAdvisor2.0* requires the timestamps for each system-call invocation, which is achieved by running the `strace` listener with different option parameters; this, in effect, simply involves changing the bash script that configures the `strace` execution options.

Recording the timestamps at the beginning and the end of each method invocation is a more complex endeavor. Since we are envisioning that the *GreenAdvisor2.0* is used by developers who have access to the source code, *GreenAdvisor2.0* uses a Java source file manipulation library that allows easy parsing and formatting of methods within java source files. More specifically, *GreenAdvisor2.0* uses *Roaster* to instrument the source files of the application under test. When this instrumented application is executed it emits a log with the timestamps of the entry and exit boundaries of all method invocations. In order to distinguish between these events, which are really close to each other, timestamps must be recorded in absolute microseconds, which is accomplished by instrumenting the beginning of each method with calls to the *System.currentTimeMillis()* and *System.nanoTime()* methods. The former reports the current absolute time in milliseconds, and the latter reports the current time in nanoseconds, relative to the time of JVM boot up. Composing these two timestamps enables us to compute the times of all method-call boundary hits in absolute microseconds. However, since the reference timestamp of this calculation has an actual accuracy of milliseconds, it is, in principle, possible that all inferred timestamps contain a random error between 0 and 999 microseconds. This error is expected to decrease, since *GreenAdvisor2.0* executes the test script multiple times in order to get an average of the system-call counts which is more reliable. Figure 3.6 shows an example

method which was instrumented.

```
1   public void example() {
2     _Logger.keepLog(
3       "(" + System.currentTimeMillis() + ")" + System.nanoTime() +
4       ":de.danoeh.antennapod.activity.MainActivity:loadData:start\n"
5     );
6     try {
7       // ...
8       // The Original Method Body
9       // ...
10    } finally {
11      _Logger.keepLog(System.nanoTime()
12          + ":de.danoeh.antennapod.activity.MainActivity:loadData:end\n");
13    }
14  }
```

Figure 3.6: An Example of Instrumentation in a Method

As discussed in section 1, *GreenAdvisor2.0* uses an application-level instrumentation to obtain timing information of method executions. By definition, this means changing the application code which will most likely cause a change in system-call and eventually energy profiles. This undesired impact is called *Heisenberg Effect*.

The instrumentation could have three different effects on the number of times system-call $s$ is invoked:

***No Effect on Either of the Versions:*** The instrumentation does not change the count of $s$ in runs of version 1 and 2. In this case, the timestamped system-call profile produced could be trusted in the process of blame assignment.

***Same Effect on Both Versions:*** The instrumentation changes the count of $s$ in runs of version 1 and 2 but does not have a significantly different effect on the runs of version 1 compared to the runs of version 2. In other words, the difference between count of $s$ in version 1 and count of $s$ in version 2 before the instrumentation is not significantly changed after the instrumentation. Therefore, the timestamped system-call profile produced could again be

trusted in the process of blame assignment.

***Significantly Different Effect on the Two Versions:*** The problem arises when the instrumentation has a significantly different effect on the count of $s$ in runs of version 1 compared to the runs of version 2. In this case, if invocations of $s$ is increased in method $m$, it is not clear whether the increase is caused by the instrumentation or a change in the code of method $m$. Therefore, the timestamped system-call profile produced could not be trusted in the process of blame assignment.

In order to deal with this challenge, *GreenAdvisor2.0* takes several actions to minimize the Heisenberg effect:

***Keep Logs in Memory, Write Only Once:*** Instead of writing the timing information of every method on disk at the time of method returns like what Hao *et al.* did in their study of Elens [7], we tend to keep all the information in memory and write everything only once after the test script is finished executing. We expect that this optimization significantly decreases the invocation of system-calls which are made for writing data on disk during the execution of instrumented application.

***Instrument Only Changed Methods:*** Instead of instrumenting all the methods, we only instrument the ones whose code was touched since the last version. This is reasonable because we only expect the methods whose code was touched to be the cause of changes in system-call and energy profiles.

***Do Not Blame Methods Based on System-calls on Which Instrumentation Had Significantly Different Effect:*** Finally, comparing the effects of instrumentation on the count of system-calls in version 1 to that of version 2, *GreenAdvisor2.0* finds system-calls which were affected undesirably due to the instrumentation and does not look for the cause of any change on the count of these system-calls. This way we make sure when

*GreenAdvisor2.0* blames a method for increasing the count of a system-call, it could not have been due to the instrumentation.

Towards correcting the testing procedure of *GreenAdvisor*, *GreenAdvisor2.0* uses use-case-specific manually written test scripts to execute the application under test. A manually written script is a set of bash commands that uses *adb* to interact with the emulator and sends tap, swipe and key-press events to the application. We used Android Developer Options, found in Android settings, to show pointer locations which helped us manually write these scripts.

### 3.2.2 Architecture

Figure 3.7 shows a composition of different components in our implementation of *GreenAdvisor2.0* .

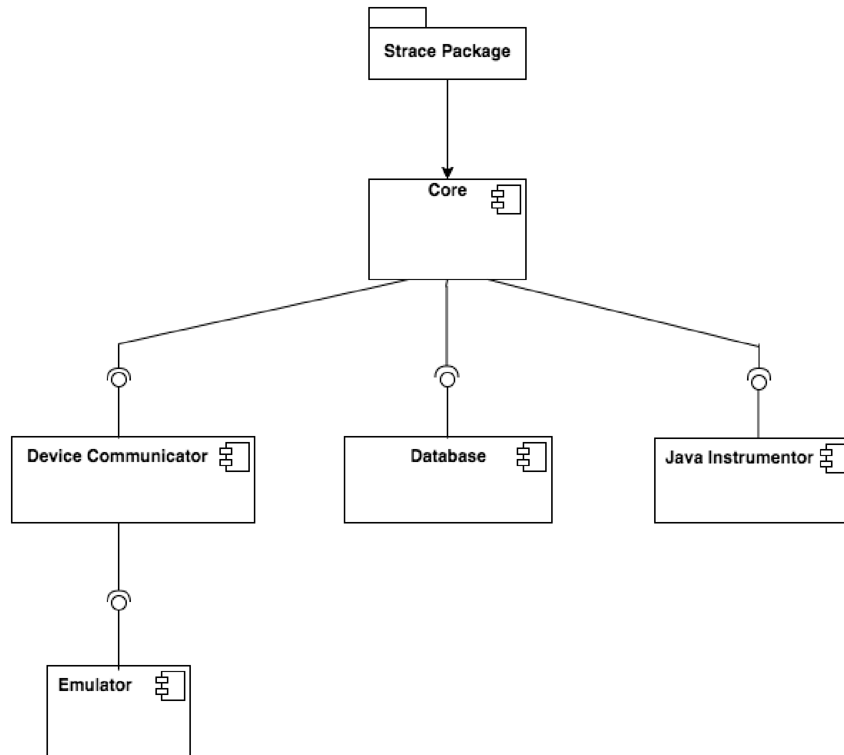


Figure 3.7: GreenAdvisor2.0 Components

**Core:** This component provides the interface for working with *GreenAdvisor2.0* . It sits in the middle and talks with other components to offer services like running test cycles, instrumenting projects, and retrieving profile information and assigning blame to methods.

**Java Instrumentor:** This component is used to instrument a given set of methods in a given project.

**Device Communicator:** This component uses `adb` (Android Debug Bridge) to communicate with the emulator. It offers services such as downloading or uploading files, installing or uninstalling applications, opening or closing processes, and running shell commands on the emulator.

**Emulator:** This component is a virtual Android device, which is used for developing and testing purposes.

**Database:** Implemented with SQLite, this is the repository of all information, recorded through the execution of the various application versions.

**Strace Package:** This package contains the executable file of the `strace` utility and a bash script that executes it with appropriate options to record the system-call profile of the application under test.

Figure 3.8 shows an example output of *GreenAdvisor2.0* blaming methods for changing invocation count of a particular system-call.

## 3.3 Chapter Summary

This chapter, first reviews the original *GreenAdvisor* and then explains the improvements applied by *GreenAdvisor2.0*.

The original *GreenAdvisor* predicted energy-consumption changes by looking at the changes in the counts of system-calls and then attempts to find the

```
1      SYSCALL TO BLAME: mprotect
2        TYPE OF CHANGE: Increase ~ SUM [873.0] -> [53949.0]
3     P-VALUE OF CHANGE: 8.73E-9
4  BONFERRONI THRESHOLD: 0.1 / 20 = 0.005
5      METHODS TO BLAME:
6  1 - de.danoeh.antennapod.fragment.QueueFragment:saveScrollPosition
7                      P-VALUE: 6.391E-17
8    CHANGE OF INVOCATION AVG: [0.0] -> [3501.8]
9  2 - de.danoeh.antennapod.adapter.NavListAdapter:getFeedView
10                     P-VALUE: 6.395E-17
11   CHANGE OF INVOCATION AVG: [0.0] -> [3500.4]
12 3 - de.danoeh.antennapod.activity.MainActivity:handleNavIntent
13                     P-VALUE: 6.395E-17
14   CHANGE OF INVOCATION AVG: [0.0] -> [3500.1]
15 ...
```

Figure 3.8: An Example output of *GreenAdvisor2.0*

code responsible for those changes using verbal connections between the name of system-calls and code of methods. *GreenAdvisor2.0* refines the blame assignment procedure of its previous version, the original *GreenAdvisor*, by (a) adding application-level instrumentation, and (b) performing a richer system-call profiling. *GreenAdvisor2.0* also controls for undesired impacts of the instrumentation on the application's energy profile by identifying the system-calls whose count might have changed due to the instrumentation.

# Chapter 4

# Evaluation

As discussed earlier, *GreenAdvisor2.0* puts the selected Android application under a test in order to obtain system-call and method-call profile data. We established a number of criteria that our chosen application had to meet:

1. Our first criterion was that it had to be open-source. This was because *GreenAdvisor2.0* relies on source-code instrumentation.

2. Our second criterion was that the application had to be well-known. This was because we wanted to avoid running into unexpected crashes due to poor code quality as much as possible. We referred to the number of active installs and user rating on Google Play as a measure of user satisfaction which we thought could indirectly imply the quality of the application.

3. Our third and last criterion was that it had to be large in size and it had to have a relatively high complexity. This was because we wanted the application to already have a rich system-call profile and energy consuming features so that our synthetic additions of code will not be the only cause of system-call invocations and energy consumption. We relied on our own subjective judgment as software engineers to decide whether or not an Android project is large in size and contain enough energy-consuming features.

The Android application we selected for this purpose is called AntennaPod. It is a podcast manager and player that gives the user instant access to millions

of free and paid pod-casts. This application provides some potentially energy-consuming features, including live streaming, downloading, audio playback, and management of a cache of local library of episodes and pod-casts. These features rely on network access, disk management, and IO control (e.g. phone speaker). By the time we conducted our experiments, AntennaPod was rated 4.6 by 12K users on Google play and had been actively installed on between 100K to 500K Android devices.

The test we wrote, is a usual use-case of the application that invokes 165 unique methods out of a total of 729 unique methods which existed in the source code of the latest version of application by the date we performed the experiments (Feb 2017). We argue that the application developer is best suited for creating this test because they know what part of the application interface they should interact in order to cover the method whose energy change they are about to assess.

The flow of this test is as follows:

```
1  Start the main activity
2  Tap on Subscriptions
3  Tap on Add Podcast button
4  Tap of Search From Itunes button
5  Tap on the first item in the list
6  Tap on Subscribe button
7  Tap on back arrow
8  Tap on back arrow
9  Tap on back arrow
10 Tap on Podcast to open
11 Tap on the first track
12 Tap on the Stream button to start listening
```

For each experiment, we selected 10 application methods for injecting system-call-free code snippets. We call this set $A$. we injected each method in set $A$ with one of the code snippets described below:

```
(a) Append 100 String values to an ArrayList object 100 times

(b) Insert at the beginning of an ArrayList object and shift
    all items 100 times

(c) Sort an Array of 100 Integers 100 times using bubble-sort
    algorithm
```

(d) `Sort an Array of 100 Strings 100 times using bubble-sort algorithm`

(e) `Put 100 elements in a HashMap and traverse the map 100 times`

(f) `Multiply two 10x10 matrices 100 times`

(g) `Add 100 elements to a PriorityQueue and pop all of them 100 times`

(h) `Calculate Fibonacci of 1000 using array implementation 100 times`

(i) `Find and replace a certain String in a list of 100 Strings 100 times`

(j) `Append a list of 100 Strings to a String 100 times`

We also selected 10 application methods for injecting system-call-producing code snippets. We call that set $B$. Methods in set $A$ and $B$ are invoked anywhere from 1 to 85 times during each execution of the test script.

Since the goal of *GreenAdvisor2.0* is to point to the methods which actually impact the system-call profile, for each experiment, *GreenAdvisor2.0* injects the same system-call-producing code snippet to all methods in set $B$. This is to make sure that all methods in set $B$ are equally involved in any impact on the system-call profile. Therefore, our experiments are characterized by the choice of system-call-producing code snippets we injected in the methods of set $B$.

It is now clear that in every experiment, *GreenAdvisor2.0* should blame all the methods in set $B$ and not blame any method in set $A$. The reason for this, is all methods in set $B$ contain system-call-producing code snippets which will make changes to the system-call profile where as the system-call profile of methods in set $A$ should not have been changed. As a result, True-Positive, False-Positive, True-Negative, and False-Negative outcomes for each experiment can be counted and is defined as follows:

**True-Positive:** If a method in set $B$ is blamed for increasing the invocations of any system-call in $\delta$

**False-Positive:** If a method in set $A$ is blamed for increasing the invocations

of any system-call in $\delta$

**True-Negative:** If a method in set $A$ is not blamed for increasing the invocations of any system-call in $\delta$

**False-Negative:** If a method in set $B$ is not blamed for increasing the invocations of any system-call in $\delta$

Using the counts above, we then calculate *Accuracy*, *Precision*, *Recall* and $F_1$ measures:

**Accuracy:** Out of all decisions regarding blaming or not blaming a method, what percentage are actually true decisions

**Precision:** Out of all blames, what percentage are actually true blames

**Recall:** Out of all those decisions which should have been a blame, what percentage were actually a blame

**$\mathbf{F_1}$:** This measure conveys a balance between *Precision* and *Recall*.

### 4.0.1 Results and Analysis

Below we examine the research questions mentioned earlier and then analyze the result of supporting experiment(s) for each:

**Q1: Does the new approach of blame assignment work when the system-call profile is impacted by a re-factoring commit which synchronously consumes more energy?**

The *GreenAdvisor2.0* works based on the *Rule of Thumb*, which correlates the changes in energy profile to that of system-call profile. Therefore, if more energy is consumed during the life-time of a method, this change should be reflected in the application's system-call profile throughout the execution of that method. Since *GreenAdvisor2.0* records the times of both system-calls and method-calls, it should be able to attribute the invocation of system-calls to method-calls and eventually find out which methods where involved in a change of counts of a system-call.

In order to support our theoretical answer for the question Q1, we designed experiment I in which we injected a code snippet that makes a long list of system-calls to perform a `ls` (list segments) command. This code invokes a `ls` command 10 times in a loop. Each time a `ls` command is issued, a new process is forked, one (or more) context switches occurs, and finally the result is accessed by reading the process `InputStream`. These steps involve a number of OS operations which significantly impact the application's system-call profile. We ensured that the energy is consumed synchronously in this case by waiting for the forked process to finish.

The calculated measures for this experiment is reported in Table 4.1.

Table 4.1: Calculated Measures for Experiment I

| Method | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| GA2.0 | 92% | 89% | 96% | 92% |
| GA | 54% | 93% | 10% | 18% |
| Random | 49% | 49% | 49% | 49% |

In this table and future tables, GA2.0 refers to GreenAdvisor2.0, GA refers to the original GreenAdvisor, and finally Random is a method in which for every system-call $s$ in $\delta$ and method $m$ we flipped a coin to randomly decide whether or not $m$ should be blamed for changing counts of $s$.

**Q2: Does the new approach of blame assignment work when the system-call profile is impacted by a re-factoring commit which asynchronously consumes more energy?**

To answer this question, we argue that the asynchronous energy can be of two types: Interior, and Exterior. Interior asynchronous energy-consumption is when the consumption of energy may happen outside the life time of the method but is bound to the context of application. For example, in multi-thread programming, threads are owned by the application's process and whatever impact they might have on the system-call profile can be captured using profilers like `strace`. However, Exterior asynchronous

energy-consumption is when the consumption of energy happens outside the context of application. For example, Android lets developers set event-handler methods for receiving GPS location update interrupts. Such cases are not reflected in the application's system-call profile and thus cannot be captured using profilers like `strace`. Therefore, we admit that *GreenAdvisor2.0* fails in identifying the cause of exterior asynchronous energy-consumption.

In order to support our theoretical answer for the question Q2, we designed experiments II in which we injected a code snippet that requests a single GPS location update 10 times in a loop. We believe that the energy is consumed asynchronously in this case because Android requires passing an interrupt handler method to deal with GPS location updates, thus system-call profile is impacted outside the lifetime of the method that passes the interrupt handler to the Android interface. The calculated measures for this experiment is reported in Table 4.2.

Table 4.2: Calculated Measures for Experiment II

| Method | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| GA2.0 | 42% | 20% | 5% | 8% |
| GA | 61% | 90% | 25% | 39% |
| Random | 49% | 49% | 49% | 49% |

**Q3: Does the new approach of blame assignment work when the system-call profile is impacted by a re-factoring commit which we may not be able to categorize as completely synchronous or asynchronous**

If the energy-consumption of an asynchronously coded task changes, it is not guaranteed that the change is reflected and can be captured in the system-call profile of the method that wraps the task. Therefore, the *GreenAdvisor2.0* might make mistake in deciding whether or not a method increases invocations of a system-call.

Another potential problem is that if the system-call profile of the whole application significantly changes during an execution of a test, it is likely that

a set of methods evenly contribute to this significant change and the system-call profile of none of these methods has changed significantly.

Depending on the way a task is handled by Android, the energy of the task may be consumed synchronously or asynchronously. Examples of this would be sending HTTP requests and accessing memory stick for read and write operations. For the case of sending HTTP requests, Android only allows asynchronous programming (i.e. Threads) in order for the UI to be responsive during the life-time of HTTP requests. However, for the case of accessing memory-stick, Android does not necessarily require asynchronous programming. It is also worth mentioning that the system-call profile of a task is not necessarily stable every time the task is executed. Agarwal *et al.* also recognized this issue in their study [1].

In conclusion, it can not be expected that *GreenAdvisor2.0* works 100% accurate in case of all the common Java API calls since it is not always guaranteed that the change in energy profile is reflected in the system-call profile.

In order to support our theoretical answer for the question Q3, we designed experiments III and IV. In experiment III, we injected a code snippet that makes a `HTTP GET` request to a RESTful API 10 times in a loop and in experiment IV, we injected a code snippet that writes a string in a file on disk 10 times in a loop. The calculated measures for experiments II and III are reported in Tables 4.3 and 4.4 respectively.

Table 4.3: Calculated Measures for Experiment III

| Method | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| GA2.0 | 78% | 91% | 62% | 74% |
| GA | 54% | 100% | 8% | 15% |
| Random | 49% | 49% | 49% | 49% |

Table 4.4: Calculated Measures for Experiment IV

| Method | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| GA2.0 | 69% | 90% | 41% | 57% |
| GA | 53% | 100% | 6% | 11% |
| Random | 49% | 49% | 49% | 49% |

The reported values for the *Accuracy* across all four experiments suggest

that *GreenAdvisor2.0* makes sensibly more correct decisions than the other two methods if the change in energy profile is reflected in the system-call profile and the energy is consumed synchronously.

The reported values for *Precision* of both *GreenAdvisor* and *GreenAdvisor2.0* across the first three experiments are quite high. However, the high Precision of the *GreenAdvisor* is less valuable than that of *GreenAdvisor2.0*, since the *GreenAdvisor*'s *Recall* is always very low which means that it rarely blames methods which should have been blamed. The reported $F_1$ measures across the first three experiments suggest that making a balance between *Precision* and *Recall*, our method is better than random guess and definitely better than *GreenAdvisor* in those specific cases.

## 4.1 Chapter Summary

This chapter, evaluates the effectiveness of *GreenAdvisor2.0* by presenting the results of four synthetic experiments that are designed to support the theoretical answers to the questions explored in this research.

The results reported for these experiments suggest that *GreenAdvisor2.0* makes sensibly more correct decisions compared to the original *GreenAdvisor* and random guess if the change in energy profile is reflected in the system-call profile.

# Chapter 5

# Threats to Validity

In this work, internal validity in threatened by the choice of application that we perform our experiments on. In particular, we subjectively decided whether the application we are testing is large in size and already consists of enough energy-consuming use-cases which might become problematic as testing a not complex enough application might bias the decisions made by *GreenAdvisor2.0* towards correct ones. Another threat to internal validity was our use of Java Virtual Machine absolute boot-up-time as the basis for converting relative times of system-call and method-call invocations to more accurate absolute times. This may have resulted in a global shift of at most 999 microseconds in all recorded timestamps which might lead into failure in identification of method(s) invoking a system-call. Internal validity is also jeopardized in cases where Linux kernel moves the code of a frequently used system-call routine to the application's space. In these cases, the application does not require to make a call to the kernel code thus the evidence of energy-consumption (e.g. system-call profile) is compromised.

External validity is threatened by our choice of system-call-producing code snippets injected in cases described by the research questions. For example, the behaviour of `ls` command in experiment I cannot be generalized to all kinds of other re-factoring commits which synchronously consume more energy. In fact, system-call profile is not a very stable metric by nature. Agarwal *et al.* admitted this issue in their study [1] as well. Another threat to external validity is our choice of number of methods to inject system-call-free and system-call-

producing code snippets. In particular, we subjectively chose the size of 10 for sets A and B. In fact, there is no guarantee that the results observed for size 10 of sets A and B can be generalized to other re-factoring commits which affect different number of methods with system-call-free and system-call-producing code snippets. External validity is also threatened by our choice of size of loop each code snippet is executed in. For example, there is no guarantee that the results observed for sending an HTTP request 10 times generalizes to other re-factoring commits which adds different number of HTTP requests to methods.

# Chapter 6

# Conclusion and Future Work

In this thesis, we presented our work on enhancing the ability of *GreenAdvisor* to recognize the cause for increases in the energy-consumption of an evolving Android application. The original *GreenAdvisor* uses system-call profiling to detect changes in the energy profile of Android applications and then uses a naive bag-of-words method to identify the code parts that caused the change. The bag-of-words approach is based on verbal connections between the name of the increased system-call and the tokens in code. *GreenAdvisor2.0* refines the blame assignment procedure of its previous version, the original *GreenAdvisor*, by (a) adding application-level instrumentation, and (b) performing a richer system-call profiling. As a result, *GreenAdvisor2.0* produces and stores time-stamped records of the method-calls and system-calls during the execution of the application under test. Finally these records are composed and Student's T-Test is used to determine what method(s) are to blame for the changes in the application's energy profile.

*GreenAdvisor2.0* also controls for undesired impacts of the instrumentation on the application's energy profile by identifying the system-calls whose count might have changed due to the instrumentation. *GreenAdvisor2.0* then takes three actions to minimize False-Positive outcomes in its decisions.

*GreenAdvisor2.0*'s effectiveness in identification and localization of the changes in the application's energy profile were evaluated in three conditions: a re-factoring commit that synchronously consumes more energy, a re-factoring commit that asynchronously consumes more energy, and two common Java

API calls. Evaluation of each condition starts with a theoretical reasoning which is followed by one or two supporting experiments. For each experiment, four measures were computed and used to compare *GreenAdvisor2.0* against the original *GreenAdvisor* and random guess in that particular condition. The reported values computed for these measures across all experiments suggest that *GreenAdvisor2.0* makes sensibly more correct decisions than the other two methods if the change in energy profile is reflected in the system-call profile.

In this work, we used only times and counts of system-calls to talk about energy and eventually monitored times of method-calls to talk about blames. We observed that these metrics are limited and cannot be used to fully identify asynchronous energy-consumption cases. This work can be extended by using more metrics and evidence for talking about energy and blame. A possible direction of research could be using system-call parameters for capturing asynchronous energy-consuming changes.

# Bibliography


[1] K. Aggarwal, A. Hindle, and E. Stroulia. Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 311–320, Sept 2015.

[2] Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. The power of system call traces: Predicting the software energy consumption impact of changes. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, CASCON '14, pages 219–233, Riverton, NJ, USA, 2014. IBM Corp.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[4] Lingfeng Bao, David Lo, Xin Xia, Xinyu Wang, and Cong Tian. How android app developers manage power consumption?: An empirical study by mining power management commits. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 37–48, New York, NY, USA, 2016. ACM.

[5] C. A. Chan, W. Li, S. Bian, C. L. I, A. F. Gygax, C. Leckie, M. Yan, and K. Hinton. Assessing network energy consumption of mobile applications. *IEEE Communications Magazine*, 53(11):182–191, November 2015.

[6] Shaiful Alam Chowdhury and Abram Hindle. Greenoracle: Estimating software energy consumption with energy measurement corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 49–60, New York, NY, USA, 2016. ACM.

[7] Shuai Hao, Ding Li, W.G.J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 92–101, May 2013.

[8] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 12–21, New York, NY, USA, 2014. ACM.

[9] Ding Li and William G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 46–53, New York, NY, USA, 2014. ACM.

[10] Ding Li, Shuai Hao, Jiaping Gui, and W.G.J. Halfond. An empirical study of the energy consumption of android applications. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 121–130, Sept 2014.

[11] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, New York, NY, USA, 2013. ACM.

[12] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys 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, pages 2–11, New York, NY, USA, 2014. ACM.

[13] Q. Lu, T. Wu, J. Yan, J. Yan, F. Ma, and F. Zhang. Lightweight method-level energy consumption estimation for android applications. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 144–151, July 2016.

[14] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[15] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[16] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[17] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 22–31, New York, NY, USA, 2014. ACM.

[18] S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin. Profiling software for energy consumption. In *2012 IEEE International Conference on Green Computing and Communications*, pages 515–522, Nov 2012.

[19] C. Wilke, S. Richly, S. Gtz, C. Piechnick, and U. Amann. Energy consumption and efficiency in mobile applications: A user feedback study. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 134–141, Aug 2013.