UNIVERSITÉ DE MONTRÉAL

ASSISTING DEVELOPERS AND USERS IN DEVELOPING AND CHOOSING EFFICIENT MOBILE DEVICE APPS

RUBÉN SABORIDO INFANTES
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

ASSISTING DEVELOPERS AND USERS IN DEVELOPING AND CHOOSING EFFICIENT MOBILE DEVICE APPS

présentée par: SABORIDO INFANTES Rubén
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. BOIS Guy, Ph. D., président
M. KHOMH Foutse, Ph. D., membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et codirecteur de recherche
M. ADAMS Bram, Doctorat, membre
M. NAGAPPAN Meiyappan, Ph. D., membre externe

DEDICATION

All began on Wednesday, February 19, 2014, at 13:01, when I received the following email:

"*Tengo algunas novedades sobre posibilidades de irte al exterior: si mañana puedes sobre las 10:45h un ratito hablamos.*"

After a short meeting and two possible alternatives, we made the best decision we have ever made: choosing Canada as the place to conduct my PhD. I will never forget that time when my wife, Mariví, helped me to make this decision while she was at Víveros Guzmán.

We started our adventure on Tuesday, August 19, 2014, and, from that moment, we knew that our life would change. Although we have had hard times, we only keep positive memories: my first visit to Poly with Mariví and Hero (my dog), my first talk in English (in Bram's course), my first group seminar, our first winter, Mariví's first job, the courses at La Maison de l'Amitié de Montréal, or the visit of Javi 'el Meri'. However, I have special memories of all the friends we have made during the way: Samuel and Gloria, Pepe and Robin, Felipe and Laura, José and Esperanza, our Canadian neighbor and best friend Carol, and our Spanish family in Canada (Eduardo, Manolita, Cecilia, and Yolanda). Without them this experience would have been different.

I want to thank our families (Ana, Juan, Noé, María, Antonio, Cuñao, Toni, and Laura), for supporting us to finish what we decided to start.

Before finishing, I would like to dedicate some words to all friends at the lab, in particular to Mohammed Sayagh and his wife Fatima, Armstrong Tita, Rodrigo Morales, and Fabio Petrillo, who have always been there to share some jokes, but also to encourage me. To all of them, thank you very much, I will miss you.

To Mariví and Hirito,
for deciding to share with me what
has been the best experience of our lives.

We will always be together!

Rubén Saborido Infantes

ACKNOWLEDGEMENTS

I would like to thank every professor who woke up on me the interest to study, learn, think, and research.

I have good memories from professors Manuel Ujaldón and Juan Antonio Fernández (Jafma), who taught me at University of Málaga long time ago.

I am also really grateful with my friends Francisco Ruiz (Curro) and Mariano Luque, professors at the Faculty of Economics and Business Studies, who showed me the research fields of optimization and decision-making. I spent three wonderful years in their research group. I cannot forget my office colleague Dr. Ana Belén Ruiz, with whom I started my research in evolutionary multi-objective optimization. Professor Óscar Marcenaro was also important for me, because he was always there when I needed some advice.

I especially thank professor Enrique Alba at the University of Málaga, who offered me the chance to do my PhD abroad. I am also grateful to professor Giuliano Antoniol at l'École Polytechnique de Montréal, who helped me to come this dream true. I learned much from him. Not less important are professors Foutse Khomh and Yann-Gaël Guéhéneuc who always trusted me. I am especially grateful to Yann, because he was always available for me, even in hard times, but throughout all time with a big smile on his face.

I also have good memories from professors Bram Adams and Ettore Merlo, both at l'École Polytechnique de Montréal.

I am very grateful with Saad Chidami and, especially, Bryan Tremblay, both at the Department of Electrical Engineering at l'École Polytechnique de Montréal.

To finish, I especially want to thank again Curro, Enrique, Yann, and Bram. They showed me that it is possible to be the best researcher but also the best person.

To all of them,
thank you very much for
having been part of this experience.

Rubén Saborido Infantes

RÉSUMÉ

Les applications pour appareils mobiles jouent, de nos jours, un rôle important dans nos vies. Même si la consommation énergétique affecte la durée de vie de la batterie des appareils mobiles et limite l'utilisation des appareils, nous les utilisons presque partout, tout le temps et pour presque tout.

Avec la croissance exponentielle du marché des applications pour appareils mobiles, les développeurs ont été témoins d'un changement radical dans le paysage du développement du logiciel. Les applications mobiles présentent de nouveaux défis dans la conception et l'implantation logicielle dus aux contraintes des ressources internes (tel que la batterie, le CPU et la mémoire) et externes (l'utilisation de donnés). Donc, les exigences traditionnelles non-fonctionnelles, tels que la fonctionnalité et la maintenabilité, ont été éclipsées par la performance.

Les chercheurs étudient activement le rôle des pratiques de codage sur la consommation énergétique. Cependant, le CPU, la mémoire et les utilisations du réseau sont aussi des mesures importantes pour la performance. Même si le matériel informatique des appareils mobiles s'est beaucoup amélioré dans les dernières années, des nouveaux utilisateurs arrivent, possèdant des appareils bas de gamme avec accès limité aux données. Les développeurs doivent donc gérer les ressources attentivement car les nouveaux marchés possèdent une part importante des nouveaux utilisateurs qui se connectent en ligne pour la première fois. La performance des applications pour les appareils mobiles est donc un sujet très important.

Des études récentes suggèrent que les ingénieurs logiciels peuvent aider à réduire la consommation énergétique en tenant compte des impacts de leurs décisions de conception et d'implantation sur l'énergie. Mais les décisions des développeurs ont un impact aussi sur le CPU, la mémoire et l'usage du réseau. Les développeurs doivent aussi prendre en considération la performance au moment d'évoluer le design de l'application des appareils mobiles. Le problème est que les développeurs n'ont pas de soutien pour comprendre l'impact de leurs décisions sur la performance de leurs apps. Ce problème est aussi vrai pour les utilisateurs d'appareils mobiles qui installent des apps en ignorant s'il existe des alternatives plus efficaces.

Dans cette dissertation, nous aidons les développeurs et les utilisateurs à connaitre d'avantage l'impact de leurs décisions sur la performance des applications qu'ils développent et qu'ils consomment. Nous voulons aider les développeurs et les utilisateurs à développer et choisir des applications performantes. Nous fournissons des observations, des techniques et des lignes

directrices qui aiderons les développeurs à prendre des décisions informées pour améliorer la performance de leurs applications. Nous proposons aussi une approche qui peut servir de complément aux marchés des applications pour appareils mobiles pour qu'ils puissent aider les développeurs et les utilisateurs à chercher des applications efficientes.

Notre contribution est un pas précieux vers l'ingénierie de logiciels performants pour les applications des appareils mobiles et un avantage pour les utilisateurs d'appareils mobiles qui veulent utiliser des applications performantes.

ABSTRACT

Mobile device applications (apps) play nowadays a central role in our life. Although energy consumption affects battery life of mobile devices and limits device use, we use them almost anywhere, all the time, and for almost everything.

With the exponential growth of the market of mobile device apps in recent years, developers have witnessed a radical change in the landscape of software development. Mobile apps introduce new challenges in software design and implementation due to the constraints of internal resources (such as battery, CPU, and memory), as well as external resources (as data usage). Thus, traditional non-functional requirements, such as functionality and maintainability, have been overshadowed by performance.

Researchers are actively investigating the role of coding practices on energy consumption. However, CPU, memory, and network usages are also important performance metrics. Although the hardware of mobile devices has considerably improved in recent years, emerging market users own low-devices and have limited access to data connection. Therefore, developers should manage resources mindfully because emerging markets own a significant share of the new users coming on-line for the first time. Thus, the performance of mobile device apps is a very important topic.

Recent studies suggest that software engineers can help reduce energy consumption by considering the energy impacts of their design and implementation decisions. But developers' decisions also have an impact on CPU, memory, and network usages. So that, developers must take into account performance when evolving the design of mobile device apps. The problem is that mobile device app developers have no support to understand the impact of their decisions on their apps performance. This problem is also true for mobile device users who install apps ignoring if there exist more efficient alternatives.

In this dissertation we help developers and users to know more about the impact of their decisions on the performance of apps they develop and consume, respectively. Thus, we want to assist developers and users in developing and choosing, respectively, efficient mobile device apps. We provide observations, techniques, and guidelines to help developers make informed decisions to improve the performance of their apps. We also propose an approach to complement mobile device app marketplaces to assist developers and users to search for efficient apps. Our contribution is a valuable step towards efficient software engineering for mobile device apps and a benefit for mobile device users who want to use efficient apps.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
APOA	An App Performance Optimization Advisor
ASP	App Selection Problem
ART	Android Runtime
CSV	Comma Separated Values
EA	Evolutionary Algorithm
EARMO	Energy-Aware Refactoring approach for MObile apps
EMO	Evolutionary Multi-objective Optimization
GUI	Graphical User Interface
IDE	Integrated Development Environment
MCDM	Multiple Criteria Decision Making
MOCell	Multi-objective Cellular Genetic Algorithm
NSGAII	Non-dominated Sorting Genetic Algorithm II
OLED	Organic Light Emitting Diode
PSS	Proportional Set Size
SDK	Software Development Kit
SPEA	Strength Pareto Evolutionary Algorithm
TPL	Third-Party Library

CHAPTER 1 INTRODUCTION

Android is an open-source operating system for mobile devices developed by Google, while iOS is a mobile operating system created by Apple for its hardware. Of the 379 million smartphones sold in the first quarter of 2017 [1], 327 million ran Android (86%) and 52 million ran iOS (14%). Thus, Android is the most popular mobile operating system globally. There are now two billion monthly active Android devices and the number of developers with more than one million monthly apps installs grew by 35% year on year [2]. Both platforms offer apps through their marketplaces, the Google Play Store App and the Apple App Store. Mobile device apps play a central role in our life today. Although energy consumption affects battery life of mobile devices and limits device use, we use them almost anywhere, all the time, and for almost everything: to check our email, to browse the Internet, and to access critical services such as banking and health monitoring.

The pace of mobile devices growth around the world is unprecedented. Billions of new users from emerging markets are coming on-line for the first time. It is what Google calls *build for the next billion* [3]. However, a majority of users in these underdeveloped markets face constraints not commonly seen in developed markets: limited access to data connections, high costs when data connections are available, low-end devices with reduced memory, and limited opportunities to recharge batteries during the day.

To address the needs of users, apps performance metrics such as energy consumption and CPU, memory, and network usages must be aligned closely with users limitations. Mobile device apps are today so useful and convenient that their users depend entirely on them, and thus on their performance.

Apps performance is also important for developers. Apps marketplaces are provided with customer ratings as a quality metric. In an internal analysis of app reviews on Google Play, Google realized that half of one-star reviews (the lowest rating) mentioned app performance [4]. Thus, performance metrics are proxies for quality of mobile apps. Therefore, developers who focus on app performance can see improvements in their ratings and, thus, their retention and monetization.

Developing efficient apps is a challenging task. Efficient mobile device apps are apps that make an optimal use of device resources, what improve their performance. Even using the

1. https://www.gartner.com/newsroom/id/3725117
2. https://android-developers.googleblog.com/2017/05/whats-new-in-google-play-at-io-2017.html
3. https://developer.android.com/distribute/best-practices/develop/build-for-the-next-billion.html
4. https://android-developers.googleblog.com/2017/08/how-were-helping-people-find-quality.html

standard process and rules of object-oriented design for the development of mobile device apps, developers may introduce anti-patterns (wrong design choices) that could have a negative impact on apps performance (Li et Halfond, 2014a). In addition to this, during the development of mobile device apps, developers must make design and implementation decisions such as deciding about which data structure implementation to use to store information, to choose a Third-Party Library (TPL) to monitor crashes or users activity, and to decide to include or not ads. However, they do not know the impact of these decisions on the performance of their apps. For example, developers could want to include ads in their apps but they have different TPLs among which to choose, and some could be more efficient than others. Android offers specific implementations for different data structures to be more memory efficient than the ones offered by Java. However, there are no evidences of the magnitude of the improvement in memory usage, if any, or their impact on energy consumption.

To develop efficient mobile device apps, developers must collect and analyze performance metrics during and after the development process. With Android Vitals [5], which has been announced by Google at the I/O 2017 conference, Google plans to help developers find different performance issues in their mobile apps. Android Vitals identifies different issues in Android apps and reports them to their developers. The data is collected from Android devices whose users have opted-in to automatically share usage and diagnostics data. The Play Console aggregates this information and displays metrics about stability, rendering time, and battery usage, in a specific dashboard. However, this information is private to developers and it is only accessible by them.

We believe that making performance information of apps publicly available would be an essential step towards efficient software engineering for mobile device apps. It would put pressure on developers to build efficient apps, which benefits mobile device users who install these apps. Today, mobile device users do not have access to information about apps performance and they compare and select apps based on their rating and numbers of downloads. Thus, users choose apps that other users choose (popularity). Even if they consume more energy or transmit more data over the network than other apps with similar functionalities (Saborido *et al.*, 2016).

Even if mobile device apps performance would be available in marketplaces, the choice of optimal apps would be complicated for users because of the cognitive effort imposed to discriminate between different apps and many different possible metrics with several values. In addition, mobile device users could need different kinds of performance depending on their locations, needs at certain times, and usages. Thus, developers should optimize their apps

5. https://developer.android.com/distribute/best-practices/develop/android-vitals.html

taking into account their users' context. For example, users in emerging markets have limited access to data connections and experience high costs when data connections are available. They also own devices with low memory, processing power, and screen resolution. In this context it would be better for users to install efficient apps in terms of CPU, memory, and network usages, and developers could have this fact in mind when they develop apps for those markets. Some performance metrics could be more important than others depending on the context and one app could be preferred over others because of its performance. The context of use affects users and developers' preferences about the metrics to be optimized when they choose and develop mobile device apps, respectively.

1.1 Research Goal

The main goal of our research is to assist developers and users of mobile device apps in developing and choosing, respectively, efficient apps. We address our research goal through the following key directions:

1. Providing techniques and guidelines to help developers make informed design and implementation decisions to improve the performance of their apps.
2. Defining an approach to complement mobile device app marketplaces to assist developers in the comparison of apps performance, and help users make informed decisions to choose their apps.

Our thesis is that multi-objective approaches support developers and users to implement and choose efficient mobile device apps, respectively.

We focus our research on the Android operating system. We choose this platform because it is the most popular mobile device operating system globally. However, our solutions are not intrinsically dependent on Android and they could also apply as well to iOS minus implementation details.

1.2 Research Contributions

To reach our research goal we do the following:

1. We conduct different research investigations to help mobile device app developers make informed decisions about the design and implementation of their apps.
   (a) We first study the impact of eight well-known object-oriented and Android specific anti-patterns on energy consumption. Then, we propose an Energy-Aware

Refactoring approach for MObile apps (EARMO). It is a multi-objective refactoring approach to detect and correct anti-patterns in mobile device apps, while improving energy consumption. Our approach leverages information about the energy cost of anti-patterns to automatically generate refactoring sequences that developers can apply to improve the quality of the source code and–or to reduce the energy consumption of their apps. We describe this research in Chapter 5.

(b) From our previous study we obtain that the Android anti-patterns studied have a negative impact on energy consumption of apps. One of these anti-patterns exists in Android apps when developers use a Java map implementation instead of a specific implementation offered by the Android Application Programming Interface (API). Although Android offers different map implementations to choose from, the official documentation is ambiguous about the performance benefit of using these specific implementations. We study the use of map data structures by Android developers. First, we perform an observational study of 5,713 Android apps. Second, we conduct a survey to assess developers' perspective on Java and Android map implementations. Then, we perform an experimental study comparing their performance. We conclude with guidelines for choosing among these map implementations. We describe this research in Chapter 6.

(c) Analyzing the source code of apps in our previous study, we observe that mobile device apps implement some functionalities integrating TPLs. We propose an approach to assist mobile device app developers in choosing an efficient TPL for a concrete task. We evaluate this approach by performing a case study over the three most popular TPLs in each of the three most popular categories of Android TPLs. Specifically, we obtain quality metrics for these TPLs and we analyze their impact on the performance of mobile device apps. We provide a catalog of performance metrics for these popular TPLs. We describe this research in Chapter 7.

(d) One of the most popular TPL integrated by developers in their apps is for advertising. Ads allow developers to keep their content free and available, reaching more users, while still making revenues. We perform a study about ads-supported Android apps and their corresponding paid versions to understand their differences in terms of performance, functionalities, permissions, implementation, and development and release processes. We also define different equations to estimate the network usage of independent ads, the percentage of battery drained due to ads, and the time in which an ads-supported app costs more than its paid version because of the presence of the ads. From our observations, we provide developers advices about the usage of ads in their mobile device apps. We describe this

research in Chapter 8.

2. Apps with similar functionalities could have a different performance due to design and implementation developers' decisions such as data structures used, TPLs integrated, and–or the inclusion or not of ads. For example, to visit an article in Wikipedia, a browser could consume more energy and transmits more data over the network than another browser, because of the inclusion of ads in the former. It means that there exist a trade-off in terms of performance between different apps. We propose an App Performance Optimization Advisor (APOA) for mobile device app marketplaces. It is a recommendation system that can be implemented in any marketplace for helping users and developers to compare apps in terms of different metrics. APOA takes as input metric values of apps and a set of metrics to optimize. It solves a combinatorial multi-objective optimization problem and it generates, as output, optimal sets of apps. To highlight the capabilities of APOA, we provide an Android case study. We describe this research in Chapter 9.

Parts of this dissertation have been published at conferences such as SANER (Saborido *et al.*, 2016), ICPC (Saborido *et al.*, 2017), and MCDM (Saborido et Khomh, 2017). In addition, parts of this dissertation have been accepted for publication at IEEE Transactions on Software Engineering (Morales *et al.*, 2017a) or are being reviewed at Journal of Empirical Software Engineering and at Journal of Sustainable Computing: Informatics and Systems.

1.3 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we give some background information related to our work. Then, we present a discussion of the existing literature related to efficiency of mobile device apps in Chapter 3. We describe the process we use in our research to collect and analyze performance metrics of mobile device apps in Chapter 4. Next, in the following four chapters, we present our research to help developers make informed design and implementation decisions to improve the performance of their apps. In Chapter 5, we study the impact of anti-patterns on the energy consumption of mobile device apps. Then, we propose EARMO, an approach to propose refactoring sequences to correct anti-patterns while improving the energy efficiency of apps. In Chapter 6, we study the performance of Java and Android map implementations for different operations and data sizes. In addition to energy consumption, we also study CPU time and memory usage. Then, we provide guidelines to help developers choose map implementations to improve their apps performance. In Chapter 7, we focus on TPLs and we study the impact of popular TPLs

on the performance of apps. We also propose an approach to assist developers in choosing efficient TPLs for their apps. Then, in Chapter 8, we study ads-supported and paid versions of Android apps, to understand their development process and to analyze the impact of ads on apps performance in terms of energy consumption and CPU, memory, and network usages. We conclude providing developers with some advices to improve the performance of free and paid versions of apps. Next, in Chapter 9, we present APOA, an approach to assist developers and users in the comparison and selection, respectively, of efficient mobile device apps. Finally, in Chapter 10, we present our conclusion, the limits and constraints of our research, and future work.

CHAPTER 2 BACKGROUND

In this chapter, we introduce the basic concepts used in this dissertation. We describe the performance metrics on which we focus and how they are usually measured in Android and iOS platforms. Then, we briefly introduce the concept of scenarios of usage and the idea of test cases for mobile devices. Finally, we present some concepts and techniques related to multi-objective optimization.

2.1 Performance Metrics

In the rest of this dissertation, we consider the following performance metrics: energy consumption (power usage), and CPU, memory, and network usages. We describe these performance metrics next.

Energy Consumption (Power Usage)

Energy consumption determines the battery life of mobile devices and, therefore, their availability. Without energy, a mobile device cannot be operated. Energy is defined as the capacity of doing work while power is the rate of using energy. Energy (E) is measured in Joules (J) while power (P) is measured in Watts (W). Energy is equal to power times a time period T in seconds. Therefore, $E = P \times T$. Thus, if an app uses two Watts of power for five seconds it consumes 10 Joules of energy.

CPU Usage

CPU usage describes the proportion of time that the processor is in use. A mobile device's CPU usage can vary depending on the types of tasks that are being performed by an app. It is usually measured in percentage (%), which indicates how much of the processor's capacity is currently in use by the system, or in the time that a method is actually running in the system. Typically, CPU is one of the primary sources of energy consumption.

Memory Usage

Memory usage is the amount of memory (RAM) that a task uses when it is running. This memory is used to save internal data and instructions to be executed. Memory limits the number of apps users can run and the amount of data they can work with.

Network Usage

Network or data usage refers to the amount of data moving across a network (Wi-Fi, 3G, 4G, ...). This metric is important for users and developers because network access could be expensive in terms of bandwidth costs for some users. Typically, network usage is one of the primary sources of energy consumption.

Measurements

Energy impact of apps running on mobile devices can be measured using hardware based approaches as the popular Monsoon Power Monitor. However, energy measurements can also be collected using software based approaches as the developer tools designed by Google or Xcode, for Android and iOS platforms, respectively. Because Android is based on Linux, CPU, memory, and network usages can be collected using well-known Linux commands, as `top` and `tcpdump`, or using the Android command `dumpsys`. For iOS, Xcode instruments can be used for the collection of these performance metrics in Apple platforms.

2.2 Scenarios of Usage and Test Cases

Independently of how performance metrics are measured, the software under test should be run while performance metrics are collected. Developers must define scenarios to simulate the user interaction or implement test cases to test app methods independently.

Appium is an open source test automation framework for Android and iOS apps. Calabash is a different alternative that enables developers to write and execute automated acceptance tests of mobile apps. It is also cross-platform, supporting Android and iOS native apps. There also are specific automation tools for each platform: such as Monkeyrunner or Robotium for Android platforms, or EarlGrey for iOS.

2.3 Multi-objective Optimization

Multi-objective optimization problems are mathematical programming problems with a vector-valued objective function that is usually denoted by $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), ..., f_m(\mathbf{x}))$, where $f_j(\mathbf{x})$, for $j = 1, \ldots, m$, is a real-valued function defined on the feasible region $F \subseteq \Re^M$. The decision space belongs to $\Re^M$ while the criterion space belongs to $\Re^m$, and the multi-objective

optimization problem can be stated as follows:

$$\begin{array}{ll} \text{optimize} & [f_1(\mathbf{x}), ..., f_m(\mathbf{x})] \\ \text{s.t.} & \mathbf{x} \in F \end{array}$$

In the functional space of criterion, some objective functions should be maximized ($j \in J_{max}$) while other objective functions should be minimized ($j \in J_{min}$). These subsets of indices verify that $J_{max} \cup J_{min} = \{1, ..., m\}$. In this context, optimality is defined on the basis of the concept of dominance, in such a way that solving the above problem implies finding the subset of non-dominated solutions, that is those feasible solutions which are not dominated by any other feasible one. A feasible solution $\mathbf{x}^0$ dominates another solution $\mathbf{x} \in F$ if and only if $f_j(\mathbf{x}^0) \geq f_j(\mathbf{x})$, for every $j \in J_{max}$ and $f_j(\mathbf{x}^0) \leq f_j(\mathbf{x})$, for every $j \in J_{min}$, with at least one strict inequality. The set of non-dominated solutions will also be referred by Pareto optimal solutions and define the efficient frontier or Pareto optimal front of the multi-objective optimization problem (see, for example, Miettinen, 1999). The concepts of dominance and Pareto optimal front are graphically shown in Figure 2.1: for a bi-objective optimization problem.

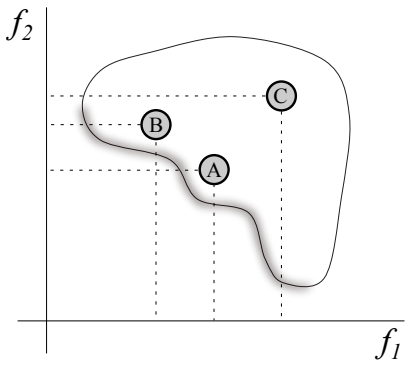


Figure 2.1: Criterion space of a multi-objective optimization problem with two objectives to be minimized. Points A and B Pareto dominate point C, but points A and B are Pareto equivalents. The highlighted line is the efficient frontier or Pareto optimal front.

In this dissertation we want to optimize different metrics. Thus, we must handle several objectives in conflict each other at the same time. Different methodologies exist to solve multi-objective optimization problems. Multiple Criteria Decision Making (MCDM) (Hwang et Masud, 1979; Miettinen, 1999) and metaheuristics, such as Evolutionary Multi-objective Optimization (EMO) (Deb, 2001; Coello *et al.*, 2007), are the most popular methodologies. They have contributed with several different approaches to solve real problems. EMO algorithms, in general, find a evenly distributed set of Pareto optimal solutions to approxi-

mate the Pareto optimal front while MCDM takes into account some user preferences to find a reduced set of optimal solutions.

We use EMO algorithms because we prefer to find large set of optimal solutions. Evolutionary Algorithms (EA) are nature-inspired search strategies based on the natural selection of evolution. An EA usually uses a single population (panmixia) of individuals and apply the operators to them as a whole. Well-accepted subclasses of EAs are Genetic Algorithms (GA), Genetic Programming (GP), Evolutionary Programming (EP), and Evolution Strategies (ES). There also exist the so called structured EAs, in which the population is decentralized. Among the many types of structured EAs, distributed and cellular models are two popular optimization variants. In many cases, these decentralized algorithms provide a better sampling of the search space, resulting in an improved numerical behavior with respect to an equivalent algorithm in panmixia.

EAs are especially well suited for tackling multi-objective optimizations problems because of their ability to find multiple trade-off solutions in one single run. Thus, the use of EMO algorithms for solving multi-objective optimization problems has become very popular in the last two decades and, currently, it is an active research field. In the EMO field, solving a multi-objective optimization problem is understood as finding a set of non-dominated solutions as close as possible to the Pareto optimal front (convergence) and that represents the entire Pareto optimal front (diversity). EMO algorithms start with a randomly created population of individuals. Afterwards, the algorithm enters in an iterative process that creates a new population at each generation, by the use of operators which simulate the process of natural evolution: selection, crossover, mutation and–or elitism preservation. These variation operators allow to transform a candidate solution to explore the decision space in the search of most attractive solutions and to escape from local optima. One of the main advantages of EMO algorithms is that they are versatile and can deal with multi-objective optimization problems having variables and objective functions of different nature; that is, they can easily handle discontinuous and concave Pareto optimal fronts, and binary and integer-valued variables.

In this dissertation we use the EMO algorithms NSGAII, SPEA2, and MOCell. We choose them because they have been successfully applied during the last decade to solve hundreds of multi-objective optimization problems in many fields (software engineering, medicine, portfolio selection, etc). Next, we introduce these algorithms.

Non-dominated Sorting Genetic Algorithm II

The Non-dominated Sorting Genetic Algorithm II (NSGAII), proposed by Deb *et al.* (2002), is based on the Pareto dominance and has stood out by its fast non-dominated sorting procedure for ranking solutions and its elite-preserving mechanism for the selection of the best individuals. Starting from a randomly generated population, at each generation an offspring population is created through selection, crossover, and mutation operators. Then, both parents and offspring are joined and classified into non-dominated fronts as follows. From the resulting population, the individuals which are not dominated by any other solution constitute the so-called first non-dominated front. These individuals are temporarily removed and, subsequently, the second non-dominated front is formed by the next individuals which are not dominated by any other solution in the population. This process continues until every individual has been included into some front. Afterwards, the population to be passed to the next generation is formed by the solutions in the lower level non-dominated fronts. If there are more solutions in the last front allowed than the remaining space in the new population, the individuals in this front are sorted according to a crowding distance. Somehow, this distance measures the objective space around each solution that is not occupied by any other solution in the population. Subsequently, the new population is completed with the individuals with the least crowding distance.

Strength Pareto Evolutionary Algorithm 2

Zitzler et Thiele (1999a) developed an algorithm to multi-objective optimization, named the Strength Pareto Evolutionary Algorithm (SPEA). It uses a mixture of established and new techniques to find multiple Pareto optimal solutions in parallel. This algorithm stores the non-dominated solutions found as generations elapse in an external or archival population set. The SPEA uses the concept of Pareto dominance in order to assign scalar fitness values to individuals. SPEA2 (Zitzler *et al.*, 2001) added several improvements to the previous version of the algorithm to enhance the robustness of the fitness assignment scheme. The improvements take into account how many individuals each individual dominates and is dominated by through the use of a nearest neighbor density estimation technique and an archive truncation method that guarantees the preservation of boundary solutions. The algorithm uses variation operators to evolve a population, like NSGAII, but with the addition of an external archive. The archive is a set of non-dominated solutions, and it is updated during the iteration process to maintain the characteristics of the non-dominated front. In SPEA2, each solution is assigned a fitness value that is the sum of its strength fitness plus a density estimation.

Multi-objective Cellular Genetic Algorithm

The Multi-objective Cellular Genetic Algorithm (MOCell) is a cellular genetic algorithm (cGA) proposed by Nebro *et al.* (2007). In a basic cGA, the population is usually structured in a grid of different dimensions, and a neighborhood of solutions is defined on it. An individual may only interact with individuals belonging to its neighborhood, so its parents are chosen among its neighbors with a given criterion. Recombination and mutation operators are applied to the individuals with given probabilities. Afterwards, the algorithm computes the fitness value of the new offspring and inserts it into the equivalent place of the current individual in the new (auxiliary) population following a given replacement policy. After applying this reproductive cycle to all the individuals in the population, the newly generated auxiliary population is assumed to be the new population for the next generation. The overlapped small neighborhoods of cGAs help in exploring the search space because the induced slow diffusion of solutions through the population provides a kind of exploration (diversification), while exploitation (intensification) takes place inside each neighborhood by genetic operations. Besides, the neighborhood is defined among tentative solutions in the algorithm, with no relation to the geographical neighborhood definition in the problem space.

MOCell includes an external archive, like SPEA2, to store the non-dominated solutions found during the search process. It uses the crowding distance of NSGAII to maintain the diversity in the Pareto front. The selection consists in taking individuals from the neighborhood of the current solution (cells) and selecting another one randomly from the archive. After applying the variation operators, the new offspring is compared with the current solution and replaces the current solution if both are non-dominated, otherwise the worst individual in the neighborhood will be replaced by the offspring.

CHAPTER 3 RELATED WORK

There is a growing body of work on analyzing and optimizing the performance of mobile devices. Most of them focused on energy consumption as the most important performance metric. Although software energy consumption optimization is a relatively new topic, several works have studied the impact of developers and users' decisions on battery life. In this, chapter we comment relevant works related to the improvement of mobile device apps performance.

3.1 Developers' Decisions and Performance of Apps

Cuervo *et al.* (2010) proposed an approach that enables fine-grained energy-aware offload of mobile code to the infrastructure. It decides at runtime which methods should be remotely executed, driven by an optimization engine that achieves the best energy savings possible under the mobile device's current connectivity constrains. Kemp *et al.* (2012) proposed a framework for computation offloading for mobile devices that can be used to reduce the energy consumption on and increase the speed of computing intensive operations. They evaluated the proposed approach with two real world mobile device apps, an object recognition app and a distributed augmented reality game, and showed that little work was required to enable computation offloading for these two apps.

Energy has also been studied in test cases generation. Li *et al.* (2014b) proposed an approach for minimizing the energy consumption of in-situ test suites. It encodes the test suite minimization problem as an integer linear programming problem. Solving this problem generates a minimized test suite that is guaranteed to satisfy the tester's minimization goals and use as little energy as possible. Sahin *et al.* (2014b) studied the impact of code obfuscation on energy consumption. They conducted an empirical study of the effects of 18 code obfuscation techniques on the amount of energy consumed by executing a total of 15 usage scenarios spread on 11 Android apps. They concluded that obfuscations can have a statistically significant impact on energy usage, however, the magnitudes of these impacts are unlikely to impact mobile device users. Field *et al.* (2014) presented a modular energy-aware computing framework that provides a layer of abstraction between sources of energy data and the apps that exploit them. This approach replaces platform specific instrumentation through two APIs to allow the development of code that may portably provide energy transparency to developers. In turn, it enables informed decisions to be made with respect to trade-offs regarding energy consumption. Linares-Vásquez *et al.* (2014) studied how dif-

ferent Android API usage patterns can influence energy consumption in mobile device apps. They measured energy consumption of method calls when executing typical usage scenarios in 55 Android apps from different domains. They analyzed the consumption of individual APIs as well as their usage patterns. Their findings indicate that some consolidated design and implementation practices, such as the use of Model-View Controller, information hiding, or the implementation of the persistence layer through a relational database may have a non-negligible impact on the app energy consumption. In addition, they found that APIs related to Graphical User Interface (GUI), image manipulation, and database, represent, all together, 60% of the energy-greedy APIs. Modern mobile devices use Organic Light Emitting Diode (OLED) displays that consume more energy when displaying light colors as opposed to dark colors. Mobile device apps can be energy efficient by modifying their GUIs. Li *et al.* (2014c) developed an approach for automatically rewriting Web apps so that they generate more energy efficient Web pages. They shown that it can achieve a 40% reduction in display energy consumption. The approach rewrites the server side code and templates of a Web app so that the resulting Web app generates pages that are more energy efficient when displayed on a mobile device. The rewritten Web app can then be made available to OLED mobile device users via automatic redirection or a user-clickable link. Something similar is done for Android apps by Linares-Vásquez *et al.* (2015). They proposed an approach for generating color palettes using a multi-objective optimization technique that produces color solutions optimizing energy consumption of GUIs and their contrast to keep them visually attractive. Wan *et al.* (2015) defined a different approach to assist developers in identifying the user interfaces of their apps that can be improved with respect to energy consumption. They used the approach to investigate 962 Android market apps and found that 41% of these apps have display energy hotspots, some of them consuming over 100% more display energy than a display-energy-optimized version. Recently, Mohan *et al.* (2017) studied the energy consumption of the storage subsystem on an Android mobile device. They analyzed the energy consumption of different storage primitives, such as sequential and random writes, on two popular mobile file systems, ext4 and F2FS. They concluded that storage can consume more energy than the network, and as much energy as the display. Lyu *et al.* (2017) studied the energy consumption of local database usage in Android apps and they found that database initialization and write operations are the most expensive. They also found that these operations appear frequently in loop structures; many of which are not properly batched in explicit transactions, which can cause significant inefficiencies.

There are much more research about developers' decisions and apps performance. However, next, we describe only closer work to our research.

3.1.1 Anti-patterns and Refactoring Operations

Gottschalk (2013) and Park *et al.* (2014) have studied the effect of applying refactorings to a set of software systems; comparing the energy difference between the original and refactored code. Li et Halfond (2014b) investigated the impact of Android developing practices. They found that bundling network packets up to a certain size, accessing class fields, extracting array length into a local variable in a for-loop, and inline getters and setters all led to reduced energy consumption. However, other practices, such as limiting memory usage had a very minimal impact on energy usage. These results serve to inform developers about specific coding practices that can help reduce the energy consumption of their apps.

Reimann *et al.* (2014) proposed a catalog of 30 quality smells specific to the Android platform. These smells were reported to have a negative impact on quality attributes like energy consumption. In another work, Sahin *et al.* (2014a) conducted an empirical study that investigates the impacts of applying refactorings on energy usage. They investigated the impact of six commonly-used refactorings on several apps. The results of their study show that refactorings impact energy consumption and that they can either increase or decrease the amount of energy used by an app. Banerjee et Roychoudhury (2016) proposed an approach to refactor mobile apps by relying on energy-consumption guidelines to control for energy-intensive device components. Their technique uses a set of energy-efficiency guidelines to refactor the design expression of an app. They define a design expression as a regular expression that represents the ordering of energy intensive, resource usages, and invocation of key functionalities within the app. To demonstrate the efficacy of their technique they analyzed ten open-source apps, reducing the energy consumption of these apps between 3% to 29% after refactoring. Hecht *et al.* (2016) conducted an empirical study with different versions of open source Android apps to determine if the correction of Android anti-patterns had a significant impact on user interface (number of delayed frames) and memory usage. They reported that correcting these Android code smells effectively improves the user interface and memory usage in a significant way for Android Dalvik runtime. They also observed that the positive and negative impact of code smells correction cumulates. Carette *et al.* (2017) conducted an empirical study on six Android apps to assess the energy impact of Android performance and also Android picture anti-patterns. For this purpose, they developed an automated approach that detects and corrects these anti-patterns, assessing their energy impact to retrieve the best version of an app. Their results confirm that Android anti-patterns have an impact on the energy consumption of apps. They concluded that by correcting only one performance Android anti-pattern one can reduce up to 3.86% the energy consumption of an app, while the correction of several anti-patterns can reduce up to 4.83% of the energy

consumption.

EARMO, our proposed approach, differs from these previous works in the sense that, beside detecting anti-patterns in mobile apps, it is a multi-objective approach to generate optimal sequences of refactoring operations that achieve a maximum removal of anti-patterns while controlling for energy consumption at the same time. We avoid a direct aggregation of different potentially conflicting objectives (number of anti-patterns and energy consumption), allowing software maintainers to select among different trades or achieve a compromise between the two objectives. Thus, EARMO allows developers to select the sequence of refactorings that decrease more the energy consumption or the one that improve more the maintainability of their code. Other developers might be more conservative and select solutions located in the middle of these two objectives. Developers have the last word, and EARMO supports them by providing different alternatives.

3.1.2 Data Structures

Chameleon, proposed by Shacham *et al.* (2009), is an automatic tool that assists Java developers in choosing the appropriate collection implementation for their apps. During program execution, it collects traces and computes heap-based metrics on collection classes to generate a list of suggested collection adaptation strategies. The tool can apply these corrective strategies automatically or present them to the developer. It focuses on memory usage and Java apps. Hunt *et al.* (2011) studied the relationship between the execution time and energy consumption of three lock-free data structures: FIFO queue, double-ended queue, and sorted linked list. Their results show that lock-free data structures cannot only provide significant performance improvements in many situations, but also that this increase in performance can improve the data structure's energy efficiency as well. Manotas *et al.* (2014) proposed SEEDS, a tool to optimize Java apps by selecting the most energy efficient library implementations for Java's collections. An empirical study and their results show that using such automation can improve energy usage of apps without requiring developers to deal with low-level energy profiling tools and–or analyses. They concluded that SEEDS reduces the energy consumption of Java apps by up to 17%. Liu *et al.* (2015) empirically studied the optimization space of application-level energy management from the data-oriented perspective. The energy optimization space is explored through multiple dimensions, ranging from data access pattern, data organization and representation, data precision, and data input/output intensity. Concerning data access patterns, they concluded that random access consumes much more energy than its sequential counterpart, due to cache locality. In addition, they also observed that read and write operations consume the same amount of energy. Regard-

ing data organization, they found that object-oriented encapsulation, although it has many benefits such as modularity, information hiding, and maintainability, it pays a toll on energy consumption. They focused on Java programs running on Intel CPUs. Hasan *et al.* (2016) created detailed profiles of the energy consumed by common operations done on Java list, map, and set implementations. They also provided a guideline about the scenarios in which the energy consumption of alternative collections classes becomes an issue.

There are previous research studying the impact of data structures on different performance metrics. Most of the existing research focus on desktop computers and Java programs. Although Hasan *et al.* (2016) ran their experiments in an Android phone, none of the previous works have considered specific Android data structures. We study Android map implementations because the official documentation is ambiguous about the the performance benefit of using these implementations. However, the Android documentation recommends them as more efficient alternatives to the Java map implementation `HashMap`. We analyze and compare the performance of these map implementations to conclude with guidelines for developers.

3.1.3 Third-party Libraries

Minelli et Lanza (2013) observed that external calls represent more than 75% of the total number of method invocations in mobile device apps. They concluded that to comprehend Android apps it is important to also understand the behavior of the used external libraries. Gorla *et al.* (2014) studied the inconsistencies between app description and requested permissions due to the usage of TPLs. Ruiz *et al.* (2014) analyzed the relationship between the number of TPLs in Android apps and their user ratings. They focused on advertising TPLs, concluding that the number of ad libraries in Android apps is not related to their ratings. Wang *et al.* (2015) proposed WuKong, an accurate and scalable approach to detect Android app clones that includes a clustering-based approach to detect TPLs efficiently and accurately without prior knowledge. Using WuKong they studied the impact of TPLs on app clone detection. Based on their evaluation, they concluded that more than 60% of the sub-packages in Android apps are from TPLs. Therefore, detecting and filtering these TPLs is important, because if apps are dominated by these TPLs, app clone detection results could be significantly skewed. Ma *et al.* (2016) and Li *et al.* (2017) have recently proposed LibRadar and LibD, respectively. They are tools to detect TPLs in Android apps. LibRadar can detect the TPLs used in a given app instantly based on static analysis and feature comparison. LibD is a more recent approach to identify TPLs. Different from LibRadar, LibD is based on feature hashing and can better handle code whose package and method names are obfuscated. We

also found some works focusing on code completion that recommend API usages for mobile apps (Nguyen *et al.*, 2015), learn API usages from bytecode (Nguyen *et al.*, 2016), or mine and recommend API usage patterns (Niu *et al.*, 2017).

Although TPLs account for most of the code of Android apps, there are no studies about the impact of TPLs on apps performance. Thus, developers do not know if TPLs they choose to integrate in their apps are less efficient than others.

3.1.4 Ads Business Model

The impact of ads on apps performance has been studied in different works. Wei *et al.* (2012) proposed a monitoring and profiling system for characterizing Android app behaviors at multiple layers. After analyzing top free and paid Android apps, they observed that ads-supported versions of apps could end up costing more than their paid versions due to increased advertising/analytics traffic. Vallina-Rodriguez *et al.* (2012) analyzed four ad networks and concluded that ad traffic can be a significant fraction of the total traffic of the users. They also found that mobile ad traffic is responsible for important energy overhead by forcing off-line apps to become on-line apps. Rasmussen *et al.* (2014) studied the effects of ads on energy consumption and the effects of attempts to block the ads. They compared different methods of blocking ads and compared the power efficiency of these methods. They concluded that there are many cases where ads-blocking software resulted in increased power usage. Gui *et al.* (2015) analyzed 21 highly-rated free Android apps and showed that the use of ads leads to mobile device apps that consume more energy and use more CPU, memory and network. They found that the median relative energy, CPU, memory, and bandwidth costs of ads are 15%, 56%, 22%, and 97%, respectively.

There are several approaches focusing narrowly on how to improve the performance of mobile ads. Khan *et al.* (2012) proposed a new advertisement delivery approach to mitigate the stress from network traffic related to ads in free mobile device apps. The approach predicts user context to identify relevant advertising content, and then opportunistically use inexpensive wireless networks to predictively cache advertisement content on mobile devices. Recently, Pamboris *et al.* (2016) proposed AD-APT, a system to avoid the adverse implications of mobile advertising on energy consumption and network usage. AD-APT strikes a balance between these two performance metrics refactoring ads-supported apps automatically to adjust the frequency of mobile ad occurrences at runtime. It is based on policies that consider the device's battery life, the type of network connectivity, and limits on network usage. Authors evaluated it on 10 popular ad-supported Android apps and showed that it can yield reductions of up to 40% in energy consumption and 30 times in network usage.

No previous work studied if the impact of ads on performance is statistically significant. The impact of ads on some performance metrics could be small and it could be unperceivable by users. We study the ads-business model and differences between ads-supported and paid Android apps. We do that to know more about the impact of developers' decisions on apps performance due to advertising TPLs. We also define different equations to help developers and users know when the hidden costs of ads-supported apps overtake the clear cost of paid apps.

3.2 Users' Decisions and Performance of Apps

Amsel et Tomlinson (2010) proposed a tool for estimating the energy consumption of currently installed software systems. This tool determines which software systems are the most efficient given the user's current computer configuration. It presents this information to the user in the form of a chart comparing the CPU usage and the energy consumption of software systems in the same category. They used the results to make suggestions to the users about which software systems to use. They tested Internet browsers, word processors, and audio software running their approach on a desktop computer. Zhang *et al.* (2014) analyzed the energy consumption of text editing desktop apps, email clients, and music players. They highlighted the perils that users face and the ultimate responsibility users have for the battery life of their mobile devices. They investigated multiple scenarios demonstrating that apps can consume energy differently for the same task thus illustrating the trade-offs that end-users can make for the sake of energy consumption. For example, they obtained that a command line music player uses more than six times less energy than a GUI one, or that web-based desktop apps tend to consume more energy than non-web based. Behrouz *et al.* (2015) proposed EcoDroid, an hybrid static and dynamic analysis technique that estimates the energy cost of Android apps, from a given category, and ranks them accordingly to help users make informed decisions. This approach uses a combination of dynamic and static analyses and test cases, to execute apps and estimates their energy cost based on their API usage. These estimates take into account the energy cost of the paths executed by test cases. They also conducted a preliminary evaluation of EcoDroid to assess its overall accuracy in ranking six apps from a given category according to their energy costs.

EcoDroid is the closest work to our approach APOA. However, while the former focuses on energy consumption and only for one category of apps, APOA ranks apps from one or more categories in terms of different performance metrics taking into account developers and users' preferences. In addition to this, APOA is platform independent.

CHAPTER 4 PERFORMANCE METRICS COLLECTION

To measure performance metrics of mobile device apps we set up a measurement environment. We use a real mobile device to run our experiments. We also define scenarios to simulate the user interaction with apps, and we implement test cases to test app methods independently. We play scenarios of usage or test cases while we measure performance metrics. We have performance measurements that we must process and analyze. Next, we describe the mobile device that we use for our experiments. Then, we describe how we define scenarios and test cases and how we measure and analyze performance metrics of Android apps.

4.1 Subject Mobile Device

We use a LG Nexus 4 Android phone equipped with a quad-core Krait CPU@$1,500MHz$, a 4.7-inch screen, 16 GB internal storage, 2 GB RAM, and running Android Lollipop (version 5.1.1, Build number LMY47V). Its battery has an electric charge of 2.1 Ah and a voltage of 3.8 V. This phone is representative of the current generation of Android mobile phones. Nexus mobile phones are pure Android devices designed by Google and manufactured by one of the most important mobile companies to provide the best user experience. In addition, this model of phone has been extensively used in previous research (Linares-Vásquez *et al.*, 2014; Sahin *et al.*, 2014b; Huang *et al.*, 2016; Sahin *et al.*, 2016).

We choose Android Lollipop because it introduced one of the most significant changes in Android in recent years: the shift to the relatively new way of executing apps called Android Runtime (ART), which replaces its predecessor Dalvik. In ART the apps are compiled to native code once, which improves the memory and CPU performance. On the contrary, Dalvik runs along with the execution of an app. From Android Lollipop, ART is the only runtime environment. According to information provided by Google [1], more than 70% of the devices that are active on the Google Play Store (at October, 2017) run Lollipop or later versions.

4.2 Automation for Apps: Scenarios of Usage and Test Cases

We define scenarios of usage using the Android tool Monkeyrunner [2]. This tool provides an API for writing programs that control an Android device or emulator from outside of Android

1. https://developer.android.com/about/dashboards/index.html
2. https://developer.android.com/studio/test/monkeyrunner/index.html

code. Thus we can programmatically send actions such as touch and swipe events to it.

To collect long scenarios to simulate typical scenarios of usage of apps, we use the Android app HiroMacro [3]. This software allows to generate scripts containing the touch and swipe events while a user interacts with an app directly on the phone. We convert the resulting scripts generated by HiroMacro to Monkeyrunner format. Thus, the interaction to collect scenarios is done using the phone and the actions can be played automatically from our own scripts using the Monkeyrunner Android tool.

When we do not need to simulate the user interaction but exercise and measure the energy consumption of a particular app method, we define and use Android test cases. Android allows to write JUnit4 style tests that perform tests against the classes in a package. When we run tests from the command-line with the Android Debug Bridge [4] (`adb`) and the `am instrument` Android command [5], we get more options for choosing the tests to run. We can select individual test cases, filter tests according to their annotation, or specify testing options. Because the test run is controlled entirely from the command-line, we can customize our testing with Python scripts in various ways.

4.3 Performance Measures of Android Apps

Next, we describe how we measure performance metrics of Android apps.

4.3.1 Energy Consumption (Power Usage)

There exist software based approaches to measure the energy consumption of apps, as PETRA (Di Nucci *et al.*, 2017), but they rely on different sources of information to estimate the measurements. Imprecisions in those sources could affect the quality of the estimations. Thus, we use a hardware based approach based on a digital oscilloscope TiePie Handyscope HS5 [6]. It offers the LibTiePie Software Development Kit (SDK), a cross platform library for using TiePie engineering USB oscilloscopes through third party software. This SDK allow us to communicate with the oscilloscope in our scripts. We power the mobile phone by a power supply that is connected to the phone's motherboard to avoid any kind of interference with the phone battery. Between both we connect, in series, a uCurrent [7] device. It is a precision current adapter for multimeters, converting the input current I proportionally to the output

3. https://play.google.com/store/apps/details?id=com.prohiro.macro
4. https://developer.android.com/studio/command-line/adb.html
5. https://developer.android.com/studio/test/command-line.html
6. http://www.tiepie.com/en/products/Oscilloscopes/Handyscope_HS5
7. http://www.eevblog.com/projects/ucurrent/

voltage V_{out}. We connect the probe of the oscilloscope to the output of the uCurrent device. Thus, knowing I and the voltage supplied by the power supply V_{sup}, we used Ohm's law to calculate the power usage P as $P = V_{sup} \cdot I$. We calculate the energy associated to each sample as $E = P \cdot T$, where P is the power of the smart-phone and T is the period sampling in seconds. The total energy consumption is the sum of the energy associated to each sample. Figure 4.1: shows the connection diagram. We disconnect the positive pin of the phone's battery and the power supply powers the phone. We cannot completely remove the battery because the phone would then not turn on.

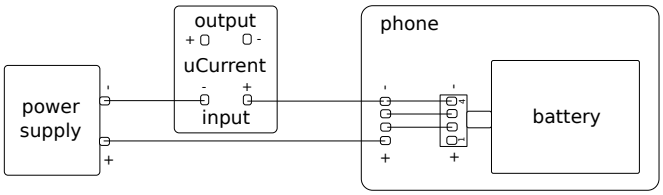


Figure 4.1: Connection between the power supply, the uCurrent device, and the phone.

To send and receive data we connect the phone via USB to a computer. We disable the USB charging on the device using an Android app that we have developed, which is free and is available for downloading [8] in Google Play.

Our oscilloscope is configurable and its frequency can be set up to 500 Mhz. The Monsoon Power Monitor [9], one of the most used energy hardware profilers, samples the energy consumption of the connected device at a frequency of $5\,kHz$. We set the frequency of our oscilloscope to 125 kHz; therefore, a measure is taken each eight microseconds. A low sampling frequency can make it hard to assess the energy consumption of short methods.

Let us consider a method M_A with an execution time of 91.96 milliseconds. Sampling at $125\,kHz$ (one sample each eight microseconds) or $5\,kHz$ (one sample each 0.2 milliseconds) does not make a difference as enough data points will be collected. However, let us consider a method M_B which execution lasts 732 microseconds. Measuring at $5\,kHz$, limits the collection of data points about M_B to no more than three samples, while measuring at $125\,kHz$ we could collect data points up to 92 samples. If a method execution last more than one millisecond, then the errors will generally averaged out, making the energy estimation error low or even negligible. However, in methods of short duration (less than one millisecond) the error may be higher.

Li *et al.* (2014a) studied what granularity of measurements is sufficient for measuring energy

8. https://play.google.com/store/apps/details?id=ruben.nexus4usbcharging
9. https://www.msoon.com/LabEquipment/PowerMonitor/

consumption. They concluded that nanosecond level measurement is sufficient to capture all API calls and methods. But nanosecond level measurement raises another problem, the bottleneck in high-frequency power sampling due to the storage system, which cannot save power samples at the same frequency as the power meter can generate them. However, we found that sampling at $125\,kHz$ just accounts for about 0.7% underestimation error (Saborido *et al.*, 2015). Therefore, we consider that $125\,kHz$ is sufficient to measure the energy consumption of mobile apps.

Battery Usage and Battery Life

Users and developers may have difficulty interpreting energy consumption (power usage) measurements. However, this metric can be translated into battery usage and–or battery life.

Battery usage is the percentage of battery charge that is consumed by an app or a block of code. We calculate battery usage using Equation 4.1 (Sahin *et al.*, 2014b), where E is the energy consumption (in Joules) of an app or block or code, and C_B and V_B are the electric charge (in Ah) and voltage (in V), respectively, of the phone's battery.

$$Battery_{Usage} = \frac{E}{V_B} \times \frac{1,000}{C_B \times 3,600} \times 100 \tag{4.1}$$

Battery life is the duration of the battery in hours. We consider the battery life of an app or block of code to be the time (in hours) that it takes to drain the battery if a particular scenario or test case is continuously run. We calculate battery life using Equation 4.2, where *Load* is the average power usage of a load (an app or a block of code).

$$Battery_{Life} = \frac{C_B \times V_B}{Load} \tag{4.2}$$

For the Nexus 4 phone, $V_B = 3.8$ and $C_B = 2,100$.

4.3.2 CPU Usage

We collect CPU usage using the same approach than Gui *et al.* (2015). We run in background the `top` command on the phone to obtain the percentage of CPU usage associated to a particular process.

When we need to measure CPU time associated to a particular method, we use the Android profiler. Execution traces generated by the Android profiler show both the inclusive and

exclusive CPU times (as well as the percentage of the total time). Exclusive time is the time spent in the method. Inclusive time is the time spent in the method plus the time spent in any called functions.

4.3.3 Memory Usage

We measure memory usage running in background the `dumpsys meminfo` command on the phone. We measure memory using the Proportional Set Size (PSS), as the Android documentation [10] suggests. It measures the app's RAM use, including shared pages across processes. All RAM pages that are unique to a process directly contribute to its PSS value, while pages that are shared with other processes contribute to the PSS value only in proportion to the amount of sharing.

4.3.4 Network Usage

We collect network usage using the `tcpdump` [11] command on the phone. This approach has been used by Gui *et al.* (2015). `tcpdump` captures packets from the Wi-Fi and cellular connections. We use this tool via the Android command `adb` to capture the numbers of bytes transmitted over the network connection. We redirect the output of this tool to a file that we store on the phone.

4.3.5 Automation Process

To collect performance metrics of apps, we run each app under study and we play a particular scenario (or test case). We run apps and play scenarios automatically to get measurements for several runs without introducing variations in execution time due to user fatigue or skillfulness. We run each app 20 or more times to get statistical results. To avoid any kind of interference during the measures, we only run on the phone the essential Android services. In addition, after each run, we uninstall the app under study and we clean the cache to keep the phone in the same state between different apps and runs.

A description of the steps is given in Algorithm 1:, which we implement as a Python script. For simplicity we include all the performance metrics in the same script but, in fact, we collect power usage individually to avoid any impact of other metrics' measurements on it. We run all apps before moving to the next run. Thus, we reduce the chance that the cache memory on the phone stores information related to the app run, which can cause the app to

10. https://developer.android.com/studio/profile/investigate-ram.html
11. http://www.androidtcpdump.com/

run faster after some executions. Finally, we process the obtain data to generate a Comma Separated Values (CSV) file that contains, for each app under study and run, the median value of power, CPU, memory, and network usages, and the total energy consumption (sum of the energy associated to each sample).

Algorithm 1: Collecting performance metrics of Android apps.

Input: list of apps under study, scenario of usage (or test cases) for each app, and number of runs

1: **for each** run **do** ▷ At least 20 runs, to get statistical results
2: **for each** app **do**
3: Install app. ▷ `adb install` app
4: Run app. ▷ `adb shell am start` app
5: Start `top` command. ▷ `adb shell 'top -d 1 | grep` app `> cpu.txt' &`
6: Obtain PID of `top` command. ▷ `adb shell ps | grep -w top | awk 'print $2'`
7: Start `dumpsys` command. ▷ `adb shell dumpsys meminfo` app `| grep TOTAL > memory.txt' &`
8: Obtain PID of `dumpsys` command. ▷ `adb shell ps | grep -w dumpsys | awk 'print $2'`
9: Start `tcpdump`. ▷ `adb shell tcpdump -s 0 -n -B 30000 -i wlan0 -w network.txt &`
10: Obtain PID of `tcpdump`. ▷ `adb shell ps | grep tcpdump | awk 'print $2'`
11: Start oscilloscope (using the LibTiePie library). ▷ `scp.start()`
12: Play scenario (or run test case). ▷ `monkeyrunner` scenario (or `adb shell am instrument ...`)
13: Stop oscilloscope. ▷ `scp.stop()`
14: Stop `tcpdump`. ▷ `adb shell kill -SIGTERM` PID of `tcpdump` command
15: Stop `dumpsys` command. ▷ `adb shell kill -SIGTERM` PID of `dumpsys` command
16: Stop `top` command. ▷ `adb shell kill -SIGTERM` PID of `top` command
17: Stop app. ▷ `adb shell am force-stop` app
18: Clean app files. ▷ `adb shell pm clear` app
19: Uninstall app. ▷ `adb uninstall` app
20: Download file containing network usage. ▷ `adb pull` network.txt `.`
21: Delete file containing network usage. ▷ `adb shell rm network.txt`
22: Download file containing memory usage. ▷ `adb pull` memory.txt `.`
23: Delete file containing memory usage. ▷ `adb shell rm memory.txt`
24: Download file containing CPU usage. ▷ `adb pull` cpu .txt `.`
25: Delete file containing CPU usage. ▷ `adb shell rm cpu.txt`
26: **end for**
27: **end for**

4.4 Data Analysis

We perform Wilcoxon rank sum tests (Hollander et Wolfe, 1973) to determine whether the difference between performance metrics of two different variables is statistically significant. We chose to use this test because we have one nominal variable (a modification or "treatment" applied to an app or app method), one measurement value (a performance metric), and because performance metric values are not normally distributed. We chose a p-value of 0.05. For the cases where there is a statistically significant difference (p-value ≤ 0.05), we compute the effect size measure Cliff's δ. It indicates the size of the effect of applying a

"treatment" to a variable (Cliff, 2014). The effect size is small for $0.147 \leq \delta < 0.33$, medium for $0.33 \leq \delta < 0.474$, and large for $\delta \geq 0.474$ (Romano *et al.*, 2006).

CHAPTER 5 AN ENERGY-AWARE REFACTORING APPROACH

Similar to traditional desktop apps, mobile device apps age as a consequence of changes in their functionality, bug-fixing, and of new features, which sometimes lead to the deterioration of the initial design (Parnas, 1994). This phenomenon, known as software decay (Eick *et al.*, 2001), is manifested in the form of design flaws or anti-patterns (poor design choices).

An example of anti-pattern is the *lazy class*, which occurs when a class has few responsibilities in an app. A *lazy class* typically is comprised of methods with low complexity and is the result of speculation in the design and–or implementation stage. Another common anti-pattern is the *blob class*, also known as *god class*, which is a large and complex class that centralizes most of the responsibilities of an app, while using the rest of the classes merely as data holders. A *blob class* has low cohesion and hinders software maintenance, making code hard to reuse and understand. There also are anti-patterns that could cause battery drain. An example of such anti-pattern is *binding resources too early class* (Gottschalk *et al.*, 2013). It occurs when a class switches on energy-intensive components of a mobile device (Wi-Fi or GPS) when they cannot interact with the user (for example, when the app is loading). Because resource management is critical for mobile device apps, developers should avoid these anti-patterns. Recently, researchers and practitioners have proposed approaches and tools to detect (Marinescu, 2004; Moha *et al.*, 2010) and correct (Tsantalis *et al.*, 2008) anti-patterns. However, these approaches only focus on object-oriented anti-patterns and do not consider mobile development concerns.

In this chapter we want to show the impact of design and implementation choices on energy consumption. We present EARMO, a novel anti-pattern correction approach that accounts for energy consumption when refactoring mobile anti-patterns. First, we conduct a preliminary study to analyze the impact of eight well-known object-oriented and mobile device specific anti-patterns on energy consumption. Second, we propose EARMO to leverage information about the energy cost of anti-patterns to generate refactoring sequences automatically. We focus our research on a subset of Android apps, evaluating EARMO on a testbed of open-source Android apps extracted from the F-Droid marketplace [1] (an Android app repository of open-source Android apps).

1. https://f-droid.org

5.1 Preliminary Study

Understanding if anti-patterns affect the energy consumption of mobile apps is important for researchers and practitioners interested in improving the design of apps through refactoring. Specifically, if anti-patterns do not significantly impact energy consumption, then it is not necessary to control for energy consumption during a refactoring process. On the other hand, if anti-patterns significantly affect energy consumption, developers and practitioners should be equipped with refactoring approaches that control for energy consumption during the refactoring process, in order to prevent a deterioration of the energy efficiency of apps.

With this preliminary study we want to check if the energy consumption of mobile device apps with anti-patterns differs from the energy consumption of apps without anti-patterns. But we also want to analyze whether certain types of anti-patterns lead to more energy consumption than others.

5.1.1 Subject Object-oriented and Android Anti-patterns

We consider two categories of anti-patterns: object-oriented anti-patterns, and Android anti-patterns. Next we present the details of the considered anti-patterns types an the refactoring strategies used to remove them. We select these anti-patterns because they have been found in mobile device apps (Hecht *et al.*, 2015, 2016), and they are well defined in the literature with recommended steps to remove them (Brown *et al.*, 1998; Fowler, 1999; Gottschalk, 2013).

Object-oriented Anti-patterns

Blob class (BL) (Brown *et al.*, 1998) is a large class that absorbs most of the functionality of the system with very low cohesion between its constituents. *Move method* (MM) is the strategy used to remove this anti-pattern, moving the methods that does not seem to fit in the *blob class* abstraction to more appropriate classes (Seng *et al.* (2006)).

Lazy class (LC) (Fowler, 1999) is a small class with low complexity that do not justify their existence in the system. *Inline class* (IC) is the strategy used to remove this anti-pattern, moving the attributes and methods of this class to another class in the system.

Long-parameter list (LP) (Fowler, 1999) is a class with one or more methods having a long list of parameters, specially when two or more methods are sharing a long list of parameters that are semantically connected. *Introduce parameter object* (IPO) is the strategy used to remove this anti-pattern. It extracts a new class with the long list of parameters and replace the method signature by a reference to the new object created. Then access to this parameters

through the parameter object.

Refused bequest (RB) (Fowler, 1999) is a subclass that uses only a very limited functionality of the parent class. *Replace inheritance with delegation* (RIWD) is the strategy used to remove this anti-pattern. It removes the inheritance from the RB class and replace it with delegation through using an object instance of the parent class.

Speculative generality (SG) (Fowler, 1999) exists when there is an abstract class created to anticipate further features but it is only extended by one class, adding extra complexity to the design. *Collapse hierarchy* (CH) is the strategy used to remove this anti-pattern, moving the attributes and methods of the child class to the parent, and removing the `abstract` modifier.

Android Anti-patterns

Binding Resources too early (BE) (Gottschalk, 2013) refers to the initialization of high-energy-consumption components of the device before they can be used. *Move resource request to visible method* (MRM) is the strategy used to remove this anti-pattern, moving the method calls that initialize high-energy consumption hardware to a suitable Android event. For example, move method call for `requestlocationUpdates`, which starts GPS device, after the app is visible to the app/user (`OnResume` method).

HashMap usage (HMU) (Hecht *et al.*, 2016) exists when the map implementation `HashMap` is used instead of `ArrayMap`, which is defined by Google as a more memory efficient map implementation. *Replace HashMap with ArrayMap* (RHA) is the strategy used to remove this pattern. It imports `ArrayMap` and replace `HashMap` declarations with the `ArrayMap` implementation.

Private getters and setters (PGS) (Hecht *et al.*, 2016) refers to the use of private getters and setters to access a field inside a class. It decreases the performance of apps because of simple inlining of Android virtual machine that translates this call to a virtual method called, which is up to seven times slower than direct field access. *Inline private getters and setters* (IGS) is the strategy used to remove this anti-pattern inlining the private methods and replacing the method calls with direct field access.

5.1.2 Data Extraction and Analysis

In order to study the impact of anti-patterns on energy consumption, we download from F-Droid 20 apps that respect our selection criteria: apps that do not require to have username and password for specific websites, apps written in English to fully understand their

functionality, apps that compile and do not crash in the middle of execution, and apps that contain at least one instance of any of the anti-patterns studied. We detect anti-patterns using an automated approach called ReCon (Morales *et al.*, 2017b). It also allows us to obtain sequence of refactorings to remove anti-patterns, which can be applied using the Android Studio and–or the Eclipse refactoring-tool-support. We use these refactoring tools to minimize human mistakes. In addition, we verify the correct execution of apps after each refactoring is applied.

For each type of anti-pattern, there are three different apps containing an instance of the anti-pattern. We refactor these apps to obtain versions without the anti-pattern. They are what we call the refactored version of the apps. For each app we define a scenario that exercises the code containing anti-patterns. The scenarios are generated with the main objective of executing the code segment(s) related to the anti-patterns in the original version, and the refactorings applied in the refactored version. We run each app and we play its corresponding scenario while energy measurements are collected. We run 30 times each app to get median energy consumptions.

Once energy measurements are collected, we statistically compare the energy consumption between the original and refactored versions of the apps. Because we do not know beforehand if the energy consumption will be higher in one direction or in the other, we perform a two-tailed test and we estimate the differences of means between original and refactored versions.

5.1.3 Results

In total, we manually correct 24 anti-patterns inside the set of apps that make up our testbed. We compute the percentage change in median energy consumption for an app a after removing one instance of anti-pattern k at time using Equation 5.1, where $E(a^k)$ is the energy consumption (in Joules) of app a after removing an anti-pattern k, $E(a^{ori})$ the energy consumption of the original app, and $med(.)$ is the median of the energy consumption values of the 30 independent runs.

$$\gamma(E(a^k), E(a^{ori})) = \frac{med(E(a^k)) - med(E(a^{ori}))}{med(E(a^{ori}))} \times 100 \tag{5.1}$$

We show in Figure 5.1: the percentage change of the energy consumption after removing one instance of the existing anti-patterns at time for each app under study. In seven instances (30%) the differences are statistically significant, with effect sizes ranging from small to large. Specifically, we obtained three apps with large effect size (two types of anti-patterns), two cases with medium effect size, and one with small effect size. This fact suggests that different

types of anti-patterns may impact the energy consumption of apps differently.

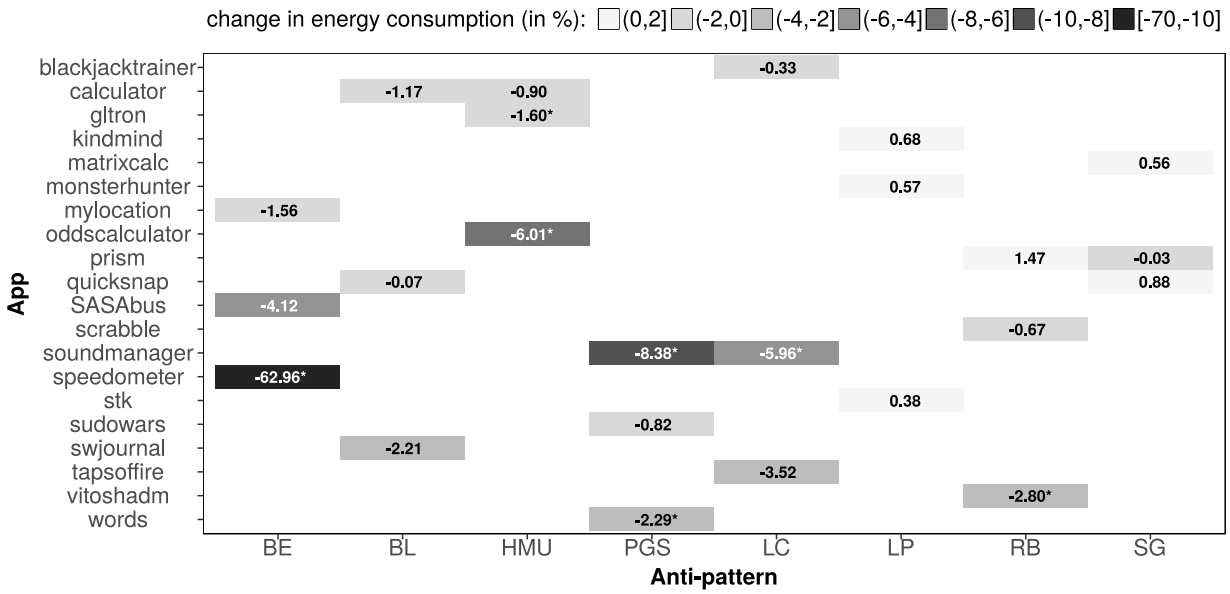


Figure 5.1: Percentage change in median energy consumption when removing different types of anti-patterns. Negative values indicate a reduction of energy consumption after refactoring, positive values indicate an increase of energy consumption. For the instances where the results are statistically significant we add an "$*$" symbol.

Regarding object-oriented anti-patterns, we observe that removing LC reduces energy consumption in app *blackjacktrainer*. This trend holds for apps *tapsoffire* and *soundmanager* respectively, with the latter one having statistically significance and large effect size. In the case of RB, two out of three apps show that removing the anti-pattern saves energy, and the difference is statistically significant for *vitoshadm*. For the BL anti-pattern, all refactored versions report a decrease in energy consumption, though the differences are not statistically significant. Concerning LP and SG, both report a negative impact on energy consumption after refactoring. While for LP, all the apps point toward more energy consumption, in the case of SG, the energy consumption is increased in two out of three apps after refactoring. We explain the result obtained for LP by the fact that the creation of a new object (the parameter object that contains the long list of parameters) adds to some extent more memory usage. For SG we do not have a plausible explanation for this trend. For both anti-patterns, the obtained differences in energy consumption is not statistically significant, hence we cannot conclude that these two anti-patterns always increase or decrease energy consumption.

Regarding Android anti-patterns, for HMU and PGS we obtain statistically significant results for two apps. For BE the result is statistically significant for one app. In all cases,

apps that contained these anti-patterns consumed more energy than their refactored versions that did not contain the anti-patterns. This finding is consistent with the recommendation of Gottschalk *et al.* (2013) that advise to remove HMU, PGS, and BE from Android apps, because of their negative effects on energy consumption. Note that the amount of energy saved is influenced by the context in which the application runs. For example, *SASAbus*, which is a bus schedule app, downloads the latest bus schedule at start, consuming a considerable amount of data and energy. As a result, the gain in energy for relocating the call method that starts the GPS device is negligible in comparison to the overall scenario. *mylocation* is a simpler app that only provides the coordinated position of mobile users. This app optimizes the use of the GPS device by disabling several parameters, like altitude and speed. For this app, we observe a consistent improvement when the anti-pattern is removed, but in a small amount. On the other hand, we have *speedometer*, which is a simple app as well, that measures user's speed but using high precision mode. High precision mode uses GPS and Internet data at the same time to estimate location with high accuracy. In *speedometer*, we observe a high reduction in energy consumption when the anti-pattern is corrected in comparison with the previous two apps.

Based on this preliminary study, there is evidence to believe that *binding resources too early*, *private getters and setters*, *refused bequest*, and *lazy class* anti-patterns can improve energy efficiency in some cases. We do not find any statistically significant cases were removing an anti-pattern increases energy consumption. Removing *blob*, *long parameter list*, and *speculative generality* anti-patterns does not produce a statistically significant increase or decrease. Anyway, the impact of different types of anti-patterns on the energy consumption of mobile apps is not the same.

Energy Consumption Coefficient for each Refactoring Operation

Based on the results obtained we estimate the impact of each refactoring operation on energy consumption. We define $\delta E(k)$ as the global refactoring energy consumption coefficient to remove anti-pattern type k. We take three apps from our testbed for each type of anti-pattern k and we compute $\delta E(k)$ using Equation 5.2, where A^k is the set of apps that were refactored to remove a single instance of anti-pattern type k.

$$\delta E(k) = med\left(\frac{\gamma(E(a^k), E(a^{ori}))}{100}\right); \forall a^k \in A^k \tag{5.2}$$

We show in Table 5.1: the energy consumption coefficient $\delta E(k)$ for each refactoring operation. Note that for the MM refactoring, we did not use the energy consumption measured for

the correction of BL, as correcting a BL requires many MM operations to be applied. Hence, we measured the same apps used for BL with and without moving exactly one method to estimate the effect of this refactoring.

Table 5.1: Energy consumption coefficient by refactoring type. Negative values indicate a reduction of energy consumption after refactoring, positive values indicate an increase of energy consumption.

Refactoring operation (k)	$\delta E(k)$
Collapse hierarchy (CH)	0.0056
Inline class (IC)	-0.0315
Inline private getters and setters (IGS)	-0.0237
Introduce parameter object (IPO)	0.0047
Move method (MM)	-0.0020
Move resource request to visible method (MRM)	-0.0412
Replace HashMap with ArrayMap (RHA)	-0.0160
Replace Inheritance with delegation (RIWD)	-0.0067

5.2 EARMO: Conceptual Sequence of Steps

Our approach is based on a search-based process where we generate refactoring sequences to improve the design of an app. This process involves evaluating several sequences of refactoring iteratively and the resultant design in terms of design quality and energy consumption. It takes as input an app to refactor, its energy consumption for a given scenario, and an energy coefficient for each refactoring strategy. It returns optimal refactoring sequences that remove the maximum number of anti-patterns while minimizing the energy consumption of the app. EARMO is comprised of three steps. They are summarized in Algorithm 2:.

Algorithm 2: EARMO approach.

Input: App to refactor (*app*) and the energy consumption of the app for a concrete scenario (E_0)
Output: Optimal set of refactoring sequences for *app*
1: AM = Code meta-model generation of *app* ▷ It generates a light-weight representation of the code
2: RA = Code meta-model assessment of AM ▷ It detects anti-patterns and generate a list of refactoring
3: Generation of optimal set of refactoring sequences using AM, RA, and E_0 ▷ Search-based approach

In the first step (line 1), the approach builds an abstract representation of the mobile app's design (a code meta-model). In the second step (line 2), the code meta-model is visited to search for anti-pattern occurrences. Once the list of anti-patterns is generated, EARMO determines a set of refactoring opportunities based on a series of pre- and post-conditions extracted from the anti-patterns literature (Brown *et al.*, 1998; Fowler, 1999; Gottschalk *et al.*, 2013). In the final step (line 3), it runs a multi-objective search-based approach to find the best legal

sequence of refactorings that remove the maximum number of anti-patterns in the system while improving the energy consumption of the app. In the following, we describe in detail each of these steps.

5.2.1 Code Meta-model Generation

In this step EARMO generates a light-weight representation (a meta-model) of a mobile app, using static code analysis techniques, with the aim of evolving the current design into an improved version in terms of design quality and energy consumption. A code meta-model describes programs at different levels of abstractions and provide methods to manipulate the design model and generate other models. The objective of this step is to manipulate the design model of a system programmatically. Hence, the code meta-model is used to detect anti-patterns, apply refactoring sequences, and evaluate their impact in the design quality of a system.

5.2.2 Code Meta-model Assessment

In this step we assess the quality of the code meta-model by identifying anti-patterns in its entities, and determining refactoring operations to correct them. The correction of certain anti-patterns requires not only the analysis of a class as a single entity, but also their relationship with other classes (inter-class anti-patterns). For example, to correct instances of BL in an app, EARMO needs to determine information related to the number of methods and attributes implemented by a given class, and compare it with the rest of the classes in the system. Then, it needs to estimate the cohesion between its methods and attributes, and determine the existence of "controlling" relationships with other classes. After performing these inter-class analysis, EARMO can propose MM refactorings to redistribute the excess of functionality from *blob classes* to related classes. Before adding a refactoring operation to the list of candidates, EARMO validates that it meets all pre- and post-conditions for its refactoring strategy to preserve the semantic of the code (Opdyke, 1992). For example, a pre-condition is that we cannot move a method to a class where there is a method with the same signature. An example of post-condition is that once we move a method from one class to another, there is no method in the source class that has the same signature as the method that was moved.

5.2.3 Generation of Optimal Set of Refactoring Sequences

In this final step, EARMO finds different refactoring sequences that remove a maximum number of anti-patterns while reduces the energy consumption of mobile apps. EARMO uses EMO algorithms to obtain optimal refactoring sequences.

Solution representation

For EARMO a solution is a refactoring sequence. We represent a refactoring sequence as a vector, where each element represents a refactoring operation to be applied. Each refactoring operation is composed of several fields like an identification number (ID), type of refactoring, the qualified name of the class that contains the anti-pattern, and any other field required to apply the refactoring in the model. For example, in a MM operation we also need to store the name of the method to be moved and the name of the target class.

Variation operators

For EMO algorithms we implement the cut and splice technique as crossover operator. It consists in randomly setting a cut point for two parents and recombining with the elements of the second parent's cut point and vice-versa, resulting in two individuals with different lengths. We provide an example in Figure 5.2:.

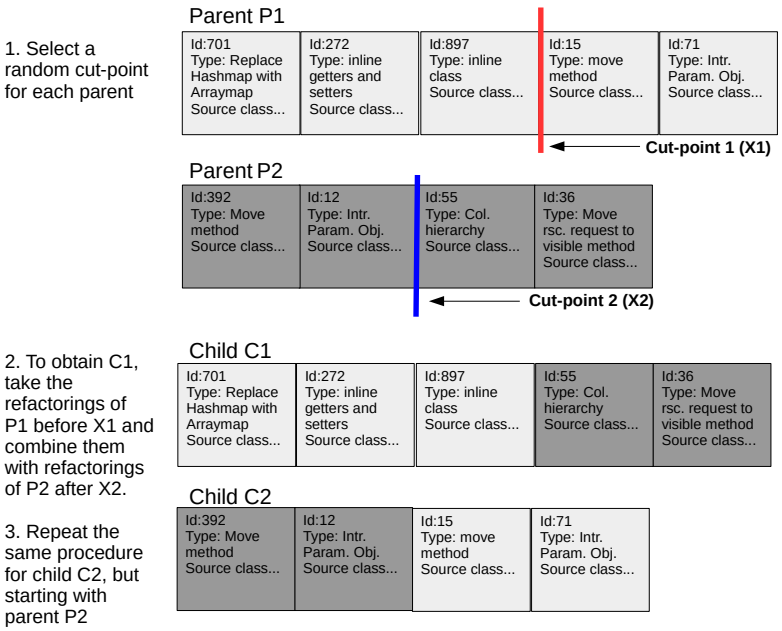


Figure 5.2: Example of cut and slice technique used by EARMO as crossover operator.

For mutation, we consider the same operator used by Morales *et al.* (2016) that consists of choosing a random point in the sequence and removing the refactoring operations from that point to the end. Then, we complete the sequence by adding new random refactorings until there are no more valid refactoring operations to add (operations that do not cause conflict with the existent ones in the sequence). We provide an example in Figure 5.3:.

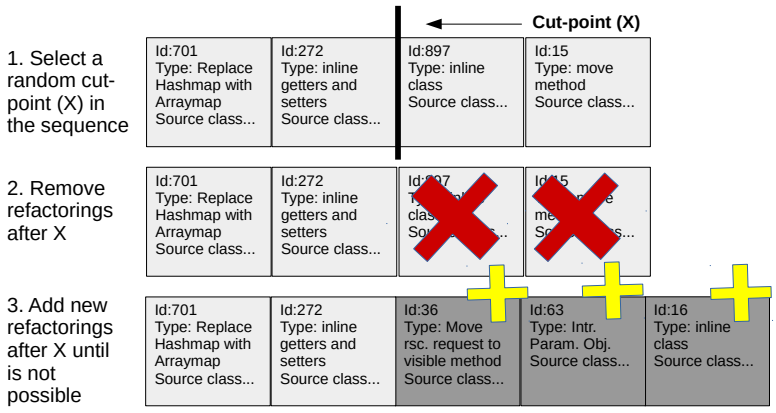


Figure 5.3: Example of the mutation operator used by EARMO.

Objective Functions.

We define two objective functions to evaluate the quality and the energy consumption of a solution, respectively.

The function to evaluate the quality of the design is $DQ = 1 - \frac{NDC}{NC \times NAT}$, where NDC is the number of classes that contain anti-patterns, NC is the number of classes, and NAT is the number of different types of anti-patterns. The value of DQ, which is normalized between 0 and 1, rises when the number of anti-patterns in the app is reduced. A value of 1 represents the complete removal of anti-patterns, hence we aim to maximize the value of DQ. This objective function was introduced by Ouni *et al.* (2013). We follow this formulation because it is easy to implement and computationally inexpensive.

To evaluate the energy consumption of an app after refactoring we define the following formulation: let E_0 be the estimated energy consumption of an app a, and r_i a refactoring operation in a sequence $S = (r_1, \ldots, r_n)$. We estimate the energy consumption of the app resulting from the application of the refactoring sequence S to the app a using Equation 5.3, where $\delta E(r_i)$ is the energy coefficient value of the refactoring operation r_i to remove an anti-pattern. These coefficient values were estimated in our preliminary study and summarized

in Table 5.1:. We aim to minimize the value of E during the search process.

$$E(a) = E_0 + \sum_{i=1}^{n} E_0 \times \delta E(r_i) \tag{5.3}$$

Note that we estimate the energy consumption of a refactoring sequence because measuring in real-time can be prohibitive. It requires to apply each refactoring element of the sequence in the code, compile it, generate the binary code and download it into the phone; all of these steps for each time the search-based process requires to evaluate a solution.

In Algorithm 3:, we present a generic pseudocode for the EMO algorithms used by our approach. The process starts by generating an initial population of refactoring sequences from the meta-model assessment step (line 1). Next, for each solution, it applies the corresponding refactoring sequence in the code meta-model and measures the design quality (number of anti-patterns) and the energy saved by applying the refactorings included in the sequence (lines 3-8). The next step is to extract the non-dominated solutions (line 9). From line 10 to 21, the main loop of the metaheuristic process is executed. The goal is to evolve the initial population, using the variation operators described before, to converge to the Pareto optimal front. The stopping criterion, which is defined by the software maintainer, is a fixed number of evaluations. Finally, the optimal refactoring sequences are retrieved (line 22).

5.2.4 Output

As output, EARMO returns optimal refactoring sequences (a Pareto optimal front) in terms of design quality and energy consumption. According to the concept of Pareto dominance, every Pareto point is an equally acceptable solution of the multi-objective optimization problem (Miettinen, 1999), but developers might show preference over the ones that favors the metric they want to prioritize. They could select the refactoring sequence that improves more the energy consumption, or the one that improves the maintainability of their code. Other developers might be more conservative and select solutions located in the middle of these two objectives. Developers have the last word, and EARMO supports them by providing different alternatives.

5.3 Case Study

We evaluate the effectiveness of EARMO at improving the design quality of mobile device apps while optimizing energy consumption. The quality focus is the improvement of the design quality and energy consumption of mobile apps, through search-based refactor-

Algorithm 3: Generation of optimal set of refactoring sequences in EARMO.

Input: Code meta-model generation and assessment (AM and RA) and energy consumption (E_0) of *app*
Output: Set of non-dominated solutions

P_0 = Generate initial population from RA
$PF = \emptyset$ ▷ PF is the set of non-dominated solutions
for each $S_i \in P_0$ **do** ▷ Evaluation of individuals (S_i is a refactoring sequence)
    $AM' =$ clone AM
    Apply sequence of refactorings S_i in AM'
    Compute design quality for solution S_i using AM'
    Compute energy consumption for solution S_i using AM' and E_0
end for
$PF =$ non-dominated solutions in P_0
$t = 1$
while not stopping criterion **do**
    $P_t =$ Apply variation operators to P_{t-1} ▷ Generation of new individuals in the population
    for each $S_i \in P_t$ **do** ▷ Evaluation of individuals
        $AM' =$ clone AM
        Apply sequence of refactorings S_i in AM'
        Compute design quality for solution S_i using AM'
        Compute energy consumption for solution S_i using AM' and E_0
    end for
    $PF =$ non-dominated solutions in $(PF) \cup (P_t)$
    $t = t + 1$
end while
return PF ▷ The set of non-dominated solutions (the Pareto optimal front)

ing. The perspective is that of practitioners interested in improving the design quality of their apps while controlling for energy consumption, and researchers interested in developing automated refactoring tools for mobile apps. The context consists of the 20 Android apps and anti-patterns studied in our preliminary study. Given an app, we use as input for EARMO the energy consumption of the app and the energy coefficient of refactoring strategies, both also computed in our preliminary study. The code meta-model is generated using Ptidej Tool Suite (Guéhéneuc, 2005). We select this tool suite because it has more than ten years of active development and it is maintained in-house. We use three multi-objective metaheuristics (MOCell, NSGAII, and SPEA2) for the generation of optimal refactoring sequences. We choose them because they are well-known evolutionary techniques that have been successfully applied to solve optimization problems including refactoring (Harman *et al.*, 2012b; Ouni *et al.*, 2013).

Results

In order to assess the effectiveness of EARMO, we study to what extent the approach can remove anti-patterns while controlling for energy consumption. We also examine what is the

precision of the energy improvement reported by EARMO. Lastly, we study to what extent is design quality improved by EARMO according to an external quality model.

The search-based approach has been implemented in jMetal, an object-oriented Java-based framework for multi-objective optimization (Durillo et Nebro, 2011). We use number of evaluations as the stopping criteria. We empirically tried different number of evaluations in the range of 1,000 to 5,000 and found 2,500 to be the best value. We use for each EMO algorithm as selection operator the default one proposed by jMetal for each algorithm. We use a default value of 100 individuals for population size, and we set the crossover and mutation probabilities to 0.8. We selected these parameters using a factorial design and choosing the best values in terms of two popular metrics, the *hypervolume* (Zitzler et Thiele, 1999b) and the *spread* (Deb *et al.*, 2002). Because the EMO algorithms employed are non-deterministic, the results might vary between different executions. Hence, we run each metaheuristic 30 times, for each studied app, to provide statistical significance. As a result, we obtain three reference Pareto front approximations (one per algorithm) for each app. From these fronts, we extract a global reference front that combines the best results of each metaheuristic for each app and, after that, dominated solutions are removed.

Concerning the particular problem of automated-refactoring, the initial size of the refactoring sequence is crucial to find the best sequence in a timely manner. If the sequence is too long, the probability of conflicts between refactorings rises, affecting the search process. On the other hand, small sequences produce refactoring solutions of poor quality. To obtain a trade-off between this two scenarios, we experimented running the EMO algorithms with four relative thresholds: 25, 50, 75, and 100 percent of the total number of refactoring opportunities, and found that 50 percent is the most suitable value for our search-based approach.

Removal of Anti-patterns while Controlling for Energy Consumption

EARMO found 2.5 non-dominated solutions on average with a maximum of eight solutions for one of the apps under study. Thus, for the studied apps, a software maintainer has approximately three different solutions to choose from. The number of non-dominated solutions are the number of refactorings sequences that achieved a compromise in terms of design quality and energy consumption. We also observe that more than 65% of the apps contain more than one solution. To have an insight on those apps, we present in Figure 5.4: the Pareto front for apps where EARMO finds more than one non-dominated solution.

To evaluate the Pareto optimal solutions found by EARMO we compare the improvement in design quality and energy consumption for each of them with respect to the corresponding

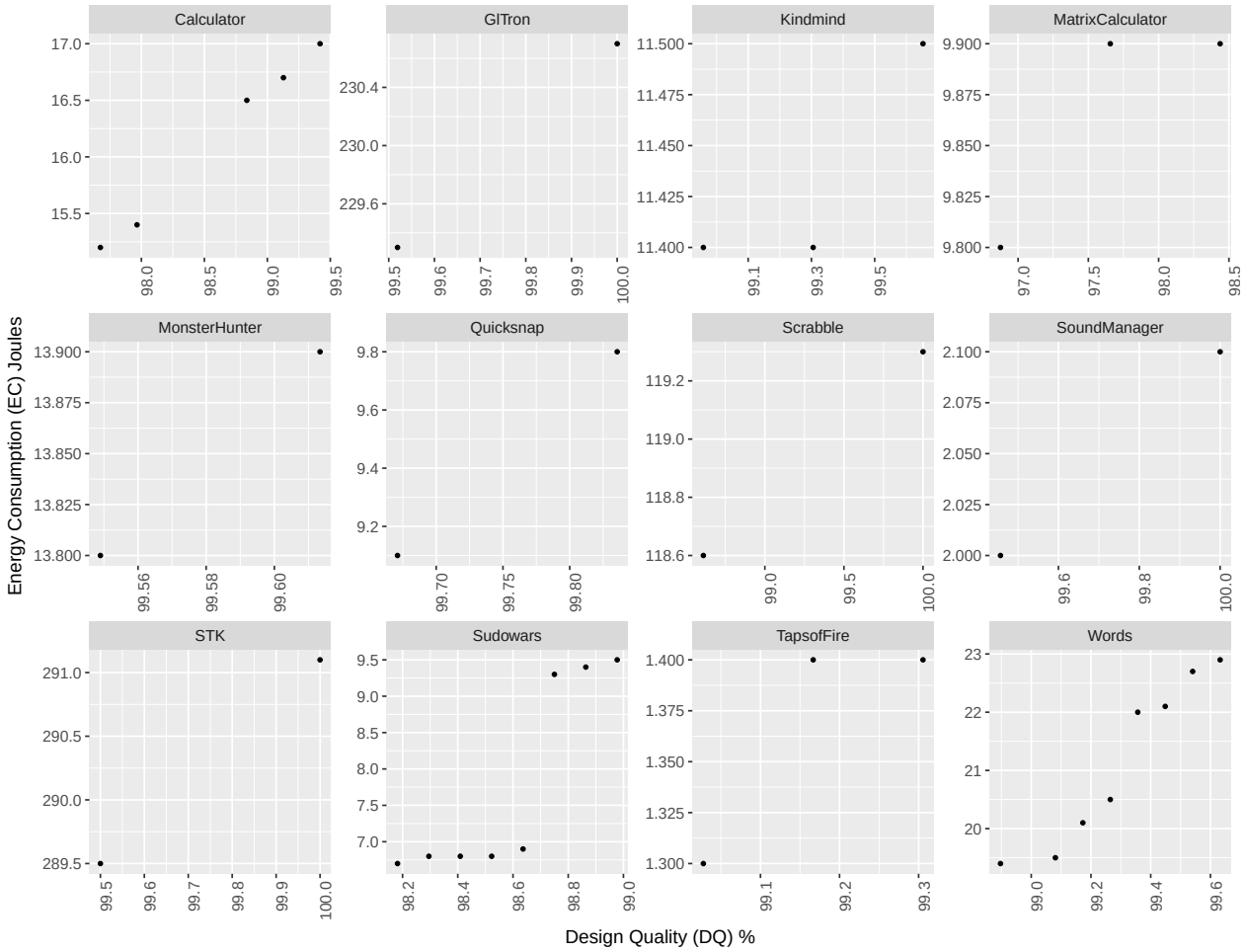


Figure 5.4: Pareto front of apps with more than one non-dominated solution found by EARMO for the case study. Each point represents a solution (refactoring sequence) with their corresponding values, design quality (*x-axis*) and energy consumption (*y-axis*). The most attractive solutions are located in the bottom right of each plot, because they maximize design quality while minimizing energy consumption.

original apps. We define design improvement (DI) as the delta of anti-patterns occurrences between the refactored (a') and the original app (a). It is computed using Equation 5.4, where $AC(a)$ is the number of anti-patterns in an app a with $AC(a) \geq 0$. The sign of DI expresses an increment (+)/decrement (-) and the value represents the improvement amount in percentage. High negative values are desired.

$$DI(a) = \frac{AC(a') - AC(a)}{AC(a)} \times 100. \tag{5.4}$$

We define estimated energy consumption improvement (EI) as the improvement in the energy consumption of an app a after refactoring operation(s). It is computed using Equation 5.5, where $E(a)$ is the energy consumption of an app a with $E(a) > 0$. The sign of EI expresses an increment (+)/decrement (-) and the value represents the amount in percentage. High negative values are desired.

$$EI(a) = \frac{E(a') - E(a)}{E(a)} \times 100. \tag{5.5}$$

We highlight a median correction of 84% of anti-patterns and estimated energy consumption improvement of 48%. We present, in Table 5.2:, the number of non-dominated solutions found for each app (column 2), the minimum and maximum values with respect to DI (columns 3-4), and EI metrics (columns 5-6). The number of non-dominated solutions are the number of refactorings sequences that achieved a compromise in terms of design quality and energy consumption. We observe that the results for DI and EI metrics are satisfactory, and we find that in nine apps EARMO reaches 100% of anti-patterns correction with a maximum EI of 89%.

We conclude that including energy-consumption as a separate objective when applying automatic refactoring can reduce the energy consumption of a mobile app without impacting the anti-patterns correction performance.

Precision of the Energy Improvement Reported by EARMO

We perform an energy consumption validation experiment to evaluate the accuracy of EARMO using our measurement setup. This is important to observe how close is the estimated energy improvement compared to the real measurements.

For each app we do the following. First, we compute refactoring recommendations using EARMO and implement the refactorings in the source code of the app. To validate the estimations of EARMO, we play the role of a software maintainer who wants to prioritize

Table 5.2: Minimum and maximum values of DI and EI for each app after applying EARMO.

	Solutions	DI		EI	
App		Min.	Max.	Min.	Max.
blackJacktrainer	1	-75	-75	-6.14	-6.14
calculator	5	-75	-93.75	-48.07	-53.55
gltron	2	-93.75	-100	-25.85	-26.32
kindmind	3	-80	-93.33	-18.42	-18.76
matrixcalculator	3	-33.33	-66.67	0.28	-0.67
monsterhunter	2	-81.63	-83.67	-43.95	-44.42
mylocation	1	-100	-100	-2.05	-2.05
oddscalculator	1	-100	-100	-14.64	-14.64
prism	2	-85.71	-100	-7.94	-9.18
quicksnap	2	-92.31	-96.15	-83.65	-84.88
SASAbus	1	-81.82	-81.82	-27.09	-27.09
scrabble	2	-85.71	-100	-12.36	-12.92
soundmanager	2	-94.44	-100	-35.36	-35.83
speedometer	1	-100	-100	-6.17	-6.17
stk	2	-83.33	-100	-11.05	-11.53
sudowars	8	-60.29	-76.47	-48.77	-63.93
swjournal	1	-100	-100	-5.67	-5.67
tapsoffire	3	-82.93	-87.8	-88.26	-89.21
vitoshadm	1	-100	-100	-3.57	-3.57
words	8	-75	-91.67	-56.83	-63.37

the energy consumption of his/her app over design quality. Second, we define a new scenario for the app if we consider that the scenario used in the preliminary study do not reflect a typical usage. The reason is that we want to reflect the actions that a user typically will perform with an app, according the purpose of their creators. Third, we measure the energy consumption of the original and refactored versions of the apps while the new scenario is played. Then, we compute the difference between the obtained values for both versions, the original one with anti-patterns and the refactored one. To get statistical results we run each version of each app 30 times.

Concerning the accuracy of the energy estimation, EARMO values are more optimistic than the actual measurements but in an acceptable level. If we compare the results obtained by EARMO compared with the preliminary study, the energy consumption trend holds for all the apps. However it is hard to make a fair comparison because in the preliminary study we measure the effect of one instance of each anti-pattern type at a time, but in the energy consumption validation of EARMO we apply several refactorings. Yet, the median error is in acceptable level of 12%.

We also compute the percentage of battery charge that is consumed by each app using Equation 4.1. Then, we use this information to compute the battery life using Equation 5.6, where ET is the execution time of the app (in seconds). We consider the battery life of an

app to be the time (in hours) that it takes to drain the battery if the scenario associated to the app is continuously run.

$$Battery_{Life}{}' = \frac{(ET \times 100)/Battery_{usage}}{3600} \tag{5.6}$$

Finally, we calculate the average battery life for each app (refactored and original) and subtracted these values to obtain the difference of battery life. We obtain that after applying refactoring sequences proposed by EARMO, duration of the battery can be extended from a few minutes up to 29 minutes.

Improvement of Design Quality According to an External Quality Model

We use the Quality Model for Object-Oriented Design (QMOOD) (Bansiya et Davis, 2002) to measure the impact of the refactoring sequences proposed by EARMO on the design quality of the apps. QMOOD defines six design quality attributes in the form of metric-quotient weighted formulas that can be easily computed on the design model of an app, which makes it suitable for automated-refactoring experimentations. Another reason for choosing the QMOOD quality model is the fact that it has been used in many previous works on refactoring (O'Keeffe et Cinnéide, 2006; Ouni *et al.*, 2015), which allows for a replication and comparison of the obtained results.

Next, we present a brief description of the quality attributes used in this study. Formulas for computing these quality attributes are described in Table 5.3:. Details about them can be found in the original source (Bansiya et Davis, 2002).

— *Reusability*: the degree to which a software module or other work product can be used in more than one software program or software system.
— *Flexibility*: the ease with which a system or component can be modified for use in apps or environments other than those for which it was specifically designed.
— *Understandability*: the properties of a design that enables it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
— *Effectiveness*: the design's ability to achieve desired functionality and behavior using object-oriented concepts.
— *Extendibility*: the degree to which an app can be modified to increase its storage or functional capacity.

We compute the quality gain (QG) for each quality attribute and app using Equation 5.7, where $Q_y(a)$ is the quality attribute y for an app a, and a' is the refactored version of the app a. Note that since the calculation of QMOOD attributes can lead to negative values in

Table 5.3: QMOOD evaluation functions, where DSC is design size, NOM is number of methods, DCC is coupling, NOP is polymorphism, NOH is number of hierarchies, CAM is cohesion among methods, ANA is average number of ancestors, DAM is data access metric, MOA is measure of aggregation, MFA is measure of functional abstraction, and CIS is class interface size.

Quality attribute	Quality attribute calculation
Reusability	-0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibility	0.25 * DAM - 0.25 * DCC + 0.5 * MOA +0.5 * NOP
Understandability	-0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * CAM -0.33 * NOP - 0.33 * NOM - 0.33 * DSC
Effectiveness	0.2 * ANA + 0.2 * DAM + 0.2 * MOA + 0.2 * MFA + 0.2 * NOP
Extendibility	0.5 * ANA -0.5 * DCC + 0.5 * MFA + 0.5 * NOP

the original design, it is necessary to compute the absolute value of the divisor.

$$QG(Q_y) = \frac{Q_y(a') - Q_y(a)}{|Q_y(a)|} \times 100 \tag{5.7}$$

In general, the refactored apps report a median slight decrease that ranges from 0.9% to 4% for reusability, understandability and flexibility. Concerning *effectiveness* we report a median gain of 3.14%. Regarding *extendibility* we report a considerable median improvement of 41%. We attribute this increment to the removal of unnecessary inheritance (through *inline class*, *collapse hierarchy* and *refused bequest* refactorings). In fact, the *extendibility* function assigns a high weight to metrics related to hierarchy. These are good news for developers interested in improving the design of their apps through refactoring, as the highly-competitive market of Android apps requires adding new features often and in short periods of time. Hence, if they interleave refactoring before the release of a new version, it will be easier to extend the functionality of their apps.

Overall, EARMO can improve the design quality of an app, not only in terms of anti-patterns correction but also their extendibility and effectiveness.

5.4 Discussion

EARMO is a novel approach for refactoring mobile apps while controlling for energy consumption. This approach supports the improvement of the design quality of mobile apps through the detection and correction of object-oriented and Android anti-patterns. To assess the performance of EARMO, we implement our approach using three EMO algorithms and we evaluate it on a benchmark of 20 free and open-source Android apps, having different sizes and belonging to various categories.

The results of our empirical evaluation show that EARMO can propose solutions to remove a

median of 84% of anti-patterns, with a median execution time of less than a minute. We also evaluate the overall design quality of the refactored apps in terms of five high-level quality attributes assessed by an external model, and report gains in terms of understandability, flexibility, and extendibility of the resulting designs. We also quantify the battery energy gain of EARMO and we find that in a multimedia app, when the proposed scenario is executed continuously until the battery drained out, EARMO can extend battery life by up to 29 minutes. The benefits of improving design quality of the code, and potentially reducing the energy consumption of an app should not be underestimated. Battery life is one of the main concerns of mobile device users and every small action performed to keep a moderate energy usage in apps is well appreciated. Even if there is not a noticeable gain in energy reduction, software maintainers are safe to apply refactoring recommendations proposed by EARMO without fearing to introduce energy leaks.

From this study we obtain that the Android anti-patterns studied have a negative impact on the energy consumption of apps. One of these anti-patterns exist when developers use the `HashMap` implementation instead of `ArrayMap`. Although Android offers `ArrayMap` and more map implementations to choose from, the official documentation is ambiguous about the performance benefit of using these specific implementations. This fact motivates the next chapter, where we study the use of map data structures by Android developers, we conduct a survey to assess developers' perspective on Java and Android map implementations, and we perform an experimental study comparing their performance to conclude with guidelines for choosing among these map implementations.

CHAPTER 6 GETTING THE MOST FROM MAP DATA STRUCTURES

In the previous chapter, we obtained that developers can improve energy consumption by replacing in their apps a Java map implementation by an Android map implementation. A map is a data structure that is commonly used to store data as key–value pairs and retrieve data as keys, values, or key–value pairs. Although Java offers different `map` implementation classes, Android SDK offers other implementations supposed to be more efficient than the traditional `HashMap`: `ArrayMap` and `SparseArray` variants (`SparseArray`, `LongSparseArray`, `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`).

The Android documentation states that *"`ArrayMap` is designed to be more memory efficient than a traditional `HashMap`"*[1]. When keys are defined as integer primitive types, the documentation also states that *"`SparseArray` is designed to be more memory efficient than `HashMap` to map integers to objects"*[2]. The same is stated about `LongSparseArray` and long primitive types used as keys[3]. When keys are defined as integer primitive types and values are defined as integer, long, or boolean primitive types, the documentation also states that `SparseIntArray`[4], `SparseLongArray`[5], and `SparseBooleanArray`[6], respectively, are designed to be more memory efficient than a traditional `HashMap`. Android Studio, the official Android Integrated Development Environment (IDE), warns *"Use `new SparseArray` instead `new HashMap<Integer,Object>()` for better performance"* (a similar warning is also given for `SparseArray` variants). Hence, `ArrayMap` and `SparseArray` variants should be preferred over `HashMap`, at least for maps containing up to hundreds of elements according to Android developers' reference documentation. For `ArrayMap` and `SparseArray` variants the documentation claims that *"this implementation is not intended to be appropriate for data structures that may contain large number of items. It is generally slower than a traditional `HashMap`"*.

Yet, the documentation is vague because (1) it does not provide supporting evidence and quantitative information about efficiency and (2) although it discourages their use in maps containing large number of elements, it does not provide more precise numbers (for example, performance information and–or threshold levels to consider). Consequently, although the current documentation raises the awareness of developers about the advantages and limita-

1. https://developer.android.com/reference/android/util/ArrayMap.html
2. https://developer.android.com/reference/android/util/SparseArray.html
3. https://developer.android.com/reference/android/util/LongSparseArray.html
4. https://developer.android.com/reference/android/util/SparseIntArray.html
5. https://developer.android.com/reference/android/util/SparseLongArray.html
6. https://developer.android.com/reference/android/util/SparseBooleanArray.html

tions of map implementations, it does not provide them concrete evidences that could be used to make informed decisions about the implementations that are the most suitable for their apps. Expressions such as "large number" and "generally slower" are vague and they do not help developers at all. In addition to the previous, the documentation says nothing about energy consumption and neither about performance for different map-related operations and data sizes. Saving CPU and memory is a major concern for users who own low end mobile devices. But energy is also a major concern of users wanting to increase battery life. Mobile device app developers want to develop efficient apps but they need more information for choosing among Java and Android map implementations.

In this chapter we perform an experimental study comparing `HashMap`, `ArrayMap`, and `SparseArray` variants map implementations in terms of CPU time, memory usage, and energy consumption. Before, we study the use of map implementations by Android developers in two ways: we perform an observational study of 5,713 Android apps, and we conduct a survey to assess developers' perspective on Java and Android map implementations. Thus, our goal is to know more about the usage of Android map implementations in real Android apps and conclude with guidelines for choosing among map implementations regarding their performance. Thus, developers can make informed decisions about the map implementations to use in their apps.

6.1 Observational Study

We study usage patterns of Android developers of Java map implementations and the Android map implementations `ArrayMap` and `SparseArray` variants. We conduct this study to analyze the prevalence of map implementations through an observational study of Android apps available on GitHub.

We select from GitHub (repository queried and accessed December, 2016) all the projects that are available in the official Android marketplace as Android apps (all the projects that contain a link to the Google Play marketplace in their *README.md* file). We choose open-source apps because we can then analyze their source code to study the prevalence of map implementations. Then, we develop a Python script to select and download, automatically, the zip file containing the source code of each existing Android project from GitHub. Thus, we obtain 5,713 Android apps.

We also develop a Bash script to process the source code of all the apps looking for occurrences of general-purpose Java map implementations (`HashMap`, `LinkedHashMap`, and `TreeMap`) and the ones under study offered by Android (`ArrayMap` and `SparseArray` variants). This Bash

script uses the `grep` command to search for occurrences of the map implementations under study. We manually validated the results and we found four different types of false positives: (1) apps that defined a data structure named `ArrayMap` which implements the `Map` interface, (2) apps that used a TPL named `ArrayMap`, (3) apps that contained one or more strings containing text matching our patterns, and (4) apps that contained comments with text matching our patterns. We have manually checked true positives (TP), false positives (FP), and false negatives (FN) for each map implementation and projects under study. We obtaine that, on average, the precision $\left(\frac{TP}{TP+FP}\right)$ is 93.46% and the recall ($\frac{TP}{TP+FN}$) 100%. Thus, we consider that our Bash script perform well for our observational study, concerning the apps under study.

Results

We obtain that, over 5,713 apps, 1,713 (30%) apps have at least one occurrence of any Java map implementation. For `ArrayMap` and `SparseArray` variants, 419 (7%) apps have one or more occurrences of these map implementations. In total, over 5,713 apps, 2,132 (37%) use any of the studied map implementations. From now on, in this section, percentages are given with respect to this number.

We find that `HashMap` is the most used Java map implementation with 1,640 apps (77%) while the others are used less often. We obtain that 282 (13%) apps use `LinkedHashMap` and 179 (8%) use `TreeMap`. Note that different map implementations can be used in the same app. Concerning Android map implementations, we find that `ArrayMap` and `SparseArray` variants are rarely used by Android developers. Only 19 (1%) and 413 (7%) apps use `ArrayMap` or any variant of `SparseArray`, respectively.

Table 6.1: shows the number and the percentage of apps that have one or more occurrences of any combination of the Java and Android map implementations. Second column shows the number and percentage of apps that have one or more occurrences of `ArrayMap` and one or more occurrences of a Java map implementation. Third column is similar to the previous one but for `SparseArray` variants. Last column shows the number and percentage of apps that have one or more occurrences of both `ArrayMap` and any variant of `SparseArray` map implementations and one or more occurrences of a Java map implementation. As it is shown, `ArrayMap` and `SparseArray` variants are used in combination to `HashMap`. For `SparseArray` variants it makes sense, because these map implementations are used when keys and–or values are primitive types while `HashMap` can be used to store non-primitive types as keys and–or values. On the contrary, `ArrayMap` could be used as a replacement for `HashMap` but we find that only five (0.23% over 2,132 or 0.30% over the 1,640 apps using `HashMap`) apps

use exclusively `ArrayMap` instead of `HashMap`.

Table 6.1: Number and percentage of Android apps having one or more occurrences of any combination of the Java and Android map implementations.

	Android map implementation		
Java map implementation	`ArrayMap`	`SparseArray` variants	Both
`TreeMap`	3 (<1%)	46 (2%)	2 (<1%)
`LinkedHashMap`	4 (<1%)	100 (5%)	3 (<1%)
`HashMap`	14 (<1%)	314 (15%)	8 (<1%)

The Android documentation claims that `SparseArray` variants have a better memory performance than `HashMap` and, for this reason, Android Studio suggests to replace `HashMap` by `SparseArray` variants. Because we find that `HashMap` is the most used map implementation we also study if Android developers adopt the `HashMap` implementation when keys and–or values are defined as primitive types. We develop another script to process the source code of apps looking for usages of the `HashMap` implementation using primitive types as keys and–or values. From the 1,640 apps using the `HashMap` implementation, 332 (20%) of these apps use integers as keys, 64 (4%) use longs as keys, 89 (5%) use integers as keys and values, 12 (<1%) use integers as keys and longs as values, and 13 (<1%) use integers as keys and booleans as values. However, in that cases, Android recommends to replace `HashMap` with `SparseArray`, `LongSparseArray`, `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, respectively, for better memory performance.

6.2 Developers' Perspective: a Survey

We want to know why developers mostly select the Java map implementation `HashMap` instead of `ArrayMap` and `SparseArray` variants. Particularly in the cases where the Android documentation advises the opposite. We assume that this lack of use is due to developers' reluctance to try new implementations. But it is also probably due to their lack of knowledge about the possible advantages in terms of performance of switching to `ArrayMap` or any variant of `SparseArray`. We get details about developers' perspective with respect to map implementations conducting an on-line survey. All questions are optional and the survey is anonymous to encourage developers to answer (Tyagi, 1989).

We consider the 1,744 apps that use `HashMap`, `ArrayMap`, and–or any variant of `SparseArray`. We contacted those project's owners that made their email address publicly available in GitHub. The total amount of emails sent was 656. We surveyed these 656 developers and 118 (18%) responded to our survey. It is considerably larger than the typical 5% answer

rate obtained in questionnaire-based software engineering surveys (Singer *et al.* (2008)). The survey was available on-line from December 2016 to January 2017. Participants in our survey are from 35 different countries around the world. The majority of them is from USA (17, 14%), India (15, 13%), and Spain (seven, 6%). Of all 118 participants: 93 (79%) have up to four years of experience developing mobile apps, 101 (86%) participants declare to use Java as primary programming language, and 97 (82%) participants declare to use Android Studio as IDE.

The on-line survey have 14 questions: four on the usage of map implementations, five on the participants' familiarity with Android map implementations, one about importance of performance metrics, and four on the participants' background and experience. All the questions have a closed set of answers from which a participant selects, while two of them include an additional field for open comments. None of the questions are mandatory and participants are allowed to drop out at any time. The survey consists on four different sections asking about the use of map implementations, the familiarity with map implementations offered by Android, the importance of different performance metrics, and participants' profile. We use for responses a dichotomous scale (*yes*/*no*) or Likert scales with values from 1 (*never*) to 5 (*every time*). When the question is related to importance magnitude we use a similar scale with values from 1 (*not important*) to 5 (*very important*). For questions about familiarity we use the same scale with values from 1 (*not at all familiar*) to 5 (*extremely familiar*).

The survey operates on a self-selection principle. It means that the results might be skewed towards developers who are willing to answer the survey, but avoiding the self-selection principle is not feasible in practice. Question-wording effect might bias respondents towards one object if there is not enough context when comparing different objects under the same conditions. To counteract the possible question-wording effect, we take care to make questions as specific and concrete as possible to discard leading, loaded, or double-barreled questions.

Results

Demographic and experience information about participants was reported previously but results for the other three sections are presented next. For Likert scale questions we use diverging stacked bar charts to show the frequency of responses (in percentage). Replies are positioned horizontally, so "low" responses are stacked to the left of a vertical baseline and "high" responses are stacked to the right of this baseline. We consider as baseline or "neutral response" the midpoint of the scale (value 3). For each stacked bar we also add the percentage of low, neutral, and high responses. In addition, on the right, we also show the total number of participants who answered each of the questions. In all the questions we

obtained more than 110 responses (over 118 participants).

Use of Map Implementations

We ask participants the frequency of usage of the Java map implementations `TreeMap`, `LinkedHashMap`, and `HashMap`. Figure 6.1: summarizes the participants' responses. As it is shown, `HashMap` is the most often used Java map implementation (80% of participants responded *almost every time* or *every time*).

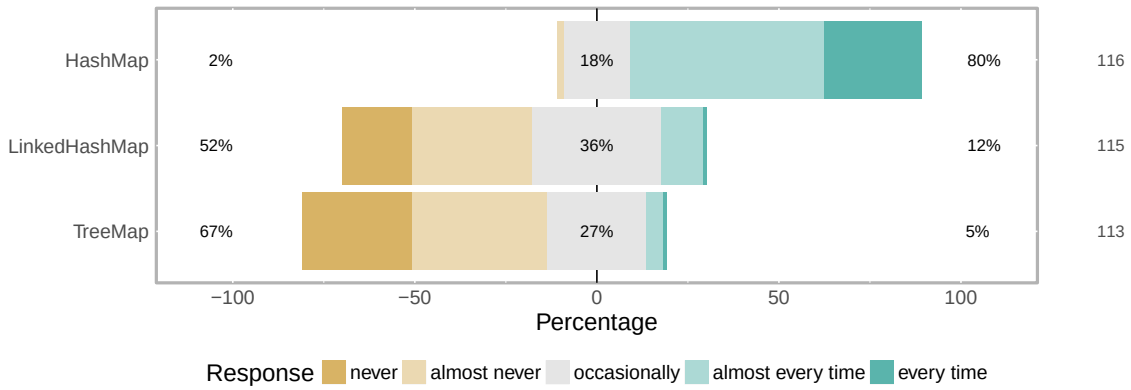


Figure 6.1: Participants' responses about the usage of most popular Java map implementations.

Concerning map-related operations (insertion, iteration, random query, and deletion), we ask participants to rate them according to their importance in their codes. Figure 6.2: summarizes the participants' responses. For insertion, iteration, and random query operations, more than 70% of the participants respond *moderately important* or *very important*. For the deletion operation, 52% of the participants consider it as an important operation while 27% of the participants have a neutral opinion about it.

Regarding the iteration operation we ask developers about the way of iterating through map structures. We receive 118 responses from participants: 94 (80%) of these participants select *for-Each loop* instead of *the Iteration pattern*, which is selected by 24 (20%) participants. In addition we ask if the iteration is used over the *key–value pairs*, the *set of keys*, or the *set of values*. We receive 118 responses: 59 (50%) participants select *key–value pairs*, 41 (35%) choose the *set of keys* because they get the value mapped to each key using the `get` method, and 18 (15%) select the *set of values* because they are not usually interested in keys.

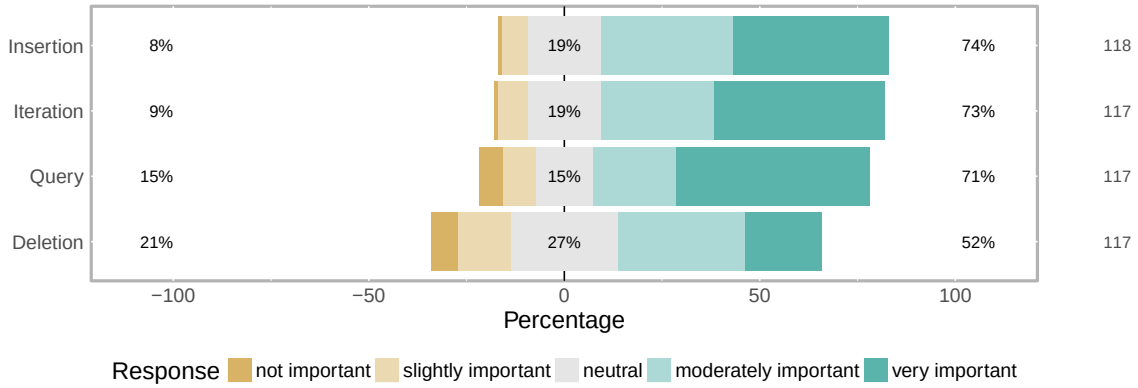


Figure 6.2: Participants' responses about importance of map-related operations.

Familiarity with Android Map Implementations

In the second section of the survey, we ask about the familiarity of developers with the Android map implementations `ArrayMap` and `SparseArray`. We focus on `SparseArray` and not on the other variants because they are used less often. Figure 6.3: summarizes the participants' responses. For `ArrayMap`, 55% of participants respond *moderate familiar* or *extremely familiar*. For `SparseArray`, 27% of participants respond *moderate familiar* or *extremely familiar*. Half of the participants respond that they are *not at all familiar* of *slightly familiar* with `SparseArray`.

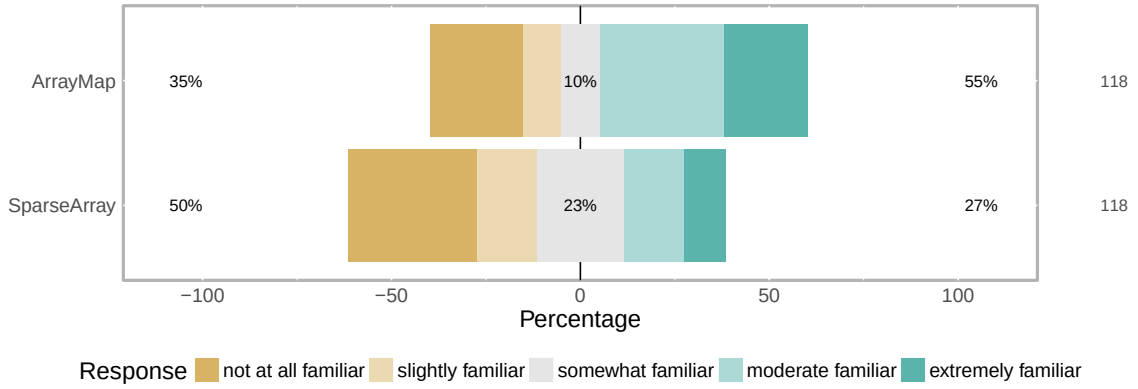


Figure 6.3: Participants' responses about familiarity with Android map implementations.

Since `ArrayMap` was introduced in Android API 19, developers from early versions of Android or even those that want to provide compatibility backwards may opt for not switch to this implementation. However, Android offers an utility class `android.support.v4.util` that

allows to use `ArrayMap` in early Android versions. We ask developers to know if they are aware of this class. We receive 118 responses from participants, from which 49 (42%) are aware. The rest, 69 (58%), could hesitate to use it if they want to bring compatibility back to previous versions, like one respondent says: "*My app is an uncommon case, I support all the way back to API level 8... I do however use SparseArray map, as it is available since API 1*".

With respect to `SparseArray`, we ask participants if they know that it is designed to be more efficient than `HashMap` when keys are integers. This question is answered by 118 participants from which 52 (44%) confirm this fact while the rest 66 (56%) answer *no*.

We ask participants if they are willing to replace `HashMap` with any of the map implementations provided by Android, if they offer better performance. We receive 118 responses from participants to this question from which 102 (86%) of these participants answered *yes*. To the 16 (14%) remaining participants, we ask why they chose *no*. In general, they answer that they use what they know and they fear of learning or using new structures wrong. Another respondent says that (s)he is concerned by the size of the app's APK, because adding the Android library would increase sizes. In addition, another respondent is worried about portability of code when Android libraries are included. We are more concrete and we also ask if they are willing to substitute `HashMap` with `SparseArray` when integers are used as key in map data structures. We receive 118 responses from participants to this question: 107 (91%) of these participants said *yes* and 11 (9%) *no*. When we ask about the reason, one participant says that "*the replacement effort may be huge because they have different interfaces*".

Importance of Performance Metrics

Finally, we ask participants to rank performance metrics (CPU time, memory usage, and energy consumption) according to the weight that they give when choosing a map implementation. Figure 6.4: summarizes the participants' responses. As it is shown, more than 70% of the participants respond *moderately important* or *very important* for CPU time and memory usage. About energy consumption, 43% of the participants respond that this performance metric is important while 27% of the participants have a neutral opinion about it.

6.3 Experimental Study

In our observational study we find that `HashMap` is the most used map implementation. From the survey we conclude that developers are not aware of the cost of selecting map implemen-

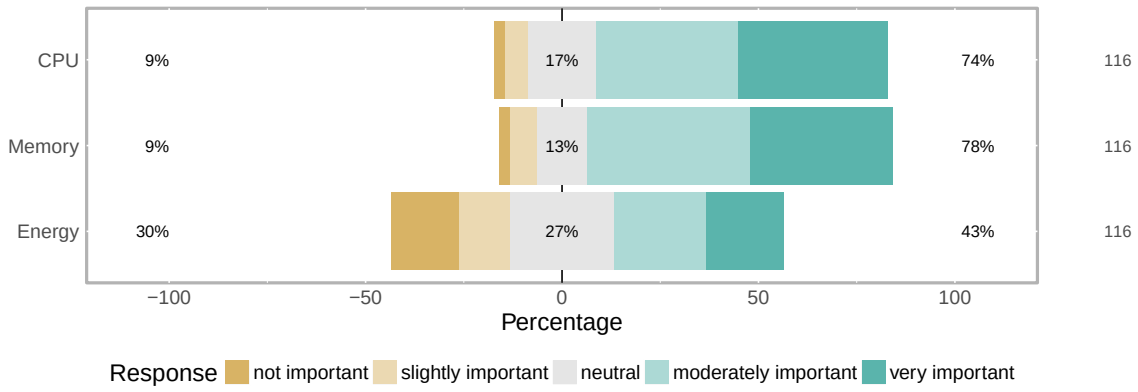


Figure 6.4: Participants' responses about importance of performance metrics when choosing a map implementation.

tations. For this reason, developers are reluctant to use new map implementations and they use what they know that works. We believe that if developers are provided with concrete results of the performance of maps in terms of critical performance metrics (e.g., CPU time, memory usage, and energy consumption), they will be able to make informed decisions. Consequently, we perform an experimental study about the performance of `HashMap`, the most popular Java map implementation, and `ArrayMap`, a map implementation proposed by Android as a replacement for `HashMap`.

In addition, as we find in our observational study, 20% of the apps using `HashMap` use integers as keys, 4% use longs as keys, 5% use integers as keys and values, <1% use integers as keys and longs as values, and <1% use integers as keys and booleans as values. In that cases the Android map implementations `SparseArray`, `LongSparseArray`, `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` are suggested by Android as more efficient map implementations. Because of this, we also study the performance of `SparseArray` variants map implementations.

Although Android proposes `SparseArray` variants as a replacement for `HashMap`, nothing is said about the usage of `ArrayMap` for primitive type keys. For this reason we also compare the performance of `ArrayMap` and `SparseArray` variants.

Thus, we analyze the performance of `HashMap`, `ArrayMap`, and `SparseArray` variants in terms of CPU time, memory usage, and energy consumption. We analyze these map implementations for different data sizes and for four operations: insertion, iteration, random query, and deletion.

6.3.1 Design

We run experiments with `HashMap`, `ArrayMap`, and `SparseArray` variants considering 30 different data sizes in the range $[1, 80000]$. The Android documentation says that `ArrayMap` and `SparseArray` variants are not intended to be appropriate for data structures that may contain large numbers of items. Because "large" is a vague quantity, we use as upper bound for data size a quantity 15 times larger than the one used by Hasan *et al.* (2016). Thus, we want to know how the performance of map implementations is affected when up to 80,000 elements are used in map data structures. For all the map implementations we use the default initial capacity and the default load factor value of 0.75 for `HashMap`, as it is proposed in the Java documentation.

Because `SparseArray` variants are designed to be used with specific primitive type keys, we compare them with respect to `HashMap` and `ArrayMap` setting the same primitive types. However, we also compare `HashMap` and `ArrayMap` implementations using objects (strings) as keys, instead of primitive types. Thus, we analyze 17 different map implementations. They are shown in Table 6.2:. First column contains the specific types used as keys and values for the map implementations shown in second column. Third column contains the syntax for the declaration of each map implementation.

Table 6.2: Subject map implementations for the experimental study.

Types for keys and values	Map implementation	Declaration
String keys and integer values	`HashMap`	`HashMap<String,Integer>()`
	`ArrayMap`	`ArrayMap<String,Integer>()`
Long keys and integer values	`HashMap`	`HashMap<Long,Integer>()`
	`ArrayMap`	`ArrayMap<Long,Integer>()`
	`LongSparseArray`	`LongSparseArray<Integer>()`
Integer keys and integer values	`HashMap`	`HashMap<Integer,Integer>()`
	`ArrayMap`	`ArrayMap<Integer,Integer>()`
	`SparseArray`	`SparseArray<Integer>()`
	`SparseIntArray`	`SparseIntArray()`
Integer keys and long values	`HashMap`	`HashMap<Integer,Long>()`
	`ArrayMap`	`ArrayMap<Integer,Long>()`
	`SparseArray`	`SparseArray<Integer,Long>()`
	`SparseLongArray`	`SparseLongArray()`
Integer keys and boolean values	`HashMap`	`HashMap<Integer,Boolean>()`
	`ArrayMap`	`ArrayMap<Integer,Boolean>()`
	`SparseArray`	`SparseArray<Integer,Boolean>()`
	`SparseBooleanArray`	`SparseBooleanArray()`

In our experiments we use four different operations over these map implementations. For each of the 30 data sizes used, map implementation, and operation, we collect performance metrics. We now explain detailed each of the four operations:

— *Insertion.* We create the data structure and we fill it inserting the number of elements desired. The insertion is done using the `put` method of the map implementations.
— *Iteration.* We create the data structure and we fill it inserting the number of elements desired. After that, we iterate, with a `for-Each` loop, over each `Entry` using the `entrySet` method of the `HashMap` and `ArrayMap` implementations. We use this approach to iterate over the elements of map data structures because, regarding our survey, it is extensively used by developers. However, `SparseArray` variants do not offer an `entrySet` method. We iterate over these implementations by modifying the index between zero and the number of elements. We obtain the key and value from each indexed key-value mapping using the methods `keyAt` and `valueAt` of `SparseArray` variants, respectively.
— *Random query.* We create the data structure and we fill it inserting the number of elements desired. Then, using the `get` method of the map implementations, we return the values to which the specified key of N random elements is mapped. Here, N is the data size of the data structure. To make a fair comparison, the same seed is used for all the data structures. Therefore, the same sequence of random numbers is always generated.
— *Deletion.* We create the data structure and we fill it inserting the number of elements desired. Then, we iterate over the data structure removing one element at time. We remove elements (accessing by key) using the `remove` method of the `HashMap` and `ArrayMap` map implementations, and the `delete` method of the `SparseArray` variants.

Next, we explain the way CPU time, memory usage, and energy consumption are collected. For each of these performance metrics we use a different approach and various scripts developed by us.

CPU Time

We create an Android app for each map implementation which runs the four operations while it collects execution traces using the Android profiler. Execution traces are used to get the CPU time associated to each operation. We run the experiments in an automatic way using a Python script. It uses as input a text file specifying, in each line, the map implementation and the data size to use. Considering the first parameter, the map implementation, the script runs the corresponding Android app, which receives as a parameter the data size. When a tap event is detected on the screen, the Android app runs the insertion, iteration, random query, and deletion operations while the app is profiled using the Android debugger. After these actions are completed, the resulting execution traces are saved on the phone and then

transferred to a server for backup and processing. Algorithm 4: shows the pseudo-code of our approach to measure CPU time of map implementations. Because we are analyzing 17 map implementations and 30 data sizes, we run 510 (17×30) experiments to get CPU measurements.

Algorithm 4: Collection of CPU time for map implementations.

```
1: for each map implementation and data sizes in input file do
2:     Install app of the current map implementation (using adb).
3:     Start app passing the data size as parameter (using adb).
4:     Wait to load the app completely.
5:     Touch the screen to run the experiment (using adb).
6:     Wait until experiment is finished.
7:     Download execution traces from the phone (using adb).
8:     Remove execution traces from the phone (using adb).
9:     Stop the app (using adb).
10:    Clean the app data (using adb).
11:    Uninstall the app (using adb).
12: end for
```

For each data size and map implementation, we obtain one execution trace per operation (insertion, iteration, random query, and deletion). Using a Bash script and the Android `dmtracedump`[7] command, we process execution traces to generate a CSV file containing the CPU time of each independent experiment. Execution traces generated by the Android profiler show both the inclusive and exclusive CPU times (as well as the percentage of the total time). Exclusive time is the time spent in the method. Inclusive time is the time spent in the method plus the time spent in any called functions. We use inclusive CPU time as CPU usage.

Memory Usage

We create an Android app for each map implementation to measure memory usage. Each app runs insertion operations over the corresponding map data structure and reports the memory difference before and after the insertion of elements. The Android app does the following: (1) get the amount of memory (in bytes) used by the Java Virtual Machine, (2) create and fill the corresponding data structure, (3) get the current amount of memory (in bytes) used by the Java Virtual Machine (using the methods `freeMemory` and `totalMemory` offered by the class `Runtime`), (4) calculate the difference between both memory values, and (5) save the resulting amount of memory in a text file on the phone. Thus, the generated file contains the memory used by the data structure, expressed in bytes. We use a Python script to collect memory

7. https://developer.android.com/studio/profile/traceview.html

usage of the studied map implementations automatically. This script uses as input a text file specifying, in each line, the map implementation to use and the data size. Considering the first parameter, the map implementation, the script runs the corresponding Android app that receives as parameter the data size of the map data structure. Algorithm 5: shows the pseudo-code of our approach to measure memory usage of map implementations. We are analyzing 17 map implementations and 30 data sizes, so we run 510 (17×30) experiments.

Algorithm 5: Collection of memory usage for map implementations.

```
1: for each map implementation and data sizes in input file do
2:     Install app of the current map implementation (using adb).
3:     Start app passing data size as a parameter (using adb).
4:     Wait to load the app completely.
5:     Touch the screen to run the experiment (using adb).
6:     Wait until experiment is finished.
7:     Download the file with information about memory usage (using adb).
8:     Stop the app (using adb).
9:     Clean the app data (using adb).
10:    Uninstall the app (using adb).
11: end for
```

Energy Consumption

We design a parametrized Android test suite for each map implementation with four Android test cases, one for each operation. The data size is considered as a parameter of each test case. We measure energy consumption in our phone while we run these test cases. Using Android test cases allows us to run experiments turning the screen off, which removes the impact on energy consumption because the screen. It also allows us to stop the measurement process automatically when a test case finished. We run automatically these Android test cases using a Python script. It reads an input text file specifying, on each line, a test case to be run. A test case is defined by the name of a map implementation, the operation to run, and the data size. Thus, a line "`ArrayMap` INSERTION 1,000" means that the test case inserts 1,000 elements in an `ArrayMap` data structure. Algorithm 6: shows the pseudo-code of our approach to measure energy consumption of map implementations. We analyze 17 different map implementations, four different operations, and we use 30 different data sizes. In total we run 2,040 ($17 \times 4 \times 30$) different test cases.

6.3.2 Data Analysis

We run experiments 20 times to obtain statistical confidence. Hence, we run $20,400$ ($510 \times 20 \times 2$) experiments for collecting both CPU time and memory usage. We run $40,800$ ($2,040 \times$

Algorithm 6: Collection of energy consumption for map implementations.

```
1: for each Android test cases in input file do
2:     Compose test case name.
3:     Start oscilloscope to measure energy consumption.
4:     Run Android test case (using adb).
5:     Stop oscilloscope.
6: end for
```

20) Android test cases for energy measurements. Overall, the collection of performance metrics took around 800 hours (over five weeks) of continuous execution time and resulted in over 600 GB of raw data.

A Wilcoxon rank sum test is carried out to check if the difference observed between the values of the performance metrics is significant. In this case, the null hypothesis is that the distribution of performance metrics of the `HashMap` implementation and performance metrics of `ArrayMap` or `SparseArray` variants differ by a location shift of μ (the average value). We consider the difference to be significant if the obtained p-value is lower than $\alpha = 0.05$. In addition, when the comparison is significant, we compute the effect size using the Cliff's δ function.

6.3.3 Results

For each performance metric and map-related operation, we compute for each data size the median value of performance metrics over the 20 runs. Figures 6.5:, 6.6:, 6.7:, 6.8:, and 6.9: show the median CPU time, memory usage, and energy consumption for each map implementation, data size, and operation. In addition, for each pair of map implementations we compute the average difference of the median values previously computed for each data size. CPU time is expressed in milliseconds (msec.), memory usage in kilobytes (kB), and energy consumption in Joules (J).

Performance of `HashMap` and `ArrayMap`

First, we compare both map implementations with object keys and primitive type values. Then, we compare `HashMap` and `ArrayMap` with primitive type keys and primitive type values.

Object keys and primitive type values.

`HashMap` is faster than `ArrayMap` for iteration, random query operations, and deletion operations. However `ArrayMap` seems a bit faster than `HashMap` for insertion operations. We observe that `ArrayMap` is, on average, 31 msec. (1%) faster for insertion operations. On the

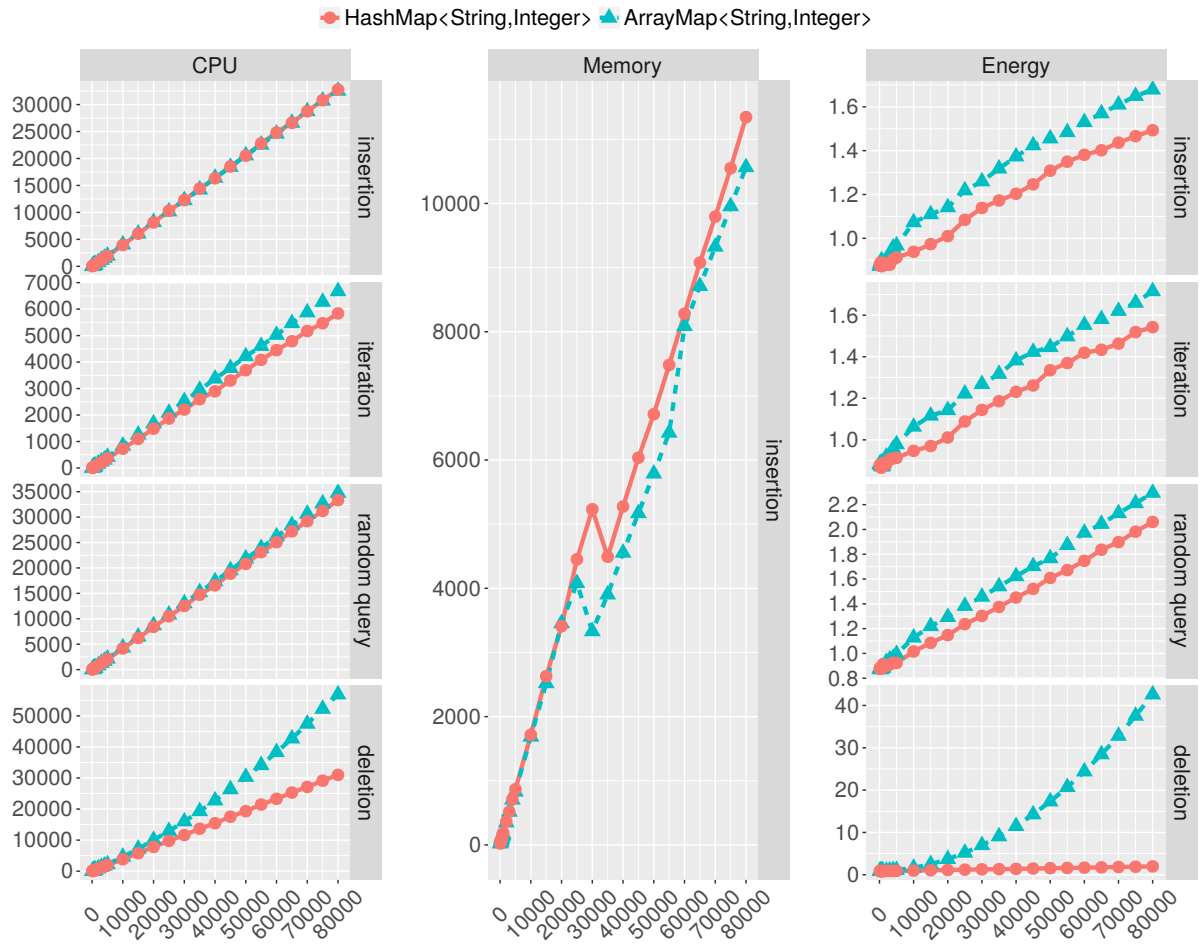


Figure 6.5: Performance metrics of map implementations using string keys and integer values, by map-related operation and data size.

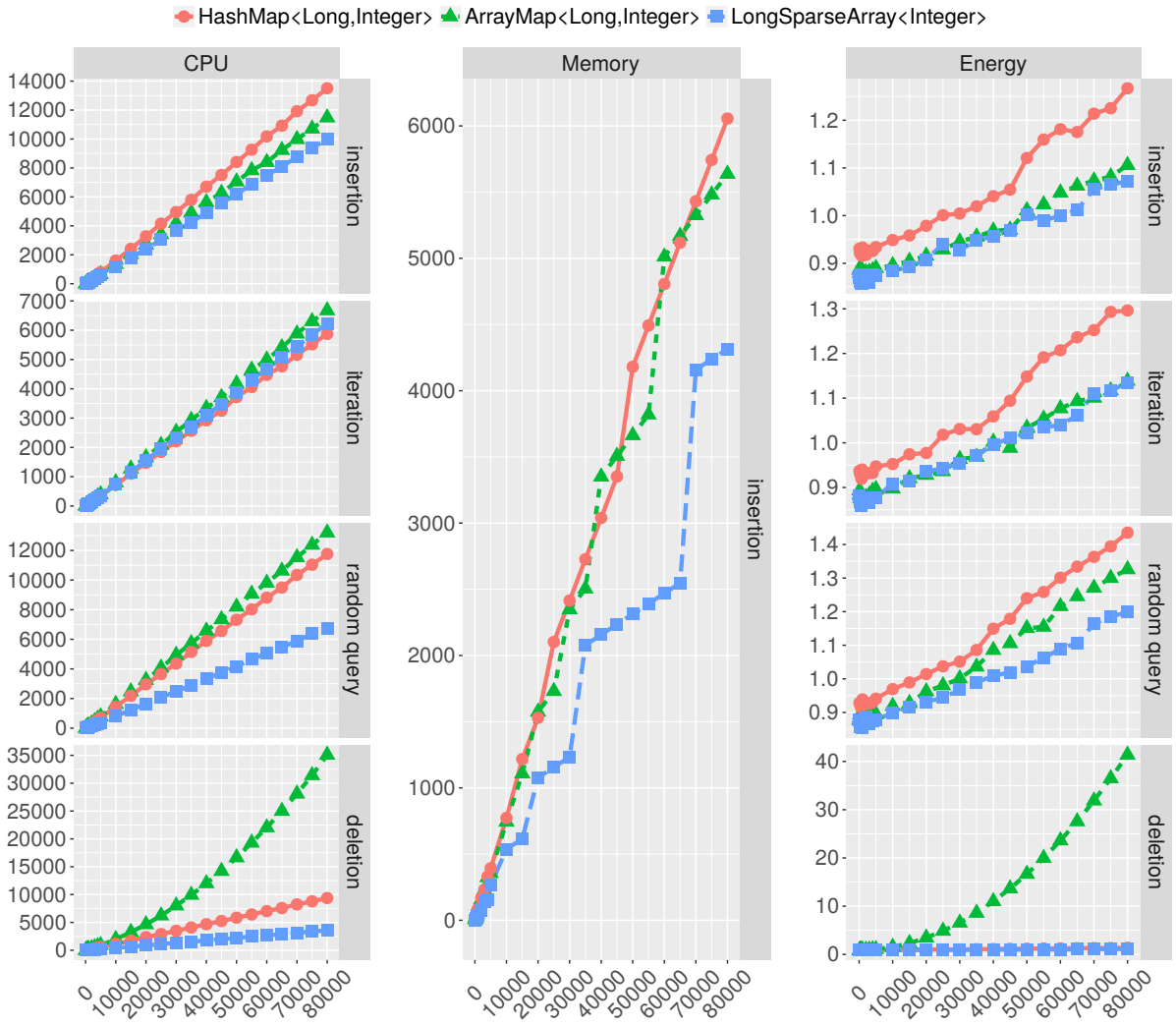


Figure 6.6: Performance metrics of map implementations using long keys and integer values, by map-related operation and data size.

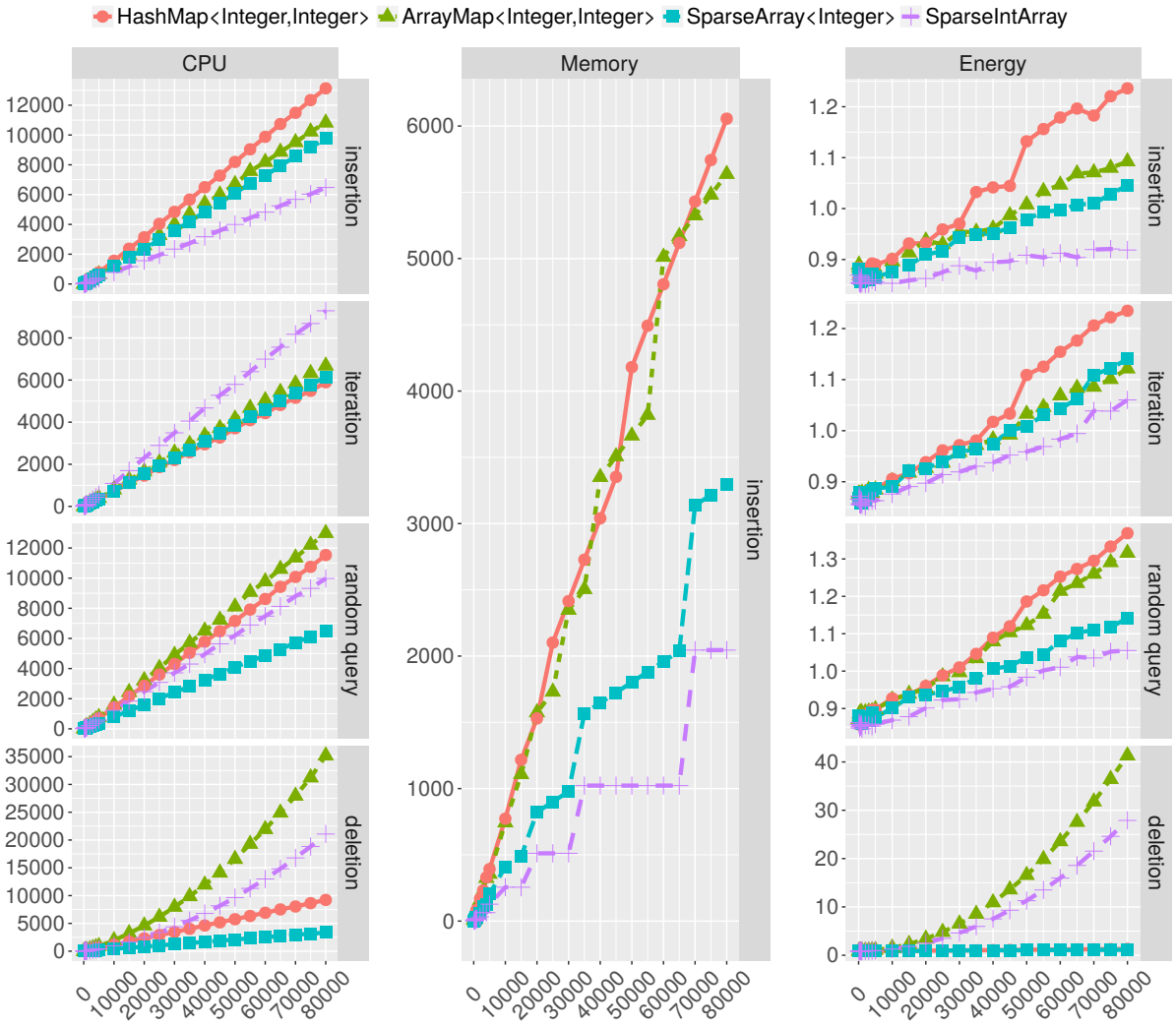


Figure 6.7: Performance metrics of map implementations using integer keys and integer values, by map-related operation and data size.

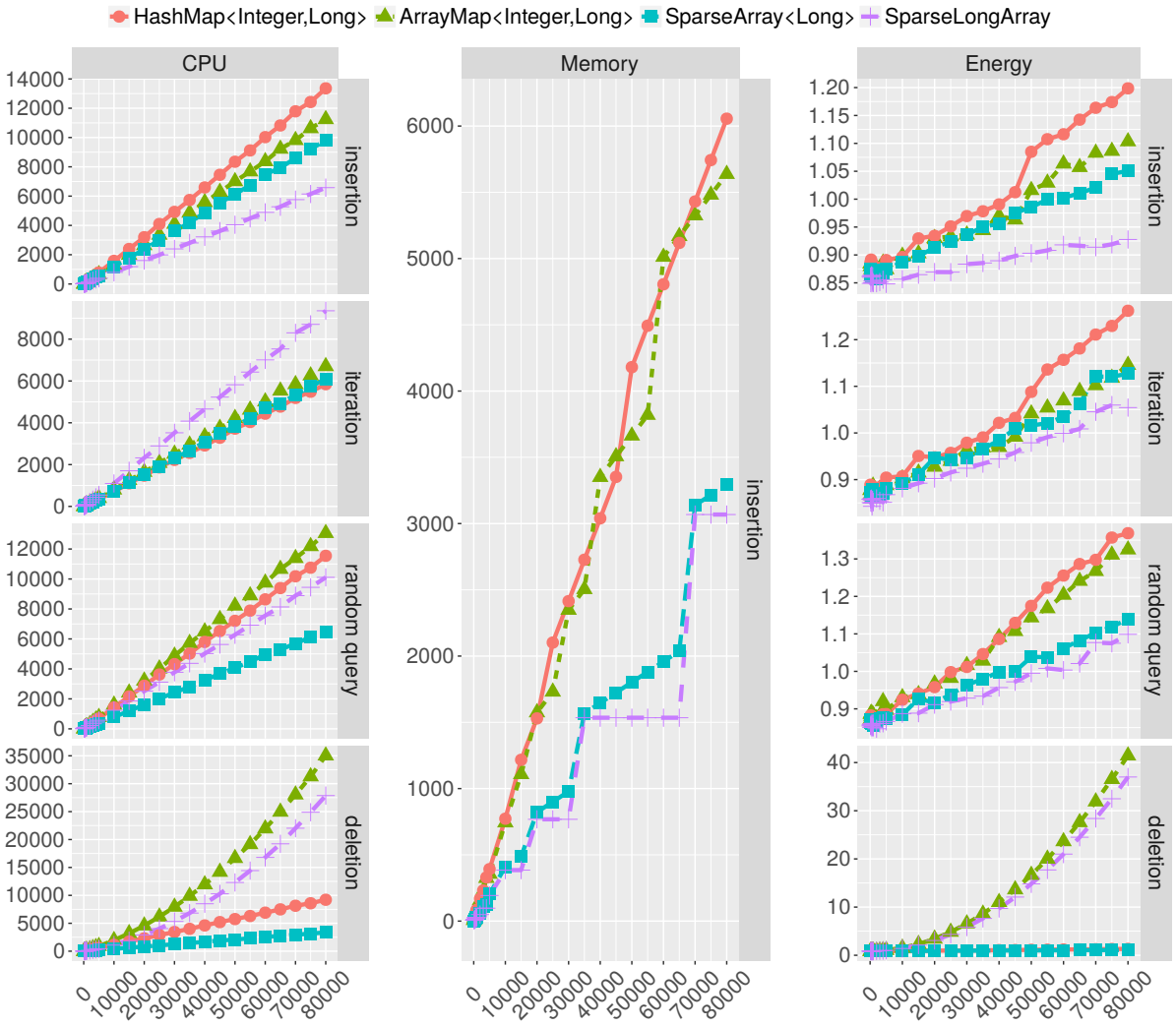


Figure 6.8: Performance metrics of map implementations using integer keys and long values, by map-related operation and data size.

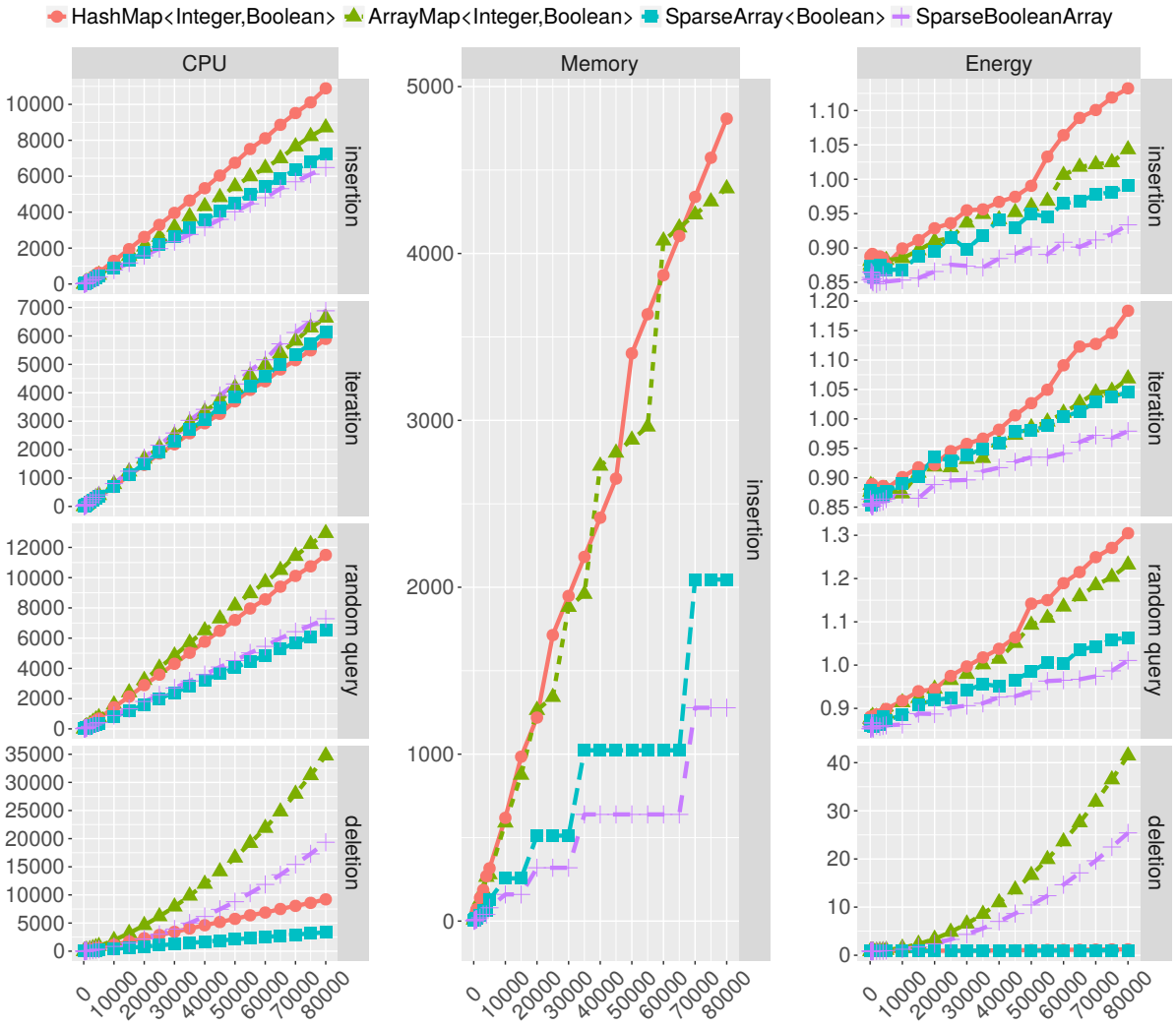


Figure 6.9: Performance metrics of map implementations using integer keys and boolean values, by map-related operation and data size.

contrary, `HashMap` is, on average, 246 msec. (13%), 422 msec. (4%), and 5,516 msec. (22%) faster than `ArrayMap` for iteration, random query, and deletion operations, respectively.

Concerning memory usage, `ArrayMap` is a bit more efficient than `HashMap`. We find that `ArrayMap` uses, on average, less memory (6%) than `HashMap`.

In terms of energy efficiency, `HashMap` consumes less energy than `ArrayMap` for all the operations (on average 7%, 6%, 6%, and 45%, for insertion, iteration, random query, and deletion operations, respectively).

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large for any data size. Differences in energy consumption are significant and the effect size is large when the data size is greater or equal than 2,000 for insertion, iteration, and random query operations. For deletion operations, differences in energy consumption are significant when the data size is greater than 1,000 elements.

Primitive type keys and values.

`ArrayMap` is faster than `HashMap` for insertion operations, but it is slower for iteration, random query, and deletion operations. We observe that `ArrayMap` is, on average, 648 msec. (19%) faster for insertion operations. On the contrary, `HashMap` is, on average, 239 msec. (12%), 434 msec. (12%), and 5,518 msec. (47%) faster than `ArrayMap` for iteration, random query, and deletion operations, respectively.

Concerning memory usage, There is no a clear trend and we consider that both implementations are memory efficient. However, `ArrayMap` uses, on average, less memory (5%) than `HashMap`.

In terms of energy efficiency, `ArrayMap` consumes less energy than `HashMap` for all the operations (on average 4%, 4%, and 2%, for insertion, iteration, and random query operations, respectively). However, `ArrayMap` consumes, on average, much more energy (45%) than `HashMap` for deletion operations.

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large when the data size is greater or equal than 100. Differences in energy consumption are significant and the effect size is large when the data size is greater or equal than 20,000 elements, for insertion and iteration operations. For random query operations, the statistical test reports that differences in terms of energy consumption are significant when the data size is greater or equal than 30,000. For deletion operations, differences in energy consumption are significant when the data size is greater than 1,000 elements.

Cost of Adopting `HashMap` instead of `SparseArray` variants

Now we focus on the cost of adopting `HashMap` with primitive type keys instead of using `SparseArray` variants.

We find that `SparseArray` variants are faster than `HashMap` for insertion and random query operations. This also keeps for deletion operations and `SparseArray` and `LongSparseArray`. Regarding iteration operations, `HashMap` is a bit faster than any `SparseArray` variant. We observe that `SparseArray` and `LongSparseArray` are, on average, 1,040 msec. (27%), 1,503 msec. (42%), and 1,746 msec. (63%) faster for insertion, random query, and deletion operations, respectively. For iteration operations, `HashMap` is, on average, 79 msec. (<1%) faster than `SparseArray` and `LongSparseArray`. Concerning `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, they are, on average, 1,768 msec. (45%) and 725 msec. (21%) faster for insertion and random query operations, respectively. However, `HashMap` is 782 msec. (25%) faster for iteration operations. Regarding the deletion operation, we observe that `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` are, on average, 63 msec. (28%) faster than `HashMap` for data sizes lower than 20,000 elements. For 20,000 or more elements, `HashMap` is, on average, 5,569 msec. (40%) faster for deletion operations.

In terms of memory usage, `SparseArray` variants are more efficient than `HashMap` for all data sizes. They use, on average, 1,025 kB (62%) less than `HashMap`.

Regarding energy consumption, `SparseArray` variants consume less energy than `HashMap` for insertion, iteration, and random query operations. This also keeps for `SparseArray` and `SparseLongArray` and deletion operations. We find that `SparseArray` and `LongSparseArray` consume, on average, 6%, 4%, 7%, and 6% less than `HashMap` for insertion, iteration, random query, and deletion operations, respectively. Concerning `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, they consume, on average, 8%, 6%, and 8% less energy than `HashMap` for insertion, iteration, and random query operations. On the contrary, `HashMap` consumes 42% less energy than them for deletion operations. Regarding the deletion operation, we observe that `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` consume, on average, 2% less energy than `HashMap` for data sizes lower than 2,000 elements. However, for 2,000 or more elements, `HashMap` consumes, on average, 66% less energy for deletion operations.

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large for all data sizes. Differences in energy consumption are significant and the effect size is large for most data sizes.

Cost of Adopting `ArrayMap` instead of `SparseArray` Variants

Now we focus on the cost of adopting `ArrayMap` with primitive type keys instead of using `SparseArray` variants.

We find that `SparseArray` variants are faster than `ArrayMap` for insertion, random query, and deletion operations. This also keeps for iteration operations and `SparseArray` and `LongSparseArray`. We obtain that `SparseArray` and `LongSparseArray` are, on average, 393 msec. (10%), 160 msec. (13%), 1,938 (49%), and 7,264 msec. (80%) faster than `ArrayMap` for insertion, iteration, random query, and deletion operations, respectively. Concerning `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, they are, on average, 1,118 msec. (32%), 1,165 msec. (31%), and 3,112 msec. (47%) faster than `ArrayMap` for insertion, random query, and deletion operations, respectively. However, `ArrayMap` is 543 msec. (14%) faster for iteration operations.

In terms of memory usage, `SparseArray` variants are more efficient than `ArrayMap` for all data sizes. They use, on average, 952 kB (60%) less than `ArrayMap`.

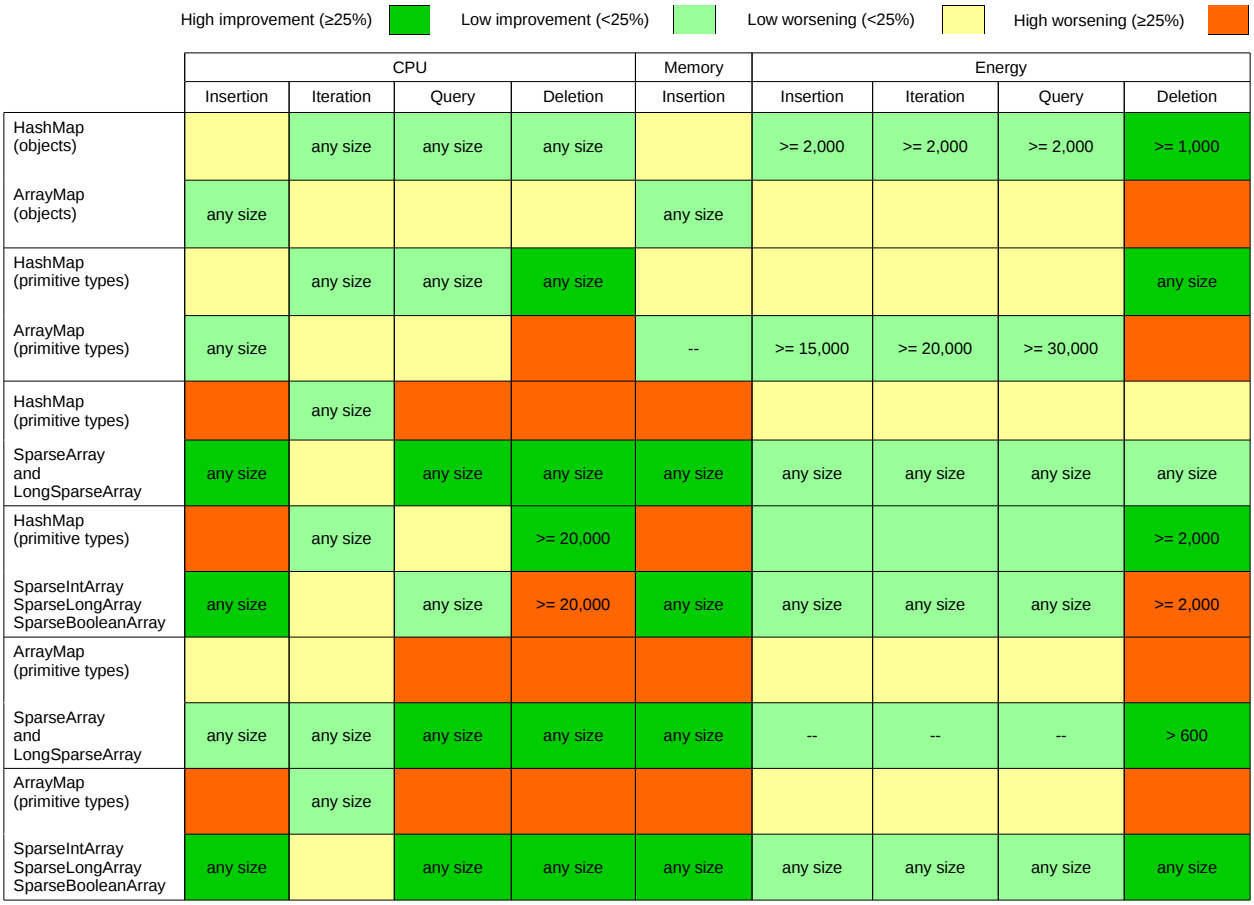
Regarding energy consumption, `SparseArray` variants consume less energy than `ArrayMap` for all the operations. We find that `SparseArray` variants consume, on average, 4%, 2%, 6%, and 32% less energy than `ArrayMap` for insertion, iteration, random query, and deletion operations, respectively.

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large for all data sizes. Differences in energy consumption between `ArrayMap` and `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` are significant and the effect size is large for most data sizes and all operations. However, although differences between `ArrayMap` and `SparseArray` and `LongSparseArray` are significant for most data sizes for random query and deletion operations, for insertion and iteration operations differences are not always significant.

6.4 Guidelines

Most developers are willing to replace `HashMap` with any of the map implementations provided by Android if they offer better performance. Figure 6.10: is a choice matrix summarizing our findings to help developers choose a map implementation. Green shades identify improvements in an operation and performance metric while yellow and orange shades mean worsening. We consider that an improvement/worsening is high/low if differences between map implementations and values of a performance metric are, on average, greater/lower than 25%. A row with stronger shades of green is likely to be more efficient on average. We also

specify a threshold in data size indicating whether our findings hold for more elements than the threshold. The symbol – indicates that, even if on average a map implementation is better than the other, there is no a threshold for which it is always true.



	CPU				Memory	Energy			
	Insertion	Iteration	Query	Deletion	Insertion	Insertion	Iteration	Query	Deletion
HashMap (objects)		any size	any size	any size		>= 2,000	>= 2,000	>= 2,000	>= 1,000
ArrayMap (objects)	any size				any size				
HashMap (primitive types)		any size	any size	any size					any size
ArrayMap (primitive types)	any size				--	>= 15,000	>= 20,000	>= 30,000	
HashMap (primitive types)		any size							
SparseArray and LongSparseArray	any size		any size	any size	any size	any size	any size	any size	any size
HashMap (primitive types)		any size		>= 20,000					>= 2,000
SparseIntArray SparseLongArray SparseBooleanArray	any size		any size	>= 20,000	any size	any size	any size	any size	>= 2,000
ArrayMap (primitive types)									
SparseArray and LongSparseArray	any size	any size	any size	any size	any size	--	--	--	> 600
ArrayMap (primitive types)		any size							
SparseIntArray SparseLongArray SparseBooleanArray	any size		any size	any size	any size	any size	any size	any size	any size

Figure 6.10: Color map showing the comparison between each pair of map implementations, operation, and performance metric. Green colors identify more efficient implementations. The greener the color, the better.

To know if performance differences are perceivable by end users when a more efficient map implementation is used, we conduct an additional experiment. We randomly select an app from our subject apps which contains occurrences of `HashMap` with primitive types as keys and values. Our guidelines suggest to use `SparseIntArray` as a more energy efficient alternative to `HashMap`. We select the app SudokuIsFun because we can compile and run it on our phone. This is a simple Sudoku game with a simple user interface. Then, we carry out an experiment comparing the energy consumption of the original version and the refactored one (by replacing `HashMap<Integer,Integer>` with `SparseIntArray`). In 30 runs of a simple scenario that introduces some values and solved the Sudoku, the refactored version consumes less energy (median of 0.38 mJ). It means that, if the battery is fully charged and a user repeats this

scenario until the battery is over, the refactored version allows users to play two minutes and forty-nine seconds more. Therefore, replacing `HashMap` with `SparseIntArray` extends battery life by 0.81% for this app and scenario. Although the improvement may seem rather marginal, it might be important enough for some developers to rethink their choices. Besides, SudokuIsFun uses maps in only one of its method, which is not heavily used. Consequently, the observed improvement is rather the lower limit of possible improvements.

6.5 Discussion

From the observational study we conclude that `HashMap` is the most used Java map implementation in Android apps. `ArrayMap` and `SparseArray` variants map implementations are rarely used in Android apps. From the survey we conclude that developers are not aware of the overhead incurred when selecting an inappropriate map implementation and that is why they use what they know that works.

From our empirical study, we partially agree with the Android developers' reference documentation: `ArrayMap` is generally slower than `HashMap` because lookups require a binary search. However, it is not true for insertion operations for which we find that `ArrayMap` is faster no matter the number of elements. We also confirm that `ArrayMap` consumes, on average, less memory than `HashMap` (by up to 6%). Although `ArrayMap` is more energy efficient than `HashMap` when keys are primitive types, we find that `ArrayMap` consumes more energy than `HashMap` for all the operations when keys are objects. The larger the number of elements, the larger the difference. We also observe that `ArrayMap` is highly inefficient for deletion operations. Figures 6.5:, 6.6:, 6.7:, 6.8:, and 6.9: show that, while for insertion, iteration, and random query operations CPU time and energy consumption grows up linearly with respect to data size, for the deletion operation the growth is exponential. Because `ArrayMap` shrinks its array as items are removed from it. However, when an element is removed in a `HashMap` the corresponding node in the entry set table is set to null. It means that `HashMap` does not shrink its entry set in deletion operations. This aggressive shrinking behavior for deletion operations in `ArrayMap` is justified in the Android developers' reference documentation to better balance memory use. Even if this fact is true, the impact of this decision on CPU time and energy consumption is not negligible. We recommend to use `HashMap` instead of `ArrayMap` when keys are objects to improve the energy efficiency of Android apps. However, `HashMap` is often adopted with primitive type keys. In that cases, the official Android IDE, Android Studio, warns about replacing `HashMap` with `SparseArray` variants for better performance. We confirm that `SparseArray` and `LongSparseArray` are more efficient than `HashMap` in terms of memory usage. But we also extend this fact to energy consumption.

Contrary to the documentation, we found that `HashMap` is faster than `SparseArray` and `LongSparseArray` but only for iteration operations. For insertion and random query operations, they are faster than `HashMap` no matter the number of elements. We also observe that the higher the numbers of elements, the higher the differences in the three performance metrics in favor of `SparseArray` and `LongSparseArray` (except for the iteration operation for which `HashMap` is slightly faster). The same keeps for `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` if values are also primitive types, excepting that these map implementations are faster than `HashMap` for iteration operations and they are less efficient than `HashMap` for deletion operations.

Android proposes `SparseArray` variants as a replacement for `HashMap` when keys are primitive types for better performance. We find that this proposition also hold for `ArrayMap`. Thus, `ArrayMap` should be replaced by `SparseArray` variants if primitive types are used as keys.

Although `SparseArray` variants are efficient alternatives to `HashMap` and `ArrayMap`, we strongly recommend to review the implementation of `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, which are highly inefficient when removing elements in comparison to `SparseArray` and `LongSparseArray`. Figures 6.6:, 6.7:, 6.8:, and 6.9: show that, while for insertion, iteration, and random query operations CPU time and energy consumption grows up lineally with respect to data size, for the deletion operation the growth is exponential. The reason is that removes require deleting entries in the array of these map implementations. On the contrary, `SparseArray` and `LongSparseArray` include an optimization when removing keys to help with performance. Instead of compacting its corresponding array immediately, it leaves the removed entry marked as deleted. The entry can then be re-used for the same key, or compacted later in a single garbage collection step of all removed entries. Considering our experiments, this implementation detail seems to make the difference between these map implementations for the deletion operation.

We conclude that `ArrayMap` consumes less energy than `HashMap` when keys are primitive types. This confirms what we found for the HMU Android anti-pattern in the previous chapter, because most of the studied apps in our previous study use primitive type keys. However, `SparseArray` variants are more efficient choices than `HashMap` and `ArrayMap`. Thus, developers must consider the use of `ArrayMap` with primitive type keys as a new Android performance anti-pattern. Following our guidelines developers can make informed decisions about the map implementations to use in their apps.

In the observational study we focus on the use of Java and Android map implementations in Android apps. During the analysis of the source code of the apps under study, we observe

that developers usually integrate in their apps more than one TPLs to implement different functionalities. For example, during the validation of our observational study we found apps that integrate TPLs that offer specific implementations of data structures, or TPLs to connect to social networks. This fact motivates the next chapter, there we study the performance of TPLs.

CHAPTER 7 HELPING DEVELOPERS CHOOSE THIRD-PARTY LIBRARIES

Mobile device apps are complex mostly because they rely on calls to the Android API and the integration of multiple TPLs. However, TPLs make mobile device app development much more convenient by offering implementations of specific functionality. For example, app developers often use advertising libraries as a source of revenue, integrate social networking libraries to simplify the login process, or include crash reporting tools to monitor crashes in theirs apps.

The size and complexity of mobile device apps grow in correlation with the addition and usage of TPLs. As Minelli et Lanza (2013) obtained, external calls represent more than 75% of the total number of method invocations in Android apps. To comprehend mobile device apps is important to understand the behavior of the used external libraries. Previous experiments carried out by Wang *et al.* (2015) on more than 100,000 Android apps from five different Android markets, shown that more than 60% of the sub-packages in Android apps are from TPLs. On average, TPLs account for more than 60% of the code in Android apps. Inefficient TPLs could have an important negative impact on the quality of mobile device apps.

To help developers monitor and control TPLs, the company SafeDK [1] provides an end-to-end TPLs management platform on the Internet. SafeDK also offers a marketplace that lets developers explore through hundreds of reviewed and rated TPLs by category. However, to the best of our knowledge, there is no previous research concerning the impact of TPLs on performance of mobile device apps. Consequently, developers are forced to integrate a TPL and monitor user's comments to, hopefully, detect performance issues that could be related to that TPL.

In this chapter, we propose a methodology to compare the performance of different TPLs. It allows developers to make informed decisions about the TPLs to be included in their apps to improve their performance. This methodology can be applied at any time during the software development process cycle. However, we recommend its usage during the design, implementation, or testing and integration phases.

1. https://www.safedk.com/

7.1 Methodology

In this section we describe the steps of our methodology to compare performance metrics of TPLs. First, developers create a minimal app for each TPL under study. Second, they define a scenario to exercise TPL's functionality. Then, developers run each minimal app and play the defined scenario while performance metrics are collected over several runs. Finally, data are aggregated and a comparative between TPLs for each performance metric is generated. We describe these steps next. As a practical case, let us suppose that a development team want to include a TPL for advertising in a new mobile device app. Ads allow developers to keep their content free and available, reaching more users, while still making revenues. Let us also consider that developers want to include a banner ad format in the new app to increase their revenue. There are several TPLs for choosing from but let us suppose that developers are particularly interested in three of the most popular TPLs for advertising.

7.1.1 Creating a Minimal App for Each Subject TPL

Developers create a minimal mobile device app for each TPL under study. By minimal we mean an app with the simplest GUI. We recommend using dark colors for the GUI to minimize the impact of the screen on energy consumption measurements. Next, developers modify the source code of each minimal app to include the corresponding TPLs under study. After this, developers build the release signed version of the apps. We suggest to build apps in this way because it does not introduce information about debugging, and because apps should be signed before submitting them to marketplaces. Thus, at the end, there is a minimal app for each of the TPLs under study. Regarding the practical case, developers build three different minimal apps, one for each advertising TPL. All of them use a similar GUI that only contain a banner at the top. In addition, each ad library is configured in a similar way to make a fair comparison. For instance, setting similar refresh rate of ads for all of them.

7.1.2 Defining Scenarios to Exercise TPLs' Functionality

Developers define playable scenarios to exercise the TPLs under study. Assuming that TPLs in consideration offer similar functionalities, developers must define a unique scenario which is common for all the TPLs under study. For the practical case the scenario could be really simple. It only waits several seconds until an ad is loaded and showed to the user.

7.1.3 Collecting Performance Metrics

Developers run each minimal app and play the defined scenario while performance metrics are collected. This should be done in an automatic way and, at least, 20 times. It allows to collect enough data to carry out an automated statistical study.

7.1.4 Statistical Analysis

Once performance metrics are available, statistical tests are carried out to check whether differences between each pair of TPLs are significant. We recommend to perform the pairwise comparisons between different TPLs using the Wilcoxon rank sum test, with a confidence level of 95% (p-value = 0.05). It allows to check if the difference observed between the values of the performance metrics of different TPLs is significant. In this case, the null hypothesis is that the distribution of performance metrics of a TPL and performance metrics of a different TPL differ by a location shift of μ (the average value). In addition, when a pairwise comparison is significant the effect size is computed. We recommend to compute the effect size using the Cliff's δ method.

7.1.5 Output

As output, information about performance metrics for each TPL is reported through two different plots. One plot shows the distribution of each performance metric for each TPL. A second plot shows the pairwise comparison of the TPLs under study for each performance metric. It includes for each pairwise of TPLs and each performance metric the median value of each TPL, the difference of the medians, if differences are significant and, in that case, the magnitude of the effect size. All of this allows developers to graphically compare performance metrics of different TPLs and make informed decision about the TPL to integrate in their mobile device apps.

7.2 Case Study

We validate our methodology proposing a case study over popular Android TPLs. We choose the Android ecosystem due to its popularity. SafeDK has done a survey of Google Play analyzing the top apps available for free worldwide [2]. This study reports the trends of app publishers using TPLs looking at data from May 2017 and analyzing more than 150,000 apps and over 900 TPLs. Based on their findings, we select for our case study three of the most

2. http://mobile-sdk-data-trends.safedk.com/full-report-May-2017

popular categories of TPLs and three of the most popular TPLs used by mobile device app developers. We summarize this information in Table 7.1: where we also show the version and provider of each TPL.

Table 7.1: Popular Android TPLs under study.

Name	Version	Provider	Category
admob	10.2.4	Google	Advertising
applovin	7.2.0	Applovin	Advertising
mopub	4.15.0	Twitter	Advertising
firebase	11.0.2	Google	Analytics
flurry	7.0.0	Yahoo	Analytics
google	10.2.4	Google	Analytics
acra	4.9.2	ACRA	Crash reporting
crashlytics	2.6	Crashlytics	Crash reporting
newrelic	5.14.0	Newrelic	Crash reporting

Advertising TPLs allow developers to keep their content free and available while still making revenues. Analytic TPLs provide insights with which developers can learn how their users are behaving, where they might be losing them, and much more. Crash reporter TPLs create detailed reports of errors in apps which are reported to developers. For each TPL in these categories we create a minimal Android app. All the apps for each TPL in a category have the same GUI. Figure 7.1: shows a screenshot of the main activity of the minimal app for each TPL category. For advertising TPLs we introduce a banner at the top of the device screen. For analytic TPLs we define in the main activity a fragment pager adapter with four different fragments, containing each of them a different image and a string used as identifier (Image1, Image2, Image3, and Image4). Every time a user swipes left or right, the previous or next fragment is loaded, respectively, and an event is sent to the corresponding TPL server. Finally, for crash reporter TPLs we introduce a button that throws a runtime exception to simulate a crash when it is clicked.

Next, for each TPL category we define an scenario to exercise TPLs functionality. For advertising TPLs we wait few second in the main activity to load an ad. For analytic TPL's we navigate through the different fragments. Finally, for crash reporter TPLs we click the button to throw a runtime exception.

Once the minimal apps have been developed and the scenarios defined, we automatically run each minimal app and we play its corresponding scenario while we collect performance metrics. We repeat this process 30 times.

Figure 7.1: Minimal app for each TPL category, all of them designed for our case study. From left to right, the GUI of the minimal Android apps for the advertising, analytic, and crash reporter categories, respectively.

Results

Figure 7.2: shows the distribution of each performance metric for each TPL, grouped by category. As it is shown, different TPLs have different performance. For example, for advertising TPLs, *applovin* uses less CPU and memory than *admob* and *mopub*. However, *applovin* transmits more data over the network than the other TPLs. Concerning the analytics TPL category, *google* seems to be more efficient for CPU, memory, and network usages, but it seems to consume a bit more energy than *flurry* and *firebase*. Finally, for the crash reporter library, *acra* is clearly more efficient than *crashlytics* and *newrelic* for memory and network usages but this fact does not keep for CPU usage.

In addition to the distribution of performance metrics, we also compute the ranking of each TPL category and performance metric. First, we aggregated the data for each TPL and metric computing the median over the 30 independent runs. Second, we obtain the ranking of each TPL in a category for each performance metric based on the aggregated data. Table 7.2: shows the ranking of subject TPLs. With this information developers can know which TPL is better than others for different performance metrics. For example, for the advertising TPL category, we confirm that *applovin* is the best one in terms of CPU and memory usages but it is the worst in terms of network usage. Regarding the analytics TPL, we confirm that *google* is the best one in terms of CPU, memory, and network usages. However, *google* is the

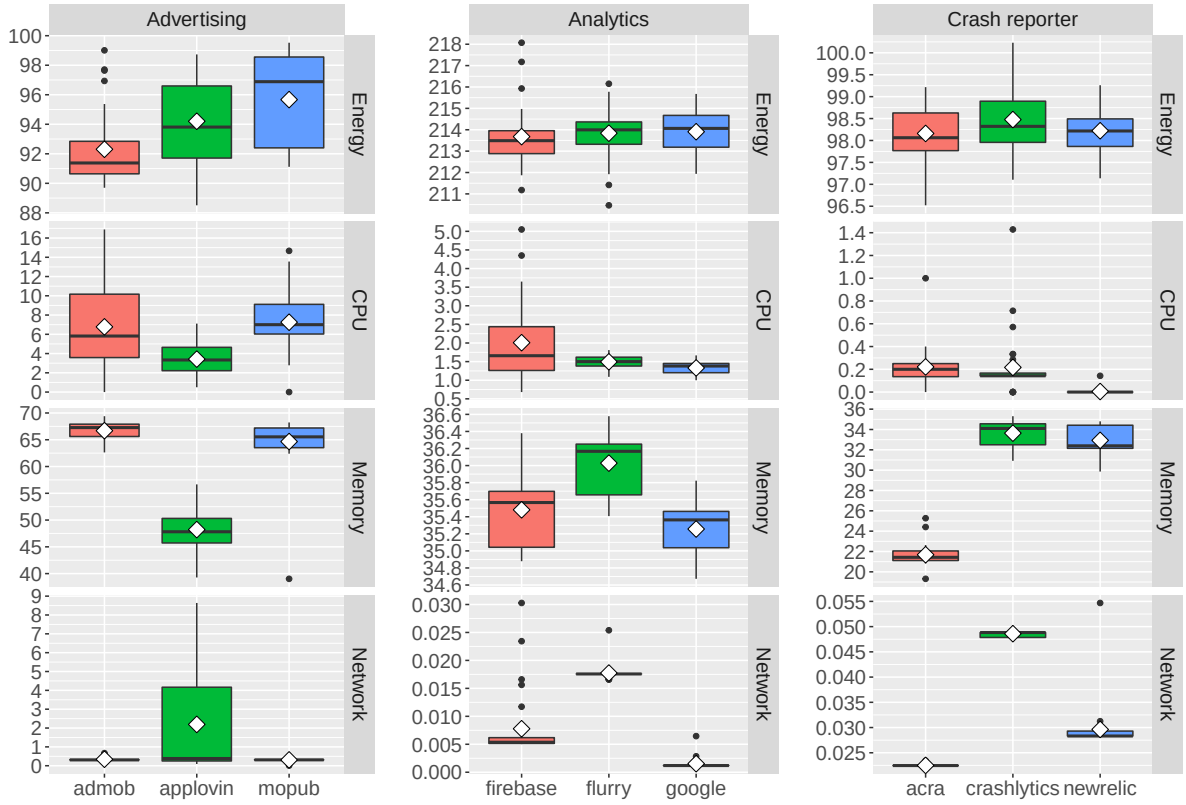


Figure 7.2: Distribution of each performance metric for each TPL, grouped by category. The lines in the boxes indicate the minimum value, lower quartile, median, upper quartile, and maximum values. The "◇" symbols represent the average value.

worst regarding energy consumption. Finally, concerning the crash reporter TPL category, we confirm that *acra* is more efficient than the others in terms of CPU, memory, and network usages. On the contrary, *crashlytics* is the less efficient for these metrics.

Table 7.2: Rankings provided by our approach for each performance metric and TPL.

TPL	Energy	CPU	Memory	Network	Category
admob	1	2	3	1	Advertising
applovin	2	1	1	3	Advertising
mopub	3	3	2	2	Advertising
firebase	1	3	2	2	Analytics
flurry	2	2	3	3	Analytics
google	3	1	1	1	Analytics
acra	1	3	1	1	Crash reporter
crashlytics	3	2	3	3	Crash reporter
newrelic	2	1	2	2	Crash reporter

Although rankings are useful to make a quick and intuitive comparison, they do not show the full picture. Differences between different TPLs could be small or even not statistically significant. Figure 7.3: shows the pairwise comparison of the TPLs under study for each performance metric. For each cell in the matrix, negative differences (green colors) mean that the TPL at the bottom is better than the TPL at the left. On the contrary, positive differences (red colors) mean that the TPL at the bottom is worse than the TPL at the left. Therefore, greener colors higher differences in favor of the TPL at the bottom and redder colors higher differences in favor of the TPL at the left. Only cases where there is a statistically significant difference are shown. From here developers can observe that the three TPLs under study for the advertising category are statistically similar in terms of network usage, because differences are not statistically significant. The same keeps for energy consumption and the analytics and crash reporter TPL categories.

For the advertising TPL category we confirm that *admob* is the most energy efficient TPL. It consumes 2.59% and 5.68% less energy than *applovin* and *mopub*, respectively. However, *admob* uses more CPU (42.72%) and memory (28,92%) than *applovin*, which is the most efficient TPL in terms of CPU and memory usages. Regarding the analytics TPL category, *google* uses 16.73% and 7.94% less CPU than *firebase* and *flurry*, respectively. Although *flurry* has a better ranking than *firebase* for CPU usage, differences are not statistically significant. The *google* TPL also uses less memory (0.57% and 2.22%) and transmits less data over the network (77.75% and 93.21%) than *firebase* and *flurry*, respectively. Even if *google* has the worst ranking for energy consumption, differences are not statistically significant. Therefore, it could be considered as the most efficient of the three TPLs under study. Finally, concerning the crash reporter TPL category, we confirm that *acra* is the most efficient in terms of memory

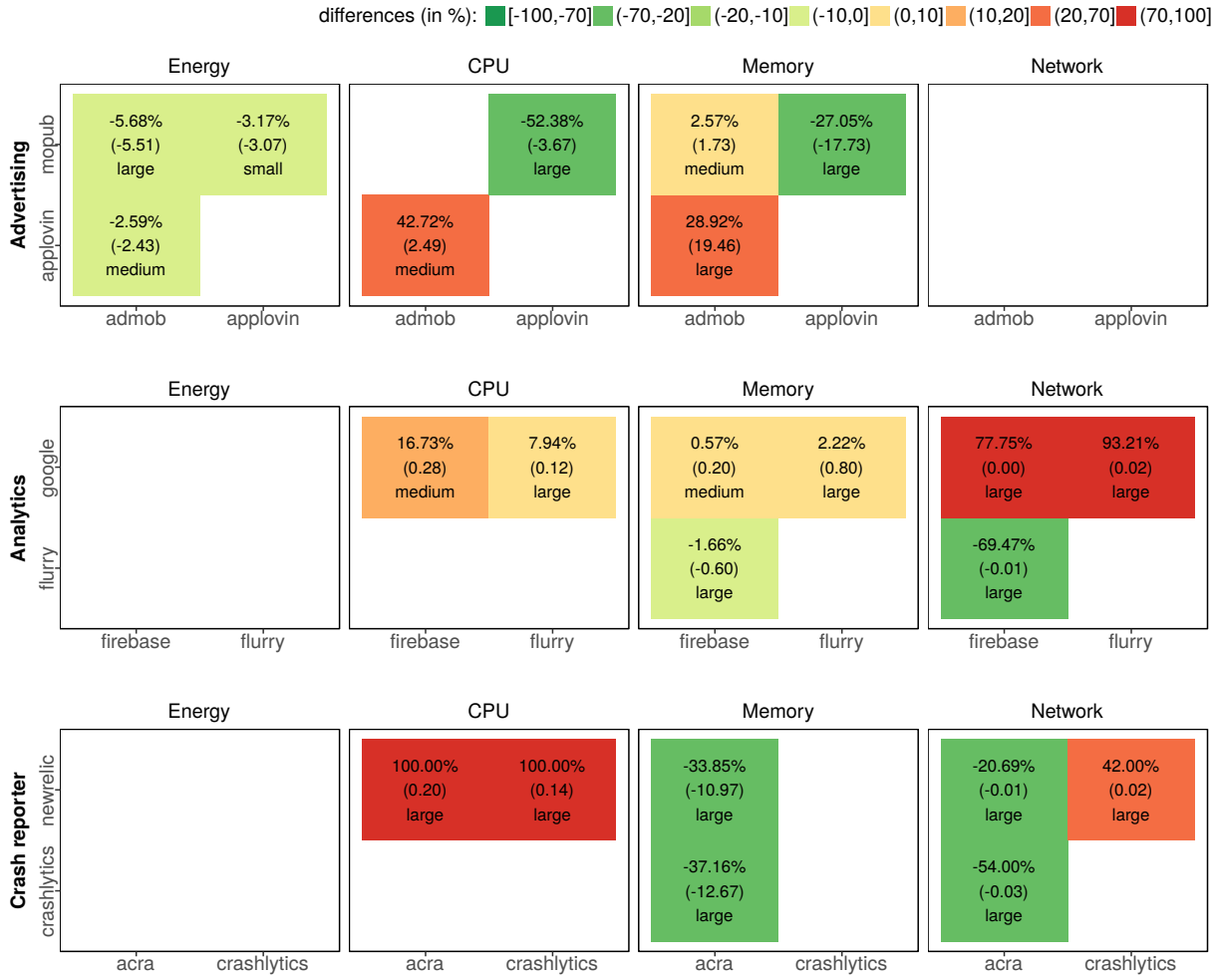


Figure 7.3: Pairwise comparison of TPLs and performance metrics. Each cell contains the difference of the medians in %, the magnitude of the difference (in Joules, %, MB, and MB, for energy consumption, CPU, memory, and network usages, respectively), and the magnitude of the effect size (small, medium, or large). Absent values indicate cases where there is not a statistically significant difference.

and network usages. It uses 37.16% and 33.85% less memory and it transmits 54.00% and 20.69% less data over the network than *crashlytics* and *newrelic*, respectively. However, *newrelic* is the most efficient in terms of CPU usage.

7.3 Discussion

For the advertising TPL category there is no a best TPL for all the metrics. Therefore, developers could make their decision about the TPL to use based on the context of use of their app. For example, if developers want to target emerging markets they should integrate in their app the *applovin* TPL. It is the most efficient in terms of CPU and memory usages, and differences are not statistically significant in terms of network usage with respect to *admob* and *mopub*. For this reason we consider *applovin* as the best choice if CPU, memory, and network usages are the most important metrics.

Concerning the analytics TPL category, we recommend to integrate the *google* TPL. Differences are not significant in terms of energy consumption and it is the most efficient in terms of CPU, memory, and network usages.

Finally, regarding the crash reporter TPL category, there is not a winner and, again, developers could decide taking into account the context of use of their app. Thus, for example, if developers are not targeting emerging markets they could integrate in their app the *acra* TPL. Although it uses more CPU than the others, it is as energy efficient as them and it uses less memory and transmits less data over the network.

Developers can use our methodology and–or our catalog to compare the performance of different TPLs and make informed decisions about the TPLs to integrate in their apps.

Advertising TPLs is one the most popular TPLs integrated by developers in their apps because they allow them to keep their content free while still making revenues. But developers can also offer paid versions of their free apps in marketplaces. In next chapter we perform a study about ads-supported apps and their corresponding paid versions to understand their performance and differences.

CHAPTER 8 COMPREHENSION OF ADS-SUPPORTED AND PAID APPS

As we obtained in the previous chapter, different TPLs have different performance. Thus, developers can improve their apps performance integrating efficient TPLs. One of the most popular TPL integrated by developers is for advertising. Ads allow developers to keep their content free and available, reaching more users, while still making revenues. However, it is well-know the hidden cost of ads and, therefore, developers must have this fact in mind to decide to include or not ads in their apps.

On the one hand, both developers and users are interested by free apps: developers to showcase their apps and users to test out these apps for free. On the other hand, developers may offer, in addition to their free versions, paid apps and include in the free versions ads. These ads-supported apps offer less or similar features than their corresponding paid versions and they use ad networks to display ads that provide revenue to developers. While paid apps have clear market values (their prices), ads-supported versions are not entirely free because ads in apps have an impact on app ratings and users' privacy (Book *et al.*, 2013) but also on performance (Wei *et al.*, 2012; Gui *et al.*, 2015). Yet, users are sometimes reluctant to pay for apps when ads-supported versions of the same or similar apps exist for free.For this reason, and to increase the numbers of purchases of paid apps, Google launched in 2016 the concept of "Family Group" [1]. When users set up a family group on Google Play, the family manager can invite up to five people to the group and they can share purchased apps. Thus, the prices of the purchased apps is divided by the numbers of family members.

First, we study the balance between the costs of ads in free apps and the costs of paid apps, while considering their performance and sharing among family groups. We carry out an experimental study to compare the performance of ads-supported and paid apps and we propose four equations to estimate the cost of ads-supported apps. We want to make explicit the hidden costs of ads when considering the possibility to form family groups. Thus, we want to provide some advices to developers, to seize and act on the balance between visible and hidden costs of paid and ads-supported apps. Past studies (Wei *et al.*, 2012; Gui *et al.*, 2015) discussed the impact and hidden costs of ads on performance metrics. However, they did not support their reports using statistical tests and, therefore, their conclusions could be statistically invalid. We complement these previous studies with measures that we can analyze statistically to check if ads-supported apps are more costly in terms of performance metrics

1. https://support.google.com/googleplay/answer/6286986

than paid apps because of the presence of ads. In addition, we propose different equations to determine the cost of free apps due to ads. Second, we carry out an exploratory study about the ads-business model comparing ads-supported and paid apps to understand their differences and development process.

8.1 Context of the Study

The context of our study is the official Android marketplace, Google Play, and the subjects are ads-supported and paid Android apps available in this marketplace. For each category in Google Play, we randomly select eight paid apps with a free version available. Over the resulting 128 apps 63 are discarded because (i) their corresponding free versions do not contain ads (31 apps), (ii) they require a GPS connection to work (four apps), (iii) their corresponding APK files are invalid and apps are not installable in our phone (10 apps), (iv) they crash on the real phone (nine apps), or (v) their corresponding paid versions do not work because they must be installed from Google Play (nine apps). Therefore, over the initial 128 apps we select 65 which contained visible ads and work on our phone. We use these 65 apps for the exploratory study. Among these 65, we randomly select 20 from different developers for the experimental study. We did that because for the experimental study we need to buy the paid version of the apps. On the contrary, for the exploratory study, we can get the information we need crawling Android app repositories. Anyway, our selection process is akin to a stratified random sampling of paid apps with two strata: ads-supported version and working on our phone. Selected apps belong to 14 different categories, the number of downloads for ads-supported and paid apps is in the range $[500, 100000000]$ and $[10, 1000000]$, respectively, and the rating is in the range $[1.2, 4.6]$ and $[1.0, 5.0]$ for ads-supported and paid apps, respectively. Therefore, we selected apps with different popularity and ratings, belonging to different categories in the official marketplace, and from different developers. For all of this we consider that we have a reduced but representative sample of Android apps.

8.2 Experimental Study

For each app, we create a simple scenario to start the app, skip the initial tutorial (if present), and wait for 100 seconds in the main activity that contains ads in the free version. These scenarios are run 30 times automatically while we collect performance metrics. We perform a Wilcoxon rank sum test to check if the differences observed between the values of the measures of ads-supported and paid versions are significant. Our null hypothesis is that the

distributions of the measures of paid apps and that of their corresponding ads-supported versions differ by a location shift of μ (the average value), expecting that paid apps have a better performance. We consider the difference to be significant if the obtained p-value is lower than 0.05. In addition, we compute the effect size using the Cliff's δ function when the comparison is significant. Using the collected data we determine the cost of ads-supported apps due to ads depending on the network usage, the battery drained, and the time in which a data plan is over, to estimate the time in which an ads-supported app overtakes its paid version.

8.2.1 Results

Ads-supported apps use more resources than their corresponding paid versions and differences are statistically significant and the effect size large, thus, we can reject the null hypotheses. Therefore paid apps are more efficient in terms of power, CPU, memory, and network usages because of the absence of ads. Figure 8.1: shows the differences in percentages for each performance metric between ads-supported and paid apps. Ads-supported apps consume, on average, 21.27%, 5.88%, 42.15%, and 93.19% more than paid apps for power, CPU, memory, and network usages, respectively.

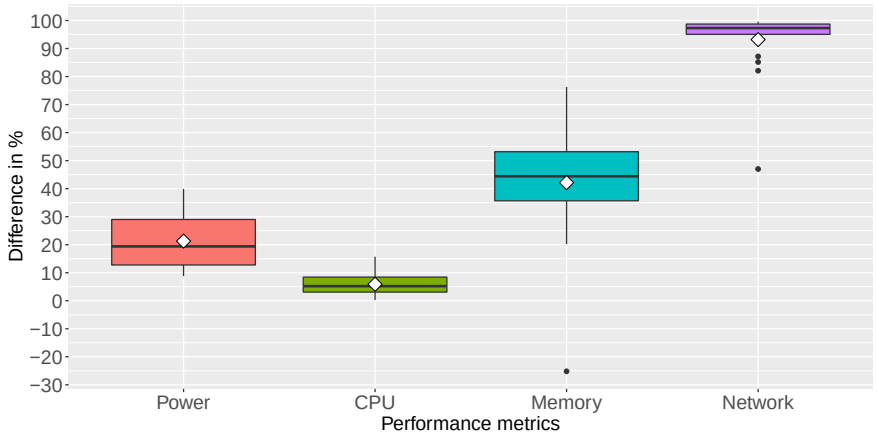


Figure 8.1: Performance metric differences for ads-supported and paid apps. The lines in the boxes indicate the minimum value, lower quartile, median, upper quartile, and maximum values. The "◇" symbols represent the average value.

From the collected data, we obtain that, on average, ads-supported apps increase network usage by 1.28MB (median 0.56MB). The average data network usage is so high because one app consumes a lot of data in its free version (average 14.43MB in comparison to the 0.06MB consumed by the paid app). Using the median value, we estimate the average network usage

of each ad. Taking into account that in 100 seconds, on average, 2.6 ads are loaded, we conclude that 0.21MB $\left(\frac{0.56MB}{2.6}\right)$ is the average network usage of each independent ad. We thus estimate the network cost of ads using the price of $15 per gigabyte as provided by the AT&T company, a popular American Internet services provider. We thus determine that each load of an ad potentially costs end users $0.00315 in network charges. Concerning power usage, on average, ads-supported apps increases it by 0.34W (median of 0.27W). Instead of power, we estimate the cost of ads in terms of energy consumption. The average difference in terms of energy consumption between ads-supported and paid apps is 36.35J (median 20.29J). Considering this fact we conclude that 13.98J $\left(\frac{36.35J}{2.6}\right)$ is the average energy consumption of each independent ad.

8.2.2 Cost of Free Apps Due to Ads

Taking into account the average energy consumption of ads we use Equation 4.1 to calculate the percentage of battery charge that is consumed by ads. We conclude that, on average, each independent ad consumes 0.0486% of the total battery.

We analyze the impact of the refresh rate of ads on the real cost of ads-supported apps calculating the total network (in MB) consumed by ads in ads-supported apps using Equation 8.1, where D is the running time of the app (in seconds), R_{rate} is the refresh rate of ads (in seconds), and $network_{ads}$ is the average network usage of ads in MB.

$$network = \left(\frac{D}{R_{rate}}\right) \times network_{ads} \tag{8.1}$$

We define a similar equation to analyze the impact of the refresh rate of ads on battery life for ads-supported apps using Equation 8.2, where $battery_{ads}$ is the average percentage of battery consumed by ads, which can be calculated using Equation 4.1.

$$battery_{drained} = \left(\frac{D}{R_{rate}}\right) \times battery_{ads} \tag{8.2}$$

In addition, given a data plan size, we calculate the time (in seconds) in which the data limit would be reached because of ads using Equation 8.3, where $datasize$ is the size of the data plan in MB, R_{rate} is the refresh rate of ads (in seconds), and $network_{ads}$ is the average network usage of independent ads in MB.

$$D_{dataplan} = \frac{datasize \times R_{rate}}{network_{ads}} \tag{8.3}$$

There is a point in time in which the hidden costs of ads-supported apps overtake the clear costs of paid apps and, thus, ads-supported apps could be more expensive. To estimate the time in which an ads-supported app overtakes its paid version, we define Equation 8.4, where $price_{app}$ is the cost of the paid app (for example, American dollars), R_{rate} is the refresh rate of ads (in seconds) in the free app, $network_{ads}$ is the average network usage of each independent ad (in MB), $price_{MB}$ is the MB price of overage data (in \$/MB), and $members$ is the number of people in the Google Family Group sharing the app.

$$amortization = \frac{price_{app} \times R_{rate}}{network_{ads} \times price_{MB} \times members} \tag{8.4}$$

8.2.3 Practical Case

Although the cost of network usage and battery life could be considered small, it depends on the use of the app as follows. Let us suppose that we use the ads-supported version of *com.foobnix.pdf.reader*, which is a popular PDF book reader that we used in our case study. Developers use a refresh rate of 60 seconds for this app. If a user uses this app 100 minutes every day for a month, Equation 8.1 shows that about 630MB of data would be spent only in ads. If the data plan is limited, for instance, to 500MB, Equation 8.3 shows that it would be over in 24 days and the user would be billed for 130MB that supposes \$1.95. Regarding battery life and considering Equation 8.2, because the app is used every day for 100 minutes, the battery percentage would be decreased an additional 4.86% each day, for only loading ads. Using Equation 8.4, and considering that A11 costs \$3.49, we obtain that if the data plan is over and the free app is used only by one member of the family group for 19 hours, the cost of data overage due to ads would be higher than the price of the paid app. If we consider that up to five users can belong to a family group, the cost of data overage would be higher than the price of the paid version in only four hours.

8.3 Exploratory Study

We conduct this study about the ads-business model comparing ads-supported and paid apps to understand their differences and development process. We analyze 130 Android apps, 65 ads-supported apps downloaded from Google Play and their corresponding paid versions bought in the same market. We analyze the frequency of releasing of ads-supported and paid apps and the evolution of the prices of paid apps across releases. In addition, we collect and process information about all the developed Android apps offered in Google Play by developers of the selected apps to know the proportion of paid apps with respect to free ones in the marketplace. We also compare both ads-supported and paid apps in terms offered

features, required permissions, and used ad networks.

We collect information about stats of apps, their prices, release dates, and features from Google Play and AppBrain [2]. The latter is a public source for information about Android apps and we cannot guarantee that it does not introduce any bias. However, we compared information existing in AppBrain and Google Play for the last version of the apps used in our study and confirmed that the information offered by AppBrain was consistent. We only consider the last versions because they are the only version available in Google Play. We analyze permissions using the Android tools `aapt` and `dumpsys`, respectively. We developed a Python script that obtains, from the APK file of each app, the list of granted permissions for both ads-supported and paid apps. Then, we apply the `diff` command to analyze the differences between ads-supported and paid versions of each app. Finally, we compute, for the paid apps, the numbers of granted permissions removed and the numbers of granted permissions added in comparison to their corresponding ads-supported versions. Lastly, we use AppBrain Ad Detector to obtain information about the ad networks used in the apps, which is available for free in Google Play. We verified the information offered by this app using the free app Addons Detector, also available in Google Play.

We crawl the Google Play marketplace and the AppBrain website to extract information about ads-supported and paid apps. Then, we analyze permissions using the developed script and we use the app AppBrain Ad Detector to get integrated ad networks in apps. Next, we summarize our findings.

What Mobile Device App Developers Prefer

We analyze the number and type of apps offered in the marketplace by the developers of the subject apps to understand what type of apps developers usually prefer. In total, developers offer 565 apps in Google Play. Out of these 565 apps, 118 free apps have the corresponding paid versions, 167 are without paid apps, and 22 are without free version. Thus, 66.55% of the free apps have corresponding paid versions, 29.56% of apps do not have corresponding paid versions, and only 3.89% of paid apps do not have corresponding free versions.

What Mobile Device Apps Users Prefer

We compare ads-supported and paid apps in terms of numbers of downloads and users' ratings, which are considered measures of success (Harman *et al.* (2012a); Gomes *et al.* (2016)). Ads-supported apps are always downloaded more than their paid counterparts. Considering

2. http://www.appbrain.com

the central values of the ranges defining the numbers of downloads we conclude that ads-supported apps are downloaded, on average, 115 times more than their corresponding paid versions. Regarding ratings, for 56 apps (70.77%), paid versions have better ratings than their corresponding ads-supported versions. For 19 apps (29.23%), free versions have ratings greater than or equal to their paid versions. In average, ratings for paid apps is 4.00 while for ads-supported apps is 3.85.

How Often Developers Release Apps

We analyze the frequency of releasing of ads-supported and paid apps to understand how developers release free and paid apps. Figure 8.2: shows the distribution of the frequencies of releasing of ads-supported and paid versions of the subject apps. In average, ads-supported apps have 21.10 releases while paid apps have 17.90 releases. We also study the number of days between consecutive releases for ads-supported and paid apps. Ads-supported apps are released, on average, every 84 days (median 43 days) while paid apps are released less frequently: every 101 days on average (median 51 days). In addition, we observe that developers usually release ads-supported apps before their corresponding paid versions. Thus, ads-supported apps usually have more releases and are released more often than their corresponding paid versions.

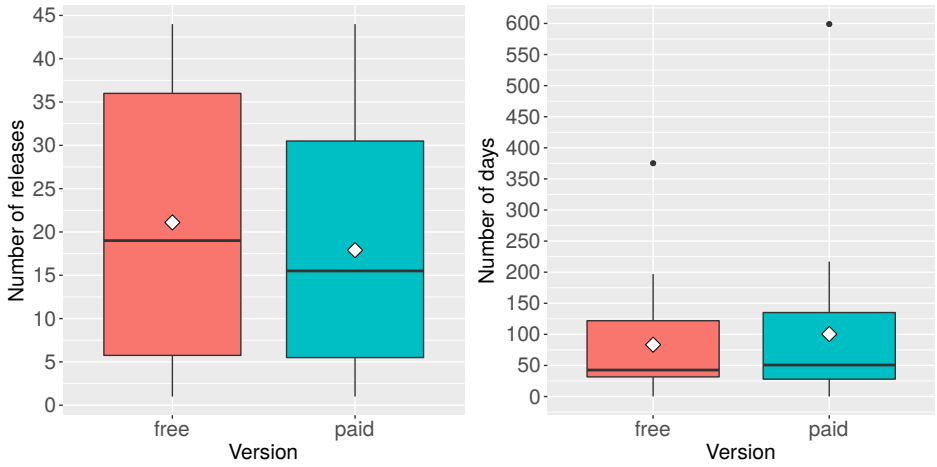


Figure 8.2: Release frequencies for ads-supported and paid apps. The lines in the boxes indicate the minimum value, lower quartile, median, upper quartile, and maximum values. The "◇" symbols represent the average value.

Price Evolution of Apps

We study the evolution of the prices of different releases of paid apps to understand how developers set and update prices. Out of 65 apps, 21 (32.31%) maintained their prices constant over different releases while prices were increased for 13 (20.00%) apps and decreased for seven (10.77%). For the other 24 (36.92%) apps the price fluctuated over the time. We also analyze release dates to check the existence of any seasonal pattern (e.g., sales around Christmas time). There is no a common pattern and price fluctuations are not explainable by seasonal events.

Number of Features of Apps

We compare the features of ads-supported and paid apps to know if ads-supported apps have as much features as the paid ones. In 18 cases, paid apps offer more features than their ads-supported versions. It means that only 27.69% of the studied apps offer more features in their paid versions. In the majority of cases (72.31%), the ads-supported and paid versions of an app are identical in terms of features. Therefore paid apps never offer less features than their corresponding ads-supported versions, which is expected.

Impact of Ads on App Permissions

Permissions is a mechanism that enforces restrictions on the operations that apps can perform. We compare ads-supported and paid apps in terms of numbers and types of granted permissions. For nine apps (13.84%), the paid versions include new permissions that do not exist in the ads-supported versions. For example, the CHECK_LICENSE permission is used in some paid versions to apply license controls to apps published through Google Play. The GET_ACCOUNT, MANAGE_ACCOUNTS, and USE_CREDENTIALS permissions are used for logging in with Google and validate the user. Or the READ_PHONE_STATE permission that is used in combination with the CHECK_LICENSE permission for licensing validation. For 21 apps (32.30%), both ads-supported and paid versions have exactly the same numbers and types of permissions. Concerning paid versions, for 30 apps (46.15%), permissions are removed in comparison to ads-supported versions (mostly permissions related to network access and location, which are required to load ads). Thus, in general, paid apps require less permissions than their corresponding ads-supported versions. Yet, the validation of their licensing may require extra permissions which are not needed by their corresponding ads-supported versions.

Number of Ad Networks in Apps

Ads-supported apps use ad networks, which allow developers to include ads in their apps by providing API and content. We investigate the numbers of ad networks. For each app, for both their ads-supported and paid versions, we retrieve the ad networks that they use. The number of ad networks used by ads-supported apps is in the range $[1, 6]$ (average 1.80). Considering paid apps, six of them contain ad networks and these networks are identical to those in their corresponding free versions. The official Android documentation [3] states that the usage of ad networks increases the sizes of apps and users often avoid downloading large apps [4]. Developers do not always remove ad networks in paid versions of their ads-supported apps, which increases the apps sizes.

Refresh Rate of Ads

When using ad networks, developers set a refresh rate defining how often ads are reloaded. The lower the value the more often are ads reloaded. We also investigate the values of the refresh rates used by developers in their apps. Ad networks recommend to set it to a value in the range of $[30, 120]$ seconds, although the default value use to be 60 seconds. A zero value means that ads would only be loaded once. We observe that for three apps developers set null refresh rates. However, most developers set the refresh rate to 60 seconds. We estimate the average refresh rate of ads as $\lceil \frac{100}{1.88} \rceil = 56$ seconds. Where 1.88 is the average number of ads reloaded in 100 seconds by ads-supported apps. This average refresh rate value has been estimated omitting the three apps with a null value.

8.4 Discussion

Ads-supported apps use more resources than their corresponding paid versions with statistically significant differences. We estimate that the average network usage and energy consumption of ads is 0.21MB and 13.98J, respectively. We offer different equations to estimate the network usage of ads-supported apps, the percentage of battery drained due to ads in free apps, the time in which a data plan is over due to the presence of ads in free apps, and the time in which an ads-supported app overtakes its paid version. Thus, depending on the context of use, paid apps could be less expensive because their costs could be amortized in a short period of time.

Although paid apps are more efficient than their free versions, we compare them to understand

3. https://firebase.google.com/docs/admob/android/lite-sdk
4. https://developer.android.com/topic/performance/reduce-apk-size.html

their differences and development process. From our exploratory study we conclude that developers do not usually offer a paid app without a corresponding free version. Although paid apps have better ratings, ads-supported apps are downloaded much more by users. We observe that ads-supported apps do not usually include less features than their corresponding paid versions. However, ads-supported apps usually require more permissions than paid apps. We also observe that developers usually start releasing free apps and later modify these apps to release them into the marketplace as paid versions. Possibly because of this, developers forget to remove ad networks in paid versions of their ads-supported apps.

From our observations we advise developers to take into account the impact of ads on their apps. If developers want to include ads in their apps we recommend them to integrate the most appropriate advertising TPLs for their app users. As we obtained in the previous chapter, different TPLs have different performance. Offering both ads-supported and paid versions of an app is a good idea. But we suggest developers to remove ad networks in paid apps because they uselessly increase the app size and the number and types of Android permissions. We also recommend developers to take into account the numbers and types of required permissions when a license validation approach is used. License validation requires communication with a server, which could increase the network usage and energy consumption of apps.

CHAPTER 9 AN APP PERFORMANCE OPTIMIZATION ADVISOR

From previous chapters we confirm that developers' decisions have an impact on apps performance. Thus, developers can improve the performance of their apps if they make informed decisions during the development process. We discussed the impact of anti-patterns on energy consumption, but also the impact of map implementations, TPLs, and ads on apps performance.

Developers who focus on apps performance can see improvements in their ratings and, thus, their retention and monetization. However, developers ignore if their apps are as efficient as apps with similar functionalities in the same marketplace. For example, to visit an article in Wikipedia the browser Chrome consumes more energy and transmits more data over the network than the browser Opera (mini). However, the former uses less CPU. It means that there exist a trade-off in terms of performance between different apps. The same keeps for any category of apps in any marketplace and mobile device platform.

Making performance information of apps available in marketplaces would be useful for developers to compare their apps performance with respect to their competitors. But availability of app performance metrics in marketplaces would also be useful for users to select and install efficient apps. However, the choice of efficient (optimal) apps would be complicated because of the cognitive effort imposed to discriminate between different apps and metrics. This is what we define as the App Selection Problem (ASP): the search of optimal mobile device apps regarding different metrics. In this chapter we propose APOA as a mechanism to be implemented in mobile device app marketplaces to make recommendations of efficient apps.

9.1 The App Selection Problem

Given the huge number of available apps in mobile device apps marketplaces, the number of existing categories, and taking into account that, in a category, apps often share similar functionalities, we define the ASP as a combinatorial problem. Let $\mathcal{C} = \{C_1, ..., C_N\}$ be a set of N categories. Further, assume that each category C_i contains a set A_i of apps. An element $\mathbf{x}$ of the search space F, $\mathbf{x} = (x_1, \ldots, x_N)$, is a set of apps where x_l is an app selected from A_l (with $l \in \{1, \ldots, N\}$). A solution $\mathbf{x}$ contains one app from each category in $\mathcal{C}$. Considering the previous, the size of the search space is given by $\prod_{i=1}^{N} |A_i| = |A_1| \cdot |A_2| \cdots |A_N|$, where the operator $|B|$ represents the number of elements in a set B. Because we consider five metrics and we are interested in any combination of them, there exist $\binom{5}{1} + \binom{5}{2} + \binom{5}{3} + \binom{5}{4} + \binom{5}{5} = 31$

different combinations. The binomial coefficient $\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!}$ whenever $k \leq n$ and which is zero when $k > n$. Each of these 31 possible combinations of metrics is a particular instance of the ASP. We summarize all of them in Table 9.1:. The first column contains an identifier associated with each instance. Performance metrics and the rating are shown from the second to the sixth column, respectively. In these columns the symbol "∘" means that the corresponding metric is considered in a particular instance of the problem. On the contrary, the "-" symbol means that the corresponding metric is not considered. Finally, the last column specifies the number of objective functions of each instance. Thus, Instance 1 considers power usage as metric to be optimized while Instance 31 considers power, CPU, memory, and network usages, and the rating.

Table 9.1: Instances of the ASP when up to five different metrics are considered. The symbol "∘" means that the corresponding metric is considered in a particular instance of the problem. The "-" symbol means that the corresponding metric is not considered.

Instance	Power	CPU	Memory	Network	Rating	#Obj
1	∘	-	-	-	-	1
2	-	∘	-	-	-	1
3	-	-	∘	-	-	1
4	-	-	-	∘	-	1
5	-	-	-	-	∘	1
6	∘	∘	-	-	-	2
7	∘	-	∘	-	-	2
8	∘	-	-	∘	-	2
9	∘	-	-	-	∘	2
10	-	∘	∘	-	-	2
11	-	∘	-	∘	-	2
12	-	∘	-	-	∘	2
13	-	-	∘	∘	-	2
14	-	-	∘	-	∘	2
15	-	-	-	∘	∘	2
16	∘	∘	∘	-	-	3
17	∘	∘	-	∘	-	3
18	∘	∘	-	-	∘	3
19	∘	-	∘	∘	-	3
20	∘	-	∘	-	∘	3
21	∘	-	-	∘	∘	3
22	-	∘	∘	∘	-	3
23	-	∘	∘	-	∘	3
24	-	∘	-	∘	∘	3
25	-	-	∘	∘	∘	3
26	∘	∘	∘	∘	-	4
27	∘	∘	∘	-	∘	4
28	∘	∘	-	∘	∘	4
29	∘	-	∘	∘	∘	4
30	-	∘	∘	∘	∘	4
31	∘	∘	∘	∘	∘	5

Considering that different apps belong to N different categories in a marketplace, and that a solution $\mathbf{x}$ of the ASP is a combination of apps, metrics to be optimized can be calculated as follow:

$$power(\mathbf{x}) = \frac{\sum_{i=1}^{N} power(x_i)}{N} \tag{9.1}$$

$$CPU(\mathbf{x}) = \frac{\sum_{i=1}^{N} CPU(x_i)}{N} \tag{9.2}$$

$$memory(\mathbf{x}) = \frac{\sum_{i=1}^{N} memory(x_i)}{N} \tag{9.3}$$

$$network(\mathbf{x}) = \frac{\sum_{i=1}^{N} network(x_i)}{N} \tag{9.4}$$

$$rating(\mathbf{x}) = \frac{\sum_{i=1}^{N} rating(x_i)}{N} \tag{9.5}$$

In Equations (9.1), (9.2), (9.3), and (9.4), $power(x_i)$, $CPU(x_i)$, $memory(x_i)$, and $network(x_i)$ are the average values of power (in W), CPU usage (in %), memory usage (in MB), and network usage (in MB) for app x_i in a certain number of runs and for a given number of exercised app functionalities. In Equation (9.5), $rating(x_i)$ is the rating of the application x_i in the marketplace. Notice that the constant N is just a rescaling factor and thus, in this case, optimizing $\frac{\sum_{i=1}^{N} rating(x_i)}{N}$ is the same as optimizing $\sum_{i=1}^{N} rating(x_i)$. The same holds for performance metrics.

If only one metric is optimized, the problem is considered as a single objective optimization problem. On the contrary, if the number of metrics to optimize is greater than one, the ASP is considered as a multi-objective optimization problem. A solution for any instance of the ASP is a recommendation of optimal apps for one or more categories of apps. We believe that users and developers likely prefer to optimize more than one metric, for example maximizing the rating while at least one performance metric is also optimized. Consequently, we are more interested in instances of the ASP in which two or more metrics are involved. Thus, we focus on multi-objective optimization problems.

9.2 APOA: Conceptual Sequence of Steps

The APOA process is shown in Figure 9.1:. APOA uses as input the set of metrics to be optimized and metric values of mobile device apps. This data can be given as a CSV file. Using this information it solves the corresponding instance of the ASP generating, as output, a Pareto optimal set of apps over which users choose the most preferred solution (decision making). If the input contains metrics for apps in a single specific category APOA will found optimal apps in that category. On the contrary, if metrics are given for sets of apps in different categories, our approach will found optimal combinations of apps. APOA could be considered as a solver of the ASP and it is transparent to the data collection process.

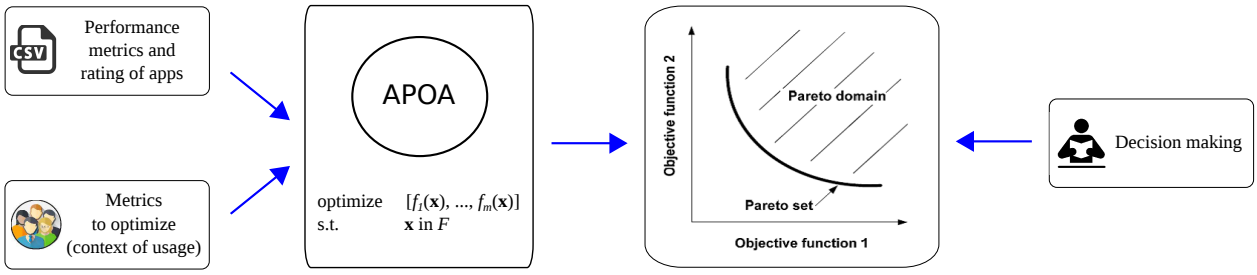


Figure 9.1: APOA conceptual sequence of steps. It uses as input a set of metrics to optimize and metric values for a set of apps belonging to different categories. It solves an optimization problem and it generates, as output, a Pareto optimal front. Each solution in the Pareto optimal front represents an optimal set of apps. Over the resulting Pareto optimal front the user selects the most preferred solution.

9.2.1 Search Space Reduction

Given a set of apps in a category, not all of them have similar metrics. For example, if a user is interested in installing a popular and energy efficient camera app, not all the existing camera apps in a marketplace should be considered because some of them could be less popular and more energy greedy than others in the same category. In that case, the number of apps to take into account could be reduced because only energy efficient and popular apps should be taken into account. It means that only camera apps which are Pareto equivalent considering these two metrics, rating and battery life, should be shown to the user. If we extend this fact to all the categories, the search space can be reduced removing Pareto dominated apps in each category. APOA always applies this reduction approach, which is formally described in Algorithm 7:.

Algorithm 7: Search space reduction in APOA.

Input: Metrics to optimize and apps' metrics.
Output: Pareto optimal apps in each category.
1: **for each** A_i where $i \in \{1, \ldots, N\}$ **do** ▷ A_i is the set of apps contained in category C_i
2: A'_i = Apply Pareto dominance to apps in A_i.
3: **end for**
4: **return** A'_i $\forall i \in \{1, \ldots, N\}$

9.2.2 Solving the ASP

After the search space reduction, depending on the number of categories and apps by category, the decision space could still be huge. In that case, EMO algorithms can be used to generate a set of close to optimal solutions. On the contrary, if the search space is small, an exhaustive search can be applied. In this case, APOA enumerates all the possible combinations and applies the Pareto dominance relation to select the Pareto optimal combinations of apps. Algorithm 8: shows the exhaustive search used by APOA. First, it generates (line 1) all the possible combinations of apps belonging to each category. It means that, if there are N categories and each category contains a set of A'_i apps after the search space reduction, $\prod_{i=1}^{N} |A'_i|$ different combinations of apps are generated. Second, for each combination, objective values associated to the metrics to be optimized are calculated using equations (9.1), (9.2), (9.3), (9.4), and–or (9.5) (line 2). Third, the Pareto dominance relation is applied over the combinations of apps to select the Pareto optimal ones (line 3).

Algorithm 8: Exhaustive search in APOA.

Input: Metrics to optimize and apps' metrics.
Output: Pareto optimal set of apps (the Pareto optimal front).
1: $Comb$ = Combinations of apps belonging to each category.
2: Calculate objective functions of each combination in $Comb$.
3: $output$ = Apply Pareto dominance to $Comb$.
4: **return** output

APOA and instances of the ASP have also been implemented in jMetal (as EARMO). An additional input parameter is used to specify if APOA solves an instance of the ASP performing an exhaustive search or running an EMO algorithm.

9.2.3 Output

As output, APOA shows a Pareto optimal front and the existing trade-off for each solution and optimized metrics. All the solutions in a Pareto optimal front are equivalents considering the Pareto dominance relation, because if an objective is improved another objective is

worsened. For this reason, there is not a best solution and the trade-off between different solutions and objectives should be analyzed by the user or decision maker (DM) to choose the most preferred solution (decision making). The trade-off of a solution and an objective function specifies the difference in percentage with respect to the best value of this objective function in the Pareto optimal front. APOA returns this information in a table enumerating all the Pareto optimal solutions and their trade-offs. In addition, to make easier the comparison between different solutions, APOA shows trade-offs in a stacked bar graph. The bars in a stacked bar are divided into different categories, one for each optimized metric.

9.3 Case Study

In order to evaluate APOA we propose a case study over a subset of Android apps. We choose the Android ecosystem because it is the most popular mobile operating system globally. The goal of this study is to assess APOA capabilities with the purpose of understanding APOA applicability to find optimal sets of apps in terms of different metrics.

The evaluation of the case study is executed from the perspective of users who wish to select and install a set of apps from a set of categories, and the perspective of developers who need to benchmark their apps against apps in the same category. First, we define different context of use to simulate users' preferences. Second, we simulate the availability of performance metrics of apps in the Android marketplace. For that, we select a subset of the most popular Android apps in the marketplace, and we collect performance metrics for them in a typical usage scenario. We analyze metric values of apps grouped by category, but we also analyze the performance metrics of apps in a particular category (browsers). This shows the fact that, as it was expected, different apps have different performance. Finally, using all of these measures, we evaluate APOA considering different context of use to make recommendation of apps for all the categories. In the following subsections we detail how we handle each of these steps.

9.3.1 Contexts of Use

There are many different contexts of mobile device use due to the wide variation among users' usage (Falaki *et al.*, 2010). Depending on the context, some performance metrics are more important than others. Therefore, even if different apps offer similar functionalities, one app could be preferred over others because of its performance in that context. Thus, we consider that the context of use affects users' preferences about the metrics to be optimized. We define two different contexts of use which are associated to instances of the ASP. Next,

we present these contexts of use which are used in this case study to evaluate APOA.

Travel Abroad

This context considers users who travel abroad, whether for work or leisure. In this case we suppose that most important performance metrics are battery life, as the time between consecutive charges is likely longer, and network usage, as data roaming is usually expensive. This context of use corresponds to Instance 8 of the ASP.

Emerging Markets

CPU and memory usages are important metrics because apps that are CPU and–or memory greedy, slow down devices, impacting negatively the users' experience. This is specially important for mobile device users in emerging markets, who own low-end mobile devices with low memory and power processing. But emerging market users also have expensive and slow data connections. We consider that most important metrics are CPU and memory usages, to accommodate low-end devices, and network usage, to minimize data transmission. This context of use corresponds to Instance 22 of the ASP.

Both contexts of use and their correspondences to different instances of the ASP are summarized in Table 9.2:. The first column contains contexts of use while the second one indicates the associated instance of the ASP. Metrics are shown from the third to the seventh column. In these columns the symbol "○" means that the corresponding metric is considered in that particular instance of the problem, as it was shown in Table 9.1:. On the contrary, the "-" symbol means that the corresponding metric is not considered. The last column specifies the number of objective functions to be optimized.

Table 9.2: Correspondence between contexts of use and instances of the ASP. The symbol "○" means that the corresponding metric is considered in a particular instance of the problem. The "-" symbol means that the corresponding metric is not considered.

Context of Use	ASP instance	Power	CPU	Memory	Network	Rating	#Obj
Travel abroad	8	○	-	-	○	-	2
Emerging markets	22	-	○	○	○	-	3

9.3.2 Simulating the Availability of Performance Metrics of Apps

In order to simulate the availability of performance metrics in a marketplace, we define a subset of the categories and apps existing in the official Android marketplace. We select

the seven ($N = 7$) most common app categories used by mobile device users: browsers, cameras, flash lights, music players, news viewers, video players, and weather forecast. For each category, we select the most popular 100 apps and we automatically download (using a Perl script and the tool Play Store Crawler [1]) their descriptions, statistics (including the rating), and APK files. Finally, we select for each category the 20 apps with the highest rating. Therefore, our homemade marketplace contain 140 apps for all of the seven selected categories.

Definition of Typical Usage Scenarios

For each app in a category, we propose a typical usage scenario that we automatically play while we collect performance metrics. We use the scenarios defined by Saborido *et al.* (2016), which were collected interacting with each app under study using the Android app HiroMacro.

Table 9.3: Typical usage scenario defined for each app category for the APOA case study.

Category	Scenario description
Browsers	Search and read an article in Wikipedia.
Cameras	Take three pictures.
Flash lights	Use the torch during 10 seconds.
Music players	Play two songs during 20 seconds.
News	Read the two first news.
Video players	Play a movie for 30 seconds.
Weather	Get the forecast for two different cities.

Although the scenarios are representative of the overall app behavior, we are aware that different features have different performance characteristics. Thus, more experiments considering different scenarios is required. However, our goal is to show APOA feasibility and support the evidence of its usefulness.

Data Collection and Processing

For each app we run a simple scenario to start the app, skip the initial tutorial (if it exists), and interact with the app to simulate the user interaction. We run these scenarios automatically while performance metrics are collected. We run each app 20 times to get statistical results.

1. https://github.com/Akdeniz/google-play-crawler

9.3.3 Results

In total, for this case study, we collect 11,200 files (more than one Terabyte of data). We present metrics of the selected apps grouped by category in Figure 9.2:. The *x-axis* shows the categories and the *y-axis* shows the distribution of metric values. Power usage of flash lights apps is much lower than apps belonging to other categories. On the contrary, cameras are the apps which consume more power (battery life will approximately be 2.29 hours while for flash lights apps, battery life will approximately be more than eight hours). In terms of CPU and median values, news and video players apps use more CPU (10.46% and 9.34%, respectively) while browsers and cameras apps do a low usage of it (2.76% and 1.78%, respectively). Regarding memory usage and median values, apps belonging to the browsers or news readers category use more memory than other apps (125.00MB and 153.57MB, respectively). The same keeps for network usage. In this case, apps belonging to browsers or news readers category consume more data over the network than apps belonging to other categories (0.62MB and 1.36MB, respectively). In addition to performance metrics, we also analyze the rating of selected apps. As it is shown, the median rating is in the range $(4.20, 4.40)$ which makes sense because we selected the most popular apps by category.

As it is expected, apps belonging to different categories have different performance. But it also keeps for apps in a category. In Figure 9.3: we show, as example, the performance metrics of the most popular 20 browsers. Although for apps in this category there are not huge differences in terms of power usage, some apps are better than others. Something similar happens for network usage. On the contrary, there exist important differences for CPU and memory usages even if we are comparing apps with similar functionalities (browsers) playing the same scenario (searching and reading the same article in Wikipedia).

Although APOA can be applied on a single category, like browsers, from now on we focus on all the categories at the same time. Thus, the resolution of each instance of the ASP generates optimal combination of apps for all the categories.

Resolution of the ASP

Next, we present the results of using APOA to solve all the instances of the ASP for the Android case study. We use both the exhaustive search and an EMO algorithm. Table 9.4: shows the number of solutions obtained by APOA running the exhaustive search. Each row corresponds to each instance of the ASP. From the third to the ninth columns the number of Pareto optimal apps in each category is shown. The tenth column shows the number of possible combinations of apps (solutions) after the search space reduction considering all the

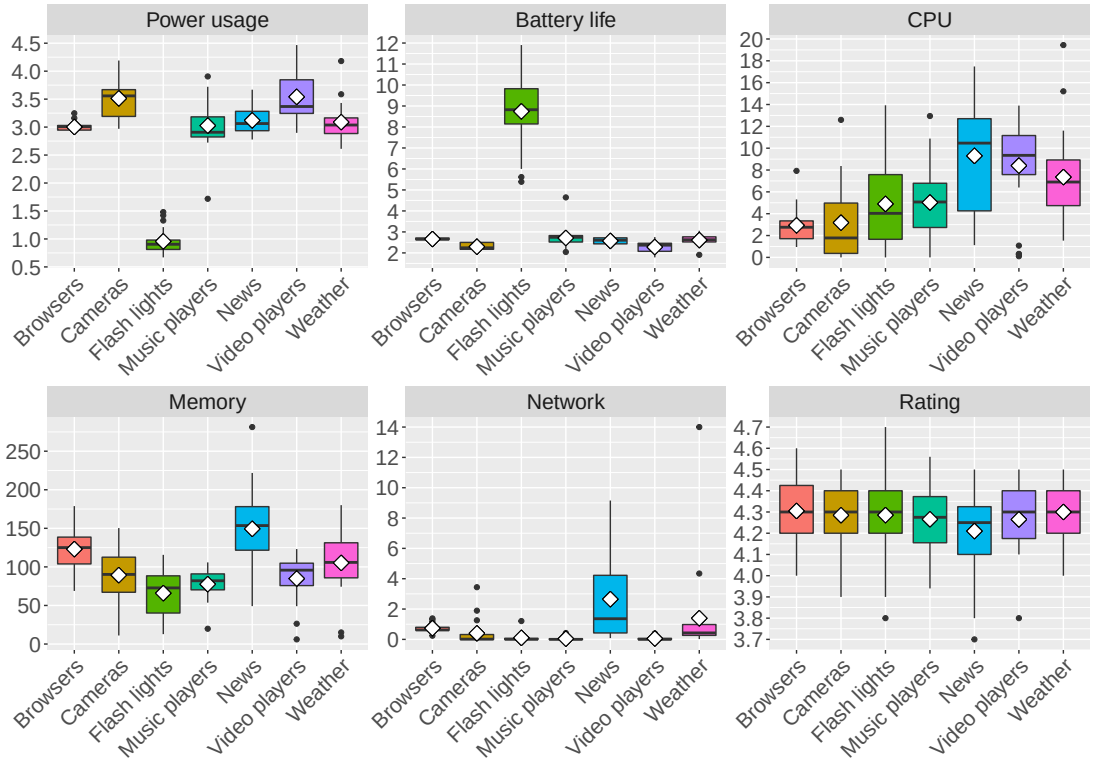


Figure 9.2: Metrics of the analyzed apps for the APOA case study, grouped by category. The lines in the boxes indicate the minimum value, lower quartile, median, upper quartile, and maximum values. The "◇" symbols represent the average value.

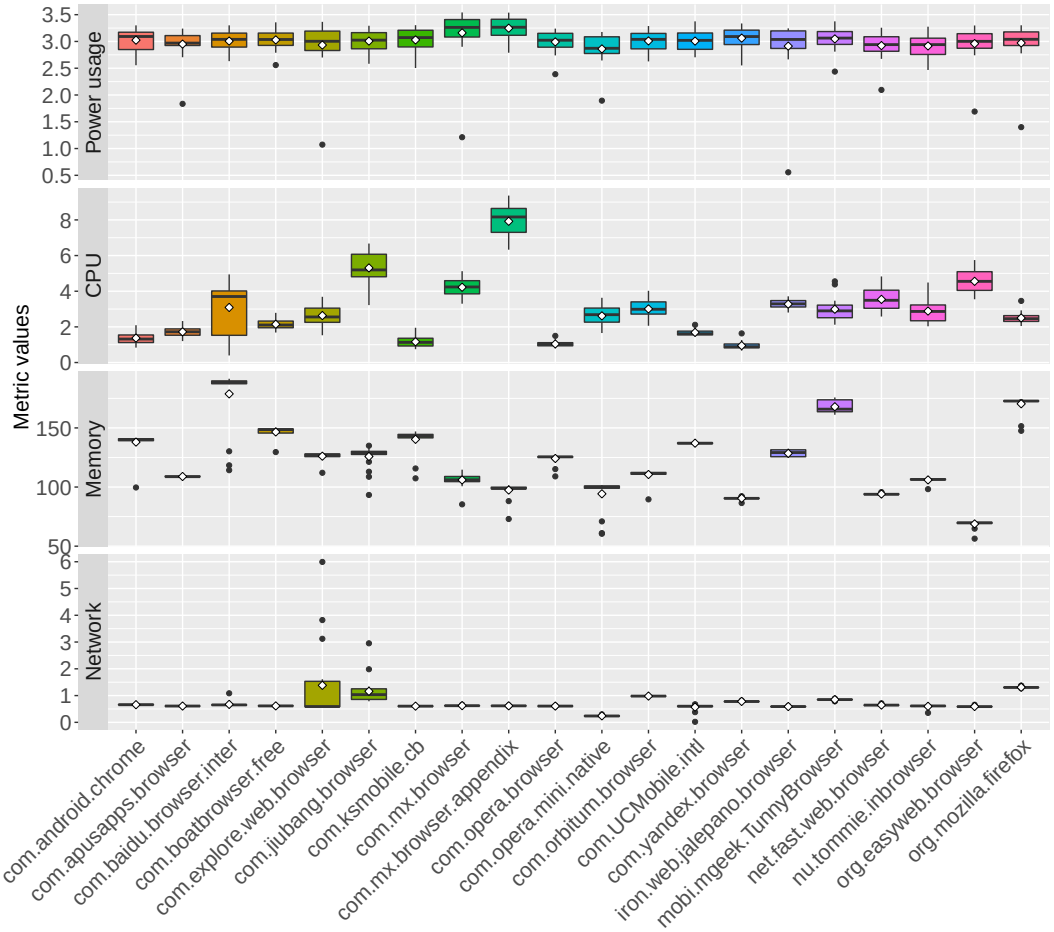


Figure 9.3: Metrics of the analyzed apps for the APOA case study in the browsers category. The lines in the boxes indicate the minimum value, lower quartile, median, upper quartile, and maximum values. The "◇" symbols represent the average value.

categories. Finally, the last column, specifies the number of Pareto optimal solutions over all the possible solutions (previous column). As it is shown, the search space reduction reduces the number of optimal apps in each category and, consequently, the number of candidate solutions. For example, if we consider Instance 31 and the browsers category, out of 20 apps nine (45.00%) are Pareto optimal. It means that 11 apps are discarded and they are not considered in the optimization process. If we consider again Instance 31, 23,591 (0.15%) over 15,459,444 existing solutions are Pareto optimal. This shows the need of a recommendation system as APOA to filter out non-optimal solutions and help users to reduce the cognitive effort to choose the most preferred ones.

Table 9.4: Number of solutions and Pareto optimal solutions for each instance of the ASP for the APOA case study.

Instance	#Obj	Browsers	Cameras	Flash ...	Music ...	News	Video ...	Weather	Solutions	Optimal
1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
5	1	2	1	1	1	1	1	2	4	4
6	2	4	3	2	5	3	3	2	2,160	48
7	2	3	1	1	3	2	4	4	288	18
8	2	1	5	1	3	1	2	4	120	27
9	2	2	2	3	4	2	3	2	576	15
10	2	2	4	2	4	1	1	5	320	31
11	2	5	2	2	5	3	4	2	2,400	70
12	2	2	3	4	3	2	3	1	432	20
13	2	2	4	1	3	2	2	3	288	19
14	2	5	2	3	4	3	3	4	4,320	38
15	2	3	4	5	2	2	2	2	960	29
16	3	6	5	2	7	3	4	13	65,520	309
17	3	6	7	2	9	3	7	6	95,256	608
18	3	6	5	6	9	4	8	3	155,520	461
19	3	3	7	1	4	2	4	6	4,032	184
20	3	6	2	6	5	5	7	9	113,400	440
21	3	4	9	9	4	2	4	6	62,208	668
22	3	8	5	2	7	3	5	8	67,200	367
23	3	7	5	8	8	4	5	6	268,800	535
24	3	5	5	7	8	4	9	2	100,800	638
25	3	8	7	7	4	4	4	7	175,616	466
26	4	8	9	2	10	3	8	13	449,280	1,733
27	4	9	6	10	11	6	10	14	4,989,600	4,243
28	4	6	10	9	11	4	12	7	1,995,840	5,916
29	4	8	10	10	5	5	7	10	1,400,000	3,424
30	4	9	7	9	12	6	10	10	4,082,400	4,272
31	5	9	11	11	13	6	13	14	15,459,444	23,591

We also run APOA using NSGAII instead of the exhaustive search. NSGAII is an state-of-the-art EMO algorithm that has been successfully applied in combinatorial multi-objective optimization problems in different fields. Table 9.5: shows the parameters and operators used in NSGAII, which are proposed as default parameters by its authors. We use the single point crossover operator because it is one of the simplest crossover operators and it works reasonably well in combinatorial problems. When two parents are selected, with a probability

of P_x the operator creates new individuals. It selects a point on both parents and all data beyond that point in either individual is swapped between the two parents. The resulting solutions or individuals are the offspring. Considering the mutation, we use the flip mutation operator. It changes the value of a gene in the individual, with a probability of P_m, with a new value generated randomly in the lower and upper bounds range. We also use the binary tournament selection operator to select individuals in the population to create the offspring. This operator selects two solutions randomly in the population and chooses the best one, or one of them with a probability of 0.5 if they are equivalents.

Table 9.5: Parameters settings for the EMO algorithm NSGAII for the APOA case study.

Parameter	Value
Population size	200
Generations	300
Crossover operator	Single point crossover
Crossover probability (P_x)	0.9
Mutation operator	Flip mutation
Mutation probability (P_m)	$1/C = 0.125$
Selection operator	Binary tournament

Figure 9.4: shows the solutions obtained by APOA for two dimensional instances of the ASP for the Android case study (instances with more than two objectives are not shown because the resulting plots are hard to read). It shows all the possible solutions, the Pareto optimal solutions found by APOA applying the exhaustive search, and the Pareto optimal ones found by APOA running the EMO algorithm NSGAII. This figure allows to compare the real Pareto optimal front obtained by the exhaustive search with respect to the approximation of the Pareto front generated by NSGAII. As it is shown, solutions found by the exhaustive search and the EMO algorithm are overlapped which means that the latter is able to find the optimal solutions for these instances. This fact is expected, given that the search space is small in that cases. Concerning the rest of the instances, where the search space is bigger and more objective functions are involved, we check that NSGAII is able to find solutions close to the optimal ones. From this we conclude that EMO algorithms, as NSGAII, are a good alternative to the exhaustive search to solve the ASP.

We ran the experiments in a Lenovo ThinkPad laptop (4×Intel Core i5-6200U CPU @ 2.30GHz) running Debian GNU/Linux Stretch. The exhaustive search took around 22 hours to solve all the instances of the ASP for the Android case study. Although Instance 31 took 14 of those hours to solve. However, the EMO algorithm NSGAII took less than two minutes to solve all the ASP instances.

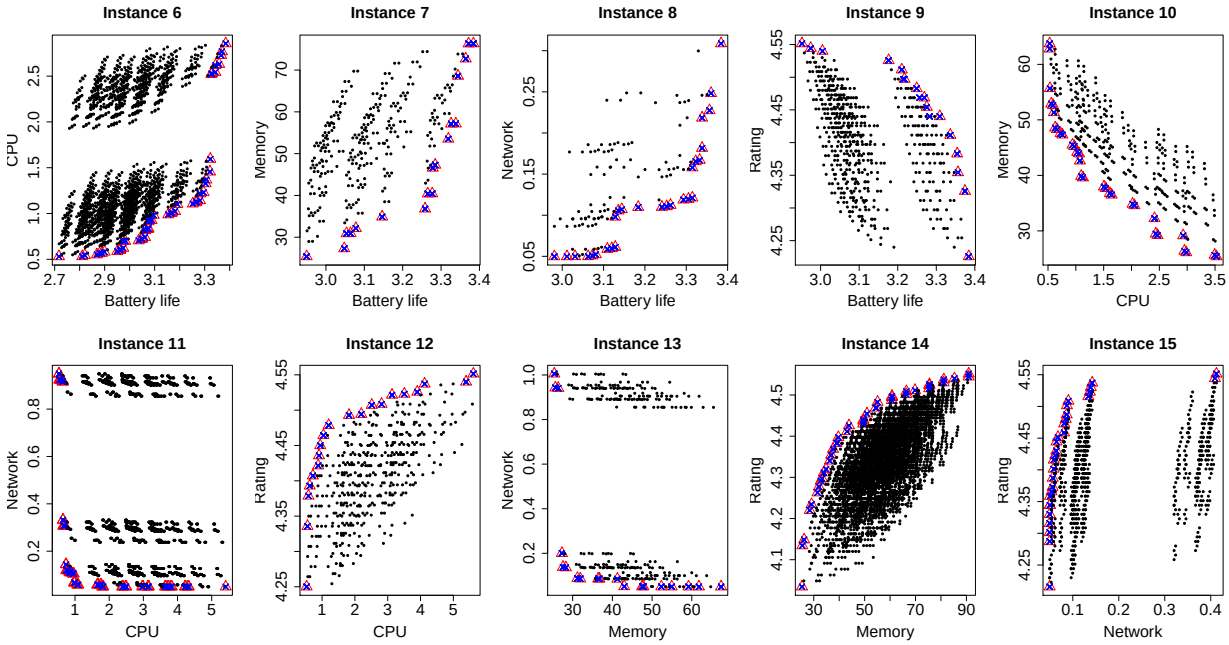


Figure 9.4: Solutions of the bi-objective instances of the ASP for the APOA case study. Symbol (•) is used for all the possible solutions while Pareto optimal solutions are shown using the symbol (△). Solutions found by APOA running NSGAII are shown using the symbol (×).

9.3.4 Helping Users in Choosing Optimal Apps

Next, we use APOA in the contexts of use described previously in Section 9.3.1. For each context, we show the existing trade-off associated to each optimal solution and performance metric. As we defined in Section 9.2.3, the trade-off of a solution and an objective function specifies the difference in percentage with respect to the best value of that objective function in the Pareto optimal front. It allows to compare the performance of optimal solutions.

We also compare the performance of optimal solutions found by APOA for each context of use with respect to the set of apps with the highest rating per category. This is what we define as the user solution, the set of apps with the best rating. This comparison allows to know the benefit of using efficient apps instead of popular ones. Considering the case study, the user solution is defined by the apps shown in Table 9.6:. Let us define $\mathbf{x}^{user}$ as the user solution. Based on our study, $power(\mathbf{x}^{user}) = 2.81$, $CPU(\mathbf{x}^{user}) = 8.92$, $memory(\mathbf{x}^{user}) = 103.72$, $network(\mathbf{x}^{user}) = 1.27$, and $rating(\mathbf{x}^{user}) = 4.55$, calculated using equations (9.1), (9.2), (9.3), (9.4), and (9.5). Based on power usage, and using equation (4.2), $battery(\mathbf{x}^{user}) = 2.84$.

Table 9.6: User solution – Apps with the best rating per category for the APOA case study.

Category	App
Browsers	mobi.mgeek.TunnyBrowser
Cameras	com.roidapp.photogrid
Flash lights	goldenshorestechnologies.brightestflashlight.free
News	com.guardian
Music players	com.tbig.playerprotrial
Video players	video.player.audio.player.music
Weather	com.handmark.expressweather

Travel Abroad

This context of use considers users who travel abroad (for working or holidays). In this case we consider battery life and network usage the most important metrics. First, because likely the time between consecutive charges uses to be longer and, second, because data roaming is usually expensive. This context of use corresponds to Instance 8 of the ASP, whose Pareto optimal front was previously presented in Figure 9.4:. Out of 120 possible solutions 27 (22.50%) are Pareto optimal in terms of battery life and network usage. Table 9.7: shows these solutions. The second and third columns show the objective values of these performance metrics. Columns fourth and fifth show the trade-off associated to each optimal solution and performance metric. Figure 9.5: shows using a bars plot the trade-off of each optimal solution for battery life and network usage. This plot and the previous table, used together,

are useful to visualize and compare Pareto optimal solutions. If the user prefers battery life to network usage, Solution 1 could be chosen. In that case network usage is increased 0.26 MB (83.88%) with respect to Solution 27, which has the lowest network usage. On the contrary, if network usage is preferred, Solution 27 could be chosen decreasing battery life up to 24 minutes (11.92%) with respect to Solution 1. If both performance metrics are equally important, Solution 19 could be chosen as the preferred one because the trade-off is almost similar for both objective functions. In that case, battery life is decreased up to 15 minutes (7.56%) respect to Solution 1 and network usage is increased 0.01 MB (3.70%) with respect to Solution 27.

Table 9.7: Pareto optimal solutions found by APOA and their associated trade-off for the travel abroad context of use. Objective values for battery life and network usage are expressed in hours and MB, respectively.

	Objective values		Trade-off (in %)	
Solution	Battery life	Network	Battery life	Network
1	3.38	0.31	0.00	83.88
2	3.36	0.25	0.71	64.30
3	3.36	0.23	0.83	57.54
4	3.34	0.22	1.36	54.60
5	3.34	0.18	1.36	42.70
6	3.33	0.17	1.53	37.96
7	3.32	0.17	1.81	37.45
8	3.31	0.16	2.05	35.02
9	3.31	0.12	2.05	23.12
10	3.31	0.12	2.33	22.61
11	3.29	0.12	2.69	22.35
12	3.26	0.11	3.61	20.24
13	3.25	0.11	3.89	19.72
14	3.24	0.11	4.23	19.47
15	3.18	0.11	5.92	19.42
16	3.14	0.11	7.10	18.54
17	3.14	0.11	7.35	18.03
18	3.13	0.10	7.56	15.60
19	3.13	0.06	7.56	3.70
20	3.12	0.06	7.81	3.18
21	3.11	0.06	8.13	2.92
22	3.08	0.05	8.96	0.82
23	3.07	0.05	9.20	0.30
24	3.06	0.05	9.50	0.04
25	3.03	0.05	10.44	0.04
26	3.01	0.05	11.02	0.00
27	2.98	0.05	11.92	0.00

We also compare the performance of optimal solutions found by APOA for the travel abroad context of use with respect to the performance of $\mathbf{x}^{user}$. Solution 1 extends battery life up to 32 minutes and decreases network usage 0.96 MB with respect to $\mathbf{x}^{user}$. If we consider

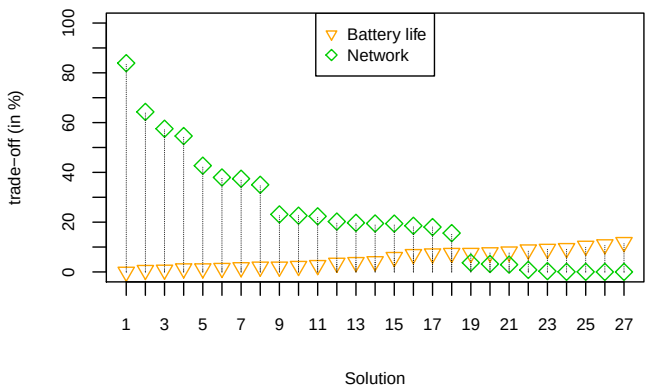


Figure 9.5: Trade-off (in %) for battery life and network usage of each optimal solution found by APOA for the travel abroad context of use.

Solution 27, it extends battery life up to 10 minutes and decreases network usage 1.22 MB with respect to $\mathbf{x}^{user}$.

Emerging Markets

This context of use considers emerging market users who have limited access to data connections and low-end devices. In this case we consider CPU, memory, and network usages as the most important performance metrics. This context of use corresponds to Instance 22 of the ASP. As it was shown in Table 9.4:, out of 67,200 possible solutions 367 (0.55%) are optimal in terms of CPU, memory, and network usages. Figure 9.6: shows the trade-off of each optimal solution for CPU, memory, and network usages. If memory and network usages are less important than CPU usage, Solution 325 could be chosen because it is the solution that uses less CPU. However, it uses more memory and data over the network than other optimal apps. Something similar happens if CPU and network usages are less important than memory usage. In this case Solution 350 could be chosen. It is the solution that uses less memory, but it uses more CPU and network than other optimal solutions. If data usage is a priority, Solution 1 could be chosen because it is the solution that transmits less data over the network. However, it increments CPU and memory usages with respect to the solutions with the lowest value for these two metrics.

We also compare the performance of optimal solutions found by APOA for the emerging market context of use with respect to the performance of $\mathbf{x}^{user}$. Solution 325, that has the lowest CPU usage, decreases CPU, memory, and network usages 8.39%, 39.98MB, and 0.32MB, respectively, with respect to $\mathbf{x}^{user}$. If we consider Solution 350, that has the lowest memory usage, it decreases CPU, memory, and network usages 5.39%, 78.34MB, and 0.26MB,

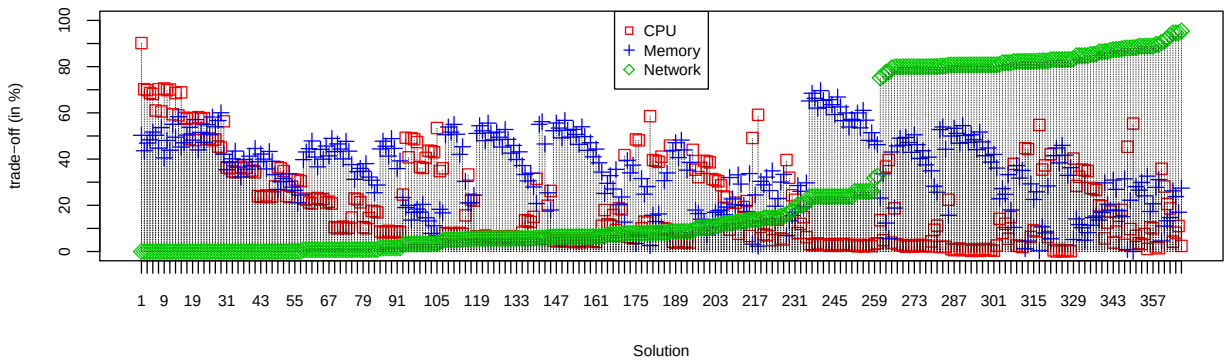


Figure 9.6: Trade-off (in %) for CPU, memory, and network usages of each optimal solution found by APOA for the emerging market context of use.

respectively, with respect to $\mathbf{x}^{user}$. Finally, if we consider Solution 1, that has the lowest network usage, it decreases CPU, memory, and network usages 3.51%, 36.34MB, and 1.22MB, respectively, with respect to $\mathbf{x}^{user}$.

9.3.5 Assisting Developers in Comparing Apps

Performance metrics are not only important for mobile device users, they are also important for developers. Their availability in mobile device app marketplaces would allow to know how close or far is a new app with respect to its competitors in the marketplace, in terms of different performance metrics. In this subsection we show how APOA can assist developers in comparing new apps with respect to their competitors in the marketplace.

Let us suppose that a company develops a new Android browser app. The scenario associated to the browsers category is played and performance metrics are collected. Let us consider that the associated power (battery life), CPU, memory, and data usages are 3.10W (2.57 hours), 9.00%, 65.00MB, and 0.40MB, respectively, for this new app. Using histograms, developers can visualize and compare performance metrics of apps belonging to the same category. It allows developers to know how their new app is positioned with respect to the others. Figure 9.7: shows the histograms of apps in the browsers category for each performance metric (where the red bar represents the new app). The new app is the third worst in terms of power usage (battery life) and the worst in terms of CPU usage. However, the new app is the best one in terms of memory usage and the second best regarding data usage. Thus, in this case, developers could focus on improving power and CPU usages before releasing the new app into the marketplace.

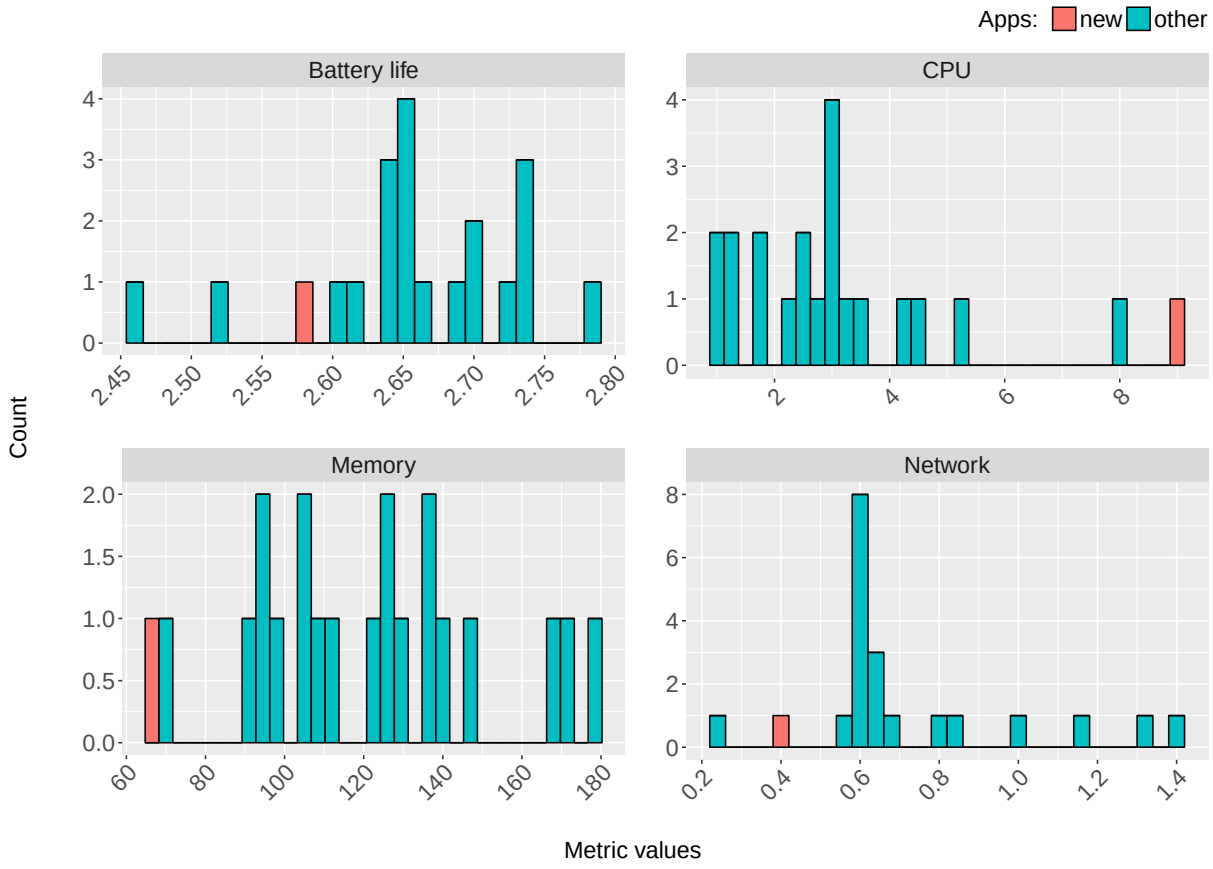


Figure 9.7: Histograms for performance metrics of a new browser and the browsers selected for the APOA case study.

Now, let us suppose that developers want to optimize their new browser for a concrete context of use. For instance, for emerging market users. In this case, CPU, memory, and network usages are the metrics to be optimized, and developers want to compare the new app with respect to the optimal ones. APOA found eight optimal browser apps for our case study considering these three metrics. Figure 9.8: shows and compare performance metrics of the new app with respect to the optimal browsers for the emerging market context of use. From here, developers obtain that their app is the most efficient in terms of memory usage but, on the contrary, there exist large differences in terms of CPU usage with respect other apps. For instance, the browser *com.yandex.browser* uses 0.95% of CPU while the new app uses 9.00%. Concerning network usage, although the new app is better than others, it transmits almost the double of *com.opera.mini.native*. The former uses 0.40 MB while the latter uses 0.24 MB. After this analysis, developers could focus on the study of *com.yandex.browser* and *com.opera.mini.native* to understand why they are more efficient for CPU and network usages, respectively.

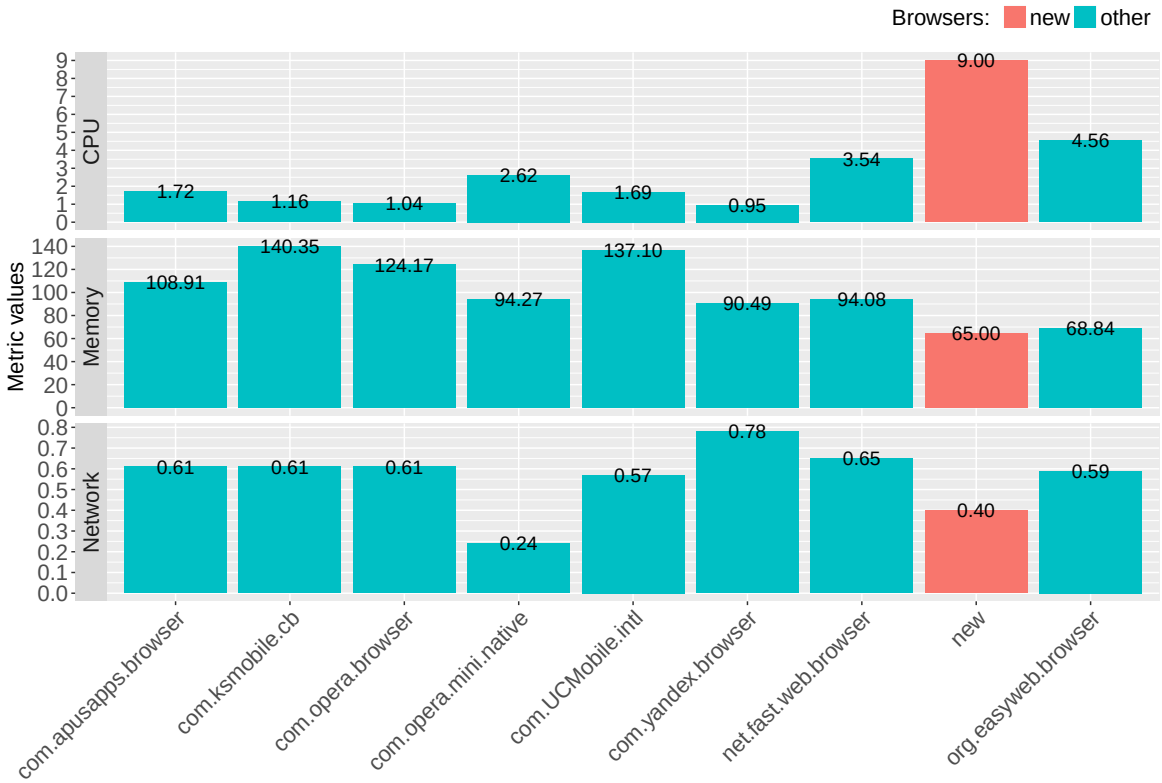


Figure 9.8: Comparison of performance metrics of a new browser with respect to optimal browsers found by APOA.

9.4 Discussion

The official marketplaces of Android and Apple platforms offer more than a million mobile device apps belonging to different categories. In both marketplaces information about performance metrics is not available and, therefore, mobile users select apps based on other criteria, as the rating. Even if performance metrics are available, different apps have different performance and, depending on the context of use, some metrics could be more important than others. All of this makes the comparison of apps difficult in terms of performance because the required cognitive effort. We proposed APOA as a recommendation tool to complement mobile device app marketplaces allowing developers and users to compare optimal apps or to rank them relevant to their current context and needs.

We evaluated APOA over an Android case study. Out seven categories and 140 apps, we defined typical usage scenarios and we collected information about performance metrics. We show the benefit of using APOA to find optimal combinations of apps and compare them regarding their efficiency. Finally, we illustrate how the availability of performance metrics and the usage of APOA can be helpful for developers before releasing a new app into a marketplace. Thus, they can know more about differences in terms of performance between a new app and similar existing apps. This fact would motivate the development of more efficient apps, which benefits final mobile device app users.

CHAPTER 10 CONCLUSION

A majority of users in emerging markets face constraints not commonly seen in developed markets: limited access to data connections, high costs when data connections are available, low-end devices with reduced memory, and limited opportunities to recharge batteries during the day. Thus, mobile device users want to use efficient apps that make an optimal use of device resources, what improves their performance: energy consumption and CPU, memory, and network usages.

Performance metrics are proxies for quality of mobile apps. Therefore, developers who focus on app performance can see improvements in their ratings and, thus, their retention and monetization. However, developing efficient apps is a challenging task because developers ignore the impact of their decisions on apps performance.

The main goal of our research is to assist developers and users in developing and using, respectively, efficient apps. Our thesis is that multi-objective approaches support developers and users to implement and choose efficient mobile device apps. We have answered this thesis positively addressing our research goal using multi-objective approaches (1) providing techniques and guidelines to help developers make informed design and implementation decisions to improve the performance of their apps and (2) assisting developers in the comparison of apps performance and helping users make informed decisions to choose their apps.

In this final chapter we summarize the research contributions, the limitations of our research, and the threats to validity of our studies. Then, we explore related future work.

10.1 Advancement of Knowledge

Software developers can follow good design practices, as the standard process and rules of object-oriented design, but it does not guarantee efficient mobile device apps. Following these practices, developers could still introduce anti-patterns that reduce the performance of mobile device apps. Thus, one could argue that improving traditional quality attributes like readability, flexibility, extendability, and reusability, and improving the efficiency of an app, do not arise at the same time during the software development process. However, we proposed automated refactoring generation as a way to support software developers to write quality and energy efficient code. The refactoring operations proposed by an automated approach would show design choices that developers could follow to produce more efficient apps. To assist developers in producing more energy efficient apps we proposed EARMO.

A multi-objective approach to detect and correct anti-patterns in mobile device apps while improving their energy-efficiency. In our validation, we obtained that EARMO can propose sequence of refactorings that remove a median of 84% of anti-patterns while improving energy usage of apps extending battery life from a few minutes up to 29 minutes.

One of the most popular Android anti-patterns is HMU, which exists in Android apps when developers use a Java map implementation (`HashMap`) instead of an specific Android implementation (`ArrayMap`). The Android developers' reference documentation recommends `ArrayMap` and `SparseArray` variants as more memory efficient alternatives to `HashMap`. However, information about Android map implementations performance is vague. We studied the use of map implementations in more than 5,000 Android apps and we performed an empirical study comparing map implementations performance. We confirm that Android map implementations are more memory efficient than `HashMap`. But Android map implementations also are more efficient in terms of energy consumption and CPU usage, but not for all the operations and data sizes. We recommend to use `HashMap` instead of `ArrayMap` when keys are objects to improve the energy efficiency of Android apps. We also recommend `SparseArray` and `LongSparseArray` over `HashMap` when keys are primitive types to improve the performance of Android apps. If values are also primitive types, and deletion operations are not usual, we suggest `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` as more efficient alternatives to `SparseArray` and `LongSparseArray`. If primitive types are used as keys, we discourage the use of `ArrayMap` because `SparseArray` variants are more efficient. Thus, we strongly recommend that Android Studio suggests replacing `ArrayMap` by `SparseArray` variants when keys are primitive types. We proposed guidelines to help developers make informed decisions about the performance of map implementations. We also validated our guidelines by replacing in an Android app `HashMap` with `SparseIntArray` extending battery life by almost three minutes. The observed improvement is rather the lower bound of possible improvements because the selected app use maps in only one of its method, which is not heavily used.

While we analyzed the source code of Android apps to study the use of map implementations, we realized that most developers integrate TPLs in their apps to implement different functionalities. Developers have tens of TPLs to choose from, but the impact on apps performance of each alternative is ignored beforehand. We proposed an approach to help developers measure and compare the performance of different TPLs. We also used this approach to create a catalog of performance metrics for the three most popular Android TPLs in three of the most popular TPL categories (advertising, crash reporter, and analytics). We obtained that, for the advertising and crash reporter TPL categories, there is not a best choice because there exist a trade-off between different TPLs and performance metrics. Thus, developers

should make their decisions about the TPL to integrate based on the context of use of their apps. For example, if developers want to target emerging markets, they could integrate the TPL that minimizes CPU, memory, and network usages. However, concerning the analytics TPL category, we obtained that one of the analyzed TPL is more efficient than the others. Although it consumes more energy, differences are not statistically significant. On the contrary, it uses less CPU, memory, and network and differences are statistically significant and the effect size is large.

Advertising TPLs use more CPU and memory and transmit more data over the network than other TPLs. Although the impact of ads on performance have been studied before, we extended previous research comparing ads-supported apps with respect to their paid versions. We confirm that ads-supported apps use more resources than their corresponding paid versions, but we extend this claim stating that differences are statistically significant and the effect size large. We also studied 130 Android apps comparing differences between their free and paid versions. We observed that users prefer ads-supported apps although they rate paid apps better. Developers prefer offering both free and paid version of their apps but ads-supported apps usually have more releases and are released more often than their corresponding paid versions. We also observed that paid apps do not usually include more features than their corresponding ads-supported versions. However, paid apps usually require less permissions than their corresponding ads-supported versions. Yet, the validation of their licensing may require extra permissions which are not needed by their corresponding ads-supported versions. In addition, we found that developers do not always remove ad networks in paid versions of their ads-supported apps, which increases the apps sizes. If developers decide to include ads in their apps, they can set higher values for ads refresh rates to improve apps performance loading ads less often. But developers can also compare different advertising TPLs to select the most appropriate for their app users.

From our previous contributions summarized above, it appears that developers' decisions have an impact on the performance of mobile device apps. However, even if they make an effort to develop efficient apps, they ignore if their apps are, as least, as efficient as their competitors in a marketplace. Making app performance metrics available in marketplaces is useful for developers and users to compare apps in terms of performance. However, the selection of optimal apps is not trivial because of the cognitive effort imposed to discriminate between different apps and metrics. It is what we defined as the App Selection Problem: the search of optimal apps in terms of any combination of different metrics. To assist developers and users in comparing and choosing, respectively, optimal apps in terms of various conflicting performance metrics we proposed APOA. A multi-objective approach that can be used as a recommendation system for mobile device app marketplaces. We validated this approach

over a set of 140 popular Android apps. Among the 20 most popular Android browsers, we found that eight apps are more efficient than the others in terms of CPU, memory, and network usages. Thus, developers should consider these apps as references against which to compare with if they want to release a new and efficient browser app for emerging market users. But the search of optimal apps is also beneficial for users. We observed that choosing optimal apps instead of popular ones, users can extend battery life by up to 32 minutes, while decreasing CPU, memory, and network usages. APOA can search for optimal apps in one or more categories at the same time. Therefore, users can use our approach to search for optimal apps in a particular category, or they can search for optimal combinations of apps for multiple categories. Although APOA recommends optimal choices, users have the last word about apps to install in their phones.

10.2 Limits and Constraints

We used in our research a specific methodology to collect performance metrics of apps. Although developers could replicate our measurement environment, they need some knowledge to make the required connections to be able to collect energy measurements. However, they can use software based approaches instead of hardware based approaches. Once developers have a measurement environment available, they can collect performance metrics of their apps. Nonetheless, software based approaches rely on different sources of information to estimate energy consumption. Thus, imprecisions in those sources could affect the quality of the estimations.

We based our research on the fact that mobile devices have limited resources. Technology evolves fast and maybe in the future batteries will last longer, RAM memories may be in the order of Terabytes, or anyone has free access to Internet anywhere. Although the hardware of mobile devices has considerably improved in recent years, emerging market users own low-end devices and have limited access to data connections. In addition to hardware improvements, Google and iOS are constantly working and updating their operating systems to improve their performance. In the future, operating systems could be as efficient that developers' decisions do not have a significant impact on apps performance. However, even if mobile device operating systems are frequently updated to include performance improvements, these updates can introduce performance bugs. This happens often on the Android platform due to the Android fragmentation problem, because an operating system optimized for a specific device could not be efficient in other devices with different hardware.

As we obtained from our research, making performance metrics available in marketplaces is useful for developers and users. However, it is a big challenge. Android Vitals allows to

collect information about issues of apps to share it with their developers. However, Vitals does not collect performance metrics but rendering times, crash rates, frozen frames, or issues related to wake locks. In addition, Vitals does not share this information with users and neither among developers. Even if we consider that app performance metrics are available in marketplaces, the comparison among different apps must be done considering the same scenario. This fact opens a new challenge, the automatic generation of scenarios of usage to simulate user interactions with mobile device apps.

We intentionally focused our research on the Android operating system. We chose this platform because, today, it is the most popular mobile device operating system globally. However, our solutions are not intrinsically dependent on Android and most of them could also apply as well to iOS minus implementation details.

10.3 Threats to Validity

Threats to internal validity concern factors, internal to our research, that could have influenced our results. We computed performance metrics using well-known approaches. In addition, we replicated several times our measures to ensure statistical validity.

Threats to construct validity concern relationship between theory and observation and the extent to which the measures represent real values. We used a Nexus 4 phone, the same model used in previous research. We measured power usage using a sampling frequency one order of magnitude higher than past studies. Overall, our power usage measures were more precise or at least as precise as those in previous studies. However, with the physical measurement of energy consumption comes all of the limitations of measurement and experimentation that exist in the natural sciences and engineering (Hindle, 2016). CPU, memory, and network usages were collected using the commands `top` and `dumpsys`, and the tool `tcpdump` on the phone, respectively, which may have introduced extra energy consumption. To avoid any impact of other metrics' measurements on it, we collected power usage individually.

Threats to external validity concern the generalization of our findings. Our findings are based on the data collected, which were limited to the Nexus 4 phone and to the Android Lollipop. We chose this Android version because it introduced the ART, a new way of executing apps. The option to use ART has been available since Android 4.4 (KitKat), although KitKat users had a choice between ART and its predecessor Dalvik. However, from Android Lollipop on, ART is the only runtime environment. By targeting Android Lollipop and later apps will run on approximately 71.3% of the devices active on the Google Play Store (at of October, 2017). For this reason we think that our findings are valid for most of the active devices.

However, further validations on different marketplaces, larger set of apps, and–or different phones is desirable to make our findings more generic. There exist different factors (each one with several possible levels) to control, due to the fragmentation problem of Android. There are hundreds of different devices with different hardware configurations, more than 10 different versions of Android OS, and different ways of executing apps (Dalvik vs. ART). To complete the study, we should do a factorial experiment design taking into account all possible combinations of these levels across all factors, which would require more resources than usually available in research labs.

Threats to conclusion validity concern the relationship between experimentation and outcome. We conducted our research with real Android apps and we applied appropriate statistical procedures. We paid attention not to violate assumptions of the constructed statistical models. In particular, we used non-parametric tests that do not make assumptions on the underlying data distribution.

10.4 Future Work

EARMO generates sequence of refactoring operations to remove anti-patterns while reducing energy consumption. We intend to extend EARMO to detect and correct more mobile anti-patterns. We also plan to apply EARMO on larger datasets, and further evaluate it through user studies with mobile apps developers. Then, we want to create a new tool to apply these refactoring operations automatically.

Concerning the efficiency of data structures, we intend to study the impact of parameters capacity (of `HashMap`, `ArrayMap`, and `SparseArray` variants) and load factor (of `HashMap`) on performance metrics. We also want to assess the feasibility of proposing semi-automated refactoring tools to detect uses of `HashMap` and–or `ArrayMap` implementations replacing them by `SparseArray` variants. Applying static and–or dynamic analysis we can know more about the usage of map implementations and map-related operations in Android apps. Then, based on our guidelines, we can refactor the code to use a more adequate map implementation taking into account developers' preferences about the performance metrics to be improved. Another interesting point to explore is the use of our guidelines to dynamically adapt Android apps allowing them to decide which implementation to use depending on available resources (CPU speed, memory, and–or battery) or concerning users' preferences. We also want to adapt the implementation of `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` to improve their performance when deleting elements. The same keeps for `ArrayMap` and the deletion operation, which is highly inefficient in comparison to `HashMap`.

We plan to extend our catalog of TPLs performance including much more categories and new metrics such as app permissions and APK sizes. Once we extend our catalog, we could use tools as LibRadar and–or LibD to detect TPLs in Android apps and warn developers about the availability of more efficient choices. We also want to study the feasibility of a tool to automatically replace TPLs in Android apps with more efficient alternatives. Regarding advertising TPLs, we plan to study and compare the performance of existing Android licensing validation approaches, usually used by paid apps. We discovered that licensing validation approaches require extra permissions but we also suspect that these license controls also have an impact on the performance of apps. A catalog of existing license validation approaches and their impact on permissions and performance of apps would be useful for developers to help them make informed decisions.

Concerning APOA, we intend to extend our experiments over larger set of apps. Thus, we can evaluate APOA in a more realistic way using tens of categories and million apps. In that case the exhaustive search could not be feasible and the algorithm NSGAII could have problems solving instances of the ASP with more than three objective functions. Therefore, we should study more EMO algorithms and, in particular, we want to focus on preference-based EMO algorithms. They are methods that take into account user's preferences to approximate a region of the Pareto optimal front. Thus, users introduce their preferences about the desired improvement to be obtained for each performance metric and they obtain solutions satisfying these constraints. We also want to extend the ASP including more metrics. In addition to apps performance and rating, we also want to consider app permissions (safety) and their APK size. APK size is particularly important for emerging market users because they will pay for data as they use it and due to their mobile device disk space, which is lower than in developed markets. We also want to investigate how mobile device app developers could use inter-app comparison and performance measures to continuously evaluate their apps performance in the app store market. When integrated into continuous integration, the developers could get relative app ranking per each software change. By integrating comparison into continuous integration (continuous inspection) developers could maintain constant awareness of performance relevant issues their apps might face.

Unfortunately, part of our current and future research rely in the availability of performance metrics which are not publicly available. However, Google plans to collect information about issues of apps through Android Vitals and maybe, in a future, they also collect performance metrics. If we could make a wish to Google, it would be to make performance metrics of mobile device apps available in app marketplaces for both developers and users. This would be an essential step towards efficient software engineering for mobile device apps and a benefit for mobile device users who want to use efficient apps.

REFERENCES

Amsel, Nadine and Tomlinson, Bill (2010). Green tracker: A tool for estimating the energy consumption of software. *CHI '10 Extended Abstracts on Human Factors in Computing Systems*. ACM, New York, NY, USA, CHI EA '10, 3337–3342.

Banerjee, Abhijeet and Roychoudhury, Abhik (2016). Automated re-factoring of android apps to enhance energy-efficiency. *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, New York, NY, USA, MOBILESoft '16, 139–150.

Bansiya, Jagdish and Davis, Carl G. (2002). A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, *28*(1), 4–17.

Behrouz, R. J. and Sadeghi, A. and Garcia, J. and Malek, S. and Ammann, P. (2015). EcoDroid: An Approach for Energy-Based Ranking of Android Apps. *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. 8–14.

Book, Theodore and Pridgen, Adam and Wallach, Dan S. (2013). Longitudinal Analysis of Android Ad Library Permissions. *CoRR*, *abs/1303.0857*.

William J. Brown and Raphael C. Malveau and William H. Brown and Hays W. McCormick III and Thomas J. Mowbray (1998). *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons.

Carette, A. and Younes, M. A. A. and Hecht, G. and Moha, N. and Rouvoy, R. (2017). Investigating the energy impact of Android smells. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 115–126.

Cliff, Norman (2014). *Ordinal methods for behavioral data analysis*. Psychology Press.

C. A. C. Coello and G. B. Lamont and D.A.V. Veldhuizen (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems. Second Edition*. Springer, New York.

Cuervo, Eduardo and Balasubramanian, Aruna and Cho, Dae-ki and Wolman, Alec and Saroiu, Stefan and Chandra, Ranveer and Bahl, Paramvir (2010). MAUI: Making Smartphones Last Longer with Code Offload. *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, MobiSys '10, 49–62.

K. Deb (2001). *Multi-objective Optimization using Evolutionary Algorithms*. Wiley, Chichester.

K. Deb and A. Pratap and S. Agarwal and T. Meyarivan (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182–197.

Di Nucci, Dario and Palomba, Fabio and Prota, Antonio and Panichella, Annibale and Zaidman, Andy and De Lucia, Andrea (2017). Software-Based Energy Profiling of Android Apps: Simple, Efficient and Reliable? *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Klagenfurt, Austria, 103–114.

Juan J. Durillo and Antonio J. Nebro (2011). jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, *42*, 760–771.

Eick, Stephen G and Graves, Todd L and Karr, Alan F and Marron, J Steve and Mockus, Audris (2001). Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, *27*(1), 1–12.

Falaki, Hossein and Mahajan, Ratul and Kandula, Srikanth and Lymberopoulos, Dimitrios and Govindan, Ramesh and Estrin, Deborah (2010). Diversity in Smartphone Usage. *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, MobiSys '10, 179–194.

Field, Hayden and Anderson, Glen and Eder, Kerstin (2014). EACOF: A Framework for Providing Energy Transparency to enable Energy-Aware Software Development. *CoRR*, *abs/1406.0117*.

Martin Fowler (1999). *Refactoring – Improving the Design of Existing Code*. Addison-Wesley.

Gomes, Ludymila L. A. and Fontão, Awdren L. and Bezerra, Allan J. S. and Dias-Neto, Arilo C. (2016). An Empirical Analysis of Mobile Apps' Popularity Metrics in Mobile Software Ecosystems. *15th Brazilian Symposium on Software Quality*. Maceió (Brazil).

Gorla, Alessandra and Tavecchia, Ilaria and Gross, Florian and Zeller, Andreas (2014). Checking App Behavior Against App Descriptions. *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, ICSE 2014, 1025–1035.

Gottschalk, Marion (2013). *Energy Refactorings*. Mémoire de maîtrise, Carl von Ossietzky University.

Gottschalk, Marion and Jelschen, Jan and Winter, Andreas (2013). Energy-efficient code by refactoring. *Softwaretechnik Trends*. Gesellschaft für Informatik, Bonn, vol. 33, 23–24.

Guéhéneuc, Yann-Gaël (2005). Ptidej: Promoting patterns with patterns. *Proceedings of the 1st ECOOP workshop on Building a System using Patterns. Springer-Verlag*.

Gui, Jiaping and Mcilroy, Stu and Nagappan, Mei and Halfond, William G. J. (2015). Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers. *Proceedings of the 37th International Conference on Software Engineering (ICSE)*.

Harman, Mark and Jia, Yue and Zhang, Yuanyuan (2012a). App store mining and analysis: MSR for app stores. M. Lanza, M. D. Penta et T. Xie, éditeurs, *MSR*. IEEE Computer Society, 108–111.

Harman, Mark and Mansouri, S. Afshin and Zhang, Yuanyuan (2012b). Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, *45*(1), 11:1–11:61.

Hasan, Samir and King, Zachary and Hafiz, Munawar and Sayagh, Mohammed and Adams, Bram and Hindle, Abram (2016). Energy Profiles of Java Collections Classes. *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, TX, US, 225–236.

Hecht, Geoffrey and Moha, Naouel and Rouvoy, Romain (2016). An empirical study of the performance impacts of android code smells. *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, New York, NY, USA, MOBILESoft '16, 59–69.

Hecht, Geoffrey and Omar, Benomar and Rouvoy, Romain and Moha, Naouel and Duchien, Laurence (2015). Tracking the Software Quality of Android Applications along their Evolution. L. Grunske et M. Whalen, éditeurs, *30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Lincoln, Nebraska, United States, Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), 12.

Abram Hindle (2016). Green software engineering: The curse of methodology. *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER 2016, Osaka, Japan, March 14, 2016*. 46–55.

Myles Hollander and Douglas A. Wolfe (1973). *Nonparametric Statistical Methods*. John Wiley & Sons, New York.

Huang, Peng and Xu, Tianyin and Jin, Xinxin and Zhou, Yuanyuan (2016). DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. *Proceedings of the The 14th ACM International Conference on Mobile Systems, Applications, and Services*. Singapore, Singapore.

Hunt, N. and Sandhu, P. S. and Ceze, L. (2011). Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. *2011 15th Workshop on Interaction between Compilers and Computer Architectures*. 63–70.

C. L. Hwang and A. S. M. Masud (1979). *Multiple Objective Decision Making – Methods and Applications: A State-of-the-Art Survey*. Springer-Verlag, Berlin.

Kemp, Roelof and Palmer, Nicholas and Kielmann, Thilo and Bal, Henri (2012). Mobile Computing, Applications, and Services: Second International ICST Conference, MobiCASE 2010, Santa Clara, CA, USA, October 25-28, 2010, Revised Selected Papers. Springer Berlin Heidelberg, Berlin, Heidelberg. 59–79.

Khan, Azeem J. and Subbaraju, Vigneshwaran and Misra, Archan and Seshan, Srinivasan (2012). Mitigating the true cost of advertisement-supported "free" mobile applications. G. Borriello et R. K. Balan, éditeurs, *HotMobile*. ACM, 1.

Li, Ding and Halfond, William G. J. (2014a). An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM, New York, NY, USA, GREENS 2014, 46–53.

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

Li, Ding and Hao, Shuai and Gui, Jiaping and Halfond, William G. J. (2014a). An Empirical Study of the Energy Consumption of Android Applications. *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*.

Li, Ding and Jin, Yuchen and Sahin, Cagri and Clause, James and Halfond, William G. J. (2014b). Integrated Energy-Directed Test Suite Optimization. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

Li, Ding and Tran, Angelica Huyen and Halfond, William G. J. (2014c). Making Web Applications More Energy Efficient for OLED Smartphones. *Proceedings of the International Conference on Software Engineering (ICSE)*.

Li, Menghao and Wang, Wei and Wang, Pei and Wang, Shuai and Wu, Dinghao and Liu, Jian and Xue, Rui and Huo, Wei (2017). LibD: scalable and precise third-party library detection in android markets. *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 335–346.

Linares-Vásquez, Mario and Bavota, Gabriele and Bernal-Cárdenas, Carlos and Oliveto, Rocco and Di Penta, Massimiliano and Poshyvanyk, Denys (2014). Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, New York, NY, USA, MSR 2014, 2–11.

Linares-Vásquez, Mario and Bavota, Gabriele and Cárdenas, Carlos Eduardo Bernal and Oliveto, Rocco and Di Penta, Massimiliano and Poshyvanyk, Denys (2015). Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach. *Proceedings*

of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, ESEC/FSE 2015, 143–154.

Liu, Kenan and Pinto, Gustavo and Liu, Yu David (2015). Data-Oriented Characterization of Application-Level Energy Optimization. A. Egyed et I. Schaefer, éditeurs, *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg. 316–331. DOI: 10.1007/978-3-662-46675-9_21.

Lyu, Yingjun and Gui, Jiaping and Wan, Mian and Halfond, William G. J. (2017). An Empirical Study of Local Database Usage in Android Applications. *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*.

Ma, Ziang and Wang, Haoyu and Guo, Yao and Chen, Xiangqun (2016). LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, New York, NY, USA, ICSE '16, 653–656.

Manotas, Irene and Pollock, Lori and Clause, James (2014). SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, ICSE 2014, 503–514.

Marinescu, Radu (2004). Detection strategies: Metrics-based rules for detecting design flaws. *IEEE Int'l Conference on Software Maintenance, ICSM*. IEEE Computer Society, 350–359.

K. Miettinen (1999). *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, Boston.

Minelli, Roberto and Lanza, Michele (2013). Software Analytics for Mobile Applications-Insights & Lessons Learned. *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. 144–153.

Moha, Naouel and Gueheneuc, Yann-Gael and Duchien, Laurence and Le Meur, A (2010). Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, *36*(1), 20–36.

Mohan, Jayashree and Purohith, Dhathri and Halpern, Matthew and Chidambaram, Vijay and Reddi, Vijay Janapa (2017). Storage on Your SmartPhone Uses More Energy Than You Think. *9th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2017, Santa Clara, CA, USA, July 10-11, 2017*.

Morales, Rodrigo and Sabane, Aminata and Musavi, Pooya and Khomh, Foutse and Chicano, Francisco. and Antoniol, Giuliano. (2016). Finding the best compromise between

design quality and testing effort during refactoring. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. vol. 1, 24–35.

R. Morales and R. Saborido and F. Khomh and F. Chicano and G. Antoniol (2017a). EARMO: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, *PP*(99), 1–1.

Morales, Rodrigo and Soh, Zephyrin and Khomh, Foutse and Antoniol, Giuliano and Chicano, Francisco (2017b). On the use of developers' context for automatic refactoring of software anti-patterns. *Journal of Systems and Software*, *128*, 236 – 251.

Antonio J. Nebro and Juan J. Durillo and Francisco Luna and Bernabé Dorronsoro and Enrique Alba (2007). MOCell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, 25–36.

Tam The Nguyen and Hung Viet Pham and Phong Minh Vu and Tung Thanh Nguyen (2015). Recommending API Usages for Mobile Apps with Hidden Markov Model. *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, *00*, 795–800.

Nguyen, Tam The and Pham, Hung Viet and Vu, Phong Minh and Nguyen, Tung Thanh (2016). Learning API Usages from Bytecode: A Statistical Approach. *Proceedings of the 38th International Conference on Software Engineering*. ACM, New York, NY, USA, ICSE '16, 416–427.

Niu, Haoran and Keivanloo, Iman and Zou, Ying (2017). API usage pattern recommendation for software development. *Journal of Systems and Software*, *129*, 127 – 139.

O'Keeffe, M. and Cinnéide, M. O. (2006). Search-based software maintenance. *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. 10 pp.–260.

Opdyke, William F (1992). *Refactoring object-oriented frameworks*. Thèse de doctorat, University of Illinois at Urbana-Champaign.

Ouni, Ali and Kessentini, Marouane and Sahraoui, Houari and Boukadoum, Mounir (2013). Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, *20*(1), 47–79.

Ali Ouni and Marouane Kessentini and Houari Sahraoui and Katsuro Inoue and Mohamed Salah Hamdi (2015). Improving multi-objective code-smells correction using development history. *Journal of Systems and Software*, *105*(0), 18 – 39.

Pamboris, Andreas and Antoniou, George and Makris, Constantinos and Andreou, Panayiotis and Samaras, George (2016). AD-APT: Blurring the Boundary Between Mobile Advertising and User Satisfaction. *IEEE/ACM International Conference on Mobile Software*

Engineering and Systems (MOBILESoft'16). IEEE, Austin, Texas, United States, vol. 1 de *Proceedings of the 3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems*.

Jae Jin Park and Jang-Eui Hong and Sang-Ho Lee (2014). Investigation for software power consumption of code refactoring techniques. *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.* 717–722.

David Lorge Parnas (1994). Software aging. *ICSE '94: Proc. of the 16th Int'l conference on Software engineering*. IEEE Computer Society Press, 279–287.

Rasmussen, Kent and Wilson, Alex and Hindle, Abram (2014). Green Mining: Energy Consumption of Advertisement Blocking Methods. *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM, New York, NY, USA, GREENS 2014, 38–45.

Reimann, Jan and Brylski, Martin and Aßmann, Uwe (2014). A tool-supported quality smell catalogue for android developers. *Softwaretechnik-Trends*, *34*(2).

Romano, Jeanine and Kromrey, Jeffrey D and Coraggio, Jesse and Skowronek, Jeff and Devine, Linda (2006). Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. *annual meeting of the Southern Association for Institutional Research*.

Israel J. Mojica Ruiz and Meiyappan Nagappan and Bram Adams and Thorsten Berger and Steffen Dienst and Ahmed E. Hassan (2014). Impact of Ad Libraries on Ratings of Android Mobile Apps. *IEEE Software*, *31*(6), 86–92.

Saborido, Rubén and Arnaoudova, Venera and Beltrame, Giovanni and Khomh, Foutse and Antoniol, Giuliano (2015). On the impact of sampling frequency on software energy measurements. *PeerJ PrePrints*, *3*, e1219.

Saborido, Ruben and Beltrame, Giovanni and Khomh, Foutse and Alba, Enrique and Antoniol, Giulio (2016). Optimizing User Experience in Choosing Android Applications. *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Osaka (Japan), 438–448.

Saborido, Ruben and Khomh, Foutse (2017). Helping Android Users to Find the Most Efficient Apps. *Scientific Program of the 24th International Conference on Multiple Criteria Decision Making (MCDM)*. Ottawa (Canada), 112.

Saborido, Rubén and Khomh, Foutse and Antoniol, Giuliano and Guéhéneuc, Yann-Gaël (2017). Comprehension of Ads-supported and Paid Android Applications: Are They Differ-

ent? *Proceedings of the 25th International Conference on Program Comprehension (ICPC)*. IEEE, Buenos Aires, Argentina, 143–153.

Sahin, Cagri and Pollock, Lori and Clause, James (2016). From Benchmarks to Real Apps: Exploring the Energy Impacts of Performance-directed Changes. *Journal of Systems and Software*, –.

Cagri Sahin and Lori L. Pollock and James Clause (2014a). How do code refactorings affect energy usage? *International Symposium on Empirical Software Engineering and Measurement, ESEM*. 36:1–36:10.

Sahin, Cagri and Tornquist, Philip and Mckenna, Ryan and Pearson, Zachary and Clause, James (2014b). How Does Code Obfuscation Impact Energy Usage? *ICSME'14*. 131–140.

Seng, O. and Stammel, J. and Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. *GECCO 2006: Genetic and Evolutionary Computation Conference, Vol 1 and 2*, 1909–1916.

Shacham, Ohad and Vechev, Martin and Yahav, Eran (2009). Chameleon: Adaptive Selection of Collections. *SIGPLAN Not.*, *44*(6), 408–418.

Singer, Janice and Sim, Susan E and Lethbridge, Timothy C (2008). Software engineering data collection for field studies. *Guide to Advanced Empirical Software Engineering*, Springer. 9–34.

Tsantalis, Nikolaos and Chaikalis, Theodoros and Chatzigeorgiou, Alexander (2008). JDeodorant: Identification and removal of type-checking bad smells. *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 329–331.

Tyagi, Pradeep K. (1989). The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science*, *17*(3), 235–241.

Vallina-Rodriguez, Narseo and Shah, Jay and Finamore, Alessandro and Grunenberger, Yan and Papagiannaki, Konstantina and Haddadi, Hamed and Crowcroft, Jon (2012). Breaking for Commercials: Characterizing Mobile Advertising. *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*. ACM, New York, NY, USA, IMC '12, 343–356.

Wan, Mian and Jin, Yuchen and Li, Ding and Halfond, William G. J. (2015). Detecting Display Energy Hotspots in Android Apps. *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

Wang, Haoyu and Guo, Yao and Ma, Ziang and Chen, Xiangqun (2015). WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. *Proceedings*

of the 2015 International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, ISSTA 2015, 71–82.

Wei, Xuetao and Gomez, Lorenzo and Neamtiu, Iulian and Faloutsos, Michalis (2012). ProfileDroid: Multi-layer Profiling of Android Applications. *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*. ACM, New York, NY, USA, Mobicom '12, 137–148.

Zhang, Chenlei and Hindle, Abram and German, Daniel M. (2014). The Impact of User Choice on Energy Consumption. *IEEE Software*, *31*(3), 69–75.

Zitzler, Eckart and Laumanns, Marco and Thiele, Lothar and Zitzler, Eckart and Zitzler, Eckart and Thiele, Lothar and Thiele, Lothar (2001). SPEA2: Improving the strength pareto evolutionary algorithm.

E. Zitzler and L. Thiele (1999a). Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, *3*(4), 257–271.

Zitzler, Eckart and Thiele, Lothar (1999b). Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *evolutionary computation, IEEE transactions on*, *3*(4), 257–271.