# Cost-effective Strategies to Develop Energy-Efficient Mobile Applications

by

Abdul Ali Bangash

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

Doctor of Philosophy

Department of Computing Science

University of Alberta

# Abstract

Smartphone users rely on mobile applications (apps) to perform various functionalities. However, if an app is developed inefficiently, such that it over-consumes energy, it could negatively impact user experience and lead to poor user reviews. To ensure that an app does not consume energy unnecessarily, app developers measure and optimize the energy consumption of their apps before releasing them to the end users.

However, the current energy optimization and measurement techniques have certain limitations. API events consume $85\%$ of the energy in an app [110], yet the current optimization techniques focus on developer-written instructions and system events only. Moreover, each API configuration may consume different amounts of energy, yet none of the current techniques guide developers on how a specific configuration may affect the energy consumption of their app. On the other hand, the current measurement techniques are cumbersome and require developers to generate test cases and execute them on sophisticated hardware. Hardware is expensive, test-case generation requires regular maintenance, and test-case execution costs time. Therefore, it is impractical for the developers to execute test cases after each code modification, such as declaring a new variable or adding a new method call.

As a solution to the aforementioned problems, this thesis explores a new direction of using static-analysis to optimize and estimate smartphone-app energy consumption, making four key contributions. The first contribution evaluates the practicality of current state-of-the-art techniques, such as search-based energy optimization and test-case execution-based energy measurement. The second contribution introduces an open-source hardware-based energy-measurement framework for iOS applications. The third contribution focuses on providing guidelines on how to configure and use an API in an energy-efficient manner, specifically the Core Location Framework API. The fourth and final contribution provides an energy-estimation model that helps identify energy-inefficient API usage in code without test case execution. The proposed energy-efficient guidelines reduce the energy consumption of real-world iOS apps by $26.91\%$. Additionally, the energy estimation model can estimate energy consumption and provide information on energy-inefficient usage at an app's version and method-level within $20\%$ mean absolute error.

The techniques presented in this thesis are helpful when developers do not have test cases readily available or hardware to run them on. Moreover, this thesis is particularly useful in an Integrated Development Environment (IDE) or a Continuous Integration/Continuous Deployment (CI/CD) pipeline. In such scenarios, developers cannot wait for test-case execution after each code modification and may require energy insights in real-time.

*Teaching and Learning. Purification and Correction. Preachment and Rectification. Service of Creation.*

– Mufti Syed Mukhtaruddin Shah DB.

# Acknowledgements

I am eternally grateful to Allah for all the blessings, and particularly for the wonderful people Allah placed in my life, including those mentioned below, who supported me throughout my Ph.D. journey.

I thank Mufti Syed Adnan Kakakhail, my spiritual mentor, for his inspiration and support throughout my Ph.D. journey. He has imparted valuable lessons on patience, perseverance, truthfulness, honesty, and humbleness, which helped me to overcome my faults such as ego, pride, arrogance, and selfishness. I am deeply grateful for his prayers and affection, which sustained me during my pursuit of knowledge.

I would like to express my gratitude to Dr. Karim Ali and Dr. Abram Hindle, my Ph.D. advisors, for their patience and guidance throughout my learning journey. They have not only helped me to enhance my research capabilities but also to refine my interpersonal skills.

I extend my sincere appreciation to my family, particularly my wife, for their unwavering support throughout my Ph.D. journey. I am also grateful to my parents for their patience and understanding, allowing me to continue my studies even when I was needed at home. I also express my gratitude for their constant support and encouragement through their prayers.

I am grateful to my teachers Mufti Imran Farooq and Mufti Faisal Al Mahmudi for their guidance and support during the demanding period of my Ph.D. I also wish to express my appreciation to my friends Rafsanjany Kushol, Haseeb Munawwar, and Mubashir Tamboli for their invaluable assistance which made my Ph.D. journey more manageable. My dear friend Mazhar Shahzad's tireless efforts in taking care of things back home while I was studying abroad will always be remembered and appreciated.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The advancement of technology has made it possible for people to access a wide range of personal and business services through software applications (i.e., apps) that run on smartphones. These apps offer a range of functionalities, such as GPS navigation, bluetooth connectivity, camera usage, and processing power from the CPU and GPU. However, when an app does not use these hardware components efficiently, it can lead to excessive energy consumption and quickly make the phone run out of battery. Excessive energy consumption of an app concerns app developers because it results in a poor user experience, negative reviews, and even the removal of their app from the smartphone [103], [142]. While hardware engineers and researchers are working to enhance the energy efficiency of hardware components [108], [128], [161], app developers are exploring software-based solutions to identify energy bottlenecks in their code and reduce energy consumption. However, the current software-based energy measurement and optimization solutions have several limitations:

**Optimization techniques and their limitations.** The current optimization techniques provide app developers with design patterns to develop energy-efficient code structure [55], user-interface (UI) guidelines to have energy-efficient design aesthetics [116], refactoring techniques to remove energy-greedy patterns from code [30], and API-guidelines [83] to choose an energy-efficient API in code.

According to studies such as Li et al. [110], API events in an app typically account for $85\%$ of its energy consumption. Therefore, measuring the energy consumption of these API events is generally considered sufficient for estimating the overall energy consumption of the app [97].

However, most of the aforementioned techniques focus on developer-written instructions (e.g., branch or arithmetic instructions) and system events (e.g., garbage collection or process switching), instead of API-level usage [110].

A few state-of-the-art techniques that focus on API-level energy consumption include a genetic search-based technique by Manotas et al. [120] that helps developers optimize API usage in an app. This technique is time-consuming because it works on the principles of genetic search, where a developer has to execute all test cases multiple times to evaluate several generations of their mutated code until they find the best version of their code. Furthermore, Hassan et al. [83] provide guidelines for developers to choose an energy-efficient collection from the Java Collections API. Although this approach helps developers make an informed decision on the class level of an API, it still does not provide information on how specific API parameters may affect energy consumption. Varying parameter configurations affect the API performance and energy consumption differently [110], yet none of the current approaches guide developers on how a specific configuration may affect an app's energy consumption and if they could configure API parameters to reduce energy consumption. For instance, suppose a developer uses the iOS Core Location API (corelocation-API) to access user-location in their app. This API offers several tuning parameters, such as `desiredAccuracy` and `distanceFilter`. The current approaches do not guide developers on how these underlying parameters could impact their app's overall energy consumption.

To address this limitation, this thesis aims to provide API parameter-level energy-efficient guidelines for developers to help them identify, avoid, and fix energy inefficiencies in their code.

**Measurement techniques and their limitations.** The current measurement techniques provide developers: with hardware-based measurement tools [87], [89] and software-based estimation models [67]. However, hardware-based techniques demand developers to establish a sophisticated measurement setup, a process that requires specialized knowledge and is therefore considered a hindrance by many developers [142]. Furthermore, acquiring the necessary hardware is costly, adding a financial burden. As a cost-effective alternative, developers use software-based techniques that rely on estimation [67], [81] and prediction [53], [106] models. However, these techniques require developers to generate and execute test cases, a process that consumes significant resources such as computing time, power, and storage [46]. Moreover, test-case generation is a difficult task because developers are required to generate test cases that cover most of an app's functionalities [95], [96]. Additionally, as software evolves, developers may need additional or updated test cases [124], [126], which may not always be readily available. After generation, execution of these test cases requires a smartphone and a script to collect the app's runtime information. In short, generating and executing test cases is a time-consuming process. A state-of-the-art technique by Jabbarvand et al. [95] takes an average of 8 minutes to generate and execute test cases. This lengthy duration means that a developer who is writing code in an Integrated Development Environment (IDE) and expecting real-time updates will have to endure an impractical wait time after each code modification.

To address these limitations, this thesis aims to use static analysis so that the energy consumption of an app can be estimated within milliseconds, without the need for generating or executing test cases.

**Thesis statement.** Static analysis can be used to optimize and measure the energy consumption of API usage in an app. This use of static analysis relieves developers from the effort of generating and executing test cases to obtain energy insights into their code.

## 1.1 Structure of the Thesis and Underlying Contributions

This thesis proposes, implements, and evaluates energy optimization and energy estimation approaches that work at the API usage level. After laying out the motivation in Chapter 1, this thesis outlines the necessary background concepts in Chapter 2. The thesis then presents four distinct contributions, with the relevant related work discussed at the end of each chapter.

To determine the feasibility of integrating current energy measurement and optimization techniques into developers' existing development tools, as its first contribution, this thesis conducts a preliminary study in Chapter 3. This chapter evaluates the state-of-the-art by adopting the practices of test case execution using search-based software engineering [120] for energy optimization. The results in this chapter lay the foundation of this thesis and explain why a static-analysis-based energy optimization and estimation approach is required to help energy-aware developers.

As its second contribution, in Chapter 4, the thesis presents an open-source hardware-based iOS apps energy measurement framework, iGreenMiner. The purpose of iGreenMiner is to obtain the hardware-based energy measurements of real-world apps. This thesis later uses these energy measurements as ground truth to validate the results of its contributions. iGreenMiner is the first-of-its-kind framework because it is publicly available to the iOS community through a web service, and its execution scripts are also open-source. iOS app developers can freely upload their apps to this web service and measure their energy consumption.

As its third contribution, in Chapter 5, this thesis proposes energy-aware configuration guidelines for API parameters to reduce an app's energy consumption. Specifically, this chapter extracts energy-aware guidelines for the iOS Core Location framework. The results in this chapter guide the developers to make an informed, energy-efficient design choice before runtime for accessing user location in an app. Furthermore, app developers may use the proposed approach to develop energy-aware guidelines for other utility APIs such as Core Graphics, Core Animation, Core Bluetooth, etc.

As its fourth contribution in Chapter 6, this thesis proposes and validates a static-analysis-based estimation model that estimates the energy consumption, or E-factor, of API usage in an app. Energy-aware developers may use this model to estimate energy consumption without a hardware setup, test-case generation, or execution. We evaluate the model on 16 versions and 11 methods of 3 real-world iOS apps and find that the estimated values of E-factor have a strong positive correlation within the hardware-based energy measurements. This result means that E-factor is a good estimator to compare the relative energy consumption of API use between the versions of an app and within the methods of an app. Furthermore, the proposed estimation approach is $10^6$ times faster than the state-of-the-art. Therefore, developers can integrate E-factor in a CI/CD pipeline and development IDEs to receive energy consumption warnings within milliseconds.

Chapter 7 summarizes the benefits of adopting a static-analysis approach for energy-efficient development and proposes potential avenues for future research, emphasizing the significance of creating methods and tools to estimate and optimize the energy consumption of apps in real-time.

## 1.2 Publication Details

During my Ph.D., I was the main author to 5 peer-reviewed publications, including 4 conference papers (at MSR-Challenge2019, ICSME2019, ICSE-NIER2022, MSR2023) and 1 journal article (at EMSE2019), under the supervision of Dr. Abram Hindle and Dr. Karim Ali. I also co-authored 2 additional conference papers (at MSR-Challenge2023).

This thesis is primarily based on 3 conference papers, for which I designed the methodology, conducted experiments, analyzed the results, and wrote the articles. Dr. Abram Hindle and Dr. Karim Ali supervised all the experiments and provided feedback on methodology and presentation of the work. In the third paper [34], Qasim Jamal helped in manual executions of the benchmarks to collect their energy measurements and Kalvin Eng helped in collecting project versions from various version-control repositories.

Chapter 3 of this thesis has been published as:

- Abdul Ali Bangash, Karim Ali, Abram Hindle, "A Black Box Technique to Reduce Energy Consumption of Android Apps." Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER, short paper). ACM, [33], 2022.

Chapter 5 of this thesis has been published as:

- Abdul Ali Bangash, Karim Ali, Abram Hindle, "Energy Efficient Guidelines for iOS Core Location Framework." 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME, technical track). IEEE, [37], 2021.

Chapter 6 of this thesis has been published as:

- Abdul Ali Bangash, Qasim Jamal, Kalvin Eng, Karim Ali, Abram Hindle "Energy Consumption Estimation of API-usage in Smartphone Apps via Static Analysis." 2023 IEEE International Conference on Mining Software Repositories (MSR, technical track). IEEE, [34], 2023.

The following is a list of peer-reviewed publications (including papers I co-authored) during my Ph.D. but have not included in this thesis.

- Abdul Ali Bangash, *et al.* “On the time-based conclusion stability of cross-project defect prediction models.” Empirical Software Engineering 25.6 (EMSE), [36], 2023.

- Abdul Ali Bangash, *et al.* “What do developers know about machine learning: a study of ml discussions on StackOverflow.” 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, [35], 2019.

- Anisha Islam, *et al.* “Evolution of the Practice of Software Testing in Java Projects.” 20th International Conference on Mining Software Repositories, [94], 2023.

- Weijie Sun, *et al.* “An Empirical Study to Investigate Collaboration Among Developers in Open Source Software (OSS).” 20th International Conference on Mining Software Repositories, [160], 2023.

# Chapter 2

# Background

The main theme of this thesis is to use *static analysis* to analyze the *energy consumption* of *API usage* in *smartphone apps* so that a developer does not have to generate and execute test cases to collect *runtime information*. This technique is expected to generate energy insights quickly so that developers can integrate it into their *IDE* or *CI/CD pipeline*. This chapter explains all the technical terminologies used in this paragraph.

This chapter first explains what a smartphone app is, what an API is, and what API usage is in the context of an app (Section 2.1). Second, it explains how apps consume energy from a smartphone (Section 2.2), how API use energy in an app (Section 2.3), and how energy consumption varies by app's versions (Section 2.4). Third, it explains what an IDE means, what a CI/CD pipeline is, and how energy insights are useful in both of these environments (Section 2.5). Finally, this chapter explains static analysis and how we use it to generate call graphs of the code to identify methods that use an API. Moreover, we explain why current static-analysis approaches that reason about runtime can not be used as a proxy for energy consumption (Section 2.6). Additional background information on topics relevant to a specific chapter, such as the Core Location API and the SQLite API, is provided in the corresponding chapters.

## 2.1 Smartphone Apps and API Usage

**Smartphone apps:** A smartphone app is a software application that developers write to help users perform specific functions on their smartphones through a UI. Depending on the functionalities that an app offers, users may perform a wide range of tasks, from essential functions such as making phone calls or sending text messages to more complicated activities like playing games, organizing and managing personal information, and accessing news, weather, and other useful information [172]. Smartphone users can download these apps from app stores, such as the Apple App Store [21] or the Google Play Store [78], and developers can download the source code of these apps from public software version control repositories unless the code is under proprietorship.

**Application Programming Interface (API):** An application programming interface (API) is a piece of code that provides app developers with a well-defined interface to an application, library, data structure, or a smartphone hardware component [44]. For example, a developer may use a networking API to add a network connection facility in their app. Similarly, developers may use a Bluetooth API to use the Bluetooth component in their app, a cloud-storage API to manage the storage of their app on a cloud service, or a collections API to maintain data structures in their code.

**API usage in apps:** Each API comes with a set of methods and parameters to configure. App developers may configure the API parameters before invoking a method to complete a task. For example, the iOS Core location API provides functionality to fetch user location via GPS and network components. The iOS Core location API provides methods to invoke a location fetching service and provides parameters, such as a `distanceFilter` parameter, to configure the service. The `distanceFilter` specifies the distance a user may travel before the service would fetch their latest location. Suppose an app developer sets the value of `distanceFilter` short before invoking the service, the service will fetch the user location after every little movement. However, if the app developer uses a considerable value for `distanceFilter`, the service will fetch the user location only after each significant user movement. The point to consider here is that each parameter configuration may exploit the hardware components, in this case, GPS and WiFi, differently.

## 2.2 Energy Consumption of an App

Smartphone apps use various hardware components to perform various functions, such as GPS for navigation, memory for computation, and camera for taking pictures. These components require power in watts (W) in order to operate properly. Power is a measure of how much energy is used per unit time and it is calculated by the product of current and voltage. Current represents the flow of electrons and is expressed in Amperes (A), while voltage represents the direction and strength of the current and is expressed in Volts (V) [127]. To control the flow of electrons (current) through the logic circuits of the components, the smartphone's transistors and semiconductors switch the current on and off. At the same time, the smartphone's battery regulates the direction and strength of the current (voltage) [162].

**Calculating energy from power:** Power is typically used in the context of long-running systems, such as power plants or home appliances, because these systems are operated continuously over a long period. For short-running tasks such as smartphone apps, energy is used to describe their consumption [85], since the duration of the task can be easily tracked.

For example, consider a smartphone camera that requires a voltage of 3V and a current of 2A. The power required to operate this component can be calculated as follows:

$$Watts(W) = ApparentPower(VA) \times PowerFactor(PF)$$

The power factor represents the efficiency of electrical power usage by indicating the phase relationship between voltage and current in an electrical system. In our case, the power factor is 1 because smartphone components operate on direct current (DC), whereas $VA = Volt(V) \times Current(I)$. Therefore, the smartphone camera would require 6W to operate.

If a smartphone app's test case utilizes this camera for a duration of 2 seconds, the energy consumed by the app can be determined using the formula [127]:

$$Energy(Joules) = Watts(W) \times Time(seconds)$$

In our case, its 6W and 2 seconds, and consequently the energy consumption required to use the camera for a duration of 2 seconds will be 12 Joules.

**Every app consumes different amounts of energy:** In a smartphone app, the amount of energy consumption may vary depending on its complexity, the hardware component it uses, and the size of the data it processes [167]. For instance, a simple weather app that retrieves and displays weather information may consume less energy over a more extended period compared to a complex 3D game that renders and animates complex graphics, even if the latter is run for a shorter period.

## 2.3 Energy Consumption of API Usage

Every API provides different functionality, such as storing data structures in memory or accessing the user's location through GPS. As a result, each API may consume different amounts of energy in an app. For example, consider an app that uses the SQLite API [88] for data persistence and the Core Location API [20] to access user location. Depending on the usage of these APIs within the app's code, each API may consume different amounts of energy. For instance, if the SQLite API is used multiple times within a loop structure, it may consume more energy than the Core Location API used outside a loop. We compare the energy consumption between the different tasks of an API, with examples in Chapter 6.

Additionally, the energy consumption of a single API can vary within an app's code based on its configuration. For instance, if a developer configures the Core Location API to retrieve location with high accuracy, even when it is not required, the API may consume more energy than needed. We discuss the impacts of different API configurations on energy consumption, along with examples in Chapter 5.

## 2.4 Energy Regression Testing of Apps

Due to code modifications, each version of an app may consume varying amounts of energy. Observing the impact of code modifications on energy consumption to alert developers of potentially costly changes is referred to as energy regression testing [147], [176]. This testing provides a relative comparison between two versions of an app and plays a crucial role in software development. It helps developers ensure a better user experience and satisfaction by mitigating the risk of introducing an energy-intensive update in a newer release of an app.

The current energy regression testing techniques face the challenges of test case generation and execution, requiring developers to dedicate effort, cost, and time to obtain insights into energy consumption of their new release. This effort involves test case generation, cost involves owning smartphones for running test cases, and time is consumed in test case execution (several hours). This thesis addresses these current challenges by providing developers with quick insights (within milliseconds) into the impact of their code modifications on energy consumption, eliminating the need for test case generation, execution, and hardware ownership. The approach in this thesis achieves this by combining static analysis and energy modeling techniques to analyze app's energy consumption without any runtime information. By integrating our approach into the software development process, particularly within a continuous integration pipeline, developers can utilize the contributions of this thesis to manage energy consumption and ensure overall software energy efficiency.

## 2.5 Energy Consumption in the Context of IDEs and CI/CD

**Integrated Development Environment (IDE):** An Integrated Development Environment (IDE) is a software application that provides a comprehensive development environment for app developers. It includes an editor for writing, debugging, and compiling code and features such as code completion and error highlighting that provide insights and speed up the development process. Popular IDEs for smartphone app development include Xcode [22] for iOS app development and Android Studio [77] for Android app development.

**Continuous Integration and Continuous Deployment (CI/CD):** CI/CD is a software development practice that regularly integrates code modifications of a project into its codebase. The CI/CD pipeline helps developers automate the process of building, testing, and releasing new versions of their apps. This process can speed up development, reduce errors, and improve the app's quality by catching bugs and performance issues before they make it to the end user. CI/CD provides several benefits, including the faster release of versions to the end user, increased collaboration among developers, and insights into code's status in terms of quality and performance.

**Energy consumption insights in IDE and CI/CD:** During app development in an IDE, if developers receive real-time feedback on the energy consumption of their code, they can save significant amounts of time. Similarly, if the CI/CD pipeline provides energy consumption insights, developers can quickly identify any energy bottlenecks in their app and take action to rectify them. Consider a scenario as an example where a developer is continuously making code modifications in an IDE or sending pull-requests to a version-control system's CI/CD pipeline. In such a scenario, real-time energy insights are most relevant because they can quickly inform the developer about possible energy-inefficient code modifications.

Some energy and runtime profilers that come with an IDE include Android Studio's Android Profiler [79], XCode's Log Instrument Profiler [92], Visual Studio's Performance Profiler [125], and JProfiler [70]. The Android Profiler provides real-time insights about an app's CPU, memory, network usage, and battery consumption [79]. The Energy Log Instrument allows measuring an app's energy consumption while running on an iOS device [92]. The Performance Profiler provides real-time information about an app's CPU, memory, network, and disk I/O usage [125]. JProfiler is a profiling tool for Java applications. It provides real-time information about various performance metrics, including energy consumption. It can be used to identify performance bottlenecks and memory leaks in applications [70]. However, all of these tools are limited in requiring runtime information of an app to provide insights about its energy consumption, while this thesis aims to provide the app-developers energy insights without any runtime information.

## 2.6 Static Analysis of Smartphone Apps

Static analysis, in the context of smartphone app development, refers to examining and analyzing an app's code without executing it. This practice allows developers to uncover the semantic properties of the app, which are crucial to ensuring its proper functioning [146]. Examples of these properties include correctness (i.e., whether the program produces the correct output), termination (i.e., whether the program terminates correctly), and reusability (i.e., whether the program can be used inside other programs).

Researchers and developers initially used static analysis to optimize compilers and generate efficient machine code. However, over time, it has become an integral part of various phases of app development, including security verification, evaluation of code quality, and maintenance and refactoring of code [130].

In this thesis, we employ several static analysis techniques, including byte-code transformations and call graph analysis, to analyze the energy consumption of mobile applications. These analyses operate at different granularity levels, allowing us to optimize energy consumption and comprehensively understand an app's energy consumption. To optimize the app's byte-code, we use byte-code level transformations [71] to relieve the developers from code modification. Additionally, we utilize call graph analysis [150] to analyze the app code's API usage and configuration in terms of energy consumption. At a fine-grained level, we analyze the app's code to detect potential energy-consuming patterns, such as inefficient API usage, excessive loops that use APIs, and unnecessary computations. By scrutinizing these fine-grained details, we can identify specific energy-consuming operations, enabling developers to optimize critical code segments that significantly impact their apps' energy consumption.

**Intermediate Representation (IR):** IR is an intermediate form of a program that the compiler generates during the compilation process. IR provides a platform-independent representation of the source code, which allows compilers to perform various transformations and optimizations before generating the final machine code. IR is an integral part of the compiler architecture, as it bridges the source code and target machine code [165]. An example of IR is Swift Intermediate Language (SIL) [27], a high-level representation of the semantics of the Swift language program. The Swift language compiler generates SIL from the source code and then uses it to perform optimizations and transformations into machine code.

**Call Graph (CG) from IR:** A CG is a graphical representation of a program's control flow. It shows the relationships between functions in the program and how they call each other [2]. A CG may be generated from the control flow information present in the IR of a program. The process of generating a CG from an IR involves identifying the function and subroutine calls in the IR and using that information to build a graph [150]. Developers may use this graph for various purposes, such as identifying performance bottlenecks and security vulnerabilities.

## 2.7 Performance Testing and Energy Consumption

In this section, we discuss the relevant literature on performance testing and its relationship to this thesis. Performance testing is a technique used to assess a system's stability and response time by subjecting it to various system workloads. Danciu et al. [63] conducted a survey that highlights the common areas of research in performance testing, including network optimization, architecture analysis, response time evaluation, execution time measurement, data structure optimization, power consumption analysis, database performance, and energy consumption analysis. We have extensively covered the literature related to energy consumption in the related work section of each chapter in this thesis. Therefore, in this section, we focus on the remaining performance estimation and optimization aspects.

Carvalho et al.[64] build upon the notion that hand-written APIs outperform auto-generated code in terms of execution speed. They develop a tool that automates the replacement of application code with API calls known for their speed efficiency. This approach aligns with our work in Chapter 5, although there is a slight difference in focus. Instead of replacing energy-consuming code with better API calls, our objective is to identify energy-efficient configurations of an API itself.

In order to enhance the throughput and latency of various workloads in an application, Berube et al.[39] introduce the concept of developing combined profiles that represent the combined behavior derived from multiple program runs. These profiles can then be utilized in *Feedback-directed optimization* (FDO) of programs [154]; FDO optimizes program performance based on past runtime information. Our contributions to this thesis are closely related to this work. In Chapter 5, we develop an energy profile of iOS core location frameworks to identify inefficient API usages. Additionally, in Chapter 6, we develop an energy profile of SQLite API to estimate its energy consumption in smartphone apps' API usages.

Developers fine-tune the hyper-parameters of their Deep Neural Network (DNN) models to maximize training accuracy and minimize training loss. Liao et al.[112] investigate the impact of hyper-parameter tuning on the optimized models' inference latency, accuracy, model size, and energy consumption. They identify a trade-off between parameter tuning and the aforementioned performance metrics, and highlight the parameters that influence specific types of tuning for optimal performance. Similarly, in Chapter 5, we analyze the impact of different parameter configurations of an API on the energy consumption of an app.

Chen et al. [51] examine performance regressions resulting from source code changes, focusing on response time degradation and increased resource utilization in software systems. They categorize and identify performance regressions that developers can avoid and those that are unavoidable. In contrast, this thesis takes a different approach to addressing the issue of source code changes. In Chapter 6, our approach uses static analysis techniques to assess the impact of specific code modifications on energy consumption.

Similar to this thesis, Bezemer et al. [40] highlight the detrimental impact of long execution times in performance load testing, which can negatively affect the efficiency of a CI/CD pipeline.

Their industrial survey reveals the need for simpler performance analysis tools that seamlessly integrate into the CI/CD pipeline for practical adoption. This finding further reinforces the contributions made in this thesis. Recognizing that energy testing can also be time-consuming due to the need for executing test cases, we address this challenge in Chapter 6 by providing energy consumption details within milliseconds. Developers can easily integrate our approach into their CI/CD pipeline.

Schuler et al. [151] conducted a study to investigate whether the utilization profiles of APIs can be utilized to estimate their energy consumption. By analyzing the usage patterns of an external library API, they identified the specific System API calls it relied on to access system resources. Based on this usage information, they successfully predicted the energy consumption of the API. This work is highly relevant to Chapter 6 in this thesis as it complements our proposed approach. In Chapter 6, we profile the energy consumption of tasks associated with external APIs and leverage these profiles to estimate an app's API usage energy consumption without the need for actual execution. Although our approach does not require executing the app, it does require pre-existing energy profiles for the APIs, which typically necessitate the use of hardware resources. However, the work by Schuler et al. [151] contributes to our energy profile collection process by enabling developers to utilize an API's utilization profile for estimating its energy profile. Schuler et al.'s work [151], in turn, facilitates the reasoning about the energy consumption of an app based on its API usage.

**Can execution time be used as a proxy to reason about energy consumption:** Researchers identify performance bottlenecks by profiling the execution time of each function in a program using a profiling tool. These tools generate a call graph to represent the functions that execute in a program most frequently and those that will take the most time to execute [68], [173]. This information can assists developers in optimizing functions that are slowing down a program. However, can this information be used to reason about energy consumption as well?

The idea to use execution time as a proxy to reason about energy consumption has been widely held [167]. This idea is based on the belief that energy consumption is simply the integration of power over time. Therefore, measuring execution time statically should be sufficient to reason about energy consumption. However, recent research by Weber et al. [167] has challenged this hypothesis for configurable systems. Configurable systems provide performance customization to the developers and users to meet the needs of different use cases or scenarios. According to Weber et al. [167], execution time can not be used as a proxy to reason about the energy consumption of such systems. Additionally, current software estimation models require more information than just runtime data, such as execution time, to estimate energy consumption accurately. This information includes operating system logs, the types of components used, and the operations performed during execution.

## 2.8 Background Related Contributions

In this section, we explain what concepts the following chapters use in order to contribute towards this thesis.

Chapter 3 employs genetic algorithms to find optimal combinations of byte-code transformations for reducing energy consumption in apps. To understand this chapter, it is necessary to have a general understanding of byte-code transformations, which is explained within the chapter itself. While this approach alleviates the need for code-level modifications, it does suffer from the limitation of long test case execution. To address this limitation, we use static analysis approaches in the thesis to optimize and measure energy consumption in apps, in Chapter 5 and Chapter 6.

However, before proposing the optimization methodology (Chapter 5) and measurement model (Chapter 6), we chose to shift our focus from Android to iOS apps. Therefore, in Chapter 4, we develop an iOS-based energy measurement framework to collect ground truth energy measurements. For this chapter, an understanding of energy measurement techniques in the literature is required, and such background covered within the chapter itself. We use this framework in the subsequent chapters (Chapter 5 and Chapter 6) to validate their results.

Chapter 5 acknowledges the limitations of long test case execution identified in Chapter 3 and presents API-level configuration guidelines that developers can statically apply in their code to optimize API-usage in terms of energy consumption. For this chapter, a general understanding of how developers use and configure APIs in their code is required. The background information on how API-level configuration can affect energy consumption is provided in the chapter with an example.

Finally, Chapter 6 addresses the limitations of long test case execution identified in Chapter 3 and provides energy consumption estimates of apps' API usage based on call-graph traversal without the need for actual test case execution. This chapter requires a basic understanding of call-graph traversal, the use of intermediate representations to generate call graphs, and how APIs consume energy and can be analyzed for energy consumption within loops using static analysis. Many of these concepts are already covered in the Background chapter, while the rest are explained in Chapter 6 itself.

# Chapter 3

# Feasibility of current energy measurement approaches

This chapter proposes a novel energy optimization technique that uses byte-code transformations to optimize energy without requiring code modification. The technique's underlying methodology relies on state-of-the-art concept of search-based energy optimization and test-case execution based energy measurement. Thus, as the first contribution of this thesis, this chapter highlights the costliness of test case execution and supports the idea that there is a need for alternative methods of energy measurement. This chapter is published at ICSE-NIER 2022 [33].

## 3.1 Introduction

Energy-aware app developers continuously strive to adopt development practices that optimize the energy consumption of their apps to avoid negative user reviews and decreased market share [104], [142]. Energy efficient development practices are extremely important because apps rely on the limited energy stored in smartphone batteries to operate [142]. High energy consumption can quickly drain the battery, leading to user frustration and dissatisfaction. Therefore, developers need to optimize their app's energy usage to ensure a positive user experience and maintain market competitiveness [135], [143].

According to prior work, developers may optimize the energy consumption of an app by adopting energy-efficient design patterns [55], APIs [83], [120], and UI elements [116]. Unfortunately, all of these approaches require developers to make source-code level changes that leave them with no choice but to: (1) modify their code structure [30], [62] that could affect code-quality attributes such as maintainability, (2) select a specific API [37], [83], [120], [136] that could confine developers' choice, or (3) change the UI structure [115], [166] that could compromise the design aesthetics of graphic designers. As a solution, in this chapter, we introduce a byte-code-transformation-level energy optimization technique so that developers do not have to modify their code.

We initially investigate if byte-code transformations affect the energy consumption of an app by applying a set of byte-code transformations on real-world Android apps. After discovering that byte-code transformations affect an app's energy consumption, we identify the most effective combination of the transformations for each app. To identify the best combination, we mutate an app's byte code with random transformation combinations and execute the test cases multiple times on the mutated code to measure its energy consumption. We repeat this process until we find the least energy-consuming version of the app. During the whole process, we were able to measure the costliness of test case execution in terms of time as well.

Results show that byte-code transformations can potentially reduce real-world apps' energy consumption by up to $11\%$. However, we are inconclusive on which byte-code transformation combination is generally effective. We also found out that test case execution, in some cases, may take up to 40 hours. Therefore it is impractical to integrate such an approach in an IDE or CI/CD pipeline where developers require quick insights about the status of their code.

## 3.2 Related Work

Search-based software engineering works on the principles of genetic algorithms. Literature has used genetic algorithms to reduce the energy consumption of smart-buildings [170], wireless networks [100], subway trains [45], grid systems [107], and also smartphone apps.

Researchers have previously used genetic search to find energy-efficient libraries usage in software [48], [120]. Some researchers used genetic search to address matrix multiplication energy tuning [31], energy efficient SAT solver [47], and energy efficient assembly code exploration [152]. The closest work is by [74], [138], in which they found the effect of the GCC and LLVM compiler configurations on energy consumption. However, since they evaluated their approach on embedded systems' benchmarks, their results can not be generalized over real-world Android apps running on a smartphone device. In this chapter, we leverage genetic algorithms to optimize the energy consumption of Android apps.

Table 3.1: A brief description of the byte-code transformations used in this study [71].

| Type | Description |
|---|---|
| (CTP) | *Constant Propagation* Substitute the values of known constants in expressions. |
| (CPP) | *Copy Propagation* Replaces the occurrences of targets of direct assignments with their values. |
| (DCE) | *Dead Code Elimination* Removes dead code from methods. |
| (DSI) | *Delete Super Interface* Ensures that the method arguments pass directly through to the super invocation. |
| (MIL) | *Method Inlining* Inlines methods that are sufficiently small (or called only a few times). |
| (MRA) | *Minimum Register Allocation* Uses graph coloring to use minimum number of registers. |
| (PHO) | *Peephole Optimization* Eliminates redundant code patterns. |
| (RDB) | *Remove Duplicate Blocks* Removes duplicate blocks. |
| (REC) | *Remove Empty Classes* Removes classes with no-functionality. |
| (ROI) | *Reorder Interfaces* Reorders Interface list for each class to improve the linear walk of list. |
| (RBI) | *Rebind Invocations* Rebinds all invocations of a virtual method or interface to their most abstract type. |
| (RGT) | *Remove Gotos* Removes gotos that are chained together by rearranging the instruction blocks to be in order. |
| (RSM) | *Remove Synthetic Methods* Removes synthetic methods by javac. |
| (RUC) | *Remove Unreachable Code* Removes un-reachable methods, fields and classes. |
| (SIR) | *Single Interface Removal* Removes interfaces that are implemented only once. |
| (SMF) | *String Minification* Minify constant string literals to reduce APK size. |
| (SBR) | *Synthetic bridge removal* Removes bridge methods that javac creates to provide argument and return-type covariance. |

## 3.3 Do byte-code transformations affect energy consumption?

Developers already use byte-code transformations to minimize their app's byte-code size and optimize its runtime performance. However, in this section, we investigate if these optimizations affect energy consumption because if they do, then developers need to be careful while applying such transformations to their code.

To observe the effects of byte-code transformations on the energy consumption of real-world applications, we carry out a preliminary investigation by randomly choosing three transformations from the Redex framework [71]. These transformations include: SMF – String Minification; MIL – Method Inlining; and SIR – Single Interface Removal. Redex is Facebook Inc.'s official byte-

Table 3.2: Description of the apps used in preliminary study

| **Application** | **Category** | **No. of versions** | **No. of unique systems calls** |
|---|---|---:|---:|
| 2048 | Game | 44 | 60 |
| 24game | Game | 1 | 50 |
| Acrylic Paint | Entertainment | 40 | 49 |
| AndQuote | Utility | 21 | 51 |
| Agram | Algorithm | 3 | 46 |
| Bomber | Game | 79 | 47 |
| Budget | Utility | 59 | 56 |
| Calculator | Utility | 97 | 48 |
| Chrome | Browser | 50 | 76 |
| DalvikExplorer | Utility | 13 | 54 |
| Exodus | Utility | 3 | 60 |
| Eye in Sky | Utility | 1 | 77 |
| Face Slim | Utility | 1 | 65 |
| GnuCash | Utility | 16 | 56 |
| Memopad | Utility | 52 | 47 |
| Paint Electric Sheep | Entertainment | 1 | 66 |
| Pinball | Game | 54 | 48 |
| Sensor Readout | Utility | 37 | 51 |
| Temaki | Utility | 66 | 50 |
| VLC | Entertainment | 46 | 61 |
| Wikimedia | Utility | 58 | 67 |
| Yelp | Utility | 12 | 78 |

Table 3.3: Energy consumption difference between 22 Android apps and their transformed versions when SM, MIL, SIR and AT are applied. Negative values represent the reduced value of energy consumption in (% Joules)

| **Application** | **Original Energy (Joules)** | **SMF Affect on Energy (Joules %)** | **MIL Affect on Energy (Joules %)** | **SIR Affect on Energy (Joules %)** | **AT Affect on Energy (Joules %)** |
|---|---|---|---|---|---|
| 2048 | 58.67 | -0.79 | -3.07 | -4.12 | 0.77 |
| 24game | 93.58 | -6.19 | -5.87 | -7.09 | 1.52 |
| Acrylic Paint | 112.89 | -0.03 | 0.03 | 0.04 | 0.32 |
| Agram | 65.42 | 0.49 | 0.16 | 1.88 | 0.41 |
| AndQuote | 38.05 | -0.95 | -0.58 | -0.62 | -0.44 |
| Bomber | 171.83 | 0.33 | -0.10 | -0.83 | -0.83 |
| Budget | 112.89 | -0.03 | 0.03 | 0.04 | 0.32 |
| Calculator | 108.35 | 0.032 | 0.49 | 0.30 | 0.32 |
| ChromeShell | 109.39 | -0.39 | -0.13 | -0.19 | -0.35 |
| DalvikExplorer | 63.28 | 0.13 | -0.31 | -0.27 | 0.14 |
| Eye in Sky | 104.82 | -0.14 | -0.11 | 0.08 | 1.24 |
| Exodus | 99.88 | 0.36 | 0.75 | 0.18 | 0.57 |
| Face Slim | 62.08 | -0.35 | 0.21 | -1.14 | 0.22 |
| GnuCash | 73.07 | 0.14 | 0.01 | 0.24 | 0.19 |
| Memopad | 81.35 | -0.34 | -0.22 | -0.26 | -0.32 |
| Paint Electric | 90.69 | -0.06 | 0.11 | 0.41 | -0.13 |
| Pinball | 112.38 | 1.34 | 0.22 | 1.66 | 1.05 |
| Sensor Readout | 172.91 | -0.04 | 0.33 | -0.19 | 0.13 |
| Temaki | 71.76 | -0.09 | -0.17 | -0.23 | 0.06 |
| VLC | 114.26 | -3.89 | -3.70 | -3.99 | -3.93 |
| Wikimedia | 194.82 | -0.13 | 0.03 | -0.15 | -0.21 |
| Yelp | 251.96 | -1.08 | -0.87 | 0.77 | 0.22 |

code transformation framework designed to reduce the size and improve the runtime performance of Android apps. We provide the details of each of these transformations and the other 14 transformations that will be used later in this chapter in Table 3.1.

The reason for selecting only three transformations is to initially understand the impact of these transformations on energy consumption. By applying these three transformations to 22 applications, we will have a total of 66 transformed applications for which we need to measure their energy consumption. Considering more than three transformations would result in an impractical number of versions (22 multiplied by the number of additional transformations) to evaluate, making it infeasible in terms of energy measurement cost. Therefore, examining a smaller set of transformations allows us to gather insights and establish a foundation for further analysis and evaluation of the remaining 14 transformations, which will be explored later in this chapter.

To apply the selected transformations, we use the publicly available Android apps dataset by Chowdhury et al. [54]. The dataset contains 24 diverse apps accompanied by 106 hand-written test cases that exploit various unique system calls. These test cases aim to emulate typical user behavior when interacting with an application, encompassing actions such as opening the app and performing its core tasks. For instance, in the case of the 2048Game game, the app is launched, and multiple tiles are touched to generate a random sequence. Similarly, in the AcrylicPaint paint app, the user opens the app and proceeds to draw a picture. In the Agram app, fixed letters are added to generate multiple sequences. Lastly, in the ChromeShell browser, the user opens the app and performs a website search.

We failed to build two apps; hence we conducted our experiment on 22 Android apps: 2048-Game, 24Game, AcrylicPaint, Agram, AndQuote, Bomber, Budget, Calculator, ChromeShell, DalvikExplorer, EyeInSky, Exodus, FaceSlim, GnuCash, Memopad, PaintElectric, Pinball, Sensor-Readout, Temaki, VLC, Wikimedia, and Yelp. We present the details about these apps in Table 3.2 where category represents the type of application and number of unique system calls represent how many unique system calls are invoked by the test case of that application. To determine the energy consumption of the apps, we execute each app's test case in its entirety and record the corresponding energy consumption.

To observe the individual effect of each transformation, we apply SMF, MIL, and SIR on the apps individually. Moreover, to observe the collective effect of these transformations, we apply all three together on the apps and refer to such a collective transformation event as *aggregate transformation* (AT). Altogether, we generate $22 \times 4 = 88$ transformed solutions (i.e., transformed versions of the apps). To measure the energy consumption of the solutions, we ran each solution's test suite 10 times on GreenMiner [87], a hardware-based Android apps energy measurement framework. From the results, we found that byte-code transformations do affect the energy consumption of apps, as energy consumption got reduced for 12 out of 22 apps, with an average reduction of $1.24\% \pm 1.96\%$ joules, and with a maximum outlier reduction of $7.1\%$ joules. The detailed result of all the transformations' effect on energy consumption of these 22 applications is available in Table 3.3. The Original Energy column represents the energy consumption of test cases in the apps without any applied transformations, while the remaining columns indicate the difference between the transformed version's energy consumption and the original energy consumption.

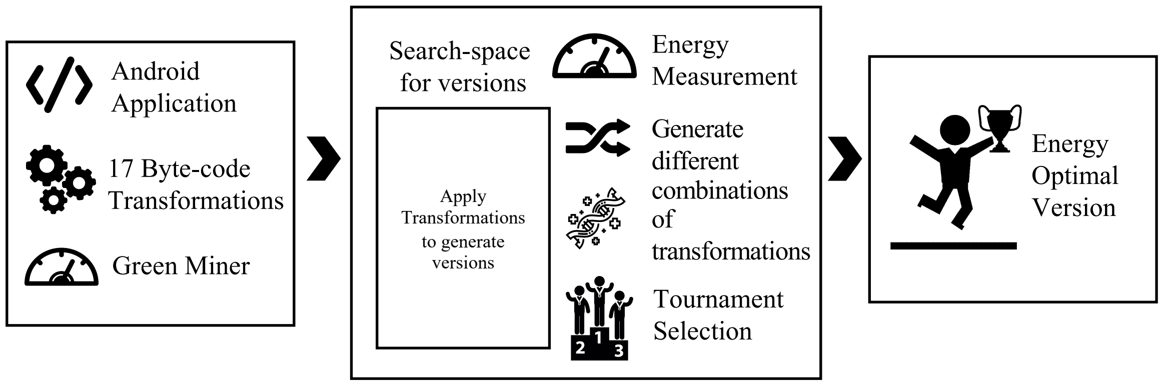
To determine if applying byte-code transformations affects all the apps similarly, we applied a Wilcoxon Signed Rank Test [84] with $\alpha = 0.05$. The results imply that each transformation affects each app differently, and none of the transformations holds a universal effect. For example, AT reduced the energy consumption of VLC, but it increased the energy consumption of 24Game. Similarly, SMF reduced the energy consumption of 24Game, but it increased the energy consumption of Pinball.

Once it is established that some byte-code transformations do affect energy consumption, we employ genetic search to identify the best combination of transformations to optimize the energy consumption of each of the apps.

## 3.4 Using Genetic Search to Optimize Energy Consumption

After confirming that certain byte-code transformations can reduce energy consumption, our investigation turns to evaluating a set of 17 Redex transformations to identify the optimal combination for each app. Table 3.1 provides a brief description of each transformation. We avoid selecting transformations that cannot be applied independently, that is, those that depend on the results of other transformations.

Applying all the combinations of these 17 transformations will yield $2^{17}$ solutions (i.e. transformed versions of the app) for each app, and measuring the energy consumption of those many solutions is impractical. As a solution, we employ genetic search to find an effective combination



Android Application
17 Byte-code Transformations
Green Miner
Search-space for versions
Apply Transformations to generate versions
Energy Measurement
Generate different combinations of transformations
Tournament Selection
Energy Optimal Version

Figure 3.1: Graphical representation of the search-based optimization approach.

of transformations that could reduce energy consumption.

Genetic search is based on the principles of the theory of evolution [82], [153]. Genetic search samples from a large population of possible solutions, mutates the solutions for a new generation and keeps sampling and mutating generations for several rounds until the best solution is found [82].

We present our methodology in Figure 3.1. To identify the best variant of each app, we use three elements: (1) the app's byte code, (2) 17 transformations, and (3) an energy measurement framework such as GreenMiner [87]. First, we generate a large search space of byte-code versions of the app by applying the transformations and then measuring their energy consumption. We then apply mutation and selection criteria to identify the most energy-optimal version of the app and repeat the process until we find the best variant.

The following sections explain our approach for selecting apps and performing a genetic search to identify the best variant.

**Selecting real-world apps** We choose 4 apps from the initially studied 22 apps of our dataset. The criteria used to reduce the 22 Android apps to the 4 apps studied was based on random selection. By randomly selecting one app from each category in the dataset, it ensured a diverse representation of app types. This approach allowed for a broader examination of energy consumption optimization across different categories of apps. The four apps include a game (2048Game);

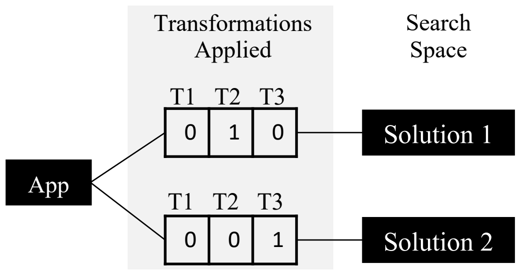


Figure 3.2: An example of applying two different combinations of three transformations on an app to generate a search space of two possible solutions (transformed apps).

an internet browser (ChromeShell); an entertainment app (AcrylicPaint); and anagram algorithm utility (Agram). The test cases of these apps that we consider to execute were auto-generated by Chowdhury et al. [54] in their study. These test cases invoke multiple unique system calls.

## Step-1: Initial population

Initially, we generate a search space for each app. We apply $50$ random combinations of 17 transformations on each app to generate this space. This process yields 50 solutions for each of the apps to begin with.

We represent each solution as a 17-bit vector showing which transformations were applied. Each index of the bit-vector represents a specific transformation, and its value 0/1 represents if that transformation was applied or not. To explain, in Figure 3.2, we demonstrate an example of applying 3 transformations' 2 different combinations on an app and show how these transformations yield 2 different solutions.

## Step-2: Fitness function

The solutions in the initial population and its successive generations are evaluated using a fitness function. A fitness function evaluates each solution's performance and assigns it a fitness score. In our case, the fitness function measures the energy consumption of each solution.

**Energy Measurement** To calculate the fitness score, we use *GreenMiner* [87], a hardware-based energy measurement tool that runs on Galaxy Nexus running Android OS 4.2.2; an Arduino Uno; and an Adafruit INA219 current sensor for monitoring power usage. We run each solution's test cases 3 times for accurate measurements and aggregate their results with a median function, a function used in literature for aggregating energy measurements [87]. The fitness score of the solution is set to the inverse of the calculated energy consumption to ensure a higher fitness score for lower energy consumption.

## Step-3: Selection

For each successive generation, a sample from the existing population is drawn out to develop a new generation. Depending on the selection technique adopted, solutions can be selected based on their fitness scores, or solutions can be randomly sampled from the population.

In this work, we used the elitist selection strategy [29] and the tournament selection strategy [43]. The elitist selection allows passing several solutions with the best fitness score to the next generation. For elitists, we kept its value to 2 solutions, i.e., forwarding the 2 best fitness score solutions in the population to the next generation. In tournament selection, multiple solutions are sampled randomly from the population, and the solution with the highest fitness score is selected. We apply tournament selection twice and select two solutions, a pair, to perform the next step of crossover and mutation.

## Step-4: Crossover and mutation

This process increases diversity among solutions in a search space.

**Crossover** We apply uniform crossover [72] among the previously selected pair of solutions. Using each half of the bit-vector values from each solution in the pair, two new solutions are generated and added to the next generation.

**Mutation** After crossover, we mutate all the solutions in the search-space [90]. To mutate each solution, we flip the bits of its 17-bit vector with a flipping probability of 1/17 for each bit.

## Step-5: Termination

Finally, we terminate the genetic search if there has not been any noticeable improvement in the fitness score of the solutions for several generations. We terminate genetic search when either the

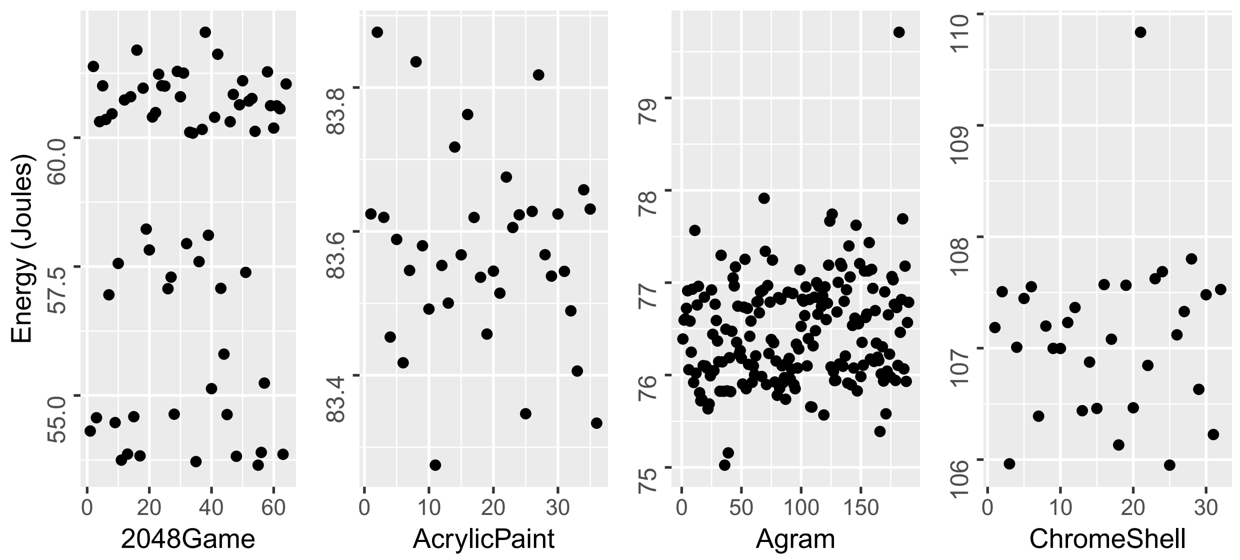


Figure 3.3: Optimal solutions' energy consumption in the search-space. The x-axis indexes the solutions of each app.

energy has not improved in the last 10 generations, or a total of 20 generations have elapsed. We chose this termination criterion according to the cost we could afford regarding test execution time for energy measurement.

## 3.5 Results

After applying multiple combinations of the byte-code transformations with the help of genetic search on the apps, we get the following results.

Table 3.4: A comparison between the energy consumption of the original version of an app and its best solution. Energy improvement shows the energy optimized using genetic search. Test suite execution is the time to run a test suite. Solutions explored are the number of solutions explored in genetic search. Analysis time shows the overall time spent for genetic search. The energy consumption is measured in Joules, the test suite execution is measured in seconds, and the analysis time is measured in hours.

| **Apps** | **Original** | **Best Solution** | **AT** | **Reduction (%)** | **Test suite exec. (s)** | **Solutions explored** | **Analysis time (hrs)** |
|---|---|---|---|---|---|---|---|
| 2048Game | 60.31 | 53.64 | 55.52 | 11.05% | 60 | 234 | 11.7 |
| AcrylicPaint | 83.62 | 83.27 | 83.74 | 0.42% | 95 | 116 | 9.2 |
| Agram | 76.39 | 75.02 | crashed | 1.79% | 77 | 637 | 40.9 |
| ChromeShell | 107.18 | 105.95 | crashed | 1.15% | 100 | 106 | 8.9 |

### 3.5.1 Genetic search reduced the energy consumption of all apps.

We present the search space of each app in Figure 3.3. This figure shows how diverse the search space is, for each app, throughout the process. To understand the variability among the solutions' energy consumption in the search-space, we measure the relative standard deviation [41] among the solutions of each app and found 6.91% among 234 solutions of 2048Game, 0.31% among 116 solutions of AcrylicPaint, 1.32% among 637 solutions of Agram, and 0.95% among 106 solutions of ChromeShell. In conclusion, we were able to identify the combinations of byte-code transformations that reduced the energy consumption of the apps and reduced energy consumption — $0.42\%$ in AcrylicPaint, $1.15\%$ in ChromeShell, $1.79\%$ in Agram, and $11.05\%$ in 2048Game app. However, it is important to note that the significant gain in 2048 might be an outlier due to the non-deterministic behavior of the game app. The randomness in the game stems from the generation of new tiles on the grid, with values of 2 or 4, which occurs at the start and after each move. This randomness introduces an element of unpredictability, resulting in a unique experience with every execution and necessitating strategic decision-making based on the randomly placed new tiles. We have also measured the effects of applying all transformations on apps and find out that such a process may increase the energy consumption of an app instead of reducing it.

Energy consumption could have been reduced even further if we had increased the search space size (i.e., exploration) and the number of generations explored (i.e., exploitation) [113]. Another aspect that should have been investigated is the impact of the order in which transformations are applied. Prior research has demonstrated that the order of compilation can produce different outcomes depending on the nature of the code [3]. However, we were constrained by the time-consuming nature of executing test cases for each solution and therefore we explored the combinations only.

### 3.5.2 Cost-effectiveness of the proposed technique

Table 3.4 presents the estimated cost in terms of time consumption to find the most energy-optimal solution for an app. In the table, *Test suite exec.* presents the total time to run a solution's test suite. *Solutions explored* are the number of solutions explored in the search space. *Analysis time* represents the overall time (actual cost of our technique) in hours to explore all search space generations, including test suite execution, energy measurement and result aggregation. The overall

Table 3.5: Difference in size of the original app and its best solution. *Overall Diff.* includes resource, binary code, and meta files, while classes.dex includes binary code.

| **Application** | **Total app size difference** | **Binary code size difference** |
|---|---:|---:|
| 2048Game | +0.37% | -23.46% |
| AcrylicPaint | -0.07% | -0.66% |
| Agram | -0.16% | +0.30% |
| ChromeShell | 0.02% | -1.26% |

cost of running a genetic search on the selected apps with energy measurement is 9–41 hours.

## 3.6 Discussion

Out of $2^{17}$ possible combinations of byte-code transformations, genetic search concludes that some combinations reduce energy consumption. To investigate the reasons for this reduction, we study the effect of byte-code transformations on the structure of the apps.

To compare the structural difference between the best solution and the original app, we used APK Analyzer [4]. Table 3.5 shows the size difference between the best solution and the original app. The overall difference includes the app's binary code, resource files, and meta files. The difference in classes.dex indicates the size difference in the binary code.

To investigate the structural differences inside the classes' byte-code, we have extracted the Chidamber and Kemerer object-oriented metrics (CKJM metrics) metrics [156] from classes.dex. In the literature, a sub-set of CKJM metrics has proven to correlate with energy consumption [149]. We used a CKJM tool [156] to measure the difference between the original app and its best solution for those CKJM metrics that have a strong correlation with energy, such as Depth of Inheritance Tree (DIT), Number of Immediate Sub-classes (NOC), Coupling between Object (CBO) and Afferent Coupling (Ca). For 2048Game, we observed the following changes: DIT +28.3%, NOC -100% and CBO +0.3%. For ChromeShell, we observed the following changes: CBO +1% and Ca +1.62%. For AcrylicPaint and Agram, only those metrics that do not correlate with energy were affected. According to Sahar et al. [149], DIT has a positive correlation while NOC, CBO and Ca have a negative correlation with energy consumption. Therefore, for ChromeShell, the change in CBO and Ca might be one of the reasons for energy reduction. However, the reasons for energy reduction remain inconclusive for the remaining apps and require further investigation.

## 3.7 Threats to Validity

We evaluated our approach on a limited number of byte-code transformations and benchmarks. Our approach may perform worse for other Android apps regarding overall analysis time cost. In this context, our approach needs to be investigated further on more Android applications. To minimize this threat, we selected 22 applications previously used in the literature [54]. Also, all applications are unique as each belongs to a separate software category, such as utility, game, algorithm and entertainment. The test suites imitate a normal application user behavior hand-written by Chowdhury et al. [54]. However, our test suite does not cover every single line of code within an app, and therefore, our test cases may fail to cover the parts of the code that were affected by a transformation. As a result, our test suite may introduce some bias when measuring the effects of transformations on the entire app. Secondly, we did not investigate if a transformation can be applied to a part of an app. Applying transformations to specific parts of an app's code can increase the capability of our genetic algorithm, as different parts of the app can be targeted for fine-grained exploration. Finally, it is important to acknowledge that we did not specifically consider whether our test case workload intersects with the effects of the transformations. In other words, we do not have information on whether our test cases cover the specific code properties that are influenced by a transformation or not.

The selected transformations are all of the optimizations that were being offered by the Redex framework during the time of this study. These transformations are prevalent since Facebook currently uses them to speed up runtime and minimize the size of their official Facebook Android application - which has over 5 billion downloads on Google Playstore.

We carried out energy measurements through the GreenMiner infrastructure on a single device to ensure consistent results, as our concern was measuring the relative effect between the transformations. However, our approach is fully-capable of tackling the problem of inconsistent energy results from multiple devices as it selects median energy measurement as the fitness score of the variant out of 10 runs.

Finally, in our evaluation, our approach consumed up to 40.9 hours in the worst case. This worst case can not be generalized over the other 2.59 million Android applications on Google

PlayStore [1]. Even though our evaluation is simple, it provides evidence that developers should be careful while applying a specific byte-code transformation since a transformation can negatively affect an app's energy consumption. Our approach helps developers ensure that they do not apply such byte-code transformations, which can increase the energy consumption of their applications. Our evaluation is limited to 4 applications because we used a hardware power profiling tool to measure energy (calculate variants' fitness). The scope of the experiment can be widely broadened if a software power profiling tool is used, as it will tremendously bring down the cost (total analysis time), and many more transformation combinations can be investigated.

## 3.8 Conclusion

To eliminate the need for developers to perform code-level modifications, we propose using byte-code transformations to optimize energy consumption of smartphone apps. However, due to the high cost of executing test cases, such as 8.9 hours for ChromeShell and 40.9 hours for Agram, we were unable to conduct the genetic search multiple times. This limitation highlights the importance of considering the potential impact on the reliability and generalizability of our findings.

Additionally, due to the time consumption of test case execution, it is impractical for the developers to integrate our approach in an IDE or CI/CD pipeline where results are required within a few milliseconds. This work highlights a significant limitation of the current energy optimization, measurement, and estimation approaches that require test case execution, and this thesis addresses this limitation in the following chapters.

As a solution, this thesis proposes API-level guidelines for energy optimization that developers can implement in their code to reduce energy consumption without executing the code (Chapter 5). Furthermore, the thesis presents an energy estimation model (Chapter 6) that eliminates the need for test case execution, providing instant insights into energy consumption based on static-analysis. Exploring static-analysis techniques in order to reason about energy consumption is a step worthwhile as it would significantly reduce the genetic search's exploration cost.

Regardless, this chapter demonstrate the effectiveness of a search-based technique in identifying the optimal combination of byte-code transformations, eliminating the need for manual code modifications. This contribution reduces the development effort and mitigates potential risks as-

[1] https://www.statista.com/

sociated with code changes, such as decreased code maintainability and the potential disruption of design aesthetics or preferred design patterns.

# Chapter 4

# iGreenMiner: An iOS Energy Measurement Framework

The contributions of this thesis, starting from this chapter, are evaluated on iOS apps [24]. Due to the unique hardware architectural requirements of iOS apps, an energy measurement infrastructure based on iOS is necessary. As a solution and the second contribution of this thesis, this chapter presents iGreenMiner, an open-source iOS energy measurement framework.

## 4.1 Introduction

As per current worldwide market share, almost all smartphones either use iOS (14.9%) or Android (85.1%) operating system [134]. While in the United States, iOS smartphones have a market share of 49%, leaving others like Android, Windows Phone, SymbianOS, and Blackberry OS with the rest of 51% [134]. Additionally, developers usually prefer the iOS platform for app development because it generates more revenue than Android [73], [145]. Furthermore, a recent survey of iOS app store reviews found that energy is a major concern for iOS users [102], [103]. iOS provides its users with an estimate to monitor the energy consumption of their apps, meaning energy-hogging apps are on the verge of getting uninstalled at any time. Moreover, iOS suspends apps that over-consume power in the background for an extended period. Because of this suspension ability, developers must ensure that their apps do not engage in unnecessary computation.

As a remedy, Apple already offers various solutions to iOS app developers. Firstly, it lets developers prioritize their tasks through a service class API that helps iOS determine the non-essential background processes of an app. Secondly, XCode [22], an iOS app development environment, provides a plugin called *instruments* that helps developers determine the improper usage of location services, networking services, and CPU. Additionally, iOS defers network and system calls to execute them as a bundle later when the user engages with the phone, a technique proven to help reduce the energy consumption of Android apps [55].

These tools are limited in their ability because they do not provide absolute energy measurements, and there is no open-source hardware-based iOS energy framework available that developers or researchers can use to measure the absolute energy consumption of their iOS apps. Due to the absence of such a framework, no public dataset with the energy measurements of iOS apps is available. Consequently, no energy estimation or prediction model is available for iOS apps.

The current hardware-based energy measurement tools [87], [89], [111], or software-based prediction and estimation models [53], [67], [106] are Android specific and do not always generalize to iOS apps. For instance, Agolli et al. [1] identified certain UI elements that affect the energy consumption of Android apps. However, these elements are irrelevant to iOS because it offers developers different UI-elements structural guidelines compared to Android [73]. Moreover, iOS runs on a different hardware configuration than Android phones, and its development framework follows a different architecture than Android. While some programming language-level contributions that suggest which programming language is energy efficient generally help developers [105], the iOS framework-specific energy consumption problems still require attention.

This chapter introduces an open-source automated energy measurement framework for iOS apps called iGreenMiner. The iGreenMiner framework allows developers to execute test cases of their app and measure their app's energy consumption. Researchers may also use the iGreenMiner framework to collect iOS apps' energy datasets. Such datasets will help build estimation and prediction models and identify iOS-related energy greedy patterns.

## 4.2 Related Work

Hindle et al. [87] proposed GreenMiner, a hardware-based tool that developers use through an open-source web service to measure the energy consumption of Android apps. Cruz et al. [61] highlighted the issue of energy measurement error in the current energy prediction models and proposed an automated framework based on the Monsoon meter. The framework proposes to measure energy when the prediction model is inaccurate; however, to this date, it has not been

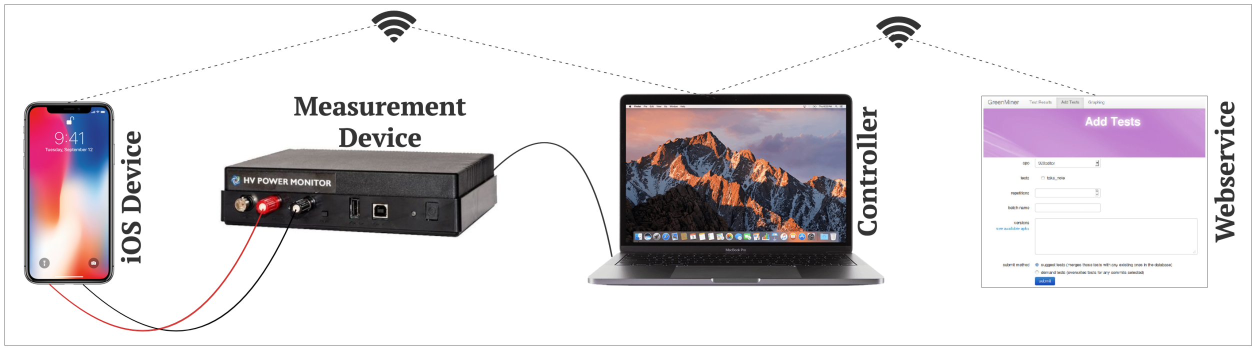


Figure 4.1: The iGreenMiner framework for measuring the energy consumption of an iOS device.

implemented. Matalonga et al. [122] collected a real-world large energy dataset to further the energy consumption and optimization research.

Unfortunately, all of these measurement frameworks are limited to the Android platform. Hence these measurement techniques can not benefit iOS developers.

## 4.3 How iGreenMiner works

To develop iGreenMiner, we use a physical device for energy measurement because hardware-based energy measurements are more accurate than the software-based prediction models [87].

The iGreenMiner framework extends GreenMiner [87], an Android-based energy measurement framework. Figure 4.1 presents an overview of our energy measurement infrastructure. This figure represents four major components: an iOS device, an energy measurement device, a controller, and a web service. To measure an app's energy consumption, a user uploads their app's code and test cases to a web service. This web service informs the controller that deploys the app's code and executes its test cases on an iOS device. During the test case execution, the controller collects the energy measurement of the iOS device through the measurement device. After completing the test case executions, the controller provides the energy measurement results to the web service, making these results available for users to download.

### 4.3.1 iOS Device

The iOS device we use for iGreenMiner is the iPhone 11 [8], running iOS version 13.4.1 [10]. This device has a non-removable 11.91 Wh (3,110 mAh) battery. Smartphone batteries degrade with use over time, and such degradation reduces their ability to hold a full charge, causing the smartphone to operate slowly. This aging factor can also affect energy measurements. To avoid such a problem, we replaced the hard-fixed battery of the iOS device with a constant direct supply of 4.2 Volts. Replacing the battery with a direct current supply required disassembling the iPhone 11, which voided its warranty. This step makes it difficult for developers to replicate this process with expensive phones.

To perform the teardown, we used a hot patch to heat the lower edge of the iPhone and soften the adhesive, making it easier to open. We then unscrewed 1.1 mm-long screws to detach the battery cover from the phone and removed the logic board cover that protected the battery connection point. We also removed the components attached to the top of the battery configuration, such as the speaker and the Taptic Engine, which supports haptic sensory and vibrations. Finally, we removed the battery itself.

After removing the battery, we performed a teardown of the battery to access its connector, which is attached to the phone's logic board, and the terminals needed to provide direct voltage. In the next step, we explain how we soldered the connector's positive and negative terminals joints with direct voltage-supplying wires.

It is important to note that the iPhone does not allow direct voltage supply and has a hardware mechanism to prevent such activity. We can overcome this constraint by connecting an earth wire to the iPhone's steel body every time once before the phone starts. To avoid noise during energy measurements, we switched off all the additional software-based services that iPhone 11 provides.

### 4.3.2 Measurement Device

For energy measurement, we use the Monsoon power monitor as a power draw measurement instrument [155]. We configure Monsoon to provide a direct voltage of 4.2 Volts to the iOS device by connecting wires to its positive and negative terminals and attaching them to their corresponding terminals on the battery connector. The Monsoon measures the current draw of the device in milliAmps (mA) at a rate of 5,000 samples per second and reports the current value with a timestamp back to the controller. The Monsoon was chosen for energy measurement because it is a reliable and accurate tool widely used in the research community for energy measurement of mobile devices [56], [87], [131] and hardware components [101].

### 4.3.3 Controller

We used a macOS computer as a controller to deploy apps and their test cases on the iPhone 11 and collect power measurements from the Monsoon power monitor. The controller collects power measurements, converts them to energy as Joules (J) by execution time, and uploads them to a web service.

If we connect the phone to the controller via USB, the USB provides extra voltage to the iOS device and automatically turns on the charging state, even if the phone is on direct current supply. The problem with the charging state is that the phone starts consuming resources at its full capacity, which is not the usual state for a user who is using a smartphone. Therefore, we connected the iOS device to the controller through WiFi to prevent the phone from going into the charging state.

In our case, the phone remains connected to the controller while the test cases run. Although this is an overhead, we could not overcome it because the controller needs a continuous connection with the phone during the test case execution to collect debug information, such as when the app started running and when it stopped. We need the start and stopping times of the app to filter out the raw power measurements that the Monsoon monitor provides. The Monsoon meter reports measurements per millisecond, whereas the iOS reports the test run start and end time via timestamps. We wrote a script that crops Monsoon's provided measurements to sync it with the actual test case executions. After time synchronization, our script integrates power and converts it into energy, and finally, uploads the results over the GreenMiner web service (explained in the next section).

We would like to mention a few more challenges that we faced because of the iOS platform and how we overcame them during the energy measurement process.

1. The iOS platform imposes several constraints regarding app development for security purposes. One of those constraints is app signing and certification. Each app deployment on the iOS device must be correctly signed and certified; otherwise, the iOS framework does not allow the app to install. Another constraint is that the app's provisioning profile expires every 10 days. We assign a unique bundle ID to each app instance to overcome this hurdle. Both of these constraints limit the automation capability of our approach, and we had to manually configure the testing apps continuously to overcome them during the energy measurements.

2. The Monsoon meter offers a serial port to connect with the controller. This serial port faces concurrency issues when the controller tries to collect measurements from multiple iOS apps. As a solution, we include a hardware piece (Apple's official USB-C to USB Adapter) and toggle this adapter through an open-source external library [129]. After every test run, the toggle resets the connection between Monsoon and the controller.

3. We needed CLI-based implementation for automation. However, the iOS framework, unlike Android, has several limitations when executing and testing applications via CLI. We employ Apple Scripts to navigate via XCode for the smooth operation of specialized test scripts, such as the ones that require user location simulation during execution. Simple scripts that do not require specialized components, such as a Camera or GPS, can be run through CLI.

### 4.3.4 Webservice

For iGreenMiner, we have extended the GreenMiner's web service, a free utility that runs 24x7 [86]. Developers and researchers can utilize this service to measure the energy consumption of their apps.

The iOS app's source code and test cases written using the XCTest test library [26], iOS's official testing framework, are required as input to use this web service. The web service then provides the controller with the app and its test cases. The controller then executes the test cases on the iPhone and simultaneously collects the power traces from Monsoon. It then aggregates the measurements into an energy consumption report and sends it to the web service.

Once the web service receives the energy reports, the reports become available for users to download.

## 4.4 Reliability and Reproducibility of iGreenMiner

The reliability of iGreenMiner depends on its power sampling rate, which is 5 kHz due to the underlying measurement device, the Monsoon power monitor.

While Saborido et al. [148] found that 125 kHz is an ideal sampling rate for energy measurement in Android apps, they also noted that the sampling rate requirement for determining an app's energy consumption is different from that for its methods. A 5 kHz sampling rate for apps is acceptable as it produces a median error of 13.27%. However, a 5 kHz sampling rate is insufficient for methods, as the error can increase to 53.15% with a median of 0.54%. To ensure reliable results, we recommend that users of iGreenMiner instrument their app to execute one method at a time in isolation, execute each method multiple times, and consider the median energy consumption of each method.

Several previous studies have also used the Monsoon power monitor to investigate energy consumption in smartphone apps. Vásquez et al.[114] mined energy-greedy API usage patterns in Android apps, while Li et al.[109] investigated best energy-saving programming practices. Nguyen et al.[131] evaluated the impact of storage configurations on smartphone energy efficiency, and Chen et al.[52] studied factors affecting power consumption in multimedia apps.

Replicating iGreenMiner is important for the research community as it contributes to the expansion of energy datasets and opens up opportunities for further research that will directly benefit the development community of iOS applications. It should be noted that successfully replicating iGreenMiner requires expertise in areas such as iOS app development, power measurements, and smartphone hardware. For clarification and guidance during the replication process, it is recommended to reach out to the author of this thesis.

To replicate iGreenMiner in your own lab, the following steps should be undertaken:

- Familiarize yourself with the principles and objectives of iGreenMiner, which is an energy-aware mining framework designed specifically for iOS applications.

- Thoroughly analyze the scripts utilized by the iGreenMiner framework to collect power measurements from the Monsoon monitor (`runSingleBenchmark.sh`), aggregate the results (`aggregateResults.py`), and the main script for executing both measurement and aggregation (`runAllBenchmarks.sh`).

- Obtain an iPhone model, a macOS machine to serve as the controller, and a Monsoon HVMP power monitor to serve as a power measurement device.

- Carefully disassemble the iPhone and establish a direct connection between its power terminals and the Monsoon power monitor.

- Utilize the iGreenMiner scripts to establish communication with the hardware setup and carry out the necessary power measurements.

It is worth noting that during the replication process of iGreenMiner, one should be fully aware of the underlying challenges that may arise at the iOS level, as discussed in Section 4.3 of the thesis.

## 4.5 Potential research direction

In this section, we outline some research problems related to iOS energy-efficient app development that researchers may address by using the iGreenMiner framework.

1. The iOS platform offers different structural guidelines for developers compared to Android [73]. Developers may be unsure how to design energy-efficient apps while following iOS regulations. By using iGreenMiner, we can develop energy-efficient guidelines on how to layout UI elements in an energy-optimal manner. Previous studies have shown a significant impact of developers' UI choices on the energy consumption of Android applications [1].

2. Apple's energy reporting tool, *Instruments* [92], reports energy ratings by measuring component usage instead of actual energy consumption. By using iGreenMiner, we can improve *Instruments* to predict actual energy measurement values for developers. A previous study shows that components' usage information can be used to build an energy-consumption prediction model [54].

3. Unlike Android, where various energy prediction models are available [54], iOS developers have to deploy iOS applications on a physical smartphone device to measure their energy consumption. XCode, the iOS development IDE, offers an emulator for simulating iOS applications. By using iGreenMiner, we can collect execution logs and energy measurements, relate simulation with system logs or code structure, and build a static energy prediction model. Using this runtime information-based model, developers can estimate their app's energy consumption without running it on a physical smartphone.

4. Developers often want to develop cross-platform apps [42] using frameworks like Flutter, PhoneGap, and ReactNative. Developers need to know how cross-platform iOS apps differ from native iOS apps in terms of energy consumption. By using iGreenMiner, we can empirically evaluate the difference between native iOS apps and cross-platform built apps.

# Chapter 5

# Energy-aware configuration guidelines for API parameters: A case-study of iOS Core Location Framework

This chapter makes a third contribution to this thesis by proposing energy-efficient configuration guidelines for API parameters aimed at reducing an app's energy consumption. This chapter focuses on the iOS Core Location framework, for which we extract the energy-aware guidelines. These guidelines help developers make informed design choices for accessing user-location in their apps. Additionally, researchers may use the approach presented in this chapter to develop energy-aware guidelines for other utility APIs, such as Core Graphics, Core Animation, and Core Bluetooth. This chapter was published at ICSME 2021 [37].

## 5.1 Introduction

Despite energy consumption being a major concern for iOS users [103], and that app developers prefer iOS over other development platforms [73], [145], the iOS platform, in general, suffers from a lack of energy-efficient guidelines for energy-aware developers. Apple advises developers to reduce the energy consumption of processing, networking, location, and graphics [9]. This advice is particularly important for location services, given the pervasive use of location information in various apps such as turn-by-turn map navigation, social networking, food and grocery ordering, carpooling, and vehicle hire. Improper access to the user-location may prevent the device from going into sleep mode, which keeps the location hardware powered on.

To access user location, Apple provides iOS developers with the Core Location framework with multiple location services: Standard Location Service, Significant Location Service, Visits Location Service, and Regional Monitoring Service. Each service is configurable to access user location with a particular frequency and accuracy. This frequency and accuracy come at the cost of battery life through energy consumption. Battery life with accuracy is an engineering tradeoff decision a software developer needs to make while developing an app. Unfortunately, the current Apple developer's guide does not provide any information about such tradeoff [12]. It suggests choosing the most energy-efficient service, but it does not provide details about which services are most energy-efficient, as it says:

*"Always choose the most power-efficient service that serves the needs of your app."* –AppleInc [20] (Core Location Documentation)

To address the limitation of these guidelines, we have extracted energy profiles of the location services using iGreenMiner (Chapter 4). To profile the location services, we created 28 microbenchmarks, each representing a different location service configuration. We then use iGreenMiner to collect the energy measurements of each micro-benchmark (i.e. configuration) and analyze these profiles to extract energy-efficient guidelines for the developers. Our analysis shows that, overall, *Visits Location Service* is the most energy-efficient service, while *Standard Location Service* is the least energy-efficient service. However, we find that Standard Location Service may be configured with particular parameter values to consume the least energy among all other configurations. Using the energy profiles of the location services, we created a graph that describes the tradeoff between location access rate and energy consumption. Developers may use this graph as a guide to prefer one location service over another. Finally, we have evaluated the effectiveness of these guidelines on three real-world apps and a benchmark app that Apple provides to its developers to fetch user location. On real-world apps, we reduced a median of 10.59% energy consumption; on the benchmark app, we reduced 11.37% energy consumption.

Our results have the potential to guide developers in making informed, energy-efficient design choices for accessing user-location in their app before execution. The underlying contributions of this chapter are as follows: *(1) Location framework benchmarks.* Microbenchmarks with multiple configurations of the iOS Core Location framework. Researchers and developers may use these microbenchmarks to investigate the effects of location services on app qualities other than energy consumption, such as runtime performance and user experience. *(2) Recommendation guide.*

Guidelines that help developers may use to make energy-efficient design choices while using the iOS Core Location framework.

## 5.2 Related Work

Recent studies have shown that developers are unaware of which parts of their code consume energy and how their code changes may affect energy consumption [119], [140], [143], [169], [177]. Several researchers have helped developers through recommendation guides and energy optimization techniques.

### 5.2.1 Recommendation Guidelines

There have been several empirical studies to help developers be aware of the development practices that affect energy consumption. Manotas et al. [120] introduce a decision-support framework for developers to make energy-efficient choices using Java Collections API. Similarly, Hasan et al. [83] identify the effect of various Java collection classes on energy consumption and provide guidelines to the developers to reduce energy. Chowdhury et al. [55] introduced an architectural design pattern developers may follow to reduce energy consumption. In a recent study, Oliveira et al. [135] have shown that different development approaches impact the energy consumption of Android apps.

### 5.2.2 Optimization Techniques

Pinto el al. [144] investigated refactoring techniques that developers can adopt to reduce energy consumption. Pambola et al. [139] have conducted an empirical study to find out how code smells affect smartphone apps energy consumption and which code smells are the most energy-hungry. Mazuera et al. [123] have extensively studied code commits of Android and iOS apps to extract performance bugs. The author then proposes a taxonomy that developers may follow to build tools for detecting and optimizing energy bugs. Couto et al. [59] statically analyzed Android apps to detect energy-greedy patterns and propose an automated refactoring technique to eradicate the detected patterns.

This chapter is closest to the investigation done by Schuler et al. [151] in terms of providing an energy recommendation guide for developers. Schuler et al. [151] evaluate the impact of third-party framework calls on energy consumption. Contrary to this chapter's technique, their study considers all calls to frameworks collectively instead of an in-depth investigation of a single framework. Their study also does not provide any recommendation guidelines to developers.

## 5.3 iOS Core Location Framework

This section provides a brief overview of the five location services that the iOS Core Location framework provides:

- Standard and Significant Location Services: tracks changes in user location with configurable accuracy,
- Regional Monitoring Service: tracks user entry and exit from specific regions,
- Beacon Ranging Service: tracks presence near a beacon,
- Visits Location Service: tracks the location where a user has spent a significant amount of time, and
- Compass Headings Service: tracks user direction.

Some of these services are configurable with additional parameters. Developers may use these parameters to change the location-accuracy and location-access-frequency. In this chapter, we investigate how these parameter configurations affect energy consumption. We exclude the Beacon Ranging Service and Compass Headings Service from our investigation because they do not report user location. Beacon Ranging Service detects a user's presence near a beacon, and Compass Headings Service detects a user's movement direction.

**Standard Location Service** This configurable service provides real-time user location. According to the iOS documentation [16], it is the most power-hungry service because it provides the most immediate location with the highest accuracy. The documentation recommends using this service only when an app requires real-time location, such as for navigation instructions or recording a user's hiking path.

Developers may configure this service through two parameters: `desiredAccuracy` and `distance-Filter`. The former may be set to `Best`, `HundredMeters`, `Kilometer`, `Navigation`, `Three-Kilometers`, or `NearestTenMeters`. The latter pauses the location service from fetching further locations unless the user movement exceeds the specified distance (in meters). By default, `desiredAccuracy` is set to `Best` and `distanceFilter` is set to `None` (i.e., the service updates the location for any user movement).

Given the lack of documentation about how these values affect energy consumption or location accuracy, developers may find it difficult to decide which value best suits their needs. In particular, Apple's documentation [16] provides a few guidelines for setting the value of `desiredAccuracy`. First, the developer should avoid setting the accuracy level to `Best` or `NearestTenMeters` unless the app requires location updates every few meters. Second, in practice, the framework provides more accurate data than requested. For instance, accuracy `ThreeKilometers` provides an accuracy closer to one-hundred meters, this guideline from Apple is vague. This brief documentation of the configuration parameters does not help developers determine how these parameters may affect the energy consumption of their apps.

**Significant Location Service** According to Apple's documentation [15], developers should use this service to get the current location and be notified only when the user has covered a significant distance. This service does not offer any configuration parameters, and its distance accuracy value is not specified in the documentation. The documentation states that Significant Location Service is the most energy-conservative service but provides the least location accuracy [15].

**Visits Location Service** This service detects only noteworthy movements of a user [17]. The service is not intended for real-time location data but rather to identify patterns in user location. The service has a parameter: `activityType`, which pauses location updates according to its type: `other`, `automotiveNavigation`, `fitness`, `otherNavigation`, and `airborne`. The `automotiveNavigation` option pauses the service location updates if it detects the user not traveling in an automotive. Similarly, `fitness` works for pedestrian activities where indoor positioning is disabled. The `otherNavigation` option works for the navigation of boats, trains, or planes. There is no information available regarding `other` or `airborne` in the documentation,

```
1    // Initializing Standard Location service
2    locationManager.desiredAccuracy = kCLLocationAccuracyBest;
3    locationManager.distanceFilter = 4096;
4    locationManager.startUpdatingLocation();
5
6    // Initializing Significant Location service
7    locationManager.startMonitoringSignfcntLocationChanges();
8
9    // Initializing Regional Monitoring service
10   let maxDistance = locationManager.maximumRegionMonitoringDistance;
11   let region = CLCircularRegion(center: center, radius: maxDistance,
        identifier: identifier);
12   region.notifyOnEntry = true;
13   region.notifyOnExit = false;
14   locationManager.startMonitoring(for: region);
15
16   // Initializing Visit Monitoring service
17   locationManager.startMonitoringVisits();
18   locationManager.activityType = CLActivityType.fitness;
```

Figure 5.1: An example illustrating how to configure various services offered by the iOS Core Location framework.

however, the default `activityType` is `other`.

**Regional Monitoring Service** This service is a low-energy alternative for a scenario where the app needs to identify the user presence within a certain geographical region [14]. A geographical region is a circle with a radius and a center point on the Earth's surface. To begin monitoring a region, the developer has to define a geographical region in their code. The service starts monitoring the user presence in the defined region immediately after the app starts.

Figure 5.1 provides a code example that shows how developers may initialize each location service to access user location. Line 2–Line 4 represent the usage of Standard Location Service, initializing `desiredAccuracy` to `Best` and `distanceFilter` to $2^{12}$ meters. Line 7 represents the usage of the Significant Location Service. Lines Line 10–Line 14 represent the usage of Regional Monitoring Service. Line 10 defines the `maximumDistance` of the region, while Line 11 defines a region using a `radius` and a `center`. The `center` contains a latitude and a longitude. Line 14 initiates the user location monitoring for the specified `radius`. Line 17–Line 18 represent the usage of the Visits Location Service, initializing its parameter `activityType` with `fitness`.

Table 5.1: The microbenchmark configurations that we have created for evaluation. Each configuration implements a different location service with different parameter values. S1–S20 represent configurations for Standard Location Service. V1–V6 represent configurations for Visits Location Service. SG and RM (not part of this table) represent the Significant Location Service and Regional Monitoring Service, respectively.

| distanceFilter | desiredAccuracy | | | |
|---|---|---|---|---|
| | Best | Hundred Meters | Kilometer | Navigation |
| $2^4$ | S1 | S6 | S11 | S16 |
| $2^8$ | S2 | S7 | S12 | S17 |
| $2^{12}$ | S3 | S8 | S13 | S18 |
| $2^{16}$ | S4 | S9 | S14 | S19 |
| None | S5 | S10 | S15 | S20 |

| | activityType | | | | | |
|---|---|---|---|---|---|---|
| | airborne | automotiveNavigation | default | fitness | other | otherNavigation |
| **Configuration** | V1 | V2 | V3 | V4 | V5 | V6 |

## 5.4 Energy Profiling of the Location Services

To help developers determine how various location services and their parameters affect energy consumption, we have created microbenchmark configurations of the location services. We use iGreenMiner to profile their energy consumption. This section presents the location services microbenchmark configurations and the test scenarios for evaluating these configurations.

### 5.4.1 Microbenchmark Creation

To extract the location services energy profile, we have created a basic iOS app structure that accesses user location. Based on this structure, we have created multiple microbenchmark configurations. Each configuration implements a different location service with various parameter values. For Significant Location Service and Regional Monitoring Service, we have created only one configuration because they do not have any parameters to configure. For Visits Location Service, we have created 6 configurations, each having a different value for the parameter `activityType`. For Standard Location Service, we have created 20 configurations; each has a different value for the parameters `desiredAccuracy` and `distanceFilter`. For each value for `desiredAccuracy`, we have selected 5 different values for `distanceFilter`: $2^4$ meters, $2^8$ meters, $2^{12}$ meters, $2^{16}$ meters, and None (i.e., zero meters). Table 5.1 outlines all 28 configurations.

We do not consider the values of `desiredAccuracy` with `ThreeKilometers` and `Nearest-TenMeters`, because their functionality may be achieved with a combination of `desiredAccuracy` values set to `Kilometer` and `Best` with a `distanceFilter` set to `3000 meters` and `10 meters`, respectively. To accurately generate the energy profiles of the configurations, we run multiple test scenarios 5 times on each configuration and compute the median energy measurement value as the representative of a configuration. We ideally wanted to profile 10 energy measurements for each configuration but that would require 46.6 hours of manually-monitored execution of one test case on 28 benchmarks. To avoid this infeasible time of execution, we had to decrease the number of test case executions to 5 for each configuration. To assess the impact of reducing the number of measurements to 5 for each configuration, we calculated the relative standard deviation of the energy measurements. This measure provides an indication of the variability within the data set. In our analysis, we found that the relative standard deviation, resulting from 5 measurements, was $7.37\%$. This indicates a moderate level of variability in the energy consumption values obtained from the test scenarios. While a larger number of measurements would have reduced the relative standard deviation and potentially increased the precision of the analysis, the decision to conduct 5 measurements strikes a balance between resource constraints and obtaining meaningful insights into the energy profiles of the configurations. However, to mitigate the risk of outliers, we used median energy from the measured samples from the iGreenMiner.

### 5.4.2 Energy Measurement through iGreenMiner

To precisely measure the location service usage while the whole app is running, we first need to know how a location service works. Per its parameter configurations, a location service initializes a handler that constantly tracks a user's movement using an accelerometer sensor. It fetches the location using GPS, cellular network, iOS beacon, or WiFi after each several meters. Since our goal is to evaluate the energy consumption of a location service usage and not its initialization, we exclude the handler initialization part from the energy measurements. To achieve that, we crop the energy measurement of the initialization of the handler through a time synchronization script and measure the energy consumption of movement tracking and location fetching only. To evaluate the

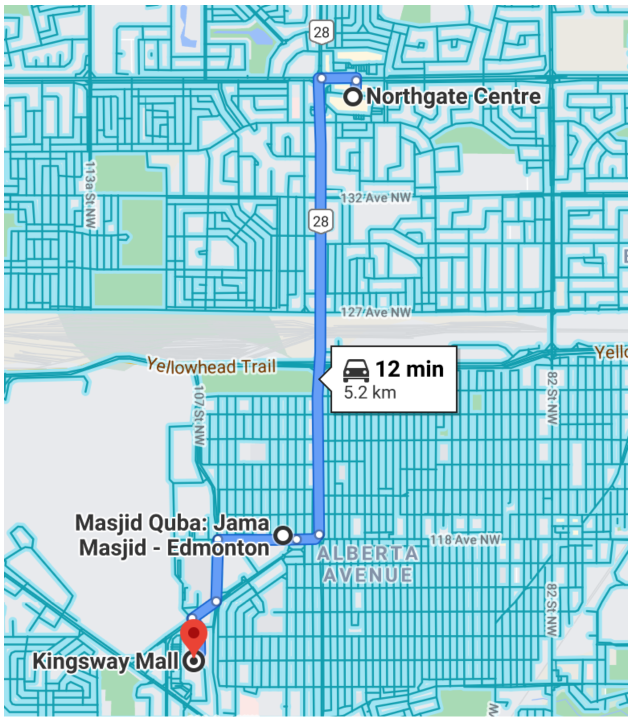


Figure 5.2: A path for a user traveling from Northgate Centre to Kingsway Mall.

configurations, we run two test scenarios on each configuration.

1. The first test scenario: measures the energy consumption of location fetching only and keeps the location stationary.

2. The second test scenario: measures the energy consumption of location fetching with user movement, simulating a user travelling from one point to another. Figure 5.2 shows the path, using Google Maps, where the user covers approximately 5.2 kilometers in 600 seconds from Northgate Centre to Kingsway Mall in Edmonton, AB, Canada. Using Xcode 11.3 [22], an iOS app development platform, we simulate the user location.

To avoid erroneous energy measurements, we keep the screen brightness constant throughout the execution of a test case [69]. To log the number of locations accessed by a configuration (i.e., location access rate) during each test case execution, we instrument each configuration and re-execute the test cases on it separately. As a result, for analysis and comparison, we collected 140 non-instrumented and 140 instrumented energy measurements.

### 5.4.3 Accurate Energy Measurement

To ensure accuracy throughout our empirical evaluation, we took 3 main measures. First, the device could not be locked during a test case execution. This is a constraint of Xcode [22] that for a locked device, it loses connection with the device under test. As an alternative, to mimic the test environment of a real user, when the app is run in the background, we decrease the display brightness to the lowest point such that the screen becomes unreadable and consumes the least energy [69]. Second, installing the app and initializing the location service handler is an overhead to our energy measurements. To avoid this overhead, we start measuring energy after the initialization of the app and the handler. Finally, installing the same configurations and accessing the same location information multiple times may introduce performance differences due to cached app data. To avoid location data caching, we completely wipe out the app's data and install altering configurations each time before each test run.

## 5.5 Which Location Service Should You Use?

To help developers understand the effect of using a particular location service on energy consumption, we have profiled the energy consumption of all the configurations from Table 5.1. We then evaluate those configurations by answering the following research questions:

**RQ1:** Does using a particular location service affect the energy consumption of an app?

**RQ2:** Which location service configuration may help developers reduce the energy consumption of their app?

**RQ3:** Do the default location service configurations yield energy-efficient code?

**RQ4:** Does the energy consumption of a location service change if it runs in a background service instead of a foreground service?

**RQ5:** Which location service configurations perform best in accessing most locations while consuming the least energy?

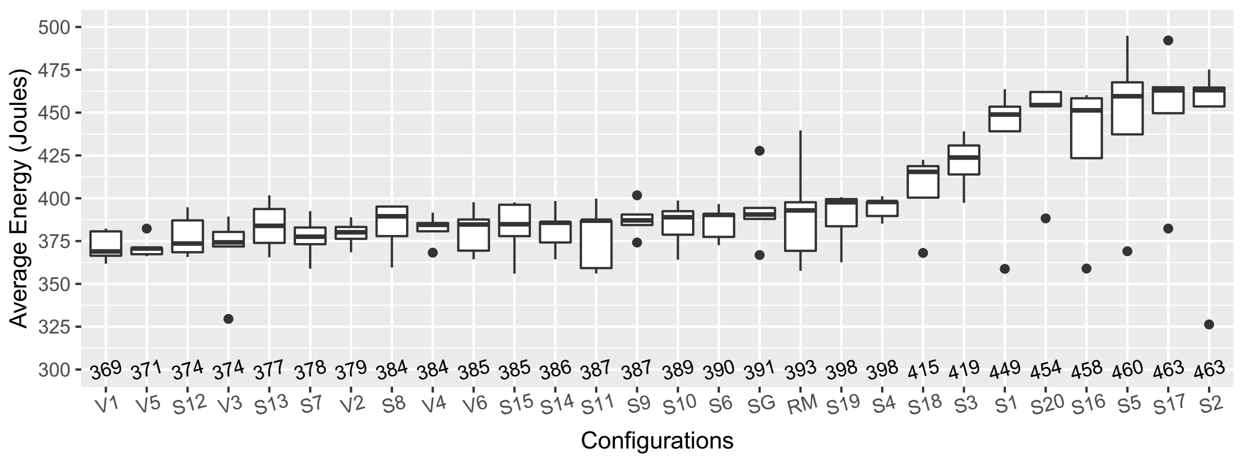


Figure 5.3: The energy profile of each microbenchmark configuration from Table 5.1. Values above the microbenchmark name along the x-axis represent the median energy consumption in Joules. Y-axis represents the average energy consumption in Joules.

### 5.5.1 Effect of Using A Particular Location Service (RQ1)

To evaluate the energy profiles of the configurations in Table 5.1, we ran the second test scenario described in Section 5.4.2 on the configurations. Figure 5.3 represents the energy profiles of each configuration. To investigate the difference in the energy consumption across the services and across configurations, we ran the Kruskal-Wallis rank sum test, and found a p-value $p < 0.05$ for both of them. This p-value indicates that both services and configurations are statistically significant factors in relation to energy consumption. It implies that different services and configurations can exhibit statistically significant differences in energy consumption. Across all configurations, the minimum energy consumed by a configuration is 373 joules, and for the same purpose (location access), the maximum energy consumed by another configuration is 463 joules. This result shows that if developers choose a particular service with certain parameter values, such as S2, it may increase the energy consumption of their app up to 23.97% while performing the same functionality.

To determine the effect of each service parameter value on energy consumption, we evaluated the configurations of Standard Location Service and Visits Location Service by applying pairwise Wilcoxon rank-sum test [84] on the energy measurements of these configurations. We keep $\alpha$ (significance level) as $0.05$ and apply the Bonferroni adjustment method to address the risk of type-I error [168]. Table 5.3 and Table 5.2 shows the results of our test. For Standard Location

Table 5.2: Energy consumption difference ($p$-values for Wilcoxon rank sum test) among the configurations of Visits Location Service for the activityType parameter.

| activityType | Airborne | AutoNav | Default | Fitness | Other |
|---|---|---|---|---|---|
| AutoNav | 1.000 | - | - | - | - |
| Default | 1.000 | 1.000 | - | - | - |
| Fitness | 1.000 | 1.000 | 1.000 | - | - |
| Other | 1.000 | 1.000 | 1.000 | 1.000 | - |
| OtherNav | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |

Table 5.3: Energy consumption difference ($p$-values for Wilcoxon rank sum test) among the configurations of Standard Location Service for the desiredAccuracy parameter.

| desiredAccuracy | Best | HundredMeters | Kilometer |
|---|---|---|---|
| HundredMeters | **0.048** | - | - |
| Kilometer | **0.048** | 1.000 | - |
| Navigation | 1.000 | **0.048** | **0.048** |

Service, there is a statistically significant difference among the configurations of different values of the parameter `desiredAccuracy`. The bold values indicate $p < 0.05$. For Visits Location Service, varying values of the *activityType* parameter do not cause any significant difference.

The p-values $< 0.05$ indicate that developers should carefully configure the parameters while using the Standard Location Service. However, in the case of the Visits Location Service, the choice of a configuration does not matter because there is no difference among the energy consumption of its configurations.

Guideline 1 (G1): Developers should be conscious about choosing a location service configuration because choosing an energy-hungry configuration may increase the energy consumption by up to 23.97%.

### 5.5.2 Effect of Configuring Location Services (RQ2)

To answer RQ2, we ranked all configurations based on the energy profiles we collected for RQ1. Across all configurations, Visits Location Service consumes the least energy, followed by Regional Monitoring Service, Significant Location Service, and Standard Location Service. In particular,

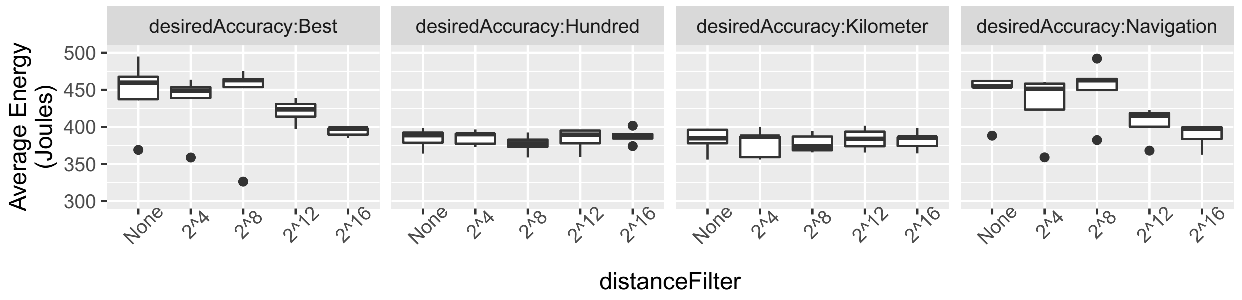


Figure 5.4: The energy profile for each configuration of Standard Location Service. The default values of `desiredAccuracy` and `distanceFilter` are `Best` and `None` (i.e., zero meters).

V1 and V5 from Visits Location Service and S12 from Standard Location Service perform better than all other configurations while providing the same functionality.

Guideline 2 (G2):Across all configurations, Visits Location Service is the most energy efficient choice for developers while Standard Location Service is the worst choice.

To identify the most energy-efficient configurations within Standard Location Service and Visits Location Service, we compare the energy profiles of the configurations of these services. Figure 5.4 shows the energy profile for each Standard Location Service configuration. Each box plot represents the five energy measurement values collected for each configuration, and the x-axis represents the values of the parameter `distanceFilter`. Each boxed section represents the parameter `desiredAccuracy`. Considering the median energy profiles, the best configuration of Standard Location Service is when the value of `desiredAccuracy` is `Kilometer` and the `distanceFilter` value is $2^8$ meters. On the other hand, this service performs the worst when the `desiredAccuracy` value is `Best` and the value of `distanceFilter` is $2^8$ meters.

Similarly, Figure 5.5 shows the energy profiles of each Visits Location Service configuration. For this service, developers may use any configuration because it has no configuration that stands out in terms of energy efficiency.

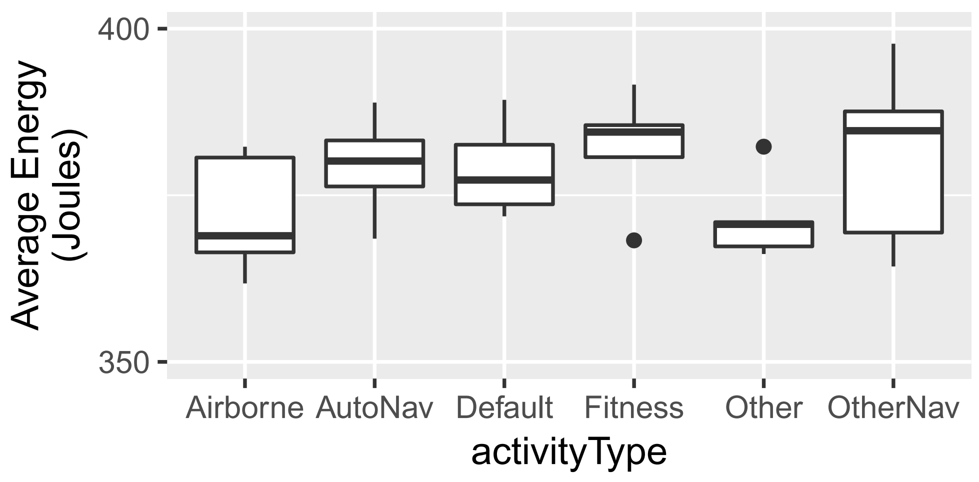


Figure 5.5: The energy profile for each configuration of Visits Location Service. The default value of parameter `activityType` is `other`.

Guideline 3 (G3): The most energy-efficient configuration for Standard Location Service is to set `desiredAccuracy` to `Kilometer` and `distanceFilter` to $2^8$ meters (configuration S12).

### 5.5.3 Effect of Default Configuration Parameters (RQ3)

If the default parameter values in the Standard Location Service and Visits Location Service are energy-efficient, developers will not have to worry about the parameter values of these location services. To evaluate the default configurations, we have executed the second test scenario described in Section 5.4.2 on the default configurations and microbenchmark configurations and compared their energy profiles.

For Standard Location Service, the default values of `desiredAccuracy` and `distanceFilter` are `Best` and `None` (i.e., zero meters), respectively. These values represent the S5 configuration in our microbenchmarks. Observing the energy measurements of Standard Location Service, Figure 5.4 shows that the default parameter values are not the most energy optimal choice. In fact, the default setting (S5) performs the third worst within the 20 configurations of Standard Location Service. Also, regarding location access rate per second, the default configuration performs second to another configuration.

For Visits Location Service, the default value of its parameter `activityType` is `other`, see Figure 5.5. As previously discussed, there is no difference across the energy consumption of Visits Location Service configurations. Therefore, the default configuration for Visits Location Service is a safe option for developers to use in terms of energy consumption.

Guideline 4 (G4): Developers should not use the default configuration of Standard Location Service, because it is not energy efficient. On the other hand, the default configuration for Visits Location Service is equally energy efficient to other location service configurations.

### 5.5.4 Background vs Foreground Service (RQ4)

Depending on its requirements, an app may access location information while running in the background of the UI thread; for instance, a fitness app that continuously accesses locations while the phone is locked and inside the user's pocket. Contrarily, an app might access location while running in the foreground of the UI thread; for instance, a map-navigation-app continuously accessing location while the phone is docked at a car's dashboard. The scope of our study is limited to apps that access locations in the background of the UI thread. However, for a brief comparison of the effects of accessing location in the background vs foreground, on energy, we re-execute the second scenario from Section 5.4.2 on our configurations in the foreground of the UI thread.

To compare the energy profiles of location access in the background of the UI thread against that in the foreground of the UI thread, we ran a paired Wilcoxon signed rank test between the two observations with Bonferroni adjustment applied to address the risk of type-I error [168]. As a result, we get $p = 7.4e - 9$ for the difference in energy consumption and $p = 0.1185$ for the difference in location access rate. This result implies that for $\alpha$ kept 0.05, there is a statistically significant difference among the observations for energy consumption. At the same time, there is no statistically significant difference among the observations for location access rate.

The average energy consumption for background and foreground services is 402.07 joules and 1,255.20 joules, respectively. This overhead energy difference is due to several factors. The iOS operating system generally provides more resources to foreground tasks because they directly interact with the user [18]. Additionally, the screen consumes more energy for foreground services [69] because the screen is dark for background services, whereas, for the foreground execution, the screen is well-lit with a white background. However, a detailed investigation is required to reason about the difference between the two observations, which is out of the scope of this study.

Guideline 5 (G5): The energy consumption of accessing location information in the foreground of the UI thread is 3× higher than accessing it in the background of the UI thread.

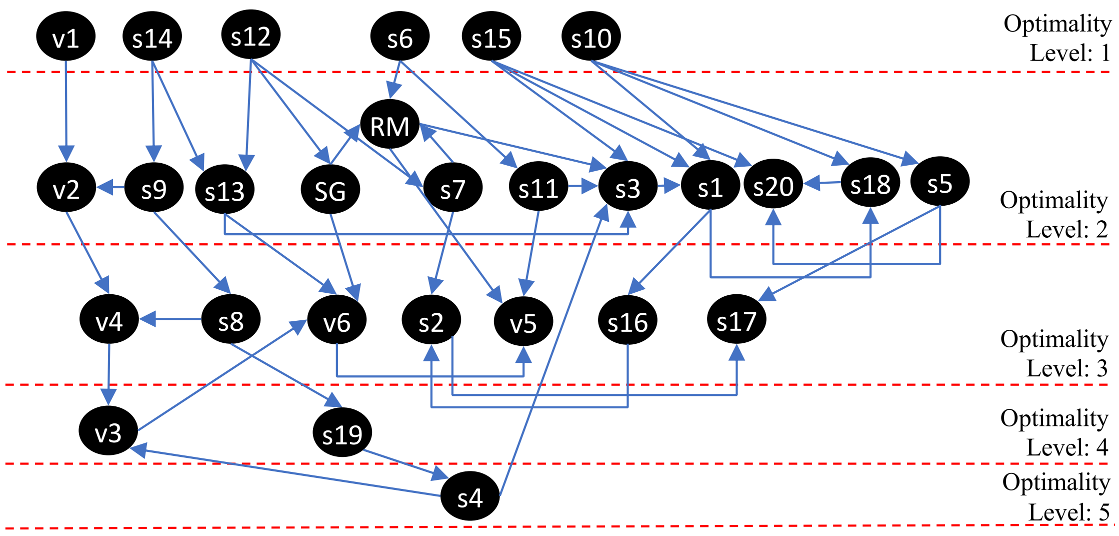
### 5.5.5 Dominating Configurations (RQ5)

We have previously shown that among all configurations, Visits Location Service is the best, whereas Standard Location Service is the worst in terms of energy consumption. However, what if the developer is interested in the frequency of location access, and Visits Location Service provides the least number of locations, whereas Standard Location Service provides the most? Suppose an app requires frequent location updates (e.g., for map navigation). In that case, the app developer typically aims to use a service that accesses locations most often while using the least energy.

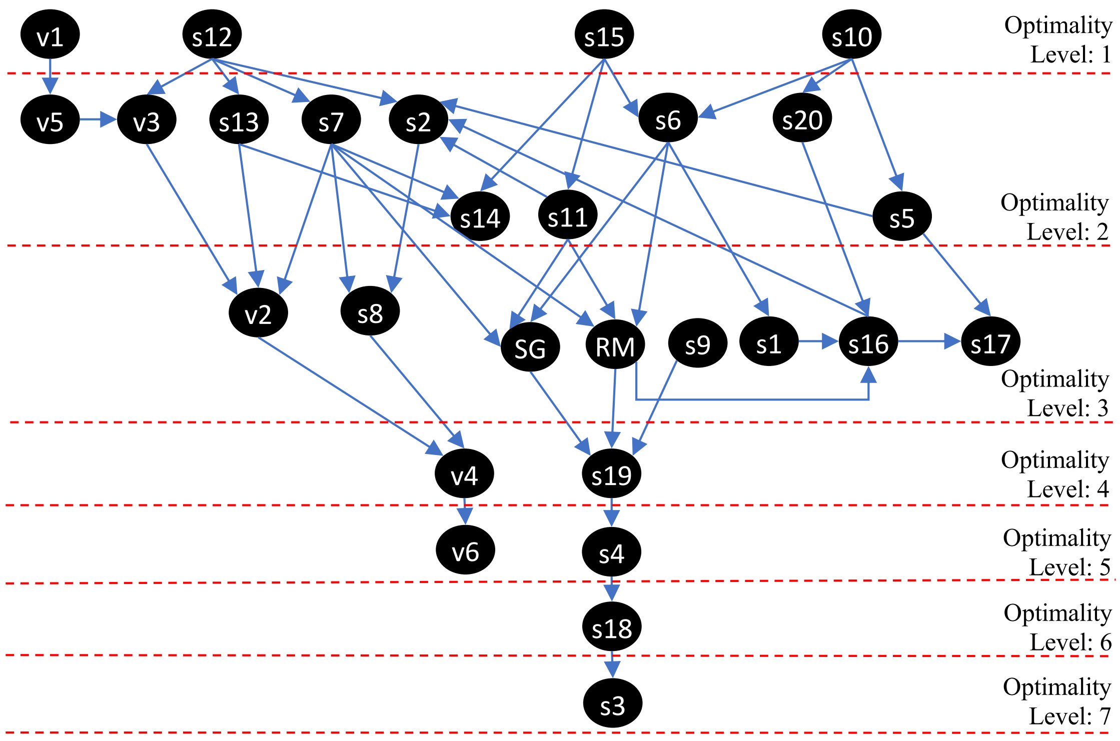
To find out the location service that has the most number of location accesses with the least amount of energy consumption, we execute the second test scenario described in Section 5.4.2 on the microbenchmark configurations twice: (1) in the background of the user interface (UI) thread and (2) in the foreground of the UI thread.

We use a better-than-graph: a directed graph where a source node represents a better configuration than the target node to present the Pareto-optimal solutions. This graph helps developers know which location service configurations dominate others in terms of less energy consumption and high location access rate. Figure 5.6 depicts this better-than-graph. For accessing locations at the foreground of the UI thread, S6, S10, S12, S14, and S15 from Standard Location Service, as well as V1 from Visits Location Service are the most dominating solutions. While for accessing locations at the background of the UI thread S10, S12, and S15 from Standard Location Service, as well as V1 from Visits Location Service are the most dominating solutions.

Guideline 6 (G6): Our graph guides developers to choose a location service based on a trade-off between location access rate and energy consumption.



(a) Benchmarks ran at Foreground



(b) Benchmarks ran in Background

Figure 5.6: The graph represents microbenchmarks which are better than the others when run at the background or foreground of a device. For example, an edge (line) from S12 to S13 means that S12 is better, it has lower energy cost and higher location access rate than S13. Optimality Level represents configurations that collectively dominate other configurations, e.g., Level 1 to Level 7 means most optimal to least optimal.

## 5.6 Discussion

### 5.6.1 Comparing Our Guidelines with the Apple iOS Energy Guidelines

We compare the results of our study with the official energy guidelines provided by Apple [11]–[13]. Although there is a lack of documentation for the location services, Apple provides some general energy guidelines. We provide a summary of the Apple guidelines (AG) and compare them to our own set of guidelines.

- **AG1:** Visits Location Service is the most energy-efficient choice when the app needs a user movement pattern instead of real-time locations. AG1 is recommended for a scenario when an app needs the location only if the user has spent a significant amount of time at that location.
- **AG2:** Significant Location Service is the most power-efficient way when the app needs continuous live locations. AG2 is recommended when real-time location updates are less frequently required.

Our results complement AG1, as we have previously shown that Visits Location Service is the most energy-efficient choice compared to the other location services (G2). However, contrary to AG2, we recommend that developers may prefer some specific configurations of Standard Location Service and Visits Location Service over Significant Location Service. Such as S6, S10, S12, S14, and S15 of Standard Location Service, and V1 of Visits Location Service, while running in the foreground of the app because these configurations always perform better than Significant Location Service in terms of both location access rate and energy consumption (G6).

- **AG3:** Standard Location Service uses significantly more power than other location services, but it delivers the most accurate and immediate location information.

Our results partially complement AG3. We discovered that Standard Location Service delivers the most frequent updates and is collectively the most energy-intensive service (G4). However, we have shown that if Standard Location Service is configured with particular parameter values (S10, S12, S15), it is more energy efficient and has a higher location access rate than Visits Location Service and Significant Location Service (G3).

- **AG4:** Developers should stop location services when the location is not needed anymore.

AG4 is not covered in our test scenarios because our goal is to evaluate the energy consumption of the service itself when it is running. However, it is reasonable to believe that stopping service will reduce energy consumption.

- **AG5:** Setting Standard Location Services' `desiredAccuracy` to `Best` is the most energy-hungry decision for developers, but it provides the most number of locations at the same time.

Unlike AG5, our experiments have shown that the suggested setting performs second best (0.83 locations per second for background, 0.84 locations per second for foreground) to another setting where the `desiredAccuracy` is set to `Hundred` meters (0.905 locations per second for background, 0.89 locations per second for foreground). This alternate setting also consumes less energy than the proposed setting in AG5. A further in-depth investigation is required to determine why `Best`, with a lower location access rate, consumes more energy than `Hundred` meters.

- **AG6:** Defer location updates from the services. Queue the locations and process them all at once some time.

AG6 is an interesting guideline. Similar to prior work [55], queuing up processes has reduced energy consumption due to fewer context switches. However, evaluating AG6 is out of the scope of our study.

### 5.6.2 Why Are There Energy Differences Among the Location Service Configurations?

The total energy consumption cost of a location service configuration comes from three properties: movement tracking sensors, service handler callbacks, and location fetching.

1. Movement tracking sensors: The `desiredAccuracy` and `distanceFilter` parameters of the API track physical activity and distance covered through the accelerometer sensor to report location after a desired interval. The computation from the accelerometer sensor requires energy consumption.

2. Service handler callbacks: The service handler sets up necessary components to manage the location service. One would expect that higher frequencies of callbacks correlate with more energy consumption.

3. Location fetching: This involves retrieving location information from various sources such as mobile data, Wi-Fi, and GPS. The process of obtaining location data requires communication with these hardware components and can consume varying levels of energy.

Considering these three properties provides insights into the energy consumption associated with a location service configuration. By optimizing these aspects, developers can effectively manage and reduce the energy cost of location-based functionality in their applications.

We want to investigate which property out of the three properties changes due to parameter values and eventually affects energy consumption. Movement tracking sensors include the accelerometer sensor, which stays constantly active when the phone is in motion. Service handler computation involves the handler initialization and call frequency. The frequency of handler calls is configured by the `distanceFilter` parameter. Location fetching involves utilizing hardware components by accessing locations through a combination of nearby iOS beacons, network routers, cellular towers, and GPS.

Among these functionalities, we are interested in the location fetching part of our study, since the primary purpose of a location service is to fetch user location. To investigate the cost of location fetching alone, we execute the first test scenario from Section 5.4.2 on all configurations in the foreground of the UI thread. We investigate the location fetch energy cost by keeping the phone location stationary. A stationary location limits the usage of movement sensors and reduces the service handler cost, including movement tracking.

We performed an unpaired Wilcoxon rank-sum test on the energy measurements to measure the difference among the benchmarks for this test scenario. The results show that for $\alpha = 0.05$, and the Bonferroni adjustment method applied, all resultant $p$-values are $> 0.05$, implying no statistically significant difference among the configurations regarding energy consumption. This result shows that the energy difference observed among the configurations is mainly due to two reasons: **(1)** frequency of the location fetching handler calls and **(2)** sensor usage for movement tracking.

### 5.6.3 Do Our Guidelines Help Improve the Energy Consumption of Real-World Apps?

To evaluate the usefulness of our guidelines, we have evaluated them on real-world apps. We replace real-world apps' current location service implementation with an energy-optimal alternative according to the better-than-graph from Figure 5.6. To profile the energy consumption, we execute the second test scenario from Section 5.4.2 on the original app and its modified (energy-optimal alternative) version. To automatically detect the usage of a location service and its parameter values, we have extended SWAN [163], an iOS static analysis framework. Our extension searches for method calls and configuration parameters of any location service usage.

#### Selecting Real-World Apps

To find real-world apps that use the Core Location framework, we have explored the occurrence of code snippets on Github [93] using the search string "`locationManager.start`" [1]. This string represents the usage of location service in iOS apps. As a result, we found 155 apps that use multiple programming languages. Since Apple recommends Swift for development on iOS [19], and iGreenMiner supports Swift, we exclude the apps that do not use Swift as a programming language. This step leaves us with 40 apps that are academic assignments, course projects, and professional apps for Apple's app store.

Across these 40 apps, we applied the following exclusion criteria: Swift version older than 5, build failure, login credentials not provided, and the app crashes. This filtration leaves us with three app categories: property search, weather, and location utility. We then picked one app from each category for this evaluation. In addition to real-world apps, we consider Apple's provided sample code implementation of the location service [6]. This sample code demonstrates the usage of location services for developers.

#### Results

**Property Search App [98]** this app helps users find real estate properties near their location. The app uses Standard Location Service with `desiredAccuracy` set to `NearestTenMeters`, and `distanceFilter` set to `None`. This setup is equivalent to setting `desiredAccuracy` to

[1] https://github.com/search?q=locationManager.start&type=code

`Best` and `distanceFilter` to `10`. The closest representative setup for this configuration in our study is S1.

To optimize this app, we replaced the current app S1 implementation with S12 because Figure 5.6a suggests that S12 accesses more locations than S1 with less energy usage. As a result, with an energy-efficient alternative, we reduced the energy consumption by 0.42%. The impact is negligible since the app only accesses the user location once during the test scenario.

**Weather App [137]** this app reports the weather of the current user location. The app continuously monitors the user's location and updates the weather at every location access. The app uses Standard Location Service with `desiredAccuracy` set to `HundredMeters` and `distanceFilter` set to `None`. This configuration represents S10 in our study.

To optimize this app, we replaced the current app S10 implementation with S12 because Figure 5.3 shows that it is the most energy-efficient configuration for Standard Location Service. As a result, by replacing the app's actual location service implementation with an energy-efficient alternative, we reduced the energy consumption by 10.59%.

**Location Utility App [159]** this sample utility app provides a user interface that continuously reports user location changes with address, speed of movement, direction and altitude. It also performs geocoding, i.e., converting latitude/longitude coordinates to a user-friendly description. The app uses Standard Location Service with `desiredAccuracy` set to `Best` and `distanceFilter` set to `None`. This configuration represents S5 in our study.

To optimize this app, we replaced the current app S5 implementation with S12 because according to Figure 5.6a, S12 accesses more locations than S5 with less energy usage. As a result, by replacing the app's actual location service implementation with an energy-efficient alternative, we reduced the energy consumption by 26.91%.

**Apple's Benchmark App [6]** this is a demo app that Apple provides iOS developers to track user location changes and draw a trail over the map as the user moves. The app uses Standard Location Service with `desiredAccuracy` set to `Best` and `distanceFilter` set to `None`. This configuration represents S5 in our study.

Similar to the location utility app, we replaced the current app S5 implementation with S12. As a result, we reduced energy consumption by 11.37%.

## 5.7 Threats to Validity

We ran all our experiments on a single smartphone device with a single operating system. This setup restricts the generalizability of our results to a single device/operating system configuration. Due to the expensive hardware costs that entail iOS measurement, we evaluate our benchmarks on one of the latest devices running iOS 13.4.1. Although our absolute energy measurements depend on a single device configuration, the relative energy difference between the location service configurations should stay the same. This hypothesis requires further investigation, which is out of the scope of this chapter. Another threat is that our results can not be generalized on iPhone versions other than 11 or iOS versions other than iOS 13.4.1. However, our methodology for evaluating the core location framework will stay relevant.

Applying our configuration guidelines to apps may impact the user experience. For example, if a developer chooses to implement Visits Location Service to save energy, it might not meet the needs of users who require real-time location updates. Users may express dissatisfaction for various reasons. Some may find the app's infrequent updates cause them to miss real-time updates or important information. Others may perceive the app's data as inaccurate, leading to unreliable or misleading results. Additionally, configurations that provide accurate data but consume more energy may raise user concerns about the negative impact on their phone's battery life. Hence, developers must consider these trade-offs, balancing energy optimization and user needs to ensure a satisfactory user experience of their apps.

Instead of physically moving the smartphone, we simulate the user's location. This limitation is because moving for 10 minutes around the city five times for each benchmark, while having 28 benchmarks, requires 23 hours of traveling along with the iGreenMiner hardware, which is a prohibitively expensive process. However, physical movement in such an experiment is essential because users use their phones in a heterogeneous network environment that causes several disconnections between WiFi and GPS. Moreover, cellular tower distance, surface elevation and signal range variation may affect energy consumption. For example, in an urban city, a user experiences frequent context switches compared to rural areas because urban cities often have more cellular towers than rural ones. Unfortunately, our mobility was restricted while conducting the experiments due to the Covid-19 pandemic. However, our experimental setup retains the relative energy difference among the location service configurations.

## 5.8 Conclusion

As a software development manager, the decision of whether to adhere to Apple's official recommendations for configuring the iOS Core Location framework or implementing custom configurations depends on various factors. These factors include the specific requirements and constraints of their app, the importance of energy efficiency in their target market, and the resources available to their development team.

It is worth considering that the Apple recommendations serve as a starting point and are designed to provide general guidelines for developers in order to use an API. However, they may not fully address the unique needs and considerations of an app or its target audience. Therefore, following the recommendations, as it is, may not always result in the most energy efficient solution for a specific scenario. On the other hand, using custom configurations of an API that are not documented in Apple's guide requires a deep understanding of the energy consumption of each configuration. Our approach is beneficial for software development teams that lack the expertise and resources to conduct extensive testing of all possible API configurations. It enables developers to make fine-grained decisions regarding API usage, so that developers can fulfill their users' needs efficiently.

Some of the guidelines derived in this study are intuitive; however, prior to our study, there was no empirical evidence available in the literature to support this intuition. To overcome these limitations, we have provided guidelines to developers for energy-efficiently configuring the usage of the Core Location framework in their code. Our guidelines demonstrate that for frequent location updates, three Standard Location Service configurations (S10, S12, S15) are the most energy efficient, while for less frequent location updates, Regional Monitoring Service is the most energy-efficient. After applying our guidelines to three real-world apps, each representing a different category, we observed energy consumption reductions of 0.42%, 10.59%, and 26.91%.

Apple Inc. can utilize our methodology to offer valuable insights and recommendations on energy-efficient configuration and utilization of their various frameworks' APIs. These guidelines will serve as a valuable resource for iOS app developers and developers within the Apple ecosystem, promoting and teaching the best practices for energy-efficient app development in the developer community.

When configuring specific API parameters, app developers may consider various factors, such as the app's unique characteristics, understanding the app's target audience, and assessing available resources. By following our guidelines and considering these factors, app developers can balance energy efficiency, user experience, and the specific goals of their app to deliver a better user experience.

In future, developers can utilize our iOS Core location API specific guidelines to create energy auto-tuning tools. At the same time, researchers can follow our general approach to establish guidelines for utility frameworks other than the location service API, such as for the Core Graphics, Core ML, Core Animation, and Core Bluetooth API.

# Chapter 6

# A Static-Analysis Based Energy Estimation Approach

Once a developer has reduced the energy consumption of their app using a recommendation guide, they need to measure its energy consumption to get an idea of how much energy their app may consume at runtime. As mentioned in the introduction of this thesis, the current energy measurement approaches are cumbersome as they require the developers to either have hardware-level expertise or to generate and execute test cases. As a solution, this chapter makes a fourth and final contribution to this thesis by proposing a static-analysis-based energy estimation model. The idea of this estimation model is to relieve the developers from expensive, sophisticated hardware and save their expenditure cost on energy testing. Additionally, the idea is to save developers' time so that they do not have to maintain or execute any test cases for the energy estimation of their app. The energy-consumption values that the estimation model represents are called the E-factor of an app. E-factor values can be calculated within milliseconds because it does not require test case execution, as opposed to the state-of-the-art approaches that take a long time to measure and estimate energy consumption (Chapter 3). Due to its quick estimation capability, E-factor can be easily integrated into an IDE so developers can get energy-related insights after each code modification. This chapter was published at MSR 2023 [34].

## 6.1 Introduction

To ensure that an app is energy efficient, energy-aware app developers have several energy profiling techniques to choose from, such as hardware-based measurement tools [87], [89], [111] and software-based estimation models [67]. To get accurate energy measurements using hardware-based tools, developers must run many test cases, which takes significant time and effort. Additionally, setting up the necessary measurement infrastructure requires developers to have specialized knowledge of smartphones and energy measurement devices, which they see as an unwanted task [142]. Furthermore, acquiring the necessary hardware may be costly, adding a financial burden. Alternatively, developers may use software-based techniques that rely on estimation [67], [81] and prediction [53], [106] models. While these techniques may not be as precise as their hardware-based counterparts, developers still widely prefer them because they do not require expensive equipment or specialized expertise to set them up. However, developers still need access to a smartphone, and they need to generate and execute test cases that measure the energy consumption of their apps.

While hardware-based techniques execute test cases to measure energy consumption, software-based techniques execute test cases to gather runtime information for energy estimation. Some runtime information, such as the execution time of a task or the response time of a component, may be obtained without test-case execution using static-analysis techniques, such as worst-case execution-time (WCET) [80] and worst-case response-time (WCRT) [133]. However, this information alone is insufficient to understand the energy consumption of a task [167], because software-based techniques still require additional information such as operating system logs, type of components used, and type of operations performed during execution. Moreover, creating effective test cases is a complex and costly process because it requires a thorough understanding of the app and its requirements and the ability to anticipate and design for potential edge cases [174]. Running these test cases also requires significant resources such as computing time, power, and storage, adding to the overall testing expense [46]. As the software evolves, developers also need additional or updated test cases [124], [126], and such test cases might not be available all the time.

To address the limitations of existing energy measurement techniques, this chapter introduces a static-analysis-based approach that estimates the energy consumption, E-factor, of an API in a given app. This chapter focuses on API usage because API events consume $85\%$ of the energy in an app compared to developer-written instructions (e.g., branch or arithmetic instructions) and system events (e.g., garbage collection or process switching) [110]. Instead of depending on app runtime information, the static-analysis-based approach uses API energy profiles. An API energy profile is the base energy cost (in joules) of the tasks that the API can perform. Obtaining a single API energy profile is a one time process and therefore it is more cost-effective than obtaining runtime information for each app because one API energy profile is relevant to all apps that use that API. To implement this approach in this chapter, my co-authors and I have developed an E-factor calculator (EC) that generates the call graph of a given app and scans that app for API uses. Based on the API uses and energy profile, EC then calculates E-factor without executing the app.

This chapter studies 56 real-world apps that use the SQLite API to evaluate its approach. It focuses on SQLite in this study because SQLite uses a smartphone's CPU, memory, and storage components, which are known to be one of the most power-heavy smartphone components [7]. To obtain the energy profile of SQLite, we created a set of 24 micro-benchmarks representing the `insert`, `update`, and `select` operations. These micro-benchmarks are available [32] as a useful resource for future researchers who wish to measure the impact of SQLite operations on other performance aspects, such as heap size, execution time, and storage size. Among the real-world apps, this chapter compares the E-factor of 16 versions and 11 methods to their hardware-based energy measurements. The findings of this chapter show that E-factor is a reliable estimate to compare the energy consumption between versions of an app and between methods of an app.

## 6.2 Motivating Example

Figure 6.1 shows a development scenario that illustrates the limitations of current approaches for measuring and estimating energy consumption. This section then presents the solution to address such limitations.

In the example, assume that a developer produces three versions of their app, each using SQLite differently. Figure 6.1d shows `DB`, a helper class that implements the SQLite API, which is commonly used by all versions. In this class, `DB.read(..)` executes the `"SELECT ..."` query, `DB.insert(..)` executes the `"INSERT ..."` query, and `DB.update(..)` executes the `"UPDATE ..."` query on a database. Figure 6.1a shows that `DB.insert(..)` and `DB.read(..)` are called in Version-1 in a UI event for tapping the screen (Line 21). Figure 6.1b shows that the event handler in Version-2 calls `DB.update(..)` (Line 28) instead of `DB.read(..)`. Figure 6.1c shows that Version-3 calls both `DB.update(..)` (Line 33) and `DB.read(..)` (Line 34).

Depending on the task that an API performs, an API may consume different amounts of en-

```
class ViewController:
    UIViewController {
  override func onTap() {
    DB.insert(id: i, name: "
        Cat")
    DB.read(id: i)
  }
}
```

(a) Version-1 of the app.

```
class ViewController:
    UIViewController {
  override func onTap() {
    DB.insert(id: i, name: "Cat"
        )
    DB.update(id: i, name: "SCat
        ")
  }
}
```

(b) Version-2 of the app.

```
class ViewController:
    UIViewController {
  override func onTap() {
    DB.update(id: i, name: "
        SCat2")
    DB.read(id: i)
  }
}
```

(c) Version-3 of the app.

```
// SQLite API usage
import SQLite3
class DB {
  // Updates username by id in
      database
  func update(id:Int, name:
      String) { ... }
  // Inserts username by id
  func insert(id:Int, name:
      String) { ... }
  // Returns all users from
      database
  func read(id:Int) -> [User] {
       ... }
}
```

(d) A common class across all versions.

Figure 6.1: Three versions of a sample Swift iOS app, each using `DB` to perform various SQLite operations.

ergy [110]. In our example, the SQLite API may perform *create, read, update, or delete* (CRUD) operations on a locally stored SQLite database. Each CRUD operation has a different workload and requires a different time duration to complete its task. Therefore, each version in Figure 6.1 consumes different energy. To inform developers how their code modifications in each version would affect the energy consumption of their app, this section demonstrates the usage of current energy measurement approaches, then demonstrates the usage and benefits of the proposed static-analysis-based estimation approach.

### Using A Traditional Approach

Following the traditional approach, we must first generate and execute test cases to collect energy measurements of the app. For each version, we have written 5 test cases. In each test case, the method `onTap()` is executed $10^0$–$10^5$ times. Following standard practice [87], we have executed each version 10 times on iGreenMiner, a state-of-the-art hardware-based energy measurement framework for iOS [37]. The results show that for $10^0$–$10^5$ executions of `onTap()`, the average energy consumption of Version-1 is **25.12 joules**, Version-2 is **46.45 joules**, and Version-3 is **25.45 joules**. The average time for executing the test cases per version is **121.33 seconds**.

### Using the proposed approach to calculate E-factor

This approach does not need to execute any test cases. It instead uses an energy profile of an API that represents the base energy cost of its tasks. The energy profile of an API is obtained by measuring the energy consumption of the API's tasks on a hardware-based energy measurement setup. This energy profile is portable and can be used across multiple apps. Using the energy profile, we calculate E-factor for each version. The results show that executing `onTap()` for $10^0$–$10^5$ reports an average E-factor value of **50.25 joules** for Version-1, **92.91 joules** for Version-2, and **50.89 joules** for Version-3.

The traditional approach typically reports a lower energy measurement compared to the E--factor estimate because an app during its execution goes through several architectural-level performance optimizations during its runtime. These optimizations result in a significant absolute difference in the energy measurement of the traditional approach and the energy estimates of E-factor. Nonetheless, since we find a strong positive correlation between energy measurements and

E-factor later in this work, we can identify the energy-intensive (hungry) version in our example using E-factor. In this example, both E-factor and the traditional approach identify Version-2 as the most energy-intensive version and Version-1 as the least energy-intensive. The average time for calculating E-factor per version is **12 milliseconds**, which is **$10^6$ times** faster than the traditional approach.

In a real-world scenario, it is impractical to follow the traditional approach and execute all test cases at every code modification [174], especially in an IDE where a developer is continuously making changes to their code. E-factor is more suitable for this scenario because it provides energy estimates that identify if a code change consumes more/less energy at app and method levels in milliseconds.

## 6.3 Related Work

In this section, we briefly discuss prior work that is closely related to this chapter in terms of methodology and results.

Lyu et al. [118] empirically investigated the effect of SQL anti-patterns on the energy consumption of smartphone apps and discovered that SQL operations performed in a loop may increase an app's energy consumption considerably and suggest that developers should use join SQL queries instead of loops when possible. Their work, however, is a recommendation model for SQL usage, unlike the proposed approach in this chapter that estimates the energy consumption of any API's usage. Hao et al. [81] used static analysis to provide accurate energy measurements at line-level granularity. However, their presented approach requires an app's runtime information and the target smartphone components' energy profile from its manufacturers. It is impractical to acquire a manufacturer's built energy profile as it may not be possible when the owner holds proprietary rights over their components, such as Apple Inc. On the other hand, the approach in this chapter does not require any runtime information or test case execution, it rather needs an energy profile and acquiring such a profile is straightforward because anyone owning a smartphone and an energy measurement instrument can obtain their selected API's energy profile.

Georgiou et al. [75] developed an Energy Consumption Static Analysis (ECSA) technique for embedded systems at both the system-architecture and LLVM-IR levels. However, this approach requires profiling system architecture-level instructions and user-provided source code annotations for loop bounds and identifying infeasible paths. Similarly, Jayaseelan et al. [99] proposed a static-analysis technique to estimate the worst-case energy consumption of software on embedded system architectures. This approach involves profiling energy consumption of assembly code and identifying code sequences that consume the most energy by determining worst-case execution time. As opposed to the technique proposed in this chapter that works on app-code-level, both of these techniques require the developers to have a deeper understanding of the underlying architecture they are testing their software in, because both of the techniques work on architecture instruction-level and assembly instruction-level.

Li et al. [110] empirically investigated 405 real-world Android apps and found some interesting facts about API usage's energy consumption in smartphone apps. They discovered that API events consume 85% of the energy in an app compared to developer-written instructions (e.g., branch or arithmetic instructions). They also discovered that SQLite queries are the third most energy-consuming factor in apps, following Network and Camera related operations. This finding signifies the importance of API usage's energy estimation in general and SQLite usage in specific. Finally, they discovered that operations in a loop represent 41.1% (average) of an app's energy consumption when in a non-idle state. Moreover, loops that use an API may consume 6–41% more energy than loops that do not use an API. This finding further signifies the importance of estimating the energy consumption of API usages in loops and UI events.

Similar to the proposed approach, Jabbarvand et al. [38] focus on measuring the relative energy consumption of the apps to rank different apps by energy cost. However, their approach involves dynamic analysis, so it executes test cases, whereas ours does not require any runtime information; thus, it relieves app developers from the burden of test case execution.

Hasan et al. [83], Oliveira et al. [136], and Pereira et al. [60], [141] use static analysis and profile the energy consumption of the Java Collections API to help developers implement the most energy-efficient collection in their code. However, we use static analysis and energy profiles differently. Instead of optimizing a developer's code, we aim to provide developers with a perspective on how expensive, in terms of energy consumption, the methods or a version of their app could become during runtime.

Table 6.1: The energy profile of SQLite (in joules). $\text{Base}_{\text{exec}}$ is the cost of an operation executed once.

| | # of Executions | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Operation | $\text{Base}_{\text{exec}}$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
| `insert` | 0.00249 | 1.13 | 1.12 | 2.49 | 22.25 | 248.72 | 2,462.33 | 23,313.01 |
| `update` | 0.00251 | 1.20 | 1.23 | 2.66 | 25.13 | 249.33 | 2,494.38 | 23,677.88 |
| `select` | 0.00023 | 1.61 | 1.73 | 1.78 | 2.28 | 15.53 | 184.99 | 1,877.99 |

## 6.4 How Do We Compute Energy Profile of an API?

In this section, we show how we obtain the energy profile of an API, which is necessary for computing E-factor.

To collect the energy profile of an API, we first have to create a set of micro-benchmarks, where each benchmark executes a separate API task. This structure enables us to measure the energy consumption of each task in isolation. We then generate the API energy profile by executing these micro-benchmarks and measuring their energy consumption on a hardware-based infrastructure. We only need to perform this process once and reuse its results in future calculations. We now demonstrate the process by calculating the energy profile of the SQLite API.

### Micro-Benchmarks Creation

We first create a simple iOS app that connects to an SQLite database stored on a smartphone. We then instrument this iOS app to create 8 micro-benchmarks for each SQLite operation (i.e., insert, update, and read), where each micro-benchmark performs an SQLite operation $10^0$–$10^7$ times. We limit the iterations to $10^7$, because executing more iterations becomes impractical as the execution thread on the smartphone tends to time out and detach from the energy measurement infrastructure.

### Energy Measurement

To measure the energy consumption of the generated micro-benchmarks, we use iGreenMiner (Chapter 4), which supports iOS apps that can run on iPhone 11 – iOS 13.4.1. iGreenMiner could only execute Apple-script-based test cases, which creates an overhead for our energy measurements. To overcome this hurdle, we extended iGreenMiner to support executing command-line-based test cases using Xcode command-line tools [22]. We have also optimized the iGreenMiner framework to decrease energy readings overhead during measurement and increase its automation capability. Since iGreenMiner required the user to manually input the test case execution time, to overcome this limitation, we have written a parser for Xcode's build log files where we store the start and teardown times of a micro-benchmark execution. This information helps us synchronize the iGreenMiner's reported energy values with test execution time to get the exact energy measurement of an SQLite operation. In Section 6.6, we use this setup to measure the energy consumption of the real-world apps' versions and methods.

To calculate the energy consumption of the micro-benchmarks, we execute them 10 times each in random order on iGreenMiner. Excluding the app setup and initialization time, this step takes an average of $1.1 \pm 0.13$ seconds to $2.26 \pm 1.52$ hours depending on the number of times that an SQLite operation executes in a benchmark. This long execution time makes it infeasible to pre-populate a database with a random number of records before each test case execution. Therefore, we keep the initial database empty for the `insert` micro-benchmarks. For the `select` and `update` micro-benchmarks, we pre-populate the database with $10^7$ records to read and update from.

Table 6.1 presents the energy cost of each SQLite operation (in joules) when it executes once ($\text{Base}_{\text{exec}}$) and when it executes $10^1$–$10^7$ times in a loop. The data shows that, after $10^2$ iterations, `select` always consumes the least energy, while `insert` and `update` consume more, but similar, amounts of energy. This result shows that one API operation, in comparison to another API operation, consumes different amounts of energy even for the same number of iterations. Therefore, profiling API operations separately is important because knowing the operation execution frequency without knowing the type of operation is insufficient to reason about its energy consumption.

## 6.4.1 Computing Energy Profile for a new API

The energy profile of an API is utilized to estimate the energy consumption associated with its usage. While numerous apps may not use the SQLite API, our creation of an energy profile for the SQLite API allows us to offer a generic guideline for the researchers to obtain energy profiles of other APIs.

To create an energy profile for a new API, certain aspects of the steps that we took to create the energy profile for the SQLite API i Section 6.4 would need to be adjusted. First, the identification stage needs adjustment to recognize the specific API and its relevant usage scenarios. This involves understanding the API's tasks and their behavior. Next, the instrumentation and energy measurement process must be tailored to capture the energy consumption of the new API. The test case design phase needs to be updated to include scenarios that cover different input values, usage patterns, and execution paths specific to the new API. During the execution of test cases, energy measurements need to be collected. The data analysis and profile generation stage should be adapted to analyze the collected energy measurements and extract energy consumption data specific to the new API tasks. By modifying these aspects, researchers and developers can generate a tailored energy profile for the new API, providing valuable insights into its tasks' energy consumption.

## 6.5 How Do We Calculate E-factor?

In this section, we first present our generalized approach to calculate the E-factor of the uses of an API. Later, we instantiate our approach to calculate E-factor of an example code that uses the SQLite API.

### 6.5.1 General Approach For Calculating E-factor

There are five steps that a researcher or developer should follow to determine the E-factor of an API. To automate these steps, we have developed an E-factor calculator (EC) that works as follows:

1. *Platform selection*: select a smartphone app development platform such as Android or iOS.
2. *API selection*: based on the selected platform, select an API to measure the energy consumption of its uses.
3. *Call-graph (CG) generation*: extract the app call-graph.
4. *API call methods identification*: using CG information, identify methods that use the selected API; produce *Method-to-API* mappings by tagging these method calls with source-level information.

5. *E-factor calculation*: use the Method-to-API mappings and the API energy profile to calculate E-factor.

### 6.5.2 Specific Instantiation for SQLite

To calculate the E-factor for the SQLite API, we discuss how we instantiate each step in our methodology:

#### Platform Selection

From the top three smartphone platforms (i.e., Android, iOS, and Windows), we chose iOS because developers prefer it for generating more revenue [73], [145].

#### API Selection

We focus on the SQLite API [88] that enables a developer to use a self-contained transactional database for persistent storage, a feature that most apps need [132].

SQLite supports several smartphone operating systems [158]. Both Apple and Google use it for their native apps [5], [50], [157]. Social-networking apps (e.g., Facebook), browser apps (e.g., Firefox), communication apps (e.g., Skype), and cloud storage apps (e.g., Dropbox) also use SQLite for data storage [157]. Therefore, our focus on studying the energy consumption of SQLite will affect millions of users and developers worldwide. Knowing how the SQLite API consumes energy, developers can identify energy-intensive operations for better-targeted optimization. These optimizations will ultimately lead to longer battery life for the device and a better user experience.

#### Call-graph (CG) generation

To detect the uses of the SQLite API in an iOS app, EC generates its CG using the Class-Hierarchy Analysis (CHA) [66] in SWAN [163]. Using CHA for a sound CG construction suits our work because our approach provides developers with energy estimation for the largest number of potential execution paths that may execute at runtime.

#### API Call Methods Identification

EC traverses the CG to identify method calls in the code that perform SQLite operations. For each method that performs an SQLite operation, EC stores a method-to-API mapping for it, which

```
48  class A {
49    func foo() {
50      DB.insert(...)
51    }
52
53    func qux() {
54      DB.insert(...)
55      DB.insert(...)
56    }
57    ...
58  }
```

(a) Class A

```
59  class B {
60    func bar() {
61      for k in 1 ... n:
62        DB.update(...)
63    }
64    ...
65  }
```

(b) Class B

```
66  class C {
67    func baz() {
68      B.bar()
69      A.foo()
70    }
71
72    override func tap(){
73      DB.read()
74    }
75   ...
76  }
```

(c) Class C

Figure 6.2: Three Swift classes: `A`, `B`, and `C` in a sample iOS app that use `DB` to perform SQLite API operations.

consists of: the method name, the SQLite operation that it performs, the source line at which the API is invoked, and the operation frequency. Figure 6.2 shows a sample iOS app that we will use to illustrate what method-to-API mappings are and how EC uses them to calculate E-factor. In this app, the primary methods that perform SQLite operations are `DB.update(..)`, `DB.insert(..)`, and `DB.read(..)`.

To create method-to-API mappings, EC first identifies the methods that call these primary methods. In class `A`, EC detects that `A.foo()` invokes `DB.insert()` (Line 50). Therefore, EC creates a method-to-API mapping of `A.foo() -> Insert @ Line 50`. EC also detects that `A.qux()` invokes `DB.insert()` twice (Line 54 and Line 55). Therefore, EC creates two method-to-API mappings of `A.qux() -> Insert @ Line 54` and `A.qux() -> Insert @`

Table 6.2: Energy consumption (in joules) of the method-to-API mappings for the sample app in Figure 6.2. The value "–" means "not applicable", and we use it for operations that are not in a loop.

| | | # of Executions | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Method-to-api mappings | Operation | Base$_{exec}$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
| `A.foo()-> Insert@Line 50` | Insert | 0.0024 | – | – | – | – | – | – | – |
| `A.qux()-> Insert@Line 54` | Insert | 0.0024 | – | – | – | – | – | – | – |
| `A.qux()-> Insert@Line 55` | Insert | 0.0024 | – | – | – | – | – | – | – |
| `B.bar()-> Update*@Line 62` | Update* | 0.0025 | 1.20 | 1.23 | 2.66 | 25.14 | 249.33 | 2,494.38 | 23,677.88 |
| `C.baz()->B.bar()-> Update*@Line 68` | Update* | 0.0025 | 1.20 | 1.23 | 2.66 | 25.13 | 249.33 | 2,494.38 | 23,677.88 |
| `C.baz()->A.foo()-> Insert@Line 69` | Insert | 0.0024 | – | – | – | – | – | – | – |
| `C.tap()-> Read*@ Line 73` | Read* | 0.0002 | 1.61 | 1.73 | 1.78 | 2.28 | 15.53 | 184.99 | 1,877.99 |

Table 6.3: E-factor values (in joules) of the aggregated methods from Table 6.2. The value "–" means "not applicable", and we use it for operations that are not in a loop.

| | # of Executions | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Method | Base$_{exec}$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
| `A.foo()` | 0.0024 | – | – | – | – | – | – | – |
| `A.qux()` | 0.0048 | – | – | – | – | – | – | – |
| `B.bar()` | 0.0025 | 1.20 | 1.23 | 2.66 | 25.14 | 249.33 | 2,494.38 | 23,677.88 |
| `C.baz()` | 0.0049 | 1.20 | 1.23 | 2.66 | 25.14 | 249.33 | 2,494.38 | 23,677.88 |
| `C.tap()` | 0.0002 | 1.61 | 1.73 | 1.78 | 2.28 | 15.53 | 184.99 | 1,877.99 |
| Total for App | 0.0148 | 4.01 | 4.19 | 7.10 | 52.55 | 513.99 | 5,173.75 | 49,233.75 |

`Line 55`. In class `B`, EC detects that `B.bar()` invokes `DB.update()` in a loop (Line 62). Therefore, EC creates a method-to-API mapping `B.bar() -> Update* @ Line 62` where `*` represents an operation in a loop. EC identifies such operations by checking if an operation is enclosed in a `for` loop, `while` loop, or enclosed in a UI event, such as tapping/swiping a screen or touching a UI button. While a UI event is not considered a loop in the traditional sense, it may be triggered multiple times by the user. Therefore, we consider a UI event as code that may execute in a loop for energy estimation. This assumption is essential for measuring the energy consumption of an app because the more frequently an event is triggered, the more energy it consumes. In class `C`, EC detects that `C.baz()` invokes `B.bar()` (Line 68), which then invokes `DB.update()` in a loop. Because of this call chain, EC creates a method-to-API mapping `C.baz() -> B.bar() -> Update* @ Line 68`. EC also detects that `C.baz()` invokes `A.foo()` (Line 69), which then invokes `DB.insert()`. Given this call chain, EC creates a method-to-API mapping `C.baz() -> A.foo() -> Insert @ Line 69`. Class `C` has an additional function `tap()` that is a user input event, and it invokes `DB.read()`. Therefore, EC creates a method-to-API mapping `C.tap() -> Read* @ Line 73`.

### E-factor Calculation

Algorithm 1 presents the algorithm of EC. Using the SQLite energy profile from Table 6.1, EC identifies the energy consumption of each method-to-API mapping. The energy consumption of mappings containing a $\star$ is represented by the energy profile's range of energy consumption values for several iterations. For example, if a mapping has a `Read*` operation, its energy consumption is 1.61 for $10^1$ iterations, 1.73 for $10^2$ iterations, etc. The energy consumption of a mapping not in a loop is the $\text{Base}_{\text{exec}}$ cost in the energy profile. Table 6.2 shows the energy consumption of each method-to-API mapping in the sample app from Figure 6.2. The table shows the method that performs an SQLite operation, its $\text{Base}_{\text{exec}}$ cost, and the energy consumption of executing that operation in a loop $10^1$–$10^7$ times. We consider the operations in a loop to execute $10^1$–$10^7$ times because our approach does not rely on runtime information, and we still want to provide the developers with an estimate of how expensive an operation in a loop can be during runtime. Choosing a specific number of iterations is ultimately up to the developer's estimation since they have the necessary expertise and understanding of the runtime behavior of their app.

**Algorithm 1:** Calculating App and Method level E-Factor for an API.

```
Input:
        $appSourceCode$;
        $energyProfile$ ;        // base energy cost of an API-operations.
Output:
        $methodsEfactor$
        $appEfactor$
Initialization:
        $cg$ = constructCallGraph($appSourceCode$)
        $executions$ = [$10^1$,$10^2$,$10^3$,$10^4$,$10^5$,$10^6$,$10^7$]
        $methodToApiMappings$ = [] ;   // the specified value can include a
    list of enum(METHOD_CALL_HIERARCHY, OPERATION_TYPE, IN_LOOP)
        $mappingsEfactor$ = [] // the specified value can include a list of
      enum(METHOD_TO_API_MAPPING, EFACTOR)
Method-to-API Mappings: Traverse through $cg$ to identify method call hierarchies that
    involve methods performing an API operation. $methodToApiMappings$ keeps a list of
    enum, where each enum has $METHOD\_CALL\_HIERARCHY$,
    $OPERATION\_TYPE$ (API operation occurred in call hierarchy), and $IN\_LOOP$
    (whether operation is in a loop or not).

E-Factor Calculation:
for $mapping$ in $methodToApiMappings$ do
    for $execution$ in $executions$ do
        $baseCost$ = $energyProfile[mapping.OPERATION\_TYPE]$;
        if $mapping.IN\_LOOP$ then
            $EFactor$ = $execution*baseCost$;
        else
            $EFactor$ = $baseCost$;
        end
        $mappingsEfactor$ += ($mapping$,$execution$,$EFactor$);
    end
end
$methodsEfactor$ = sumByEntryPoint($mappingsEfactor$) // Calculates the sum
    of the E-Factors grouped by first $methodName$ in the
    $METHOD\_CALL\_HIERARCHY$ and number of $execution$.
$totalEfactor$ = sumByExecution($methodsEfactor$) // Calculates the sum of the
    E-Factors grouped by number of $execution$.
```

To calculate the E-factor value for each method, we aggregate and add up the energy consumption values from Table 6.2 by method name. Table 6.3 shows the final aggregated values. To calculate the E-factor of the complete API usage, we take the sum of the E-factor values of all methods. Table 6.3 shows that, our sample app may consume $0.0148$ ($10^0$ iterations) to $49,233.75$ ($10^7$ iterations) joules depending on its SQLite usage.

Using EC, developers can calculate E-factor after each instance of code modification, which may be presented to them as a table similar to Table 6.3. This table displays the energy cost of a specific API usage for a method and the entire app. EC is open-source and is freely available online [32].

## 6.6 Evaluation

We evaluate E-factor through the following research questions:

**RQ1:** Can E-factor be used to compare the energy consumption difference between the versions of an app in the context of an API's usage?

**RQ2:** Can E-factor's fine-grained method-level energy estimates be used to compare the energy consumption of the methods of an app?

**RQ3:** Does our automated program (EC) accurately estimates the E-factor of real-world apps?

### 6.6.1 Dataset Preparation

To answer our research questions, we first select real-world apps and calculate their E-factor.

#### App Selection

To select apps for evaluation, we search GitHub [76], applying the following inclusion and exclusion criteria on all available iOS apps:

- Include all iOS projects written in Swift because it is the currently supported language for CG generation in our underlying tool SWAN.
- Include projects that implement the SQLite API by searching for "`import sqlite3`" within the codebase.

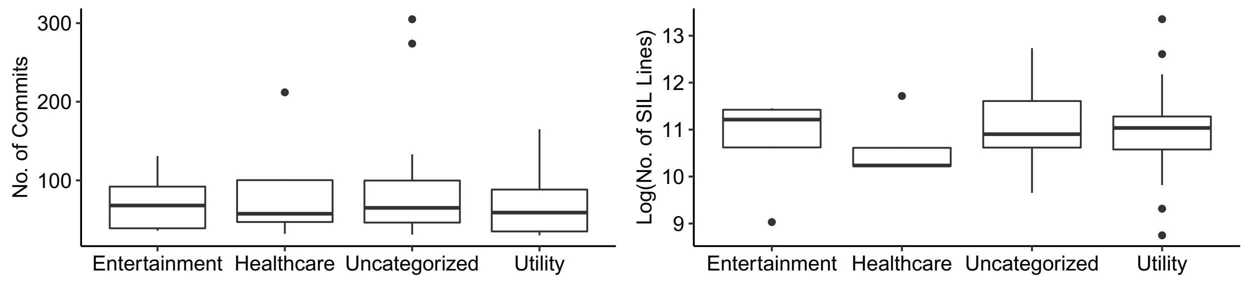


Figure 6.3: Information about the main app categories for the real-world apps in our dataset.

- Exclude projects that are libraries, APIs, or incomplete. We selected projects with a user-interface code structure to ensure this criterion. We recognize that structure by searching for a UI storyboard or an `AppDelegate`.
- Exclude projects that have less than two commits in their repository. This exclusion is because the main use case for E-factor is to compare the energy consumption between app versions.
- Exclude projects that do not contain `.xcodeproj` or `.xcworkspace` file extension. This exclusion makes sure that the projects are executable.
- Exclude projects built for iOS platform $< 9$ because our underlying energy measurement infrastructure (iGreenMiner) supports apps built for iOS $>= 9$.

After applying our selection criteria, we ended up with 59 real-world apps. We group these apps into three categories (entertainment, healthcare, and utility), while some apps remain uncategorized because either their documentation is missing or their commit messages are not clear enough to infer their category. Figure 6.3 shows the number of commits and the number of SIL lines in each app in our dataset. The number of commits shows how active the project development is, while SIL lines represent the project size.

### E-factor Calculation

To detect methods that contain a usage of SQLite, EC searches our dataset using case-insensitive regular expressions: `.*SELECT.*` for read operations, `.*UPDATE.*` for update operations, and

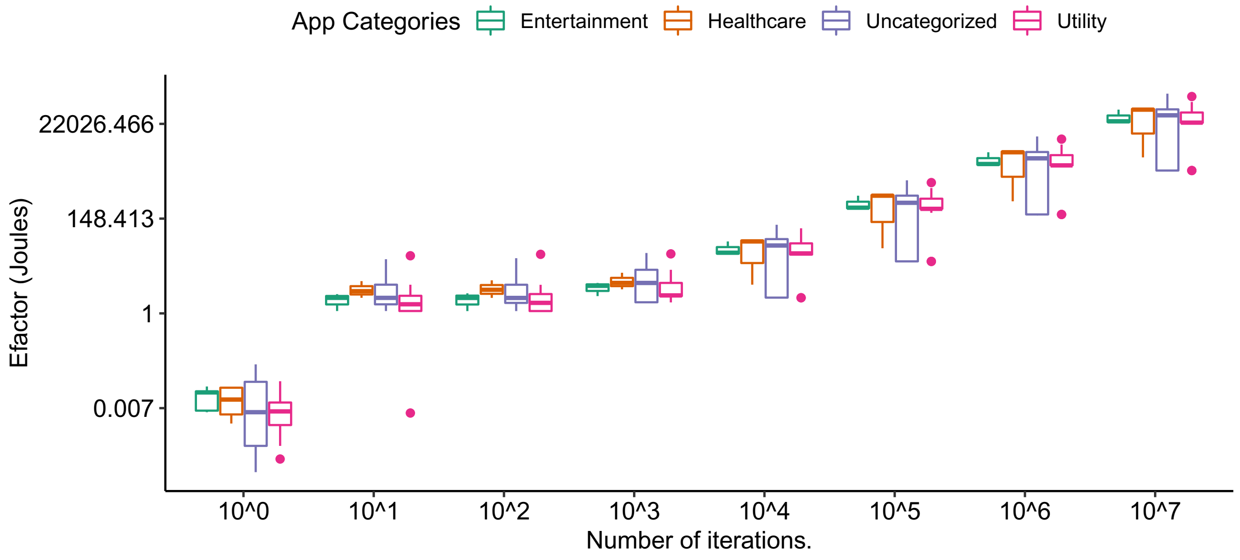


Figure 6.4: The E-factor values of each app category. The range $10^0$–$10^7$ is the number of times that each app may execute its SQLite operations during runtime.

`.*INSERT.*` for insert operations. EC then traverses the CG that SWAN generates for an app to identify those methods that invoke these SQLite methods. Using the identified method-to-API mappings, EC calculates the E-factor values of apps and their methods as explained in Section 6.5.2. Figure 6.4 shows the E-factor value for each app category for $10^0$–$10^7$ executions. The figure also shows that the rank of app categories by E-factor changes across the different number of iterations. Out of the 59 apps, there were 8 apps with zero E-factor value because the SQLite operations in these apps are never invoked by a method. We could not estimate the E-factor of 3 additional apps, because they either contained incomplete code or had a third-party API dependency that we could not resolve to build the app. It took an average time of 12.44 milliseconds to calculate the E-factor for each of the 56 real-world apps, where the majority of this time was spent in the call-graph traversal.

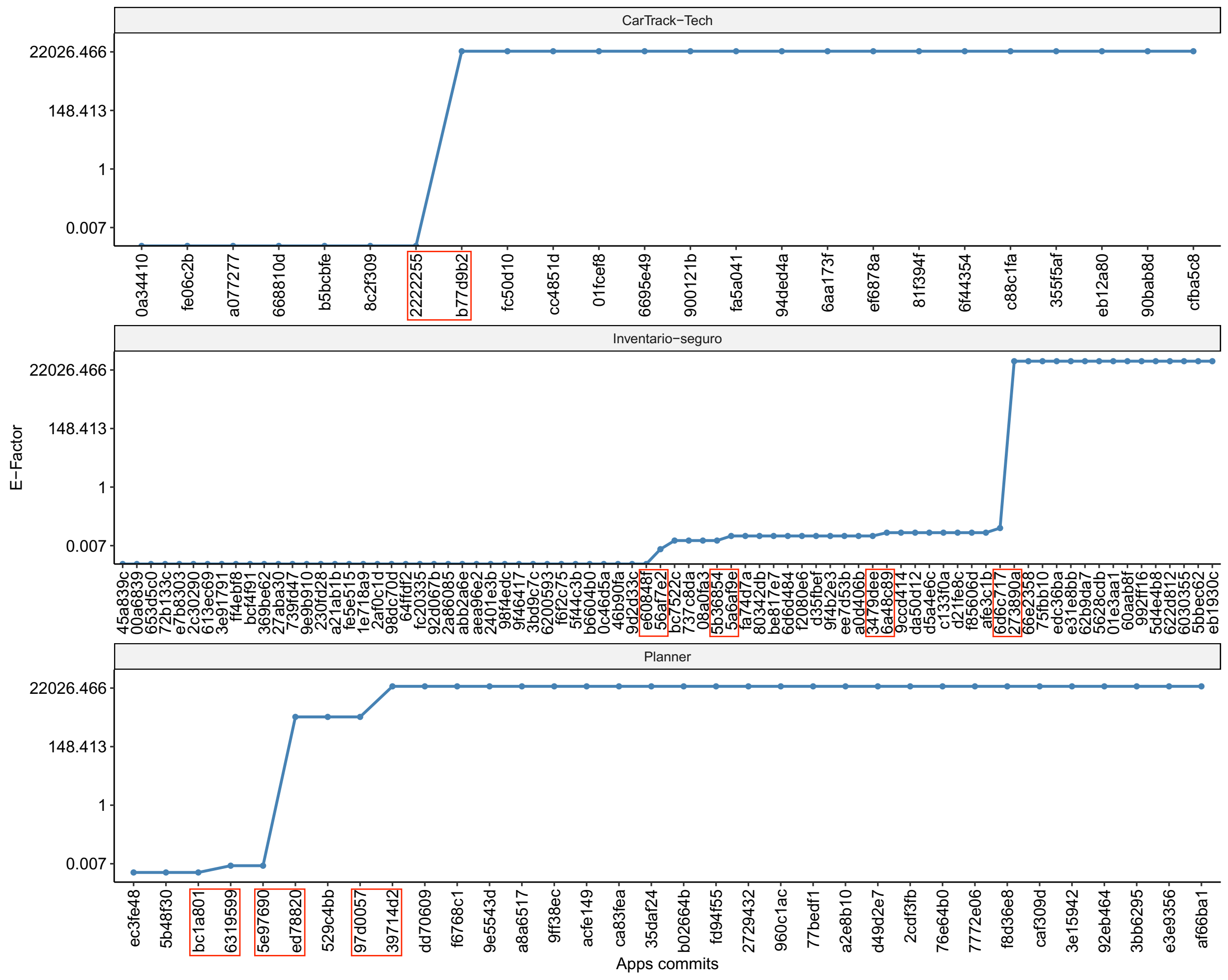


Figure 6.5: E-factor for commits from CarTrack, Inventario, and Planner. The commits are ordered left-to-right in sequence over time (i.e., oldest to latest commit). Commits whose E-factor change over time are highlighted in a red box. The number of iterations assumed for the SQLite usages that are in a loop is $10^4$.

### 6.6.2 E-factor for Comparing Versions (RQ1)

#### Setup

In RQ1, we investigate if developers can use E-factor to compare the energy consumption of an API's usage between the various versions of an app.

*Commit Selection:* To consider an app from our dataset for RQ1, it has to be supported by our energy measurement setup (iGreenMiner) by fulfilling the following criteria: (1) should have a test-suite with UI tests, (2) should not depend on a third party web API to execute (e.g., Firestore), (3) should not require login credentials from a web service that we do not have access to, (4) should not run into runtime errors, (5) and should have an API use inside a loop or a UI event. Applying these criteria, we have 3 apps to investigate:

- CarTrack [58]: This app registers people with their profiles for record keeping. We identified 10 `insert` and 7 `select` operations in this app.
- Inventario [121]: This app keeps an inventory list. We identified 8 `insert` and 5 `select` operations in this app.
- Planner [57]: This is a doodle note-taking app that users may use to save a note on each calendar day. We identified 2 `insert`, 3 `select`, and 1 `update` operations in this app.

We then extracted 34 commits from CarTrack, 83 commits from Inventario, and 35 commits from Planner to calculate their E-factor. We could not build 16/152 commits because their code was insufficient to build the app. Eventually, we calculated E-factor for 136 commits in total. Figure 6.5 shows the app-level E-factor of each commit when an SQLite use would execute $10^7$ times. The figure presents how E-factor evolves over time based on the SQLite usage in an app. Calculating E-factor for all commits was relatively straightforward because it does not require any test case execution. However, gathering hardware-based energy measurements (i.e., ground truth) for all commits was unrealistic. This is because, for each commit, we would need to instrument it, then generate and execute its test cases. Following the methodology of Romansky et al. [147] for effectively mining commits to measure energy consumption, we selected commits whose SQLite usage changed over time. Therefore, we chose 16 commits for hardware-based energy measurement (highlighted in red boxes along the x-axis in Figure 6.5). The SHAs for these commits are:

- CarTrack: `2222255` and `b77d9b2`.
- Inventario: `e60848f`, `56af7e2`, `5b36854`, `5a6af9e`, `3479dee`, `6a48c89`, `6d6c717`, and `273890a`.
- Planner: `bc1a801`, `6319599`, `5e97690`, `ed78820`, `97d0057`, and `39714d2`.

*Commit Instrumentation:* To execute each commit on iGreenMiner, we had to manually configure the dependencies of each commit, especially older commits that rely on obsolete dependencies. This instrumentation required instrumenting the UI navigations and authentications in the apps to override business rules and methods such that all SQLite operation methods would execute within a single test-case execution. For instrumentation, we chose $10^4$ iterations for the loops and UI events having SQLite operations because this is where E-factor starts to noticeably increase (Figure 6.4). It was also unrealistic to measure for more than $10^4$ iterations because, at that point, the highest energy-consuming app, Inventario, takes $\approx 57$ hours to execute.

*Hardware Energy Measurement:* To accurately measure the energy consumption of each commit, we ran the 16 instrumented versions on iGreenMiner multiple times. We execute the versions that took less than an hour 10 times, and those that took an hour or more 5 times. We compare these versions' energy measurements with their E-factor.

**Results**

To compare E-factor to hardware-based energy measurements of the commits, we use the Spearman's Rank Correlation with the Bonferroni correction method [175]. The Spearman's Rank Correlation reports $\rho = 0.60$ with a $p$-value $< 0.05$, indicating a significant correlation between E-factor and hardware-based energy measurements of the commits. Using Hopkin's scale [91], $\rho = 0.60$ indicates that if a commit has a lower E-factor than another, then it is more energy-efficient due to the large positive correlation between E-factor and energy measurements. Given this significant positive correlation, E-factor is a reliable indicator for developers to determine if their latest code change related to the SQLite API usage would result in an increase or decrease in energy consumption compared to a previous version of their app.

Table 6.4: Comparing E-factor to hardware-based energy measurements for methods from real-world apps. The number of iterations assumed for the SQLite usages that are in a loop is $10^4$.

| App | Method | Operation | Ground Truth (joules) | E-factor (joules) |
|---|---|---|---|---|
| CarTrack | `CountryRepo.-createCountry-Data()` | `insert*` | 62.27 | 22.2504 |
| CarTrack | `CountryRepo.-init(...)` | 1 `select` | 36.93 | 0.0002 |
| Inventario | `VC.dummys()` | 1 `insert` + 1 `select` | 4.03 | 0.0027 |
| Inventario | `DBUsuarioHelper-.insert(..)` | 1 `select` | 2.85 | 0.0002 |
| Inventario | `LoginVC.create-DummyUsers()` | 5 `insert` + 1 `select` | 3.25 | 0.0124 |
| Inventario | `LoginVC.do-Login(..)` | 1 `select` | 3.37 | 0.0002 |
| Inventario | `mostrarVC.Insert-Registros(..)` | `insert*` + 1 `select` | 8446.95 | 22.2504 |
| Inventario | `mostrarVC.Insert-Rollos()` | `insert*` + 1 `select` | 3899.09 | 22.2504 |
| Planner | `CanvasVC.create-NewCanvas()` | 1 `insert` | 1.63 | 0.0025 |
| Planner | `CanvasVC.load-Canvas()` | 1 `select` | 1.74 | 0.0002 |
| Planner | `CanvasVC.drawing-DidChange(...)` | `update*` + `select*` | 11.30 | 27.4114 |

### 6.6.3 E-factor for Fine-grained Energy Review (RQ2)

For a given API usage in an app, if a developer discovers that their app consumes too much energy, they may use E-factor to compare the energy consumption of different methods within the app that use that API. In this research question, we investigate whether E-factor provides this information to developers to help them focus their optimization efforts on specific methods.

#### Setup

For this investigation, we consider methods of the same apps that we used in RQ1: CarTrack, Inventario, and Planner. We have already calculated the E-factor of these apps and their methods. However, to collect the hardware-based energy measurements (i.e., ground truth) of these methods, we must first instrument the apps so that each method executes in isolation.

*Method Selection:* CarTrack has 8 methods that perform SQLite operations. For evaluation, we consider `CountryRepo.createCountryData()` that performs an `insert` in a loop. From the remaining methods that perform a single `select`, we could only instrument `CountryRepo.-init(...)` to execute in isolation. Inventario has 6 methods that perform SQLite operations. `LoginVC.createDummyUsers()` performs 5 `insert` operations and 1 `select`. `mostrar-VC.InsertRollos()` and `mostrarVC.InsertRegistros(..)` perform `insert` in a loop and 1 `select`. `VC.dummys()` performs 1 `insert` and 1 `select`, while `DBUsuarioHelper-.insert(..)` and `LoginVC.doLogin(..)` perform 1 `select`. We consider all these methods in our evaluation. Planner has 5 methods that perform SQLite operations. Out of the four methods that perform `insert` and `select` once, we consider `CanvasVC.createNewCanvas()` and `CanvasVC.loadCanvas()`. Finally, we consider `CanvasVC.drawingDidChange(...)` that performs a `select` and an `update` in a loop.

*App Instrumentation:* We created 2 instrumented versions for CarTrack, 6 instrumented versions for Inventario, and 3 instrumented versions for Planner. Similar to RQ1, each instrumented version executes a method $10^4$ times if it would occur in a loop or a UI-event and once otherwise.

*Hardware Energy Measurement:* To collect the ground truth, we executed each instrumented version 10 times on iGreenMiner and collected their energy consumption measurements. Since each instrumented version represents a method, to compare a method's energy consumption with its E-factor, we compare the instrumented versions' energy measurements with their representative method's E-factor value.

## Results

Table 6.4 compares E-factor to the ground truth of each method. The table shows that E-factor reports the highest values for the most energy-consuming methods in an app: `CountryRepo.-createCountryData()` in CarTrack, `mostrarVC.InsertRegistros(..)` and `DBUsuarioHelper-.insert(..)` in Inventario, and `CanvasVC.drawingDidChange(...)` in Planner.

To determine the relationship between E-factor and hardware-based energy measurements, we used the Spearman's Rank Correlation with the Bonferroni correction method. The Spearman's Rank Correlation reports $\rho = 0.59$ with a $p$-value $< 0.05$, indicating a significant correlation between E-factor and hardware-based energy measurements of the evaluated methods. Using Hopkin's scale [91], $\rho = 0.59$ indicates that if a method has a lower E-factor than another, it is more energy-efficient due to the large positive-correlation between E-factor and energy measurements.

Given this significant positive correlation, developers may reliably use E-factor to compare the energy consumption of API usage within methods in their code without executing them.

### 6.6.4 E-factor's Automation Capability (RQ3)

In RQ3, we want to verify if EC calculates E-factor accurately. To verify the accuracy, we compare EC with the manual computation of E-factor.

#### Setup

For this evaluation, we used 12 apps (3 from each category) with the minimum, average, and maximum automatically calculated E-factor values:

- Entertainment: InstagramProjectIos, Instant, and Priyo Movies
- Healthcare: HealthKitTest, iOS-App-Project, and Careapp
- Utility: Dulce, Storage, and Planner
- Uncategorized: TOURUS, THISAPPTEAM, and Hybrid-App

To manually compute E-factor (i.e., ground truth), we first traversed through the code files of each app. We then identified method-to-API mappings that are reachable to execute during runtime.

#### Results

To compare the automatically calculated EC E-factor values with the manually calculated E-factor, we compute the mean absolute percentage error (MAPE) [65] between both. MAPE indicates that the E-factor values calculated by EC have an average absolute error of $16.84\%$ compared to the actual values. In comparison, the state-of-the-art approach, as presented in the work of Chowdhury et al. [53], has an error margin of $10\%$ for its energy prediction model. Although E-factor has a $6.84\%$ higher error rate compared to the state-of-the-art approach, it is still considered acceptable because, unlike the state-of-the-art, it does not require the generation of test cases or the use of specific hardware for execution and energy measurement. This quality of E-factor makes it a convenient alternative that estimates reasonably accurate energy consumption without additional resources.

While Saborido et al. [148] found that 125 kHz is an ideal sampling rate for energy measurement in Android apps, they also noted that the sampling rate requirement for determining an app's energy consumption is different from that for its methods. A 5 kHz sampling rate for apps is acceptable as it produces a median error of 13.27%. However, a 5 kHz sampling rate is insufficient for methods, as the error can increase to 53.15% with a median of 0.54%. To ensure reliable results, we recommend that users of iGreenMiner instrument their app to execute one method at a time in isolation, execute each method multiple times, and consider the median energy consumption of each method.

To investigate the reason behind the error, we compare the manually detected method-to-API mappings with EC detected mappings. Upon this investigation, we found that EC identified the mappings with a precision of 0.91 and a recall of 0.84 with an F1 score of 0.87. These measurements show that EC is less likely to provide false positives but may miss some true positives.

Upon further investigation, we have also identified the following reasons for the occurrence of false positives (i.e., reporting a mapping that does not exist) and false negatives (i.e., not reporting a mapping that exists):

- EC could not identify UI actions in input apps if the responsible action implements a method other than `tapGesture` or `onClick` from the Swift UI framework.
- Some identified execution paths from the CG are infeasible because they do not execute at runtime.
- The underlying framework that generates the CG (SWAN) does not support recursion or Swift structs.

## 6.7 Threats to Validity

In this section, we discuss the threats to the validity of our results and how we addressed them.

### 6.7.1 Construct Validity

To find call sites that use SQLite queries, we construct a CG using CHA [66]. Many other CG construction algorithms, such as RTA [28] and XTA [164], are more precise than CHA, and each of these algorithms may yield different results for E-factor. However, we have chosen CHA to quickly estimate the energy consumption estimates for the developers.

### 6.7.2 Internal Validity

To measure the energy profile of the SQLite API, we pre-populated the DB records. However, our DB records may not represent real-world scenarios because we keep the initial database empty each time before executing the `insert` benchmarks and pre-populate $10^7$ records before executing the `select` and `update` benchmarks. Having a different number of records affects the runtime performance of SQLite differently [49], and realistically, users perform SQLite operations on a variable number of records. For better SQLite energy profiling, future work may adopt parallelization to run multiple instances of a DB on various phones, where each DB will have a random number of records, columns, and indexes. Regardless, we believe that the relative difference between `insert`, `select`, and `update` may stay consistent when a DB has the same structure and number of records to start with. However, this claim requires an empirical evaluation that is out of the scope of this chapter.

We have designed our approach to consider only two usage contexts for SQLite operations: those that occur inside a loop and those that occur outside of it. However, SQLite operations may occur in many other contexts, such as inside a branch condition or an asynchronous thread. In this study, we assume that all branches execute and do not consider the other possible contexts. Therefore, we are unable to provide a best-case or even an average-case energy estimate. Instead, we provide worst-case estimates only. Similarly, an operation within a loop can be executed a varying number of times based on the user's requirements. In order to give developers an indication of the potential energy consumption of a particular API usage during runtime, without requiring actual runtime information, we present them with energy consumption projections for a range of executions, from $10^1$ to $10^7$. It is important to note that when providing estimates for all the loops in an app, we assume that all operations within a loop would execute an equal number of times.

The SIL files that we analyze represent the developer's code only, and therefore our E-factor values are confined to the developer's code. We do not calculate the E-factor of third-party libraries on which an app or an API call may depend. However, this can be done by generating and analyzing the SIL files for the external libraries. Nonetheless, this is out of the scope of this chapter, and we leave it for future work.

In Table 6.4, the E-factor value of `CanvasVC.drawingDidChange(...)`, in which two different SQLite operations in a loop run in combination, is larger than the ground-truth value. This inaccuracy might occur because we had separately profiled the energy consumption of `update` and `select`, and we never profiled their combinatorial effect. Operations, when executed in combination, may follow several architecture-level optimizations. Therefore, for more accurate E-factor values, different operations combinations should also be profiled.

### 6.7.3 External Validity

For hardware-based energy measurements, we utilized iGreenMiner, which supports iPhone 11 running iOS 13.4.1. As a result, the energy profile and results of our SQLite API can be generalized to all iOS apps designed for iOS 13.4.1 and running on iPhone 11. The generalizability of our obtained SQLite API's energy profile may be influenced by various factors in newer or older phones, including processor speed, power management architecture, battery capacity, operating system version, and underlying hardware features. However, conducting such an evaluation is beyond the scope of our contribution. It is important to note that while certain factors may impact the generalizability of E-factor's calculation, several other factors remain relatively consistent across different device models. These factors include the energy measurement mechanism for energy profile collection, the code analysis technique using call graphs, and the energy model of E-factor based on software rather than hardware characteristics.

A different smartphone device and operating system may yield a different energy profile for the SQLite API. Therefore, our energy profile and results may not generalize over all smartphone platforms and devices that run them. To claim the generalization of our results over all iOS apps, the energy profile has to be obtained using other iOS versions. In future, researchers can train a prediction model based on the characteristics of APIs to auto-generate API energy profiles, this will increase the coverage of E-factor.

Our study primarily focuses on the Swift SQLite API, which leverages the processing subsystem of a smartphone. While we do not directly investigate other energy-intensive APIs such as networking, location, and graphics in this chapter, the insights and methodology derived from our research serve as a valuable foundation for estimating energy consumption in API usage.

The SQLite API shares similarities with other database-related APIs, including MySQL, PostgreSQL, Oracle Database, and Microsoft SQL Server, as it provides a structured way to interact with a relational database management system (RDBMS). Developers can use the SQLite API for common database operations like table creation, data manipulation, and executing SQL queries, similar to these other APIs. However, SQLite differs in its serverless nature, operating directly on the local file system without needing a separate server process. Additionally, it employs a unique storage model where the entire database is stored as a single file, enabling self-contained databases. Consequently, it becomes essential for the researchers to develop energy profiles for other RDBMSs beyond SQLite.

E-factor provides the API usage energy estimates while assuming that the API use will execute for a range of iterations. Future work may better estimate the runtime iterations of loops through static analysis techniques that use abstract interpretation [117] or path dependency automation [171].

## 6.8 Conclusion

The current hardware and software-based measurement and estimation techniques are cumbersome because they require developers to own a smartphone and generate and execute test cases. As a solution, this chapter proposes a static-analysis-based approach that can estimate the energy consumption (E-factor) of API usage in an app without the need to generate and execute test cases. We evaluate the approach on real-world apps that use the SQLite API and find out that E-factor is a reliable estimator to compare the relative energy consumption of API use between the versions of an app and within the methods of an app.

The proposed approach offers the advantage of providing energy estimates at compile time, which alleviates the burden on developers of executing test cases after each code modification or before releasing a new version. However, it should be noted that the success of this approach relies on having energy profiles for the APIs. The collection of energy profiles requires having and executing test cases while measuring energy consumption with hardware or a profiler. We posit that energy profile collection is a one-time process. Once generated, the profile can be utilized for all iOS applications that utilize a specific API, iOS version, and iPhone version. This allows for reusability and efficiency in estimating the energy consumption of various applications.

Additionally, this approach provides an opportunity for integration with existing Integrated Development Environments (IDE)s or Continuous Integration/Continuous Deployment (CI/CD) pipelines, providing automated energy testing as a part of the development process. Such integration would reduce the costs associated with energy testing and improve the overall development process by identifying and fixing energy-inefficient code early on in the development cycle. In future, E-factor's estimation can be improved by adding more APIs' energy profiles to its computation.

# Chapter 7

# Conclusion

To provide a variety of functionalities, smartphone apps use hardware components, such as memory and CPUs for computation, GPS for navigation, Bluetooth for connectivity, and a camera for taking pictures. Therefore, an inefficient app with energy bottlenecks in its code can make the hardware components over-consume energy, leading to rapid drainage of a phone's battery. Such inefficient apps get negative user reviews and often get removed from the phone. As a result, developers are continuously looking for ways to optimize and measure the energy consumption of their apps.

Current techniques for optimizing energy efficiency in software development include design patterns [55], UI guidelines [116], refactoring techniques [30], API choice guidelines [83], and search-based algorithms [120]. However, these optimization techniques focus on developer-written instructions and system events rather than API-level usage [110], while API-usage accounts for 85% of an app's energy consumption. Some existing techniques that focus on API-level usage [83], [120] are time-consuming because they require test-case execution and they do not provide guidance on how specific API parameter configurations affect energy consumption. Since API-usage accounts for a large portion of an app's energy consumption, API-parameter-level information is crucial for developers to optimize their code for energy-efficient API usage. On the other hand, current energy measurement techniques include hardware-based tools, such as Greenminer [87], and software-based energy prediction and estimation models, such as Greenscaler [53], Android Profiler [79], XCode's Log Instrument Profiler [92], and JProfiler [70]. Hardware-based techniques require specialized knowledge and expensive equipment, while software-based techniques rely on test-case generation and execution, which can be time-consuming and resource-intensive. Additionally, generating and executing test cases requires a smartphone and a script to collect runtime information, which may take an average of several minutes, making real-time updates impractical for energy-aware developers.

To address the current limitations of energy optimization and measurement techniques, this thesis demonstrates that static analysis can optimally configure an app's API parameters, significantly reducing its energy consumption. Moreover, this thesis presents a static analysis technique that can effectively estimate the relative energy consumption of an app's API usage without requiring the execution of test cases. In particular, this thesis makes four contributions.

The first contribution in Chapter 3 evaluates the practicality of current state-of-the-art techniques, such as search-based energy optimization and test-case execution-based energy measurement. This evaluation demonstrates that the idea of test-case execution to provide app-developers with real-time energy insights for their code modifications is impractical because of long execution times.

The second contribution in Chapter 4 is the development of an open-source iOS energy measurement framework called iGreenMiner. iGreenMiner is the first hardware-based energy measurement framework available for iOS, allowing developers to measure the absolute energy consumption of their iOS apps. The framework also enables researchers to collect energy datasets for iOS apps. Researchers may use it to build energy estimation and prediction models and identify energy greedy patterns specific to iOS. This contribution addresses a gap in the existing tools and solutions for iOS app development, which do not provide absolute energy measurements and are limited in their ability to identify energy consumption problems specific to the iOS platform. Researchers can address several problems related to energy-efficient iOS app development using the iGreenMiner framework. These include designing energy-efficient user interfaces, improving energy reporting tools, empirically evaluating Apple's energy-aware development guidelines, building a static energy prediction model, and evaluating the energy consumption of cross-platform iOS apps.

The third contribution in Chapter 5 proposes energy-efficient configuration guidelines for the iOS Core Location framework to reduce an app's energy consumption. The guidelines help developers make informed design choices for accessing user-location in their apps. This contribution provides location framework benchmarks and their energy profiles and analyzes them to extract energy-efficient guidelines for the developers. The analysis shows that, overall, energy-wise, the most efficient service is *Visits Location Service*, while *Standard Location Service* is the most expensive in energy consumption. The contribution also evaluates the effectiveness of these guidelines on three real-world apps and a benchmark app that Apple provides to its developers. The results show that these guidelines can reduce energy consumption by up to 26.91% on real-world apps and 11.37% on the benchmark app. Future work in this area could include applying the proposed techniques to explore energy-efficient development guidelines for other APIs, such as the Bluetooth API or the Camera API.

The fourth contribution in Chapter 6 proposes an energy estimation model based on static analysis to analyze an app's code and estimate the energy consumption of its API usage; the energy estimates are called E-factor. E-factor values can be calculated within milliseconds using static analysis, which is faster than the traditional approach. The benefit of this approach is that it relieves the developers from expensive, sophisticated hardware and saves their expenditure cost on energy testing. Furthermore, it saves developers time so that they do not have to maintain or execute any test cases for the energy estimation of their app. The E-factor is a reliable estimate to compare the relative energy consumption between versions of an app and between methods of an app.

## 7.1 Future Work

This thesis demonstrates the benefits of using an energy profile to estimate energy consumption. However, the capabilities of the energy profile defined in this thesis are limited in terms of context (task inside a loop or out-of-loop), platform (iOS), API (SQLite API), and device (iPhone 11). As a solution, this thesis takes an opportunity to make an open call to the community of Software Engineering to collect and submit their energy profiles on a GitHub repository [32]. The more energy profiles for different APIs we have, and the more contexts they are evaluated in, the more precise the energy estimates (E-factor) will be. In the future, with many energy profiles, we can achieve absolute energy estimates instead of relative energy estimates.

Once we have multiple energy profiles, we can use the base energy cost of their tasks and develop a weight-based estimation model. This model would assign weights to each API task based on its base cost. These weights can later be used to rank and suggest, in comparison to each other, how energy-efficient or inefficient certain apps are on an Appstore, certain versions are in a repository, and certain methods are in an app.

To understand further, let us assume two apps whose energy consumption has to be compared without execution, and both of these apps use SQLite for storage persistence, iOS Foundation API [25] for network-related requests, and iOS AVFoundation API [23] to execute media files. First of all, we will have to collect an energy profile for each of these APIs. For SQLite, we can use the energy profile from Chapter 6 that represents the base energy consumption of its operations, such as `select`, `update`, `insert`. To profile Foundation API, we can profile the base energy consumption for making a basic GET request and a basic POST request. To profile AVFoundation API, we can profile the energy consumption for executing basic video and audio files. Once we have the energy profiles for all of the APIs, we can assign each API task a weight, based on how expensive it is as compared to the other tasks in other APIs. Let us assume that AVFoundation's video execution was assigned a weight of 1.0 for being the most energy-consuming task among all API tasks, Foundation's POST request was assigned a weight of 0.8, Foundation's GET request was assigned a weight of 0.6, and SQLite's `select` was assigned a weight of 0.4 for being the least energy consuming task. Using these weights, the developer can compare the energy consumption of different apps and app versions and choose the most energy-efficient version. This weight-based approach can further be improved by considering the context in which each API-task was profiled.

## 7.2 Summary

In conclusion, this thesis provides a stepping stone toward a new direction in energy-efficient app development by proposing static-analysis techniques to estimate energy consumption. The contributions of this work include the development of an open-source iOS energy measurement framework, energy-efficient configuration guidelines for the iOS Core Location framework, and a static analysis-based energy estimation model called E-factor. While this work has shown promising results, much is still to be done in this area. We advocate that researchers invest their energies in improving the models by contributing energy profiles and exploring energy-efficient development guidelines for other APIs. Collaboration and ongoing research are crucial to improving energy-efficient app development practices and reducing energy consumption in mobile devices, which will save time in execution, effort in dealing with sophisticated hardware and generating test-cases, and money in terms of affording expensive hardware.

# References

[1] T. Agolli, L. Pollock, and J. Clause, “Investigating Decreasing Energy Usage in Mobile Apps via Indistinguishable Color Changes,” in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, IEEE, 2017, pp. 30–34.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] L. Almagor, K. D. Cooper, A. G. T. J. Harvey, S. R. D. Subramanian, L. Torczon, and T. Waterman, “Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms,” Technical report, Los Alamos Computer Science Institute, Tech. Rep., 2003.

[4] Android, *Android APK-Analyzer*, 2021. [Online]. Available: `https://developer.android.com/studio/debug/apk-analyzer`.

[5] AndroidDoc, *Save data in a local database using Room*, 2022. [Online]. Available: `https://developer.android.com/training/data-storage/room`.

[6] AppleInc, *Breadcrumb: Using CoreLocation to track user movement*, 2018. [Online]. Available: `https://developer.apple.com/library/archive/samplecode/Breadcrumb`.

[7] AppleInc, *What’s New in Energy Debugging*, 2018. [Online]. Available: `https://developer.apple.com/videos/play/wwdc2018/228/?time=265`.

[8] AppleInc, *About iPhone 11*, [Specifications], 2019. [Online]. Available: `https://www.apple.com/by/iphone-11/specs/`.

[9] AppleInc, *Improving Battery Life and Performance*, [WWDC], 2019. [Online]. Available: `https://developer.apple.com/videos/play/wwdc2019/417/?time=405`.

[10] AppleInc, *About iOS 13 Updates*, [ios13.4.1], 2020. [Online]. Available: `https://support.apple.com/en-ca/HT210393`.

[11] AppleInc, *Energy Guide iOS*, 2020. [Online]. Available: `https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/`.

[12] AppleInc, *Getting Users Location*, [Online; accessed 17-Feb-2021], 2020. [Online]. Available: https://developer.apple.com/documentation/corelocation/getting_the_user_s_location.

[13] AppleInc, *Location and Maps Programming Guide*, [Online; accessed 17-Feb-2021], 2020. [Online]. Available: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/LocationAwarenessPG/CoreLocation/CoreLocation.html.

[14] AppleInc, *Region Monitoring Service*, [Online; accessed 17-Feb-2021], 2020. [Online]. Available: https://developer.apple.com/documentation/corelocation/monitoring_the_user_s_proximity_to_geographic_regions.

[15] AppleInc, *Significant Location Service*, [Online; accessed 17-Feb-2021], 2020. [Online]. Available: https://developer.apple.com/documentation/corelocation/getting_the_current_location_of_a_device.

[16] AppleInc, *Standard Location Service*, [Online; accessed 17-Feb-2021], 2020. [Online]. Available: https://developer.apple.com/documentation/corelocation/getting_the_current_location_of_a_device.

[17] AppleInc, *Visit Location Service*, [Online; accessed 17-Feb-2021], 2020. [Online]. Available: https://developer.apple.com/documentation/corelocation/getting_the_current_location_of_a_device.

[18] AppleInc, *Avoid Extraneous Graphics and Animations*, [Graphics Guide], 2021. [Online]. Available: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/AvoidExtraneousGraphicsAndAnima
html.

[19] AppleInc, *iOS 14 features reimagined*, 2021. [Online]. Available: https://www.apple.com/ca/ios/ios-14/.

[20] AppleInc, *Core Location Framework*, [Online; accessed 25-Oct-2022], 2022. [Online]. Available: https://developer.apple.com/documentation/corelocation.

[21] AppleInc, *Apple App Store*, 2023. [Online]. Available: https://www.apple.com/app-store/.

[22] AppleInc, *Apple's integrated development environment (IDE)*, 2023. [Online]. Available: https://developer.apple.com/xcode/.

[23] AppleInc, *AVFoundation Framework*, 2023. [Online]. Available: https://developer.apple.com/documentation/avfoundation.

[24] AppleInc, *Create powerful experiences*, [Offcial], 2023. [Online]. Available: https://developer.apple.com/ios/.

[25] AppleInc, *Foundation Framework*, 2023. [Online]. Available: https://developer.apple.com/documentation/foundation.

[26] AppleInc, *iOS Official Testing Framework*, 2023. [Online]. Available: https://developer.apple.com/documentation/xctest.

[27] AppleInc, *Swift Intermediate Language (SIL)*, [Swift-IR], 2023. [Online]. Available: `https://github.com/apple/swift/blob/main/docs/SIL.rst`.

[28] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996, pp. 324–341.

[29] S. Baluja and R. Caruana, "Removing the Genetics from the Standard Genetic Algorithm," in *Machine Learning Proceedings 1995*, Elsevier, 1995, pp. 38–46.

[30] A. Banerjee and A. Roychoudhury, "Automated Re-factoring of Android Apps to Enhance Energy-Efficiency," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 139–150.

[31] T. Banerjee and S. Ranka, "A genetic algorithm based autotuning approach for performance and energy optimization," in *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, IEEE, 2015, pp. 1–8.

[32] Bangash, *E-Factor Calculation Program*, Jan. 2023. [Online]. Available: `https://zenodo.org/record/7262615`.

[33] A. A. Bangash, K. Ali, and A. Hindle, "A Black Box Technique to Reduce Energy Consumption of Android Apps," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1–5.

[34] A. A. Bangash, K. Eng, Q. Jamal, K. Ali, and A. Hindle, "Energy Consumption Estimation of API-usage in Smartphone Apps via Static Analysis," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories*, 2023.

[35] A. A. Bangash, H. Sahar, S. Chowdhury, A. W. Wong, A. Hindle, and K. Ali, "What do developers know about machine learning: A study of ml discussions on stackoverflow," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 260–264.

[36] A. A. Bangash, H. Sahar, A. Hindle, and K. Ali, "On the time-based conclusion stability of cross-project defect prediction models," *Empirical Softw. Engg.*, vol. 25, no. 6, pp. 5047–5083, Nov. 2020, ISSN: 1382-3256.

[37] A. A. Bangash, D. Tiganov, K. Ali, and A. Hindle, "Energy Efficient Guidelines for iOS Core Location Framework," in *Proceedings of the 2021 International Conference on Software Maintenance and Evolution (ICSME)*, Luxembourg City, Luxembourg, Jun. 15, 2021, pp. 1–12.

[38] R. J. Behrouz, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "Ecodroid: An approach for energy-based ranking of android apps," in *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*, IEEE, 2015, pp. 8–14.

[39] P. Berube, A. Preuss, and J. N. Amaral, "Combined Profiling: Practical Collection of Feedback Information for Code Optimization," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, 2011, pp. 493–498.

[40] C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker, "How is Performance Addressed in DevOps?" In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19, Mumbai, India: Association for Computing Machinery, 2019, pp. 45–50, ISBN: 9781450362399.

[41] K. H. Bindu, R. Morusupalli, N. Dey, and C. R. Rao, *Coefficient of Variation and Machine Learning Applications*. CRC Press, 2019.

[42] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, "A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–34, 2018.

[43] T. Blickle, "Tournament selection," *Evolutionary computation*, vol. 1, pp. 181–186, 2000.

[44] J. Bloch, "How to Design a Good API and Why It Matters," ser. OOPSLA '06, Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 506–507.

[45] M. Brenna, F. Foiadelli, and M. Longo, "Application of Genetic Algorithms for Driverless Subway Train Energy Optimization," *International Journal of Vehicular Technology*, vol. 2016, 2016.

[46] L. C. Briand, "A Critical Analysis of Empirical Research in Software Testing," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, IEEE, 2007, pp. 1–8.

[47] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 1327–1334.

[48] N. Burles, E. Bowles, A. E. Brownlee, Z. A. Kocsis, J. Swan, and N. Veerapen, "Object-oriented genetic improvement for improved energy consumption in Google Guava," in *International Symposium on Search Based Software Engineering*, Springer, 2015, pp. 255–261.

[49] M. J. Carey and D. Kossmann, "On Saying "Enough Already!" In SQL," *SIGMOD Rec.*, vol. 26, no. 2, pp. 219–230, Jun. 1997, ISSN: 0163-5808.

[50] S. Celis, *SQLite Swift Wrapper*, 2022. [Online]. Available: `https://github.com/stephencelis/SQLite.swift`.

[51] J. Chen and W. Shang, "An Exploratory Study of Performance Regression Introducing Code Changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 341–352. DOI: `10.1109/ICSME.2017.13`.

[52] X. Chen, Y. Chen, Z. Ma, and F. C. A. Fernandes, "How is Energy Consumed in Smartphone Display Applications?" In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '13, Jekyll Island, Georgia: Association for Computing Machinery, 2013.

[53] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, "GreenScaler: Training Software Energy Models with Automatic Test Generation," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1649–1692, 2019.

[54] S. A. Chowdhury and A. Hindle, “Greenoracle: Estimating software energy consumption with energy measurement corpora,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2016, pp. 49–60.

[55] S. A. Chowdhury, A. Hindle, R. Kazman, T. Shuto, K. Matsui, and Y. Kamei, “GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing,” in *Proceedings of the 41st International Conference on Software Engineering*, IEEE Press, 2019, pp. 1107–1118.

[56] J. Chung, H. Jung, J. Koo, Y. Kim, and U.-M. Kim, “A Development of Power Consumption Measurement System for Android Smartphones,” ser. IMCOM ’17, Beppu, Japan: Association for Computing Machinery, 2017.

[57] A. Clare, *Planner App*, 2021. [Online]. Available: `https://github.com/clare228/Planner`.

[58] R. Correia, *Car Track Tech Challenge App*, 2021. [Online]. Available: `https://github.com/ricardo-correia/CartrackTechChallenge`.

[59] M. Couto, J. Saraiva, and J. P. Fernandes, “Energy Refactorings for Android in the Large and in the Wild,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2020, pp. 217–228.

[60] L. Cruz and R. Abreu, “Performance-Based Guidelines for Energy Efficient Mobile Applications,” in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, IEEE, 2017, pp. 46–57.

[61] L. Cruz and R. Abreu, “EMaaS: Energy Measurements as a Service for Mobile Applications,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2019, pp. 101–104.

[62] L. Cruz, R. Abreu, and J.-N. Rouvignac, “Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring,” in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 205–206.

[63] A. Danciu, J. Kroß, A. Brunnert, F. Willnecker, C. Vögele, A. Kapadia, and H. Krcmar, “Landscaping Performance Research at the ICPE and its Predecessors: A Systematic Literature Review,” in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 91–96.

[64] J. P. De Carvalho, B. Kuzma, I. Korostelev, J. N. Amaral, C. Barton, J. Moreira, and G. Araujo, “KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls,” *ACM Transactions On Architecture And Code Optimization (TACO)*, vol. 18, no. 3, pp. 1–22, 2021.

[65] A. De Myttenaere, B. Golden, B. Le Grand, and F. Rossi, “Mean Absolute Percentage Error for regression models,” *Neurocomputing*, vol. 192, pp. 38–48, 2016.

[66] J. Dean, D. Grove, and C. Chambers, “Optimization of object-oriented programs using static class hierarchy analysis,” in *European Conference on Object-Oriented Programming*, Springer, 1995, pp. 77–101.

[67] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?" In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, IEEE, 2017, pp. 103–114.

[68] M. Dmitriev, "Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation," in *Proceedings of the 4th International Workshop on Software and Performance*, 2004, pp. 139–150.

[69] M. Dong, Y.-S. K. Choi, and L. Zhong, "Power Modeling of Graphical User Interfaces on OLED Displays," in *2009 46th ACM/IEEE Design Automation Conference*, IEEE, 2009, pp. 652–657.

[70] ejTechnologies, *Java Profiler - JProfiler*, 2023. [Online]. Available: `https://www.ej-technologies.com/products/jprofiler/overview.html`.

[71] Facebook, *Redex - An Android Bytecode Optimizer*, 2021. [Online]. Available: `https://fbredex.com/`.

[72] E. Falkenauer, "The worth of the uniform [uniform crossover]," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, IEEE, vol. 1, 1999, pp. 776–782.

[73] M. Fitzgerald, *Android vs. iOS App Development: Which Is the Better Choice for Your Business in 2019?* Oct. 2019. [Online]. Available: `https://bit.ly/3nbOCkS`.

[74] K. Georgiou, C. Blackmore, S. Xavier-de-Souza, and K. Eder, "Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '18, Sankt Goar, Germany: Association for Computing Machinery, 2018, pp. 35–42.

[75] K. Georgiou, S. Kerrison, and K. Eder, "On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs," *arXiv*, 2015.

[76] GitHubInc., *A Code Hosting Platform for Version Control and Collaboration.* 2022. [Online]. Available: `https://github.com/`.

[77] GoogleInc., *Androids's integrated development environment (IDE)*, 2023. [Online]. Available: `https://developer.android.com/studio`.

[78] GoogleInc., *Google Play Store*, 2023. [Online]. Available: `https://play.google.com`.

[79] GoogleInc., *The Android Profiler — Android Studio*, 2023. [Online]. Available: `https://developer.android.com/studio/profile/android-profiler`.

[80] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[81] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th international conference on software engineering (ICSE)*, IEEE, 2013, pp. 92–101.

[82] M. Harman and B. F. Jones, "Search Based Software Engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[83] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy Profiles of Java Collections Classes," in *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016, pp. 225–236.

[84] W. Haynes, "Wilcoxon Rank Sum Test," in *Encyclopedia of Systems Biology*, New York, NY: Springer New York, 2013, pp. 2354–2355.

[85] A. Hindle, "Green Software Engineering: The Curse of Methodology," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 5, 2016, pp. 46–55.

[86] A. Hindle, *Pizza Greenminer Service*, Jan. 2022. [Online]. Available: `https://pizza.cs.ualberta.ca/gm/`.

[87] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ACM, 2014, pp. 12–21.

[88] D. R. Hipp, *SQLite Docs*, 2000. [Online]. Available: `https://www.sqlite.org/about.html`.

[89] J. M. Hirst, J. R. Miller, B. A. Kaplan, and D. D. Reed, *Watts up? pro ac power meter for automated energy recording*, 2013.

[90] T.-P. Hong and H.-S. Wang, "A Dynamic Mutation Genetic Algorithm," in *1996 IEEE International Conference on Systems, Man and Cybernetics. Information Intelligence and Systems (Cat. No. 96CH35929)*, IEEE, vol. 3, 1996, pp. 2000–2005.

[91] W. G. Hopkins, "A New View of Statistics," 2002.

[92] A. Inc, *Xcode IDE - Instruments*, 2023. [Online]. Available: `https://developer.apple.com/xcode/features/`.

[93] G. Inc., *Internet hosting service for software development and version control using Git.* [Github], 2023. [Online]. Available: `https://github.com`.

[94] A. Islam, N. Hewage, A. A. Bangash, and A. Hindle, "Evolution of the Practice of Software Testing in Java Projects," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.

[95] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-Based Energy Testing of Android," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 1119–1130.

[96] R. Jabbarvand, F. Mehralian, and S. Malek, "Automated Construction of Energy Test Oracles for Android," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 927–938.

[97] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 425–436.

[98] N. Jaswal, *Property Search App*, 2018. [Online]. Available: `https://github.com/nitinjaswal96/CapstoneProjectPS`.

[99] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the Worst-Case Energy Consumption of Embedded Software," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006, pp. 81–90.

[100] S. K. Jha and E. M. Eyong, "An energy optimization in wireless sensor networks by using genetic algorithm," *Telecommunication Systems*, vol. 67, no. 1, pp. 113–121, 2018.

[101] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, "DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '12, Tampere, Finland: Association for Computing Machinery, 2012, pp. 353–362.

[102] H. Khalid, "On Identifying User Complaints of iOS Apps," in *2013 35th international conference on software engineering (ICSE)*, IEEE, 2013, pp. 1474–1476.

[103] H. Khalid, E. Shihab, and A. E. Hassan, "What Do Mobile App Users Complain About?" *IEEE software*, vol. 32, no. 3, pp. 70–77, 2014.

[104] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What Do Mobile App Users Complain About?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[105] Z. Kholmatova, "Impact of Programming Languages on Energy Consumption for Mobile Devices," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1693–1695.

[106] M. B. Kjærgaard and H. Blunck, "Unsupervised Power Profiling for Mobile Devices," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Springer, 2011, pp. 138–149.

[107] J. Kołodziej, S. U. Khan, L. Wang, A. Byrski, N. Min-Allah, and S. A. Madani, "Hierarchical genetic-based grid scheduling with energy optimization," *Cluster Computing*, vol. 16, no. 3, pp. 591–609, 2013.

[108] P. Kurzweil, "Lithium Battery Energy Storage: State of the Art Including Lithium–Air and Lithium–Sulfur Systems," in *Electrochemical energy storage for renewable sources and grid balancing*, Elsevier, 2015, pp. 269–307.

[109] D. Li and W. G. Halfond, “An Investigation into Energy-Saving Programming Practices for Android Smartphone App Development,” in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014, pp. 46–53.

[110] D. Li, S. Hao, J. Gui, and W. G. Halfond, “An Empirical Study of the Energy Consumption of Android Applications,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 121–130.

[111] X. Li, X. Zhang, K. Chen, and S. Feng, “Measurement and analysis of energy consumption on Android smartphones,” in *2014 4th IEEE International conference on information science and technology*, IEEE, 2014, pp. 242–245.

[112] L. Liao, H. Li, W. Shang, and L. Ma, “An Empirical Study of the Impact of Hyperparameter Tuning and Model Optimization on the Performance Properties of Deep Neural Networks,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, Apr. 2022.

[113] L. Lin and M. Gen, “Auto-Tuning Strategy for Evolutionary Algorithms: Balancing between Exploration and Exploitation,” *Soft Computing*, vol. 13, no. 2, pp. 157–168, 2009.

[114] M. Linares-Vásquez, G. Bavota, C. Bernal, R. Oliveto, M. Di Penta, and D. Poshyvanyk, “Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 2–11.

[115] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, “Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018, ISSN: 1049-331X.

[116] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, “Optimizing energy consumption of guis in android apps: A multi-objective approach,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, pp. 143–154.

[117] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, “A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models,” in *2009 International Symposium on Code Generation and Optimization*, IEEE, 2009, pp. 136–146.

[118] Y. Lyu, A. Alotaibi, and W. G. Halfond, “Quantifying the Performance Impact of SQL Antipatterns on Mobile Applications,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2019, pp. 53–64.

[119] H. Malik, P. Zhao, and M. Godfrey, “Going Green: An Exploratory Analysis of Energy-Related Questions,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, IEEE, 2015, pp. 418–421.

[120] I. Manotas, L. Pollock, and J. Clause, “SEEDS: A Software Engineer’s Energy-Optimization Decision Support Framework,” in *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014, pp. 503–514.

[121] J. Margarita, *Inventario Seguro App*, 2021. [Online]. Available: `https://github.com/ProyectoIntegrador2018/inventario_seguro`.

[122] H. Matalonga, B. Cabral, F. Castor, M. Couto, R. Pereira, S. M. de Sousa, and J. P. Fernandes, "GreenHub Farmer: Real-World Data for Android Energy Mining," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 171–175.

[123] A. Mazuera-Rozo, C. Trubiani, M. Linares-Vásquez, and G. Bavota, "Investigating Types and Survivability of Performance Bugs in Mobile Apps," *Empirical Software Engineering*, pp. 1–43, 2020.

[124] S. McMaster and A. M. Memon, "An Extensible Heuristic-Based Framework for GUI Test Case Maintenance," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, IEEE, 2009, pp. 251–254.

[125] Mikejo5000, *Measure performance in Visual Studio - Visual Studio (Windows)*, 2022. [Online]. Available: `https://learn.microsoft.com/en-us/visualstudio/profiling/?view=vs-2022`.

[126] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Automatically repairing test cases for evolving method declarations," in *2010 IEEE International Conference on Software Maintenance*, IEEE, 2010, pp. 1–5.

[127] S. Monk, *Electronics Cookbook: Practical Electronic Recipes with Arduino and Raspberry Pi*. O'Reilly Media, 2017.

[128] L. Mukhanov, P. Petoumenos, Z. Wang, N. Parasyris, D. S. Nikolopoulos, B. R. De Supinski, and H. Leather, "ALEA: A fine-grained energy profiling tool," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 1, pp. 1–25, 2017.

[129] Mvp, *USB hub per-port power control*, Jan. 2023. [Online]. Available: `https://github.com/mvp/uhubctl`.

[130] M. Nachtigall, L. N. Q. Do, and E. Bodden, "Explaining Static Analysis - A Perspective," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, IEEE, 2019, pp. 29–32.

[131] D. T. Nguyen, "Evaluating Impact of Storage on Smartphone Energy Efficiency," in *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, ser. UbiComp '13 Adjunct, Zurich, Switzerland: Association for Computing Machinery, 2013, pp. 319–324.

[132] E. Noei, F. Zhang, and Y. Zou, "Too Many User-Reviews! What Should App Developers Look at First?" *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 367–378, 2019.

[133] T. Nolte, H. Hansson, and C. Norstrom, "Probabilistic worst-case response-time analysis for the controller area network," in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, IEEE, 2003, pp. 200–207.

[134] O'Dea, *Global smartphone operating system market share 2014-2023*, Feb. 2020. [Online]. Available: `https://www.statista.com/statistics/277048/`.

[135] W. Oliveira, R. Oliveira, and F. Castor, "A Study on the Energy Consumption of Android App Development Approaches," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 2017, pp. 42–52.

[136] W. Oliveira, R. Oliveira, F. Castor, G. Pinto, and J. P. Fernandes, "Improving energy-efficiency by recommending Java collections," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–45, 2021.

[137] Oluwakemi, *Weather Access App*, 2021. [Online]. Available: `https://github.com/kemi-mabel/MyWeatherApp`.

[138] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms," *The Computer Journal*, vol. 58, no. 1, pp. 95–109, 2015.

[139] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the Impact of Code Smells on the Energy Consumption of Mobile Applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019.

[140] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What Do Programmers Know about Software Energy Consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2015.

[141] R. Pereira, P. Simão, J. Cunha, and J. Saraiva, "JStanley: Placing a Green Thumb on Java Collections," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, Montpellier, France: Association for Computing Machinery, 2018, pp. 856–859.

[142] G. Pinto and F. Castor, "Energy Efficiency: A New Concern for Application Software Developers," *Communications of the ACM*, vol. 60, no. 12, pp. 68–75, 2017.

[143] G. Pinto, F. Castor, and Y. D. Liu, "Mining Questions about Software Energy Consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 22–31.

[144] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for Energy Efficiency: A Reflection on the State of the Art," in *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*, IEEE, 2015, pp. 29–35.

[145] A. Puvvala, A. Dutta, R. Roy, and P. Seetharaman, "Mobile Application Developers' Platform Choice Model," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, IEEE, 2016, pp. 5721–5730.

[146] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. Mit Press, 2020.

[147] S. Romansky and A. Hindle, "On Improving Green Mining For Energy-Aware Software Analysis," in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, 2014, pp. 234–245.

[148] R. Saborido, V. V. Arnaoudova, G. Beltrame, F. Khomh, and G. Antoniol, "On the Impact of Sampling Frequency on Software Energy Measurements," in *PeerJ PrePrints*, 2015.

[149] H. Sahar, A. A. Bangash, and M. O. Beg, "Towards Energy Aware Object-Oriented Development of Android Applications," *Sustainable Computing: Informatics and Systems*, vol. 21, pp. 28–46, 2019.

[150] J. Sawin and A. Rountev, "Assumption Hierarchy for a CHA Call Graph Construction Algorithm," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2011, pp. 35–44.

[151] A. Schuler and G. Anderst-Kotsis, "Characterizing Energy Consumption of Third-Party API Libraries using API Utilization Profiles," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[152] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-Compiler Software Optimization for Reducing Energy," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 639–652, 2014.

[153] J. M. Smith, "Optimization Theory in Evolution," *Annual review of ecology and systematics*, vol. 9, no. 1, pp. 31–56, 1978.

[154] M. D. Smith, "Overcoming the Challenges to Feedback-directed Optimization (keynote talk)," in *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, 2000, pp. 1–11.

[155] M. Solutions, *High Voltage Power Monitor*, 2021. [Online]. Available: `https://www.msoon.com/`.

[156] D. Spinellis, "Tool writing: a forgotten art? (software tools)," *IEEE Software*, vol. 22, no. 4, pp. 9–11, Jul. 2005, ISSN: 1937-4194.

[157] SQLite, *Well-Known Users of SQLite*, 2022. [Online]. Available: `https://www.sqlite.org/famous.html`.

[158] SQLite1, *Features of SQLite*, 2023. [Online]. Available: `https://www.sqlite.org/features.html`.

[159] Sumandeep, *Location Utility App*, 2020. [Online]. Available: `https://github.com/Sumandeep2111/FinalProject-ios`.

[160] W. Sun, S. Iwuchukwu, A. A. Bangash, and A. Hindle, "An Empirical Study to Investigate Collaboration Among Developers in Open Source Software (OSS)," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.

[161] Y. Sun, L. Kong, H. A. Khan, and M. G. Pecht, "Li-ion Battery Reliability – A Case Study of the Apple iPhone," *IEEE Access*, vol. 7, pp. 71 131–71 141, 2019.

[162] S. Tarkoma, M. Siekkinen, E. Lagerspetz, and Y. Xiao, *Smartphone Energy Consumption: Modeling and Optimization*, ser. Smartphone Energy Consumption: Modeling and Optimization. Cambridge University Press, 2014.

[163] D. Tiganov, J. Cho, K. Ali, and J. Dolby, "SWAN: A Static Analysis Framework for Swift," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1640–1644.

[164] F. Tip and J. Palsberg, "Scalable Propagation-Based Call Graph Construction Algorithms," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000, pp. 281–293.

[165] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[166] M. Wan, Y. Jin, D. Li, J. Gui, S. Mahajan, and W. G. Halfond, "Detecting Display Energy Hotspots in Android Apps," *Software Testing, Verification and Reliability*, vol. 27, no. 6, e1635, 2017.

[167] M. Weber, K. Christian, S. Florian, S. Apel, and N. Siegmund, "Twins or False Friends? A Study on Energy Consumption and Performance of Configurable Software," in *IEEE/ACM International Conference on Software Engineering*, 2023.

[168] E. W. Weisstein, "Bonferroni correction," *https://mathworld. wolfram. com/*, 2004.

[169] C. Wilke, S. Richly, S. Götz, C. Piechnick, and U. Aßmann, "Energy Consumption and Efficiency in Mobile Applications: A User Feedback Study," in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, IEEE, 2013, pp. 134–141.

[170] T. Wortmann, C. Waibel, G. Nannicini, R. Evins, T. Schroepfer, and J. Carmeliet, "Are Genetic Algorithms Really the Best Choice for Building Energy Optimization?" In *Proceedings of the Symposium on Simulation for Architecture and Urban Design*, 2017, pp. 1–8.

[171] X. Xiaofei, "Static Loop Analysis and Its Applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1130–1132.

[172] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying Diverse Usage Behaviors of Smartphone Apps," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11, Berlin, Germany: Association for Computing Machinery, 2011, pp. 329–344.

[173] D. Yan, G. Xu, and A. Rountev, "Uncovering performance problems in Java applications with reference propagation profiling," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 134–144.

[174] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[175] J. H. Zar, "Spearman rank correlation," *Encyclopedia of biostatistics*, vol. 7, 2005.

[176] C. Zhang and A. Hindle, “A Green Miner’s Dataset: Mining the Impact of Software Change on Energy Consumption,” in *Proceedings of the 11th International Conference on Mining Software Repositories*, 2014, pp. 400–403.

[177] C. Zhang, A. Hindle, and D. M. German, “The Impact of User Choice on Energy Consumption,” *IEEE software*, vol. 31, no. 3, pp. 69–75, 2014.