UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO CIÊNCIA DA COMPUTAÇÃO

Wellington de Oliveira Júnior

Leveraging Design Diversity to Build Energy-Efficient Applications

Recife

2021

Wellington de Oliveira Júnior

Leveraging Design Diversity to Build Energy-Efficient Applications

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador (a): Fernando José Castor de Lima Filho

Coorientador (a): João Paulo Fernandes

Recife

2021

Catalogação na fonte
Bibliotecária Fernanda Bernardo Ferreira, CRB4-2165

O48l Oliveira Júnior, Wellington de
A Leveraging Design Diversity to Build Energy-Efficient Applications / Wellington de Oliveira Júnior. – 2021.
149 f.: il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2021.
Inclui referências.

1. Engenharia de Software e Linguagens de Programação. 2. Consumo de energia. 3. Análise de desempenho. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- CCEN 2021 - 108

Wellington de Oliveira Júnior

“Leveraging Design Diversity to Build Energy-Efficient Applications”

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 31/05/2021.

Orientador: Prof. Dr. Fernando José Castor de Lima Filho

Dedico este trabalho a minha família, meus colegas, meu orientadores e a todos que me ajudaram de alguma forma.

ACKNOWLEDGEMENTS

A vida é um grande efeito borboleta
Todos os detalhes
As pequenas coisas
Tem uma consequência

A princípio, não dá para saber qual
A princípio, não dá para saber porque
A princípio, tudo parece igual

  mas

Tudo que acontece
Por mais insignificante
Faz com a vida seja assim
Do jeito que ela é

Cada momento
Cada instante
Cada sentimento

Tudo que passa pela vida
Tem um peso
Deixa uma marca
Fica na memória

A vida é regida
Nas menores notas
Sinfonia cálida

Agradeço ao destino

E as asas da borboleta
Por me tornarem quem sou

Agradeço do fundo do meu coração ao meu orientador, Fernando Castor, ter trabalhado com você sempre será sempre um dos maiores privilégios da minha vida. E um agradecimento especial por todos esses anos me aguentando e tendo paciência comigo.

Agradeço também meu orientador João Paulo Fernandes, que me recebeu de braços abertos em Portugal, me fazendo sentir em casa desde o meu primeiro dia em solo lusitano. O próximo jogo de futebol é minha responsabilidade.

Gostaria de agradecer aos meus colegas, por todas as dificuldades, as madrugadas de trabalho, as reuniões constantes e especialmente por andar comigo nesse árduo processo. Muito obrigado a todos.

Categoricamente, minha família. Respeitosamente, durante todo o tempo que levei no projeto, sempre do meu lado. É a única certeza que tenho na minha vida. Tenho vocês do meu lado, e estarei sempre do lado de vocês. Isto aqui é para vocês. Nunca duvidem disso. Obrigado.

Agradeço também a todos que cruzaram pela minha vida nessa caminhada. Cada um de vocês, de uma forma ou de outra, contribuiu para que esse projeto se tornasse realidade.

Muito Obrigado.

ABSTRACT

Developing an application with energy consumption in mind may be difficult for a developer. First, because developers may not be familiar with techniques to reduce energy consumption. Second, because it may not be clear when and where these techniques can be applied, since apps with different characteristics require different solutions. Third, because information about energy efficiency is spread throughout multiple sources, making it difficult to make informed decisions. In this thesis, we introduce the concept of energy design diversity and how it can be used by non-specialists developers to build energy optimized applications. Our main insight is that, for many software development issues, there are multiple readily available diversely-designed solutions that have different characteristics in terms of energy consumption. Our objective is to help developers produce more energy efficient code without a significant increase in code complexity. To achieve our objective, we looked into two different aspects that impact the energy consumption of software systems: development approaches and Java collections. Our results when analyzing the different development approaches shows that using hybrid approaches to optimize CPU-intensive snippets for their code may result in an increase in energy efficiency. To compare the different development approaches, we realized empirical experiments on 33 different benchmarks and 3 applications on 5 different devices. Even with small changes the modifications made using JavaScript or C++ instead of Java can significantly reduce energy consumption. Regarding Java collections, we propose an approach for energy-aware development to help non-specialists developers. Using this approach, we implemented our energy saving tool, CT+, using energy profiles to compare the different collections implementations. Across 7 devices, 2295changes were made, achieving up to 16.34% reduction in energy consumption, usually changing a single line of code. Aside from the collections implementations itself, the results points that other factors may heavily influence collections energy optimizations such as: workload, device, development environment, energy profile and battery's age. It is also relevant to point out that some of the most commonly used implementations (`ArrayList`, `Hashtable`, and `HashMap`) can often be replaced with more energy efficient versions, usually from alternative sources to the Java Collections Framework.

Keywords: Energy Consumption. Performance Analysis. Design Diversity. Refactoring. Mobile Applications. Static Analysis.

RESUMO

Desenvolver um aplicativo com o consumo de energia em mente pode ser difícil para um desenvolvedor. Primeiro, porque os desenvolvedores podem não estar familiarizados com as técnicas para reduzir o consumo de energia. Em segundo lugar, porque pode não estar claro quando e onde essas técnicas podem ser aplicadas, uma vez que aplicativos com características diferentes requerem soluções diferentes. Terceiro, porque as informações sobre eficiência energética estão espalhadas por várias fontes, dificultando a tomada de decisões por parte dos desenvolvedores. Nesta tese, apresentamos o conceito de *energy design diversity* e como ele pode ser usado por desenvolvedores para construir aplicativos energeticamente otimizados. O raciocínio é que existem várias soluções já disponíveis com características diferentes em termos de consumo de energia. Nosso objetivo é ajudar os desenvolvedores a produzir código com maior eficiência energética sem um aumento significativo na complexidade do código. Nossos resultados ao analisar as diferentes abordagens de desenvolvimento mostram que o uso de abordagens híbridas para otimizar trechos de uso intensivo de CPU para seu código pode resultar em um aumento na eficiência energética. Mesmo com pequenas alterações, as modificações feitas usando JavaScript ou C ++ ao invés de Java podem reduzir significativamente o consumo de energia. Com relação às coleções Java, propomos uma abordagem para o desenvolvimento energeticamente consciente para ajudar os desenvolvedores não especialistas. Usando essa abordagem, implementamos nossa ferramenta de economia de energia, o CT+, usando perfis de energia para comparar as diferentes implementações de coleções. Em 7 dispositivos, foram feitas alterações 2295, alcançando uma redução de até 16,34 % no consumo de energia, geralmente alterando uma única linha de código. Além das próprias implementações das coleções, os resultados apontam que outros fatores podem influenciar fortemente as otimizações de energia das coleções, tais como: carga de trabalho, dispositivo, ambiente de desenvolvimento, perfil de energia e idade da bateria. Também é relevante apontar que algumas das implementações mais comumente usadas (`ArrayList`, `Hashtable`, e `HashMap`) podem frequentemente ser substituídas por versões mais energeticamente eficientes, geralmente de fontes alternativas ao Java Collections Framework.

Palavras-chaves: Consumo de energia. Análise de desempenho. Diversidade de design. Reestruturação. Aplicações Móveis. Análise estática.

LIST OF FIGURES

LISTA DE CÓDIGOS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADB	Android Debug Bridge
API	Application Programming Interface
CLBG	Computer Language Benchmark Game
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CT+	Collections Energy Consumption Optimization tool Plus
DAQ	Data acquisition systems
EC	Eclipse Collections
EDP	Energy-delay product
EMSE	Empirical Software Engineering
GPS	Global Positioning System
GREENS	International Conference on Green Communications, Computing and Technologies
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ICMSE	International Conference on Software Maintenance and Evolution
ICSE	International Conference on Software Engineering
ICT4S	International Conference on ICT for Sustainability
IDE	Integrated Development Environment
INES	National Institute of Science and Technology for Software Engineering
IVA	Intelligent Virtual Assistants
JCF	Java Collections Framework
JDK	Java Development Kit
JIT	Just-In-Time
JRAPL	Java Running Average Power Limit

JS	JavaScript
kLoC	1000 Lines of Code
LCD	Liquid Crystal Display
MEGSUS	International Workshop on Measurement and Metrics for Green and Sustainable Software
MOBILESOFT	International Conference on Mobile Software Engineering and Systems
MSR	Mining Software Repositories
NDK	Native Development Kit
OLED	Organic Light-Emitting Diode
RAPL	Running Average Power Limit
RQ	Research Question
SD	Standard Deviation
SDK	Software Development Kit
UI	User Interface
WALA	Watson Libraries for Analysis

CONTENTS

1 INTRODUCTION

Responsible energy consumption is a problem that permeates most modern human activity. It is not by chance that three of the UNs sustainable development goals can be linked with better usage of energy supplies[1]. Energy became a particular problem for the IT industry with the extensive adoption of battery-based devices such as mobile phones, smart watches, and laptops, in conjunction with the already very power-hungry data-centers.

Reducing the inefficiencies on energy consumption is not only a worry for IT. Across the globe, people are more aware of our impact on the environment and are trying to reduce it. One of these initiatives can be seen on the ambitious goals of reducing emissions, such as the reduction of by 40% by 2030, aiming for neutrality by 2050[2], set by Ursula von der Leyen, current President of the European Commission.

Data-centers may be seen as a good example of efficient usage of energy. If left unchecked, just the data-centers would have consumed up to 20% of global electricity and would be responsible for 5.5% of the global greenhouse gas emissions by 2025 (ANDRAE, 2017). In fact, efficient usage of data-centers made data centers consume just 1% of global electricity consumption by 2018. This represents an increase of 6% from 2010 (in the same time frame, data center usage increased by 550%) Masanet et al. (2020). The improvements on data-centers are far from done but this shows the positive impact of green computing.

Over the last years, mobile devices have been ever-present in our lives. These mobile devices do not have access to an energy outlet and rely on an external battery to function. Battery life is seen as very important to most smartphone owners, with as much as 92% considering battery life as a significant factor when purchasing a new smartphone, based on a survey with 400 responders[3]. According to the same survey, 63% of all smartphones owners are somewhat unsatisfied with their devices' battery and 66% would pay more for a device with longer battery life. Other battery-powered devices may suffer from the same problem, like smartwatches, tablets, and notebooks, among many others. Energy can be a serious problem when using those devices as they usually employ a multitude of power-hungry technologies such as high-definition screens, multicore processors, and GPS. Besides, new patterns of use for mobile gadgets are emerging, as people have the need to be constantly connected and

[1] <https://sdgs.un.org/goals>
[2] <https://ec.europa.eu/commission/sites/beta-political/files/political-guidelines-next-commission_en.pdf>
[3] <https://aytm.com/blog/smartphone-battery-survey/>

mobile. Some people have become dependent on their phone, to the point of being affected by nomophobia (that is "a psychological condition when people have a fear of being detached from mobile phone connectivity" (BHATTACHARYA et al., 2019)). Therefore, since these devices run on battery power, energy efficiency has grown in importance.

The most used mobile operating system is Android, developed and maintained by Google. Android is an open-source operating system based on a modified version of the Linux Kernel. In recent years, Android has become a ubiquitous platform, being present in many different types of devices like televisions, cars, smartwatches, game consoles, digital cameras, PCs, and, specially, on smartphones. Currently, more than 71.81% of all smartphones in the world use Android as their operating system[4] and the Google Play store has more than 2.97 million apps[5] available for download. Efficient use of energy on mobile devices is even more critical when taking into consideration that mobile development is accompanied by different challenges when compared to the desktop environment, such as: screen sizes compatibility, event-driven development, security awareness, and a considerable focus in the graphical user interface. One difference that draws attention from industry, academia and end-users is the importance of energy consumption on mobile devices.

The problem of improving energy efficiency on devices has received considerable attention in the research literature. In academia, a number of conferences and journals have presented an increasing number of papers about energy consumption (e.g., International Conference on Green Communications, Computing and Technologies (GREENS) (PEREIRA et al., 2016; ARDITO; TORCHIANO, 2018), International Workshop on Measurement and Metrics for Green and Sustainable Software (MEGSUS) (BAGNATO; ROCHETEAU, 2018; FOSSE et al., 2018), International Conference on Software Engineering (ICSE) (MCINTOSH; HASSAN; HINDLE, 2019; CRUZ; ABREU, 2019b), International Conference on Software Maintenance and Evolution (ICMSE) (CRUZ et al., 2019; ROMANSKY et al., 2017), International Conference on ICT for Sustainability (ICT4S) (HINTEMANN; HINTERHOLZER, 2019; PENZENSTADLER, 2020), International Conference on Mobile Software Engineering and Systems (MOBILESOFT) (ANWAR, 2020; MALAVOLTA et al., 2020), Mining Software Repositories (MSR) (MATALONGA et al., 2019; OLIVEIRA et al., 2019), Empirical Software Engineering (EMSE) (CHOWDHURY et al., 2019; CRUZ; ABREU, 2019a; OLIVEIRA et al., 2021)). The importance of producing green code (i.e., code that was developed with energy efficiency in mind) has become widespread.

[4] <https://gs.statcounter.com/os-market-share/mobile/worldwide>
[5] <https://www.appbrain.com/stats/number-of-android-apps>

In the past, hardware was the main focus of studies aiming to improve the energy efficiency of IT devices (TIWARI; MALIK; WOLFE, 1994). Nevertheless, as mentioned before, a number of works have shown that software can have considerable impact on energy consumption. Developers themselves also want to improve their code by producing greener code, and not just for mobile applications (MANOTAS et al., 2016). Notwithstanding their interest, many developers are not specialists and many aspects of energy consumption are not clear for them (MOURA et al., 2015). These developers have ample knowledge and experience with their development platforms of choice, but lack both knowledge and tools when it comes to making energy efficient applications (PINTO; CASTOR; LIU, 2014a). This comes from the inherent complexity of developing software systems, with some of the factors that can influence energy consumption being counterintuitive, such as internet browsers (MACEDO et al., 2020) or enabling and disabling logging (CHOWDHURY et al., 2018).

Developers have been more aware of the importance of energy for the environment and end-users, and are interested in building more energy efficient software (MANOTAS et al., 2016). Notwithstanding, thinking about algorithms and solutions that are energy efficient is not a typical skill for a developer, considering that computer science curriculum seldom offer green computing courses (SARAIVA; ZONG; PEREIRA, 2021). In addition, some solutions that developers typically employ or recommend are not backed up by scientific evidence (PINTO; CASTOR; LIU, 2014a). As pointed out by previous work (CHOWDHURY; HINDLE, 2016), energy-aware projects usually are larger and the changes that could impact the application's energy efficiency are most of the times relegated to specialists.

When trying to produce software, developers are presented with different options to solve a problem. Every non-trivial software system has parts where it is possible to use different kinds of data structures, Application Programming Interface (API) calls, or concurrency control mechanisms by means of simple source code transformations. We call these parts *energy variation hotspots* when these transformations reduce energy consumption. For example, the Java language has 9 different implementations of hash tables, with different guarantees in terms of scalability, thread-safety, and memory efficiency. Previous work (PINTO et al., 2016) has shown that changing implementations of a collection in Java lead to a reduction in energy consumption of more than 70% by an operation. Looking at functional languages, changing the thread instantiation primitive in Haskell also resulted in a reduction of energy consumption by the application (LIMA et al., 2019). Those energy variation hotspots vary in development stage, complexity, size, code location, and environment.

As research in green computing progresses, researchers have found specific constructs of software development that may have a negative impact in energy consumption, called *green smells* (GOTTSCHALK et al., 2012). Unlike the energy variation hotspots, they can be disjointed of programming constructs. Some of energy-related factors that contained green smells are: APIs, binding of resources too early, code obfuscation, code smells, collections, database usage, design patterns, development approaches, garbage collector, Hypertext Transfer Protocol (HTTP) requests, image format, information hiding, loops, screen colors, thread switching, and wake locks. In Table 1 we summarized a non-exhaustive list of topics and references to studies that analyzed them. To help researchers get an overview of the field, some works organize the knowledge about green computing (C.; CHANDRASEKARAN; CHIMALAKONDA, 2020; MYASNIKOV et al., 2020).

In this thesis, we will focus on the problem of the rapid discharge of battery-powered devices and present ways on how to reduce it. Battery-powered devices nowadays have a plethora of functionalities that have non-negligible energy footprint. Every so often, these functionalities may be used in a more energy efficient way. Our main insight is that, for many green computing issues, there are multiple readily available diversely-designed solutions that have different characteristics in terms of energy consumption, what we call *energy design diversity*. Our hypothesis is that these different solutions can be leveraged to ensure a reduction on energy consumption without increasing significantly the development complexity.

The concept of energy design diversity is inspired by the concept of fault tolerance's design diversity. In fault tolerance (LITTLEWOOD; POPOV; STRIGINI, 2001), the main idea is to avoid an unacceptable state in the program. To ensure that, one can use different techniques such as: replicate parts of the source code, use different teams to solve the same problem, defensive programming, exception handling and so on. In green computing, we want to develop as many solutions to a specific energy hotspot to ensure that the least possible energy is consumed. Our insight is that using a similar concept from fault tolerance would help practitioners save energy by selecting the right solution for the problems they face while developing. Not every energy solution is applicable to each situation, and taking into account the wide variety of mobile devices available on the market, having several different solutions to the same problem may be a necessity. The idea behind energy design diversity is to increase the probability that, for a specific hotspot, at least one of the solutions will apply.

To reduce energy consumption, researchers could focus on increasing the energy efficiency of any energy hotspot, using the concept of energy design diversity to present to developers

Table 1 – Representative list of different factors studied to reduce energy consumption in Android and theirs references.

Topic:	**Reference**:
APIs	Linares-VÁsquez et al. (2014), Chowdhury et al. (2019); Zimmerle et al. (2019)
Binding Resources	Gottschalk, Jelschen e Winter (2014)
Bundling	Chowdhury et al. (2019)
Code Obfuscation	Sahin, Pollock e Clause (2014)
Code Smells	Cruz e Abreu (2017), Palomba et al. (2019); GoaËr (2020), Iannone et al. (2020) Anwar (2020), Habchi, Moha e Rouvoy (2021)
Collections	Pereira et al. (2016), Pinto et al. (2016); Hasan et al. (2016), Saborido et al. (2018) Oliveira et al. (2019)
Databases	Lyu et al. (2017)
Design Pattern	Sahin et al. (2012), Chen e Zong (2016)
Programming Languages	Corral et al. (2014), Chen e Zong (2016); Oliveira, Oliveira e Castor (2017) Kholmatova (2020), Pereira et al. (2021)
Energy Pattern	Cruz e Abreu (2019a)
Garbage Collector	Li et al. (2013)
HTTP Requests	Li e Halfond (2015), Li et al. (2016), Anwar et al. (2020)
Image Format	Thiagarajan et al. (2012)
Information Hiding	Linares-VÁsquez et al. (2014); Li e Halfond (2014), Morales et al. (2016)
Loops	Li e Halfond (2014)
Machine Learning	Mcintosh, Hassan e Hindle (2019)
Off-loading	Kwon e Tilevich (2013), Tang et al. (2020)
Refactoring	Couto, Saraiva e Fernandes (2020)
Screen Colors	Dong e Zhong (2011), Linares-VÁsquez et al. (2018)
System Calls	Aggarwal et al. (2014)
Thread Switching	Li et al. (2013), Pinto, Castor e Liu (2014b)
Wake Locks	Pathak et al. (2012), Liu et al. (2016)

more energy efficient options. In this thesis, among those different factors that impact the energy consumption of Android apps, we choose to prioritize our effort in two different aspects: development approaches and Java collections. These factors were selected for two main reasons: (i) previous works (CORRAL et al., 2014; CHEN; ZONG, 2016; HASAN et al., 2016; SABORIDO et al., 2018) have shown that they can have significant impact on energy consumption; and (ii) they are used on a wide range of applications, running on a myriad of computing devices. Given the importance of those factors, it is not a surprise that previous works have already tried to find solutions to reduce the energy consumption of those energy hotspots.

In this thesis, we will answer the following Research Question (RQ):

- **RQ1:** how can we increase the energy efficiency of a fully functional Android application using an alternative development approach to Java?
- **RQ2:** how can we increase the energy efficiency of a fully functional software system optimizing the Java Collections used by it?

We consider that answering these questions would help to reduce energy consumption and provide evidence of the importance of the concept of *energy design diversity*.

The following sections will introduce the two main topics of this thesis: Section 1.1 we will present a summary of our work on the energy optimization based on development approaches and Section 1.2 we will present a summary of our work on Java Collection energy efficiency optimization.

1.1 DEVELOPMENT APPROACHES

When developing Android apps, the developer can choose between four development approaches: *Java*, the default language for Android development; *JavaScript*, with support from a mobile framework (such as Apache Cordova [6] Monaca [7], or NativeScript [8]), it is possible to develop a full app using the web toolkit (HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript); *Kotlin*, supported by Google as a development language since October 2017[9]; and *C/C++*, through the Native Development Kit (NDK)[10], which

[6] <https://cordova.apache.org/>
[7] <https://monaca.io/>
[8] <https://nativescript.org/>
[9] <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>
[10] <https://developer.android.com/ndk//>

makes it possible for a developer to write the majority of an application in C/C++.

The developer can freely choose any approach to develop her/his app or even make a hybrid application (i.e., using more than one programming language). When choosing among these approaches, developers need to keep in mind different factors: memory usage, performance, expertise in the programming language, ease of maintenance, number of available libraries, among many others. As stated before, energy is also seen as a very important factor when developing mobile applications. Nevertheless, developers have almost no information about the differences in energy consumption among those different approaches.

Aiming to reduce this lack of knowledge, we conducted an empirical study using a testbed of 33 different benchmarks and four applications. We executed our experiments on three[11] out of four different approaches using five distinct devices.

On the topic of development approaches, we will answer the following research questions:

- **RQ1.1:** Is there a more energy efficient approach among the most common Android development models?
- **RQ1.2:** Is it possible to reduce the energy consumption of a native app by making it hybrid?

According to our data, there is a significant difference in energy consumption between each approach and this difference varies greatly depending on the executed benchmark. In most cases, JavaScript was more energy efficient than the other approaches, with Java and C++ having a similar number of benchmarks where they were the most energy efficient approach. To further investigate the energy impact of these approaches, we tried to determine whether it was possible to reduce energy consumption of an application using a hybrid approach. Results from modified apps showed that developers can leverage energy efficiency by choosing to implement a hybrid application but only when the application makes heavy usage of the CPU. More details about this study can be found in Chapter 3.

1.2 JAVA COLLECTIONS

Java collections are widely used in Java systems, both mobile and desktop. They are the backbone of many applications as they are used to store, access, and modify information

[11] Kotlin was not officially supported when the study was being conducted.

during the execution of an application. Several papers have focused exclusively on studying the energy consumption of Java data structures: on Android (HASAN et al., 2016), on the desktop (MANOTAS; POLLOCK; CLAUSE, 2014; PEREIRA et al., 2016) or using concurrency (PINTO et al., 2016). Some similar studies with data structures have also been conducted in Haskell (LIMA et al., 2019).

We choose to study collections to delve deeper into their energy impact. Specifically, we want to investigate if alternative implementations to the Java Collections Framework have a smaller energy footprint; which collections have the greatest impact on energy consumption; and if the environment where the application is running influences the energy impact of those collections. To answer our questions, we develop a framework to analyze the energy consumption of data structures in mobile and desktop environments. Based on this framework, we created a tool named Collections Energy Consumption Optimization tool Plus (CT+).

CT+ automatically runs multiple benchmarks, on desktop and mobile environments, targeting Java's collections. Based on the benchmarks' execution data, it establishes energy consumption profiles (HASAN et al., 2016) to the implementations of these collections in an application-independent way. It then performs static analysis on the source code of the application under investigation to estimate the usage frequency of each collection's operation. With the static analysis results, it recommends the most appropriate implementation for each collection implementation.

On the topic of Java collections, we will answer the following research questions:

- **RQ2.1:** To what extent can we improve the energy efficiency of an application by statically replacing Java Collection implementations?
- **RQ2.2:** Are recommendations for Java collections device-independent?
- **RQ2.3:** How much does the workload size impact the energy efficiency of a Java collection implementation?
- **RQ2.4:** Are recommendations for Java collections profile-independent?.

We executed two different studies to analyze the influence of (i) **devices** and (ii) **energy profiles** on the energy consumption of software systems using Java Collections. Across the two studies, a total of seventeen different software systems (two mobile, twelve desktop, and three on both environments) were used, most of them mature applications with thousands of lines of code. On our study on **devices**, seven distinct devices were used while on our study about **energy profiles**, we used six different energy profiles. A total of 64software system versions were created during this project and used to measure the differences in energy consumption of the aforementioned factors.

Our tool had positive results (that is, increase the energy efficiency on the application) in most cases, up to 16.34% reduction on energy consumption. Only for a small amount (2 out of 64 versions), the recommendations degraded the energy efficiency, up to 1.21%. Overall, CT+made a total of 2295 recommendations that lead to a statistically significant impact on energy efficiency. Our experiments also showed indications that some of the most widely used collections (`ArrayList`, `HashMap`, and `HashTable`) may not be very energy efficient and should be used cautiously. More details about that study can be found in Chapter 4.

1.3 THE CONTRIBUTIONS

With this thesis, we made a number of contributions to the field of green computing. Although they differ in level of importance, we believe that they help improve the state-of-art of green computing.

Our first set of contributions are focused on our works analyzing the energy impact of the development approach of applications running on the Android operating system. These contributions are summarized as follows:

- **JavaScript as an alternative to optimize energy consumption.** In this thesis, we present data that points towards the possibility of using JavaScript to save energy in Android applications.
- **Insights on the Native Development Kit efficiency.** Another way to improve performance and energy efficiency, using the NDK. From our studies, it seems to have less probability of worsening the energy efficiency than JavaScript, even if it may increase the development complexity.
- **The importance of experiment on different devices.** While analyzing the development approaches, we found out that different devices had a similar energy consumption pattern. The result was exactly the opposite when looking into the energy consumption

of Java collections. This shows the importance of analyzing using different devices when executing empirical studies on green computing.

- **Execution time is not a proxy to energy consumption but it is a good bet.** Analyzing the energy consumption of our mobile experiments, one can see that running a task using parallelism may consume more energy than executing it sequentially, even if it takes more time to finish. Even so, our experiments indicate that in most cases, parallel executions consume less energy and time than their sequential counterparts and should be used (with caution) by practitioners when trying to optimize energy efficiency.

Our second comes mainly from our works analyzing the energy impact of Java Collections on software systems. These contributions are summarized as follows:

- **Generalist approach to design energy efficient applications.** We present in this thesis an approach that can be used for general-purpose green development, unrestricted to a scenario, development environment, device, or application.
- **Automated energy optimizer refactoring tool.** We instantiated our previously mentioned approach on a tool called CT+. This tool can recommend the most energy efficient Java collection implementation and automatically refactor the application source code to use it.
- **Insights on the importance of battery age on energy data.** Across all our experiments we noticed that the device battery age may have a heavy influence on the energy data collected, even creating so much noise that invalidates any data from the device.
- **An understanding of the energy profiles.** Energy profiles are a staple concept of green computing and in this thesis we developed techniques to optimize their creation and analyze their influence on code refactoring.
- **A glimpse on the domination on collections.** Collections implementations may present an energy domination behavior. The dominated implementation always consumes more energy than the dominating. The study of this phenomena could lead to an optimization on collections refactoring.
- **The importance of energy diversity design.** 89.6% of all energy saving collections refactoring done on our works are from alternative sources to the Java Collections

Framework. Among the most changed implementations, we had very popular ones, such as `ArrayList` and `HashMap`. Using alternatives to these implementations could lead to a more energy efficient program.

During the development of this thesis, the author also contributed with a paper analyzing the energy consumption on the field of reactive applications. More specifically, an analysis of CEP.js library energy consumption (ZIMMERLE et al., 2019). This library was developed based on the Reactive Extensions for JavaScript[12] (RxJS) but providing Complex Event Processing (BUCHMANN; KOLDEHOFE, 2009) operators.

1.4 ORGANIZATION

The remainder of this document is organized as follows:

- Chapter 2, we will present the theoretical foundation with the objective of providing the necessary knowledge to accompany the rest of this work.
- Chapter 3, we will present our study on energy consumption of the different development approaches. In this chapter, we will be looking at the possibility of hybridization of mobile applications and how much energy can be saved by using another approach together with Java, experimenting in a number of devices. We found out that both JavaScript and NDK can be used to reduce energy consumption in certain scenarios.
- Chapter 4, we will present our framework to reduce energy consumption and a study on data structures, detailing our methodology, tool and results. In this chapter, our energy-saving framework will be explained and instantiated in a tool called CT+. Using this tool, we analyzed different factors that may influence the energy consumption in an app (e.g., energy profiles, development environment, devices, collections instantiating). We found out that energy consumption optimization is heavily influenced by the applications workload and that using the wrong energy profile may increase the energy consumption of the application.
- Chapter 5, we will summarize the results, our final considerations and presents possible future works in the green computing field.

[12] https://github.com/reactivex/rxjs

2 BACKGROUND

This chapter aims to detail the pillars on which this thesis was based. The chapter organized in terms of the following topics: Android Infrastructure (Section 2.1), Android Application Development (Section 2.2), Android Energy Awareness (Section 2.3), Measuring Energy Consumption (Section 2.4), Android Power Profiler (Section 2.5), Static Analysis (Section 2.6), and Design Diversity (Section 2.7). Across this thesis, each chapter may have a background section with more specific information.

2.1 ANDROID INFRASTRUCTURE

The Android infrastructure can be separated in five different layers: Applications, Application Framework, Native Libraries and Runtime, Hardware Abstraction Layer, and Kernel, as illustrated in the Figure 1.

In this thesis, we will focus mostly on the Applications layer. As our goal is to help the developer create more energy-efficient applications, the application layer is the ideal place to target our efforts. It is worth noting that all Android applications, including native ones (e.g.,

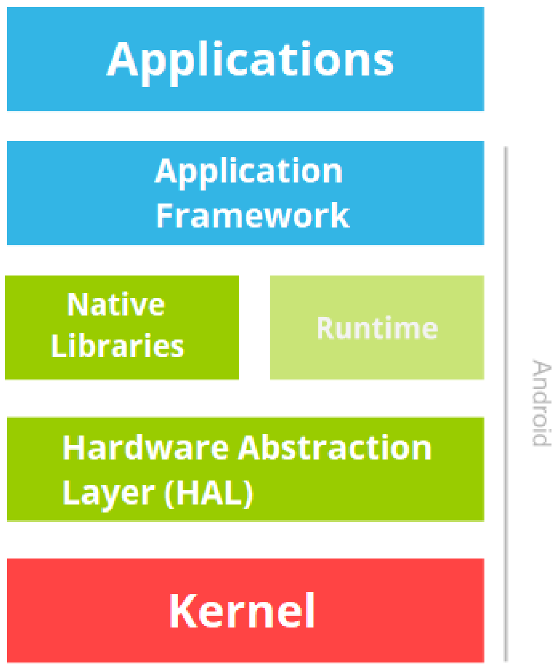


Figure 1 – The five levels of Android infrastructure.

.

Android's User Interface (UI)), run at the application level. This layer is composed of several components that allow system access to the application. The four main components (classified by Google[1]) are:

Activities: components that represent the application screens. Each activity is independent of each other and can represent any relevant element within the application. It is possible for one app to call an activity from another one. When developing, the activity is implemented as a subclass of the `Activity` class. The other UI elements inside an activity are implemented as subclasses of the `View` class. From all components, Activities are the only one that interacts with the end-user.

Services: components that perform work outside the UI, invoking some type of resource available on the device, in software or hardware. That resource usually is a long-running operation and needs to be executed in the background. For example, a service may play music while the user is in a different application, or fetch data over the network without blocking user interaction with an activity.

Broadcast receivers: components that respond to broadcast advertisements throughout the system, enabling the system to deliver events to the app outside of a regular user flow. Using broadcast receivers, the user can be notified without the need for the app to remain running. Chat notifications or telling the user that the battery is low are examples of events that send broadcasts to the screen. Usually the broadcast receiver is used to interface other components, such as initializing a service based on an event that occurred in an activity.

Content Providers: components that manage a shared set of application data. A common use is to manage databases such as SQLite. Using the content provider, other applications can take advantage of the data collected, for example, access the user's contact list to add them to a social networking application.

On our studies, we will be focusing mostly on exploring resources of the **Activities** component. Inevitably, all other components are used when execution and analyzing Android applications but their behaviors in relation to energy consumption are outside the scope of this research.

[1] <http://developer.android.com/guide/components/fundamentals.html>

2.2 ANDROID APPLICATION DEVELOPMENT

Traditionally, in mobile environments, native applications, i.e., developed using a native language of the operating system, have been regarded as faster, safer, and more adaptable to changes in the operating system. In the iOS ecosystem, native apps are written in Objective-C and Swift whereas Android native apps are written in Java and Kotlin. On the other hand, web apps, i.e., applications developed using Web technologies, are seen as more portable, with lower maintenance and development costs, and are faster to get to market[2]. At the same time, it is expected that these web apps run slower than native ones. This section provides a high level overview of web development frameworks and briefly presents the NDK, a toolkit aiming to support the development of apps with strict performance requirements through the use of C and C++ code.

Web application framework. Developers who choose to develop web apps without prior experience in mobile development may find it more difficult, as there are not as many tutorials or support as from IDEs as in the native language. Android Studio, Google's recommended Integrated Development Environment (IDE) for Android development, offers limited support for building web apps. In addition, most APIs are native, which means that they are directly accessible to Java code but require a plugin in the case of web apps.

Mobile development frameworks aim to ease the construction of smartphone applications based on web technologies, such as HTML, CSS, and JavaScript. These frameworks are intended to allow the use of standard web development technologies for cross-OS development, i.e., the application is developed once and ported to any mobile operating systems freeing the developer from having to deal with each OS native language. These applications are then wrapped, so the operating system can identify them as traditional applications and give them access to native APIs.

Among the different mobile development frameworks available, this thesis was developed using Apache Cordova. Cordova is open source and is used as the basis for other popular frameworks like Phonegap, Ionic, Intel XDK, Telerik, Monaca and TACO. Among the contributors to the Apache Cordova project are IBM, Google, Microsoft, Intel, Adobe, Blackberry, and Mozilla[3]. We employed Cordova mainly because our experiments involved the execution of multiple benchmarks that were implemented in JavaScript but did not specifically target

[2] <https://www.lifewire.com/native-apps-vs-web-apps-2373133>
[3] <http://wiki.apache.org/cordova/who>

Android.

When using Cordova, the application is developed as a web page, being able to reference CSS files, JavaScript code, images, media files, or other resources needed to run it. This app will run over an encapsulated `WebView`[4] within a native application. An application using Cordova and native components can communicate, i.e., JavaScript code can invoke native snippets directly. Ideally, JavaScript APIs for native code are consistent across multiple platforms and operating systems. There are also externally available plugins that allow a developer to use features not available on all platforms. Because of the interactions between the native and web code sections, these applications are known as *hybrid apps*. In Android, hybrid apps produced using Cordova will be compiled into an .apk file, which is the default distribution format for Android applications.

Native Development Kit. In Android apps that have strict performance requirements, it is possible to develop parts of an application using C/C++ through the Native Development Kit (NDK). This approach is often only employed in specialized scenarios, most notably games. For example, in one sample of 109 apps we examined (Chapter 3, Table 2), only 5 employed the NDK and all of them are games. Although it might seem a good idea to use the NDK as much as possible to make an application achieve maximum performance, Google does not recommend it. According to the Android website[5], there are two situations where the use of the NDK is recommended: (i) when it is necessary to *"squeeze extra performance out of a device to achieve low latency or run computationally intensive applications, such as games or physics simulations"*; and (ii) to *"reuse your own or other developers' C or C++ libraries"*.

The NDK is an optional package that can be installed using the *Android Software Development Kit (SDK) Manager*. Unlike building native and web apps, using the NDK is not straightforward, as pointed out on the Android website: *"(...)The NDK may not be appropriate for most novice Android programmers who need to use only Java code and framework APIs to develop their apps."*. One example of complication that arises from the use of the NDK is the need to manually manage memory, since Android keeps two separate heaps, one for the Android Runtime and another one for the native parts of the app. Moreover, the setup of an app that uses the NDK requires additional steps such as the construction of an additional build script and the definition of at least one method that will serve as the interface between the Java and C/C++ code.

4 <https://developer.android.com/reference/android/webkit/WebView>

5 <http://developer.android.com/ndk/guides/index.html>

2.3 ANDROID ENERGY AWARENESS

Since the release of the Android OS in 2008, Google introduced a number of changes aiming exclusively to help reduce energy consumption. Before Android 5.0, most updates were focused on reducing the energy consumption of the operating system itself. After version 5.0, besides optimizing their infrastructure, Google has been giving developers tools and information so they may be able to reduce the energy footprint of their own apps. In a more subtle way, Google also has introduced ways for the end-users to indirectly reduce the energy consumption of their devices (e.g., using dark themes on Organic Light-Emitting Diode (OLED) screens).

Here, we present some of the major updates regarding energy consumption on the Android OS:

- **Project Volta (Android v5.0):** A set of optimizations and tools aiming to reduce battery consumption. It included: a new battery saver mode, a job-scheduling APIs, batching of tasks, the Battery Historian[6] and the `batterystats` API.
- **Doze mode (Android v6.0):** Reducing CPU and network activity on apps when the device is unused for long periods of time, aiming to reduce the energy consumption while the user is not on the phone.
- **App Standby feature (Android v6.0):** Reducing background network activity on apps that had not been used recently.
- **Improved Doze mode (Android v7.0)**.
- **Battery usage alerts (Android v7.1)**.
- **Light and dark themes (Android v8.1)**. On OLED devices, darker themes can save a significant amount of energy (LINARES-VÁSQUEZ et al., 2018).
- **App Standby Buckets and Adaptive Battery (Android v9.0):** Helping prioritize the apps requests, based on how recently and frequently the apps have been used.
- **Improvements to battery saver mode (Android v9.0):** When entering battery saver mode, the OS may put apps in app standby mode more aggressively, background

6 <https://developer.android.com/topic/performance/power/setup-battery-historian>

execution limits apply to all apps, may disable location services when the screen is off, and remove network access to background apps[7].

- **New system-wide dark theme/mode (Android v10.0).**

Complementary to the Android updates, in Android Studio version 3.2 (released in 2018 soon after Android v9.0), they introduced an energy profiler, that is, a tool that makes it easier for developers to see the energy footprint of their apps. The intention was to help developers diagnose and improve the energy impact of their app.

This shows the importance of optimizing the energy consumption on mobile devices. Even so, non-specialist developers may still introduce energy leaks to their applications, reducing the overall battery life. When compared with other development metrics (such as performance), excessive energy consumption is not as easy to notice without the aid of an energy profiler.

2.4 MEASURING ENERGY CONSUMPTION

In the early days of computing, hardware and the operating system were seen as the big energy spenders. Tiwari, Malik e Wolfe (1994) showed that software can also have a big influence on energy consumption. Nonetheless, the developers still believed that hardware and operating systems were the only important components. That led developers to be unconcerned about trying to reduce the energy consumption of their applications. This has caused an escalation in energy consumption over the years (ASAFU-ADJAYE, 2000).

In recent years, several academic works have focused on reducing the energy consumption of an application by changing pieces of software, with different degrees of success (HASAN et al., 2016; OLIVEIRA; OLIVEIRA; CASTOR, 2017; PINTO; CASTOR; LIU, 2014b; HINDLE, 2012; COHEN et al., 2012; LI; HALFOND, 2014; MANOTAS; POLLOCK; CLAUSE, 2014).

With the advent of mobile technologies and the intense usage of battery powered devices, efficiency energy consumption has become essential. Without a constant flux of power, energy consumption is critical and any improvement can have a positive impact on the user experience.

To measure energy, practitioners make use of a tool called energy profiler. Jagroep et al. (2015) uses the following definition to describe an energy profiler:

[7] <https://developer.android.com/about/versions/pie/power>

> An energy profiler (EP) is a software tool that estimates the energy consumption of a system based on the computational resources used by applications, and by monitoring the hardware resources.

Making use of an energy profiler, practitioners can analyze the energy consumption of their software systems, searching for energy anomalies that could decrease the energy efficiency of their application.

In the next sections we will discuss the ways to profile energy consumption on mobile devices, following the classification of Hoque et al. (2015). The taxonomy of profilers have four different main aspects to classify the profilers: Granularity, Model Type, Model Construction and Deployment Type.

Granularity: how specific the profiler can be when looking at energy consumption from the four possibilities: system, subcomponent, application, and API.

Model Type: which heuristics the power model follows to estimate the energy consumption on the mobile device (e.g., linear regression, utilization, genetic model).

Model Construction and Deployment Type: where the power model to calculate energy consumption is constructed and where the profiler is executed. Both aspects can be classified as "on device" or "off device".

To materialize our vision, we needed a profiler fulfilling our requirements, that is, granularity of (at least) application level and "on device" deployment type, to measure the energy consumption of different constructs related to software development. Without at least a granularity of "application level", our measures would be filled with noise from other components running on the device (e.g., GPS, screen, network, other applications). Our choice to have a "on device" deployment is based on the possible future developers adherence of our energy aware tools. The tools to measure energy consumption can be complex to handle and sometimes very expensive. Cruz e Abreu (2019b) have even suggested measuring it through the cloud, using a Software-as-a-Service infrastructure, to reduce that complexity. To increase the accessibility of our tools, we wanted a profiler that could be used without external instruments.

Analyzing the available profilers, we found that each one has its own advantages and disadvantages. None of them is the best or can be used in all possible scenarios. As an example of energy profilers diversity, previous works have several different solutions to analyze the energy consumption of an application. On system level granularity, some examples are:

programmable batteries (SILVA-FILHO et al., 2012), physical devices, including current clamps and Data acquisition systems (DAQ)s (Data Acquisition System) (LI et al., 2013; PETERSON et al., 2011; SABORIDO et al., 2015; SILVA et al., 2018; BESSA et al., 2019), and even let the battery discharge completely (COUTO et al., 2014).

On the subcomponent, application and API levels, there are works focusing specifically on methods (COUTO et al., 2014), API calls, (LINARES-VÁSQUEZ et al., 2014), whole applications (WILKE et al., 2013; SILVA et al., 2018), lines of code (HAO et al., 2013; LI et al., 2013), collections (HASAN et al., 2016), and even instruction level (BESSA et al., 2019), among others. It is worth noting that as the granularity of measurement gets finer, the impact of instrumenting the code on energy consumption increases (HÄHNEL et al., 2012).

Among the mobile profilers with "application level" granularity and "on device" deployment, we selected Android Power Profiler. More details about it on Section 2.5.

Some of our studies have also analyzed the energy consumption of Java implementations on development machines and servers. For that, we used the Intel RAPL[8] (Running Average Power Limit) to perform the measurements. Intel RAPL consists of a set of low-level interfaces that provide information about energy and power consumption using a software power model. This software power model calculates power consumption using hardware performance counters and input and output models. It is supported by servers and development machines that use Intel chipsets. Several papers make use of Intel RAPL to measure energy consumption (SUBRAMANIAM; FENG, 2013; LIU; PINTO; LIU, 2015; KAMBADUR; KIM, 2014).

We used Intel RAPL via `Java Running Average Power Limit` (JRAPL) (LIU; PINTO; LIU, 2015). JRAPL is a framework for profiling Java programs running on Central Processing Unit (CPU)s with Running Average Power Limit (RAPL) support[9]. JRAPL provides an easy way to use RAPL when analyzing Java programs. It offers refined energy analysis and synchronization-free measurement. It is possible to use Intel RAPL to profile the energy consumption of Android applications using a device emulator. This solution is not without mishaps, such as: (i) system level profiling may introduce noise on the energy data; (ii) an emulated device and a real one may not behave the same; (iii) other solutions present better results, such as the Android Power Profiler.

[8] <https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl>
[9] <http://kliu20.github.io/jRAPL/>

2.5 ANDROID POWER PROFILER

The Android Power Profiler was, among other alternatives, the profiler that better fulfilled our requisites of model deployment and minimum application level, as stated in Section 2.4. The reasons for us to choose the Android Power Profiler were:

- **Availability across devices**. It is available on every Android device with version 5 or more (which corresponds to over 94.1% (April 2021) of all Android devices[10] and will only increase over time). This gives us freedom to run the experiments on almost any device. Other solutions that fulfilled our requirements (e.g., Trepn Profiler[11]) had other limitations that reduce our device pools (e.g., could only be executed in a specific chipset). Those limitations would undermine our results generality.
- **No extra application or instrumentation**. As the profiler is natively executed, no extra application is needed. As a way to decrease the noise and to make it easier for the developer to collect the results, we opted for a profiler that could be easily used, without extra installations on the device or external to it.
- **Easy of use by the developer**. The commands used to collect the battery information are quite simple and it does not require any setup or tool to be used;
- **Component consumption distinction**. It distinguishes between the different components used on the execution (e.g, Wi-Fi, CPU, Global Positioning System (GPS)). Because we used a dashboard application to store the execution data, with the distinction between the components, wifi consumption was easily discarded. More details about the dashboard application are presented next on this Section.
- **The profiler is supported by Google**. Other profilers were created in two different ways: by researchers, as a tool to help solving research problems (e.g., PowerProf (KJÆRGAARD; BLUNCK, 2011), Eprof (PATHAK; HU; ZHANG, 2012), V-edge (XU et al., 2013)); or by companies, as a way to better inform developers about the energy behavior of their apps (e.g., PowerTutor[12], Trepn Profiler). There is in fact an extensive number of frameworks and tools to measure energy consumption (C.; CHANDRASEKARAN; CHIMALAKONDA, 2020); The Android Power Profiler was made by Google, responsible for

10 <https://developer.android.com/about/dashboards>
11 <https://developer.qualcomm.com/software/trepn-power-profiler>
12 <http://ziyang.eecs.umich.edu/projects/powertutor/documentation.html>

the development of Android OS, to help developers detect fault energy behaviors on their apps. By using a tool developed by Google, we hope that our tool can be safely used by other researchers or developers in the future.

To support our experiments, we created an application to control the flux of executions on mobile apps. The `Dashboard App` is a webservice, using HTML and JavaScript. Through the dashboard, we could control the execution of our benchmarks and modified applications. It allows us to select several parameters to be executed: device; number of executions; type of experiment (e.g., benchmark or app); and main point of experiment (e.g., programming language or framework); the number of warm-up iterations; and the number of iterations. The last two factors may be used to debug the experiments. It also collects and stores all energy data from our experiments. Figure 2 represents the current state of our software interface.

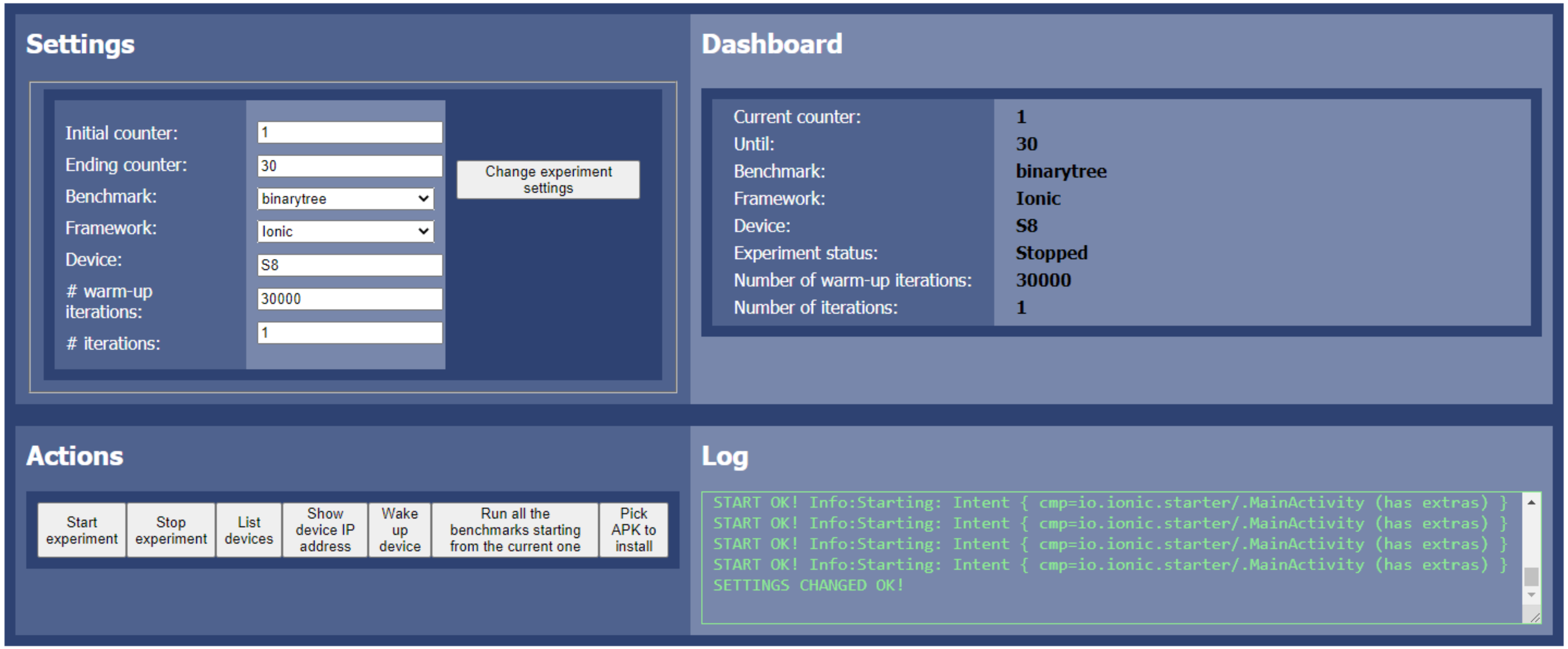
Part of our solution to measure energy consumption using wifi includes a library called `Dashbench`. This library has the objective of making a bridge between the `Dashboard App` and the mobile application. This library is easily adaptable to be used by different kinds of apps. Using its API, it is possible to make use of the `Dashboard App` almost seamless. Up to this point, `Dashbench` has been used with four different programming languages: Java, JavaScript, C++, and Dart.

To use the Android Power Profiler tools it is necessary to make use of Android Debug Bridge (Android Debug Bridge (ADB))[13]. ADB is a command-line tool that works like a communication interface, using the client-server model, where the device being used acts as the client and the development machine acts as the server. Through ADB it is possible, among a variety of commands, to: install and debugging apps, collect data about the device, and execute automated tests.

For our studies, we will be focusing on the following set of commands:

- `adb tcpip <PORT>`: used to specify a port to be used by our wifi connection. `<PORT>` is the number of the port.
- `adb connect <IPADDRESS>:<PORT>`: used to open an wifi connect to the device. With this command, we can execute any adb command via wifi, removing the necessity of an USB connection. It is important to remember that if the device is connected via USB, the battery charge will not decrease and it will not be possible to measure the energy

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



Initial counter:	1
Ending counter:	30
Benchmark:	binarytree
Framework:	Ionic
Device:	S8
# warm-up iterations:	30000
# iterations:	1

Dashboard

Current counter:	1
Until:	30
Benchmark:	binarytree
Framework:	Ionic
Device:	S8
Experiment status:	Stopped
Number of warm-up iterations:	30000
Number of iterations:	1

```
START OK! Info:Starting: Intent { cmp=io.ionic.starter/.MainActivity (has extras) }
START OK! Info:Starting: Intent { cmp=io.ionic.starter/.MainActivity (has extras) }
START OK! Info:Starting: Intent { cmp=io.ionic.starter/.MainActivity (has extras) }
START OK! Info:Starting: Intent { cmp=io.ionic.starter/.MainActivity (has extras) }
SETTINGS CHANGED OK!
```

Figure 2 – Screenshot of our Dashboard app user interface.

.

consumption. The mobile device and the development machine must be on the same wifi network. `<IPADDRESS>` is the local ip address of the device.

- `adb shell am start -n <ACTIVITY> -e param <PARAM>`: used to start the application. `<ACTIVITY>` is the main activity of the application. `<PARAM>` is used to inform a parameter to the app (e.g., a specific data structure to be analyzed by our profiler).
- `adb shell dumpsys batterystats`: used to collect energy data. Can be affixed with "> file.txt" to store the data on a file or "–reset" to reset the energy data on the device.
- `adb shell shell pm clear <PACKAGE>`: used to force stop the app, ensuring that it is terminated after the energy logs are collected. `<PACKAGE>` is the application package name.

We followed a strict procedure to execute any application or benchmark on our devices. With that, we tried to reduce the inherent noise of measuring the energy consumption on mobile devices. Experiments in this thesis were executed observing the following step-by-step procedure:

1. Close all running applications not involved in the tests, activating airplane mode, and immediately rebooting the device. The Wi-Fi must be turned on to use the dashboard app.
2. Connect the device to the webservice via Wi-fi using the Android Debug Bridge;
3. Reset all data regarding battery consumption;
4. Execute the experiment;
5. Prevent the app from running in the background at all times, not locking the screen, not allowing the screen to shut down, or changing to another app.
6. Gather the execution data from the device.

Items 1 and 2 are partially automated using shell scripts and monkey[14]. Items 3, 4, 5 and 6 were executed automatically using our web dashboard application. On Figure 3, we present a graphic illustration of the workflow when executing the `Dashboard` app steps 4, 5 and 6.

[14] <https://developer.android.com/studio/test/monkey>

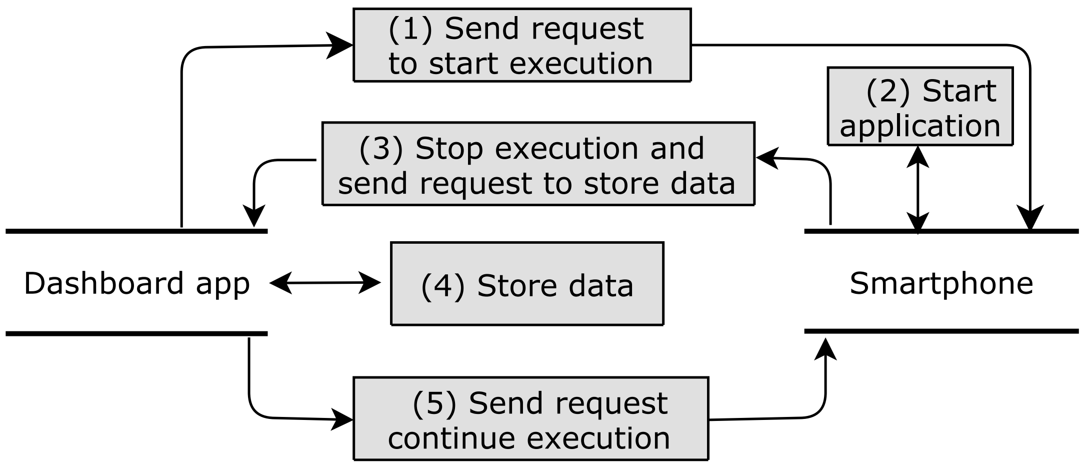

(1) Send request to start execution
(2) Start application
(3) Stop execution and send request to store data
Dashboard app
(4) Store data
Smartphone
(5) Send request continue execution

Figure 3 – Workflow of the dashboard application. The activities are executed following the number beside each step.

The energy and performance tools available in Android give us the information about execution time and CPU usage in minutes and seconds. On the other hand, the units used to measure battery discharge on mobile devices are not Joules, the unit used by the International System of Units to measure energy. Using the Android Power Profile tools, the discharge is presented in milliampere-hour mAh.

Milliampere-hour is a unit of electric charge, multiplying electric current (Ampere) by time (seconds) and its measured in Coulombs. Using that measure to analyze energy consumption is conceptually wrong, since milliampere-hour is a unit of electric charge. However, its possible to convert the electric charge (mAh) to energy (Joules) using the voltage, using the following equalities:

$$1 milliampere \times hour = 3.6 coulomb \tag{2.1}$$

$$Energy = Charge \times Voltage \tag{2.2}$$

Equality 2.1 presents how much coulombs are present in one mAh. On equality 2.2, we can find the energy expended by multiplying the charge (that is, the data value shown by Android Power Profiler multiplied by 3.6) and the voltage. As an example, if the data shows that an app used 1mAh with a voltage of 4.4v, the energy value would be

$$Energy = 1 * 3.6 \times 4.4 = 15.84\ Joules \tag{2.3}$$

To optimize power usage, devices use a power management technique called dynamic voltage scaling, that is, the voltage may increase or decrease in order to optimize the performance and energy consumption.

To verify how this variation occurs while benchmarks were executed, we collected voltage data from two different devices and compared it with a linear distribution using the Kolmogorov-Smirnov test. Both of our results suggested that the voltage of those devices decreases in a linear model (with p-values of 0.847 and 0.964). Since the voltage decay during the executions is linear, we use the mean, using the maximum and the minimum voltage during the experiments, to estimate the current voltage in Chapter 3.

In recent versions of Android Power Profiler, it is possible to gather the voltage at the moment of the execution. Because of that, in Chapter 4 we used the voltage informed by the profiler to calculate the energy consumed instead of the mean.

Nucci et al. (2017) compared Android's tools for measuring energy consumption with the measurements of an oscilloscope (coarse-grained measurement) and their results indicate that Android's tools are accurate.

2.6 STATIC PROGRAM ANALYSIS

Software systems are complex objects by nature. When this complexity rises beyond human analysis capacity, developers usually make use of two different methods to analyze computer software: Static program analysis, done without the system (usually performed on the source code or some intermediate representation); and dynamic analysis, done while the application is running.

Both methods are used to understand the behavior of software applications and have their own advantages. Dynamic analysis requires less instrumentation and tools to be done (it is currently largely used on unit tests, integration tests, system tests, and acceptance tests), can detect vulnerabilities in execution time, may be executed without the system source code, and it usually leads to less false negatives than static analysis. Still, it does not cover all execution cases and can be more difficult to find the problem's source. Static analysis can find security vulnerabilities, memory errors, resource leaks, violations of API rules, and detection of race conditions and other errors that usually do not appear in a typical execution of the application, pinpointing the exact point of error; it also detects problems much earlier in the development cycle, what has been shown to have a number of advantages (HANGAL; LAM, 2002; EMDEN; MOONEN, 2002). Automated static analysis can create a substantial number of false positives and false negatives, and it does not find any problems created by runtime variables. Static analysis can also be done manually but in that case it can become complex and time consuming.

In this work, we used T. J. Watson Libraries for Analysis (WALA[15]) to make the static analysis of benchmarks and applications. Previous works have already used WALA to help analyze energy consumption of software constructs (MANOTAS; POLLOCK; CLAUSE, 2014; HASAN et al., 2016; FERNANDES; PINTO; CASTOR, 2017).

2.6.1 T.J. Watson Libraries for Analysis

Watson Libraries for Analysis (WALA) is a library that supports static analysis of Java bytecode and related languages (e.g., Android Dalvik, .NET), and for JavaScript. Among the features provided by WALA we have: Java type system and class hierarchy analysis; Source language framework supporting Java and JavaScript; Interprocedural dataflow analysis; Context-sensitive tabulation-based slicer; Pointer analysis and call graph construction; SSA-based register-transfer language IR; General framework for iterative dataflow; General analysis utilities and data structures; A bytecode instrumentation library; and a dynamic load-time instrumentation library for Java.

Among the features offered by WALA, we will be focusing on only a subset to execute our experiments: Interprocedural data-flow analysis, Call graph construction, and Pointer analysis (within WALA, we used it with ZeroOneCFA policies).

Data-flow Analysis: Data-flow analysis is the process of gathering information about the variables used in a specific part of a program, attempting to estimate the value of it during the execution. The analysis consists in collecting information on each point of a function or procedure. The easiest way to do that is to analyze the boundaries of a determined block of code (e.g., the body of a method in Java). With that information, it is possible to know what will happen with the selected variables in a possible execution of the program.

Data-flow analysis is used in several different scenarios such as code optimization, when soundness or automation is required, or to detect unwanted behavior. By supporting data-flow analysis, we mean that we also consider the calling context of a method call. That is useful to analyze the origin of the variables.

Call graph: Call graphs are used to represent the calling relationships between functions (or

[15] <http://wala.sourceforge.net/wiki/index.php/Main_Page>

methods) in a software system. On this graph, nodes represent the functions and the edges represent method calls.

They are used to analyze the behavior of a system and can be constructed dynamically or statically. The dynamic graph represents the execution of a software system and describes exactly one run of the program. During software development, it is normal to find such graphs when debugging in the shape of stack traces. The static graph is created to represent all possible executions of a software system. Since this is an undecidable problem, when creating static graphs, the tool needs to make approximations.

On WALA, the CallGraph class represents a call graph. It is important to note that WALA implementations of pointer analysis perform on-the-fly call graph construction.

Pointer analysis: Pointer analysis (or Points-to-analysis) is a technique that establishes what are the pointers pointing (memory locations or variables) at and how they can interact with it. Among the several implementations of pointer analysis, perhaps the most used is the Andersen-style pointer analysis (ANDERSEN, 1994). It can be expressed as *subset constraints*: "different program statements induce inferences of the form 'points-to set A is a subset of points-to set B', or, computationally, 'add all elements from points-to set A to points-to set B'" [16].

WALA provides a framework using exactly an Andersen-style pointer analysis. There are three different context-sensitivity policies available to customize the analysis: ZeroCFA is a simple, cheap, context-insensitive pointer analysis. It is the fastest and simplest option available. ZeroOneCFA provides an approximation of the standard Andersen's pointer analysis. ZeroOneContainerCFA extends the previous policy with unlimited object-sensitivity for collection objects. Each one of those offers increasingly more features but also is increasingly more expensive. We experimented with the different policies and, in our experiments, opted to use the ZeroOneCFA policy. This policy is the least expensive policy (in terms of performance) that offered all the necessary resources that we needed for our studies.

2.7 DESIGN DIVERSITY

Creating reliable software and ensuring that it does not enter in an unacceptable state (i.e., a state where the program does not work as intended) should be a priority of every developer. More significantly, in certain critical systems, failure cannot be an option, since

[16] <https://yanniss.github.io/points-to-tutorial15.pdf>

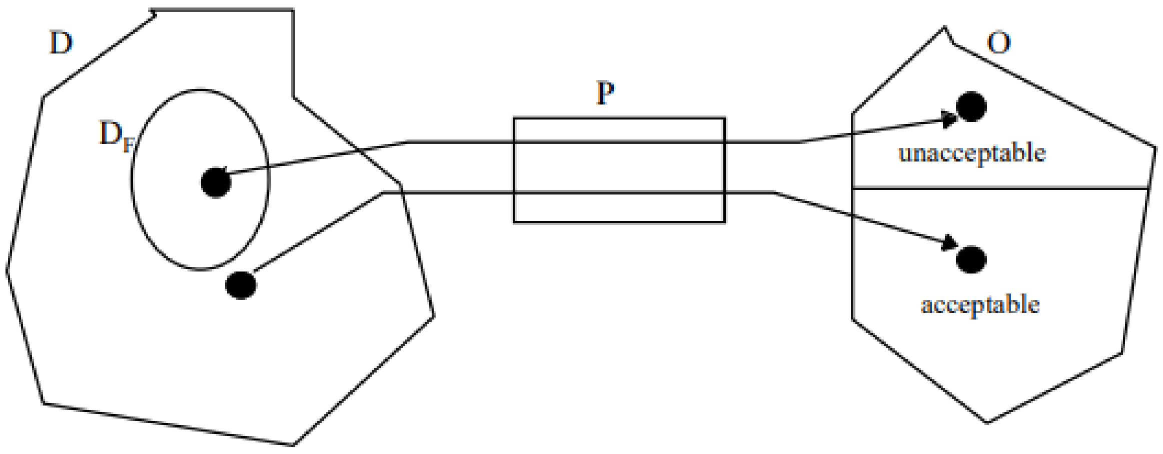


Figure 4 – Conceptual model of the software failure process. Program execution is a mapping from the set D, of all possible demands (sequences of input values), into the set of output sequences, O. Df represents the totality of all demands that the program, P, cannot execute correctly: they map into unacceptable output sequences^{14}.

the (human and financial) consequences can be catastrophic (e.g., airship control and stock market systems). To enhance reliability, and reduce the probability of system failure, one of the existing approaches is design diversity[17].

Figure 4[18] shows a representation of this model. One way to model the execution of a software-based system is to describe it by means of demands and outputs. The set of demands, D, represents all possible inputs used on the program's execution, containing traditional software inputs and possible external factors like weather or location. All points of D that incur in a failure are included in a subset called Df. The set of outputs, O, represents all the possible outcomes of the program's execution. The set of demands can be complex to enumerate or classify, on the other hand, the outputs can always be classified in one of two groups: acceptable or unacceptable.

The objective of design diversity in the context of fault tolerance is to reduce Df as much as possible. To do that, it employs several different techniques, such as: independently developed solutions to software pieces, defensive programming, recovery mechanisms, exception handling, and so on.

Diversity can be seen as a possible solution to reduce unwanted characteristics of a software-based system, often by means of redundancy. When comparing to works analyzing energy consumption, one can draw a parallel between design diversity in fault tolerance and what we call *energy design diversity*.

[17] Diversity here has the sense of a variety of solutions
[18] Figure originated from Littlewood, Popov e Strigini (2001)

2.7.1 Energy Design Diversity

Building an application can be a complex task, requiring developers to deal with interconnected and sometimes conflicting objectives in order to solve non-trivial problems. One way to mitigate this complexity is to leverage the availability of software solutions such as libraries, APIs, frameworks, gists[19], and answers from Q&A sites such as StackOverflow[20]. In this context, designing and implementing software has become a task of selecting appropriate solutions among multiple options (BALDWIN; CLARK, 2000) and combining them to build working systems. We call *energy variation hotspots* the programming constructs, idioms, libraries, components, and tools in a system for which there are multiple, interchangeable, readily-available solutions that have potentially different energy footprints.

A number of papers have measured and analyzed different types of energy variation hotspots, such as programming languages (OLIVEIRA; OLIVEIRA; CASTOR, 2017; PEREIRA et al., 2017; GEORGIOU; SPINELLIS, 2020), API usage (AGGARWAL et al., 2014; LINARES-VÁSQUEZ et al., 2014; ROCHA; CASTOR; PINTO, 2019), thread management constructs (PINTO; CASTOR; LIU, 2014b; LIMA et al., 2019), data structures (HASAN et al., 2016; PEREIRA et al., 2016; PINTO et al., 2016; LIMA et al., 2019; OLIVEIRA et al., 2021), color schemes (LI; TRAN; HALFOND, 2014; LINARES-VÁSQUEZ et al., 2018), and machine learning approaches (MCINTOSH; HASSAN; HINDLE, 2019), among many others.

Unlike low-level abstractions, such as voltage and frequency scaling, developers are familiarized with energy variation hotspots. Moreover, they make it easy to experiment with different options to analyze their impact on energy, since there are readily-available alternative implementations. Furthermore, the cost of replacing one implementation by another tends to be low, since they usually share common specifications (PINTO et al., 2016). If these algorithms adhere to a common specification, a recommendation tool can analyze their usage context so as to recommend a potentially efficient alternative. This can yield energy savings at a cheap cost in terms of development effort and does not require specialist knowledge.

The idea behind energy design diversity is using a more energy efficient option to improve an application while keeping the same functionality. Although both implementations have an acceptable output for the same demand, energy savings can be achieved by using the most efficient option.

[19] <http://gist.github.com>
[20] <https://stackoverflow.com>

In a number of cases, there is no single solution to reduce the energy consumption of an energy variation hotspot, as the solutions vary with external and internal factors. As an example, when analyzing the data structures and knowing that different implementations have different costs to each operation, we could not recommend a more energy efficient collection's implementation without knowing the number of insertions, removals, and access to variables in the original implementation. Having a variety of distinct energy efficient implementations let us make a better recommendation in a greater number of different scenarios.

This thesis tries to leverage the concept of energy design diversity by combining it with the idea of *energy variation hotspots*. We will use it as proof-of-concept of how design diversity can improve energy-aware software: When considering development approaches, we have the possible different implementations on different programming languages to solve the same problem. We noticed that using the same algorithm (only translated to another programming language) in two different approaches resulted in quite diverse energy consumption data. When considering Java collections, we have analyzed the use of different implementations to solve the same problems. Those changes were made automatically, without the interference of any developer. Usually the changes were very simple (many recommendations changed only a single line of code).

The main idea of diversity here is that by having multiple solutions for the same problem, the developer can use whatever is best, depending on the expected output. In the specific case of energy design diversity, the developer will be prioritizing energy rather than other software quality attributes (e.g., performance, memory space).

One advantage of using energy design diversity is that as the implementations are already available, there is considerable less effort to implement modifications based on energy design diversity. Developers can focus on selecting appropriate software constructs and exploring their impact, instead of building new options. Although simple, the changes can have quite positive impacts on the energy efficiency of the application (OLIVEIRA; OLIVEIRA; CASTOR, 2017).

Other ways to reduce energy consumption, like frequency and voltage scaling, and use of power efficient components, can be quite complex. That complexity can result in developers reverting their energy-improvement changes over time (MOURA et al., 2015). Using energy design diversity, we hope to reduce the complexity of optimizing the application by lowering the need to understand hardware-specific details, increase the pool of options and lower the implementation overhead.

There is a slight difference between the concept of energy design diversity and energy-aware refactoring. As defined by Fowler (2018), refactoring is:

> **noun:** a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior
>
> **verb:** to restructure software by applying a series of refactorings without changing its observable behavior.
>
> "Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring."

By definition, energy-aware refactoring would be change the code to improve energy efficiency instead of making it easier to understand or cheaper to modify.

The main concept of energy design diversity is that there is several different solutions for a specific problem and that using the most energy efficient solution would reduce the energy consumption of an application.

This concept could lead to a single refactor, a series of refactors, or to the whole rewrite of the solution. For example, two applications developed in distinct programming language could have the same functionalities. Nevertheless, in the case of a more energy efficient programming language, energy-aware refactoring would not help a developer trying to reduce the energy footprint of their app. Because of this core different, an application design using the concept of energy design diversity could be very different from a app refactored to consumed less energy. Energy design diversity gives us options to refactor a program. Since these options adopt the same interfaces and have functionally almost identical, refactoring to leverage them is intuitive.

In summary, although the concept of energy design diversity could (and often does) lead to energy-aware refactoring, ultimately, they deal with on layers of abstractions.

3 THE ENERGY FOOTPRINT OF ANDROID DEVELOPMENT APPROACHES

Currently, there are four approaches to develop apps that are directly endorsed by Google: Java, the default language for Android development; Kotlin; JavaScript, with the support of some frameworks, one can develop a full app using the web toolkit (HTML, CSS and JavaScript); C/C++, through the Native Development Kit (NDK), which makes it possible for a developer to write the majority of an application in C/C++. In the latter case, there is a trade-off between an increase in complexity and the benefit of (potentially) improved performance. In summary, Android apps can be written entirely in Java or Kotlin (**native** apps), in JavaScript-related technologies (**web** apps), or in a combination of more than one programming language (**hybrid** apps)[1]. Most of the Android apps, however, are written entirely in Java. In a sample of 109 projects we examined from F-Droid, only 4% use Javascript and 4% use the NDK resources to improve performance. Table 2 has a list of all apps from our sample.

Although one can find scientific and anecdotal evidence about the performance of Android

[1] The hybrid app denomination it is also used to refer to JavaScript-powered apps. In this work, we consider a hybrid app to be any app that uses more than one development approach.

Table 2 – Applications collected from F-Droid. From the 109 apps, 104 were developed using Java, five making use of NDK, and five were developed using JS.

Programming Languages	**Applications**
Java	1x1 clock, Anagram Solver, Allsimon/Alldebrid, AndroidRun, android-obd-reader, thialfihar/apg, bpear96/ARChon-Packager, applocker, google/google-authenticator-android, Sash0k/bluetooth-spp-terminal, boardgamegeek, jchmrt/clean-calculator, bitfireAT/cadroid, callmeter, Car Report, CineCat, Clover, CountdownTimer, Cowsay-android, CricketsAlarm,DeepScratch, derandom, dotty, Drinks, Droid-Beard, Earmouse, esms, EasyDice, EnigmAndroid, external-ip,falling for reddit, Fish,Flashlight, FreeOTP, frostwire-android, GetBack GPS, Gobandroid, HandyNotes, Hash It!, HeartRateMonitor, HeaterRC,HUD, ICSdroid, IntentRadio, JAWS, Matrix Calc, MAXS Module LocationFine, MAXS Module Ringermode, Migraine Tracker, Movian Remote, MobileOrg, MyOwnNotes, MultiPing, NetMBuddy, Network Discovery, Number Guesser, No Stranger SMS, OI About, OI Notepad, OpenMensa, Page Plus Balance, Photo Bookmark, Permissions, Pocket Talk, PocketSphinx Demo, Prism, Quest Player, RedScreenActivity, ReLaunch, S Tools, sanity, Search Light, SecDroid, Send to SD card, ShoppingList, Simply Do, Sky Map, Sokoban, SparkleShare, Speedo, StockTicker, Sudowars, TaigIME, Temaki, Timber, Toe, Torch, Tri Rose, Twister, Visualizer, Voodoo,CarrierIQ Detector, WebSMS Connector: GMX, weechat, WiFi Warning, WiGLE,Wifi Wardriving, WWWJDIC for Android, Yaacc, YubiClip, YubNub Command Line.
Java with NDK	24 Game, OpenWnn Legacy, PrBoom For Android, Lumicall, Mitzuli
JavaScript	ankidroid/Anki-Android, clipcaster, Overchan, RainTime, smeir/berlin-vegan-guide

apps written using these different development approaches[2,3], it is hard to find data about the differences in energy consumption. It is not yet clear whether the four aforementioned approaches lead to energy efficient applications.

This chapter aims to shed more light on the issue of energy efficiency among three out of the four different Android app development approaches. We compare energy consumption and performance of 33 benchmarks developed by several authors from Rosetta Code[4] and the Computer Language Benchmark Game (Computer Language Benchmark Game (CLBG))[5]. Our study consisted of executing multiple versions of each benchmark on a number of different mobile devices, while measuring execution time and energy consumption.

To measure energy consumption, we used the tools available through Android Power Profiler. These resources enable us to collect energy consumption information on a per-app basis. Our goal with this study is to provide an answer to the following research question:

- **RQ1.1. Is there a more energy efficient approach among the most common Android development models?**

We found out that for 26 out of the 33 analyzed benchmarks, JavaScript versions exhibited lower energy consumption than their Java counterparts. The Java versions of six of these benchmarks outperformed their JavaScript counterparts, even though they consumed more energy. This result indicates that, at least for CPU-intensive apps, Java may not be the most energy efficient solution.

For the CLBG benchmarks, we compared Java, JavaScript, and C++ versions, executing them on five devices with different characteristics, achieving similar results. We found out that, although there are some small variations in performance, the energy consumption relation between Java, JavaScript, and C++, remained similar across devices, with JavaScript having the edge over the other approaches concerning energy consumption, while exhibiting a good trade-off between energy and performance. We also noticed that using the NDK does improve performance.

Even though these are interesting results, Android apps in general behave differently from CPU-intensive benchmarks (RATANAWORABHAN; LIVSHITS; ZORN, 2010). Most of their execution time is spent waiting for user input or using sensors. Because of that, energy optimizations

2 <https://stackshare.io/stackups/android-vs-apache-cordova>
3 https://stackoverflow.com/questions/16156370/phonegap-vs-native-on-android-performance-test
4 <http://rosettacode.org/wiki/Rosetta_Code>
5 <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>

made on benchmarks may not have the same impact on the energy usage of real applications (SAHIN; POLLOCK; CLAUSE, 2016). This led us to question whether one could save energy by using a hybrid approach, adopting JavaScript and C++ in the more CPU-intensive parts of applications. Thus, we also provide an initial answer for the following research question:

- **RQ1.2. Is it possible to reduce the energy consumption of a native app by making it hybrid?**

We reengineered four open-source apps: three existing apps from the F-Droid repository[6] and one app developed by the National Institute of Science and Technology for Software Engineering (National Institute of Science and Technology for Software Engineering (INES))[7]. Three of those apps are also available at the Play Store. Each app was written in Java and we made parts of them run in JavaScript and C++. Our goal was to analyze whether using these approaches alongside Java impacted performance and energy consumption. From the benchmark analyses, we knew that using JavaScript and C++ often led to an improvement in performance, energy consumption, or both. However, we had no information about whether it was possible to make improvements in performance and energy consumption by using a hybrid approach. We analyzed different models for invoking Javascript and C++ snippets using Java code and measured the energy consumption in all cases. Our results indicate that it is possible to save energy using this hybrid approach - for one of the apps, a hybrid version using a combination of Java and C++ consumed 0.37J, under a certain workload, whereas the original version written entirely in Java consumed 32.92J.

Knowing whether small modifications in the code promote a non-negligible reduction in energy consumption empowers developers. Moreover, tool builders can introduce cross-language refactorings that support developers in reengineering existing applications when a hybrid approach may be beneficial. All data related to this study can be found at <https://secaada.github.io/msr2017/>.

This chapter is structured as follows: **Section 3.1** describes the methodology used on this chapter; **Section 3.2** displays the results from our experiments analyzing the energy consumption of Android development approaches; **Section 3.3** report our findings about the energy efficiency of the different development approaches; **Section 3.4** presents the threats to

[6] <http://f-droid.org>
[7] <https://github.com/ines-escin/>

validity; **Section 3.5** presents the related work, focusing on papers analyzing different aspects of energy consumption on Android, and **Section 3.6** concludes this chapter.

3.1 METHODOLOGY

The aim of this study is to analyze the most popular development approaches for Android and to establish whether they differ in terms of energy efficiency and performance. The metric we use to evaluate performance in this chapter is execution time.

In this section, we explain how we selected the analyzed benchmarks and apps (Section 3.1.1), and how we executed the experiments (Section 3.1.2).

3.1.1 Benchmarks and apps

Benchmarks were extracted from two software repositories: Rosetta Code and the Computer Language Benchmark Game (CLBG). Rosetta Code is a programming chrestomathy[8] site. It includes a large number of programming tasks and solutions to these tasks in different programming languages. CLBG is a website whose main purpose is to compare the performance of several programming languages. Both have been employed in prior work for comparing different programming languages and to analyze energy efficiency (NANZ; FURIA, 2015; COUTO et al., 2017; PEREIRA et al., 2017; LIMA et al., 2019; PEREIRA et al., 2021).

Table 3 lists all the benchmarks analyzed in this study. The benchmark set encompasses 23 benchmarks from Rosetta Code and 10 from CLBG. Most of the benchmarks from Rosetta have already been used in other studies (NANZ; FURIA, 2015).

All benchmarks from CLBG that have implementations in all three languages were used, which includes in some cases, sequential and parallel versions of the same benchmark.

Since the benchmarks from Rosetta were not built with optimal performance in mind, their performances vary widely. For example, in the NQUEENS benchmark, the solutions available at Rosetta took 20s to finish in Java and 69s in JavaScript. By converting the JavaScript version to use the same algorithm as the Java version, this new version took 12s to finish.

To illustrate the modifications made, we will present the original JavaScript algorithm for NQUEENS, the original Java version, and the modified JavaScript used on the experiment:

[8] A selection of literary passages from a foreign language assembled for studying the language; or a text in various languages, used especially as an aid in learning a subject.

Table 3 – The selected set of benchmarks and applications.

Source	**Benchmark or App**
Rosetta Code	BUBBLESORT, COMBINATIONS, COUNT IN FACTORS, COUNTINGSORT, GNOMESORT, HAPPY NUMBERS, HEAPSORT, HOFSTADTERQ, INSERTSORT, KNAPSACK BOUNDED, KNAPSACK UNBOUNDED, MATRIX MULT, MAN OR BOY, MERGESORT, NQUEENS, PANCAKESORT, PERFECT NUMBER, QUICKSORT, SEQNONSQUARES, SHELLSORT, SIEVE OF ERATOSTHENES, TOWER OF HANOI, and ZERO-ONE KNAPSACK
Computer Language Benchmark Game	BINARYTREES, FANNKUCK, FASTA, FASTA PARALLEL, KNUCLEOTIDE, NBODY, REGEXDNA, REGEXDNA PARALLEL, REVCOMP, and SPECTRAL
F-Droid	ANDOF, ENIGMANDROID, and TRIROSE
INES	NUCLEUSAPP

Código Fonte 1 – Original JavaScript NQUEENS

```
function queenPuzzle(rows, columns) {
    if (rows <= 0) {return [[]];}
    else {return addQueen(rows - 1, columns);}
}
function addQueen(newRow, columns, prevSolution) {
    var newSolutions = [];
    var prev = queenPuzzle(newRow, columns);
    for (var i = 0; i < prev.length; i++) {
        var solution = prev[i];
        for (var newColumn = 0; newColumn < columns; newColumn++) {
            if (!hasConflict(newRow, newColumn, solution))
                newSolutions.push(solution.concat([newColumn]))}
    } return newSolutions;
}

function hasConflict(newRow, newColumn, solution) {
    for (var i = 0; i < newRow; i++) {
        if (solution[i]     == newColumn          ||
            solution[i] + i == newColumn + newRow ||
            solution[i] - i == newColumn - newRow) {
                return true;}
    } return false;
}
```

Código Fonte 2 – Original Java NQUEENS

```java
    private static int[] b = new int[8];
    private static int s = 0;
    static boolean unsafe(int y) {
        int x = b[y];
        for (int i = 1; i <= y; i++) {
            int t = b[y - i];
            if (t == x || t == x - i || t == x + i) {return true;}
        } return false;
    }
    public static void putboard() {
        System.out.println("\n\nSolution " + (++s));
        for (int y = 0; y < 8; y++) {
            for (int x = 0; x < 8; x++) {
                System.out.print((b[y] == x) ? "|Q" : "|_");
            } System.out.println("|");
        }
    }
    public static void main(String[] args) {
        int y = 0; b[0] = -1;
        while (y >= 0) {
            do {b[y]++;}
            while ((b[y] < 8) && unsafe(y));
            if (b[y] < 8) {
                if (y < 7) { b[++y] = -1; }
                else {putboard();}
            } else {y--;}
        }
    }
```

Código Fonte 3 – Modified JavaScript NQUEENS

```
var b = [];
  function unsafe(y){
      var x = b[y];
      for(var i = 1; i <= y; i++){
          var t = b[y - i];
      if (t == x || t == x - i || t == x + i) {return true;}
      } return false;
  }
  function putboard(){
      for (var y = 0; y < 8; y++) {
          for (var x = 0; x < 8; x++) {
              print((b[y] == x) ? "|Q" : "|_");
          } print("|");
      }
  }
  function run(){
      var y = 0; b[0] = -1;
      while (y >= 0) {
          do {b[y]++;}
          while ((b[y] < 8) && unsafe(y));
          if (b[y] < 8) {
              if (y < 7) { b[++y] = -1;}
              else {putboard();}
          } else {y--;}
      }
  }
```

As the research focus is on the development approaches and not algorithm implementations, we manually analyzed all of Rosetta Code's benchmarks and if the solutions used distinct algorithms, we changed the slowest solution to use a similar process to the fastest.

Those modifications make the comparisons more balanced. As both languages are syntactically similar, adaptations were straightforward. Since the Rosetta Code benchmarks do not have a strong emphasis on performance, we have only used Java and JavaScript versions of them.

In CLBG, all implementations try to achieve the best possible performance. Therefore, the only modifications applied to those benchmarks were the ones necessary to execute them in the Android environment. We executed Java, JavaScript, and C++ versions of these benchmarks. Five implementations of benchmarks in Java and C++ used parallelism to solve the problems, thus improving performance. Sometimes this also led to an increase in the energy consumption, sacrificing energy efficiency (PINTO; CASTOR; LIU, 2014a). Single-core implementations of such benchmarks, whenever available, were also analyzed, as a way to verify whether a slower execution would lead Java and C++ to be more energy efficient. All benchmarks in JavaScript were optimized for a single core. As the CLBG benchmarks were more reliable, we executed them on every device available.

Benchmarks may not be useful to provide an answer for **RQ1.2**, since they work differently from apps (RATANAWORABHAN; LIVSHITS; ZORN, 2010; SAHIN; POLLOCK; CLAUSE, 2016). Therefore, we have used four real-world, open-source apps for the study. These apps appear in the last two rows of Table 3. ANDOF is an app to calculate depth of field for photography, NUCLEUSAPP[9] is an app that aims to facilitate the collection of waste cooking oil, ENIGMANDROID is an app that simulates the enigma machine from the 2nd World War and TRIROSE is an app that mathematically generates unique and intricate rose graphs. These apps were chosen because, even though they spend much of their time on input and output operations, they perform a non-trivial amount of computation. NUCLEUSAPP comprises approximately 1k lines of code (LoC), TRIROSE 1.11000 Lines of Code (kLoC), ANDOF 1.7kLoC and ENIGMANDROID 4.6kLoC. ANDOF, NUCLEUSAPP and TRIROSE can also be found at the Play Store.

The four analyzed apps were written entirely in Java. We have reengineered their most computing-intensive parts to use JavaScript and C++, creating hybrid web apps and hybrid NDK apps, respectively. To detect those parts, we used the profiler from Android Studio. It is important to determine the frequency with which Java will invoke the part written in another language. If the former invokes the latter too frequently, much of the execution time and possibly energy consumption will be dominated by the overhead of cross-language invocations. If these invocations are too infrequent, application functionality may be compromised.

In this work, we used three different models to manage cross-language invocations. In the *Stepwise* model, one method in Java is mapped directly to a function in JavaScript or C++ and each time the method is supposed to be called, the function is used instead. In the *Batch* model, one method in Java is mapped to a function in JavaScript or C++, bundling several calls of the method, returning the aggregated result to Java. This model reduces the communication overhead by dividing processing duties between Java and the other language. Finally, in the *Export* model, all the work to be performed in a sequence of method invocations in the original version is mapped to a single JavaScript or C++ function. Creating a hybrid app by modifying an existing one instead of coding a new app allowed us to verify whether it was

[9] <https://github.com/ines-escin/NucleusApp>

Table 4 – Machines used on the experiments. *Age* shows how old the device was when we executed the experiments (in years)

Device	Version	RAM	Chipset	CPU (GHz)	Battery(mAh)	Age
LG L90	5.0.2	1GB	Snapdragon 400	4-core 1.2	3000	2
Nexus 5	5.1	2GB	Snapdragon 800	4-core 2.3	2300	3
Nexus 7	5.1.1	2GB	Tegra 3	4-core 1.2	4325	4
Samsung J7	5.1	1.5GB	Exynos 7580	8-core 1.5	3000	1
Zenfone Selfie	6.0.1	3GB	Snapdragon 615	2x 4-core 1.7/1.0	3000	1

possible to increase energy efficiency with minor modifications, meaning minimum effort for developers. Each execution of the hybrid apps had the same input and output as the original version.

All benchmarks and applications were deployed using Android Studio v2.2.2.

3.1.2 Running the experiments

All benchmarks were executed using a preset workload, individual to each benchmark. The size of each workload was determined so as to guarantee it was executed for at least 20s even though sometimes that was not possible due to memory limitations. All web versions of the benchmarks were executed inside a wrapper originated from Apache Cordova.

When working with the C++ versions of the benchmarks, it was not always possible to pick the fastest version available in CLBG. This is due to the fact that some of the low-level approaches that were used in order to achieve maximum performance were not available in NDK. For example, the fastest solution of the REVCOMP benchmark uses the not-thread-safe version of the functions *fwrite* and *fgetc*. For that reason we tried to use the fastest compiling version of the benchmarks.

Table 4 lists the devices we employed in this study. Five devices were used to run the benchmarks, four smartphones and a tablet. We tried to achieve diversity by selecting devices with different manufacturers, chipsets, and CPUs. All devices run the Lollipop version of Android (5.x) or a more recent one. Updates to Android were only applied by means of its updating tool, to emulate a realistic usage scenario. Alternatively, we could have manually installed the same version on every device but this method would require unofficial versions. Thus, we ended up with some slight variations of Lollipop.

We executed every version (Java, JavaScript, and C++) of the CLBG benchmarks at least

10 times in each device. In most cases, however, each benchmark version was executed 30 times in each device, with the Nexus 5 smartphone being the only exception. The reason for this lower number of executions is that this phone stopped turning on before we could perform the full set of executions. For the remaining four devices, we performed 30 executions of each version of each CLBG benchmark. All C++ versions of the benchmarks were executed at least 30 times in each device. The versions of the Rosetta Code benchmarks were executed in the Nexus 5 only. We choose to focus on the CLBG because it was a more fair comparison. As previously mentioned, the benchmarks from Rosetta Code are not optimized as the benchmarks found on CLBG. The execution time and battery measurements were obtained using custom-built scripts. To collect measurement data, we have developed a webservice, the `Dashboard App`, that combined with the aforementioned scripts, automatically performs clean-up on the device, loads the app to be executed, executes the app, and records the results.

We have also executed each version (native, hybrid web, and hybrid NDK) of each of the apps mentioned in Section 3.1.1 30 times. Each app was executed using a predefined workload aiming to simulate realistic usage scenarios. Alternatively, we could also have employed an automated testing tool, such as monkeyrunner[10]. This approach would be particularly advantageous for apps with a stronger emphasis on user interaction, which is not the case on the selected apps.

Our results point out that the device has little influence on the outcome of which development approach is the most energy efficient on a specific application. Because of that, the experiments to answer **RQ1.2**, about the energy efficiency of applications hybridization, were executed only on Samsung J7. The benchmarks experiments answering **RQ1.1** were executed on all five devices. Each experiment starts with the device fully charged and keeps running until either all the executions are over or the battery reaches 40% charge. Using this conservative threshold, we aim to guarantee that the device will never go into battery-saving mode.

3.2 STUDY RESULTS

This section presents the results of this study. Section 3.2.1 presents the results for **RQ1.1** whereas Section 3.2.2 examines **RQ1.2**. All the data from the benchmarks and apps can be found at <https://secaada.github.io/msr2017/>

[10] <https://developer.android.com/studio/test/monkeyrunner/index.html>

3.2.1 Is there a more energy efficient app development approach?

We separated the benchmark results in two parts. The first one analyzes the data collected from the benchmarks from Rosetta Code and the second one from CLBG. For benchmarks with both sequential and parallel versions, to distinguish between the two, we marked the parallel version with a 'p' at the end of the benchmark name, e.g., FASTA and FASTAP.

Figure 5 shows the execution time (lines) and energy consumption (bars) of the Rosetta Code benchmarks. Overall, the JavaScript benchmarks exhibit lower energy consumption and execution time. The Java versions of these benchmarks consume a median 2.09x more energy than their JavaScript counterparts. Furthermore, the Java versions spend a median of 1.52x more time to finish their execution. The figure shows that in 18 out of 22 benchmarks from Rosetta, the JavaScript versions consumed less energy and in 16 they exhibited lower execution time. Finally, in 3 of the 7 benchmarks where Java was faster, it also consumed more energy than JavaScript, which suggests a non-linear relation between energy and performance.

Figure 6 shows the execution time (lines) and energy consumption (bars) of the CLBG benchmarks across all devices. The FASTAP benchmark consumed the lowest amount of energy in Java, with JavaScript, and C++ consuming a median of 1.72x and 3.33x more energy across all devices, respectively. The REGEXDNAP benchmark was the most energy efficient in JavaScript, with the other versions in Java and C++ consuming a median of 13.93x and 9.05x more energy across all devices, respectively. The REVCOMP benchmark was the most energy efficient in C++, with Java and JavaScript consuming a median of 2.80x and 19.64x more energy across all devices, respectively. In spite of these differences between the development approaches, Figure 6 shows that their relationship does not differ much across devices, in terms of energy consumption. For example, for the spectral benchmark, Java consumed the most energy in all devices and JavaScript consumed the least, although the amount of energy consumed in each device was different. This can be observed for most of the benchmarks.

When comparing approach-to-approach the 50 benchmark-device pairs, contrasting JavaScript and Java, the former exhibited a lower energy consumption in 36 out of the 50 benchmark-device pairs and better performance in 19 out of the 50. When compared with C++, JavaScript exhibited lower energy consumption in 37 out of 50 and better performance in 25 out of the 50. In a direct comparison between Java and C++, the latter had a lower energy consumption in 28 out of the 50 benchmark-device pairs and a better performance in 27 out of the 50. These results suggest there is no overall winner in terms of performance, although JavaScript

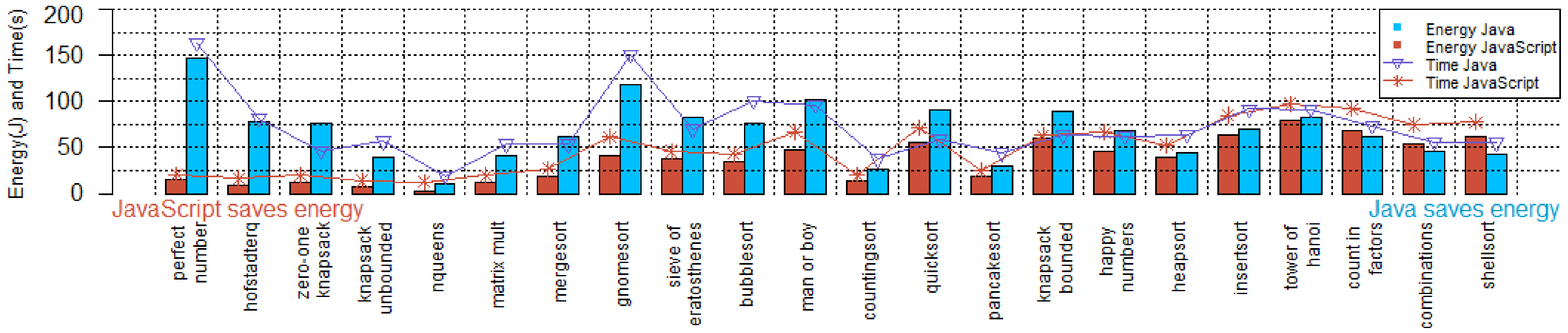


Figure 5 – Results of the benchmarks from Rosetta Code. The bars are sorted using the relative gain in energy consumption for each benchmark.

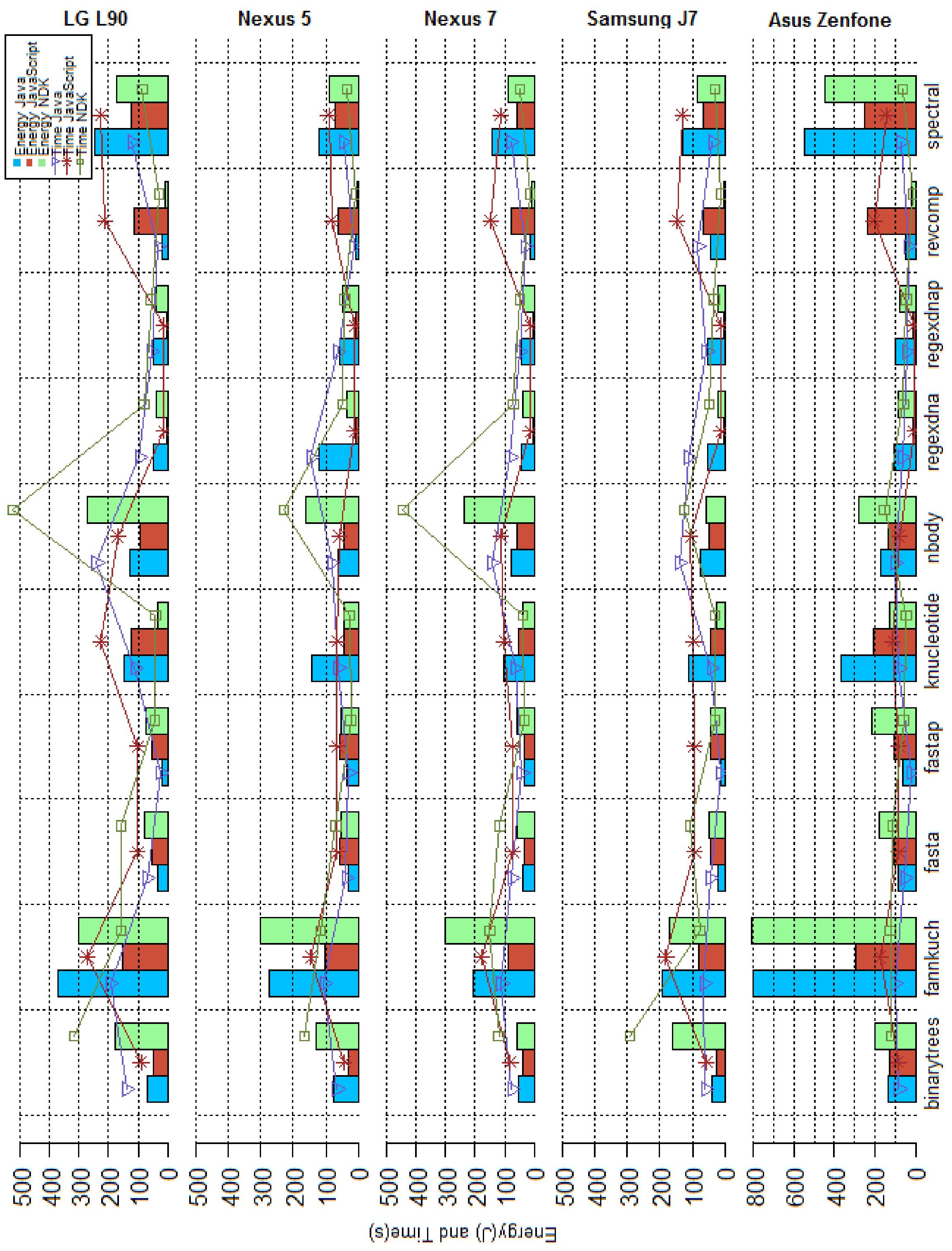


Figure 6 – Performance and energy results of the benchmarks from the Computer Language Benchmark Game (CLBG) on all devices

consumed less energy and had the worst performance on average across all devices.

To get a better understanding of which approach exhibits the best performance and energy consumption for each device, we analyzed the 50 benchmark-device pairs across all approaches. Performance-wise, Java had the best performance in 13 out of the 50 pairs, JavaScript in 18, and C++ in 19. Energy-wise, Java consumed less energy in 9 out of the 50 pairs, JavaScript in 31, and C++ in 10. Table 5 graphically summarizes these results. In this table, we use blue to indicate that Java won for that particular benchmark running on that particular device, red for JavaScript, and green for C++. Device names are abbreviated: L90 stands for LG L90, N5 for Nexus 5, N7 for Nexus 7 to N7, J7 for Samsung J7, and ZF for Zenfone. For example, JavaScript had the lowest energy consumption on the BINARYTREES benchmark on the LG L90 whereas C++ had the lowest energy consumption for the REVCOMP benchmark on the same device.

Table 5 – The right-hand side presents the development approach with the best results for energy and the left-hand side the development approach with the best performance, for each device.

Energy	L90	N5	N7	J7	ZF
binarytrees	🟥	🟥	🟥	🟥	🟥
fannkuch	🟥	🟥	🟥	🟥	🟥
fasta	🟦	🟦	🟥	🟦	🟦
fastap	🟦	🟦	🟦	🟦	🟦
knucleotide	🟩	🟩	🟩	🟩	🟩
nbody	🟥	🟥	🟥	🟥	🟥
re5gexdna	🟥	🟥	🟥	🟥	🟥
regexdnap	🟥	🟥	🟥	🟥	🟥
revcomp	🟩	🟩	🟩	🟩	🟩
spectral	🟥	🟥	🟥	🟥	🟥

Performance	L90	N5	N7	J7	ZF
binarytrees	🟥	🟥	🟦	🟥	🟦
fannkuch	🟩	🟦	🟦	🟦	🟦
fasta	🟦	🟦	🟩	🟦	🟦
fastap	🟦	🟩	🟩	🟦	🟦
knucleotide	🟩	🟩	🟩	🟩	🟩
nbody	🟥	🟥	🟥	🟥	🟥
regexdna	🟥	🟥	🟥	🟥	🟥
regexdnap	🟥	🟥	🟥	🟥	🟥
revcomp	🟩	🟩	🟩	🟩	🟩
spectral	🟩	🟩	🟩	🟩	🟩

Legend	
🟦	Java
🟥	JavaScript
🟩	C++

3.2.2 Can a hybrid approach to app development save energy?

The results discussed in Section 3.2.1 suggest that the three approaches for app development in Android have different trade-offs in terms of energy consumption. However, Android applications are predominantly written in Java - in a random sample[11] of 109 apps among the 1,600 apps in F-Droid. They noticed that only 5 apps used JavaScript and another 5 used NDK in any way[12]. Nevertheless, in Android, it is possible for Java code to invoke JavaScript and vice-versa. Thus, since Java is the predominant approach to write Android apps, it may be possible to save energy by retrofitting existing apps to perform part of their work in JavaScript or C++. The major obstacle to this approach is that there is the overhead of cross-language invocations (GRIMMER et al., 2013). In this section we examine whether it is possible to compensate for this overhead so as to make existing apps more energy efficient. It is worth noting that using two (or more) programming languages to develop an app could reduce its maintainability.

The four apps we have analyzed, ANDOF, ENIGMANDROID, NUCLEUSAPP, and TRIROSE have different behaviors. The idea is that by using different strategies to convert the source code between different programming languages we hope to achieve a better performance, both in time and energy consumption. We will use as many models as possible, within the boundary of the application behavior.

In TRIROSE's case, waiting for another development approach to perform the entire computation could impose seconds of delay (the *Export* model – Section 3.1.1) and thus is not acceptable because it could hinder the user experience. Waiting a couple seconds is not a problem in cases like ENIGMANDROID. On the other hand, the only model that makes sense for the ENIGMANDROID is *Export*. Because this app makes heavy use of the CPU, the *Stepwise* model (Section 3.1.1) would increase the overhead with more cross-language invocations with no advantage to the user and the *Batch* model (Section 3.1.1) would be suboptimal, as we would have needed to pass smaller parts of the workload as arguments. For ANDOF, the *Stepwise* model was the only realistic model. This app works updating the data to the user in real time. Any other model would hinder the user experience, imposing an unnecessary delay. NUCLEUSAPP mainly focuses on IO inputs (from sensors and web data) and the information is presented to the user in real time. Batching data would create a delay between the required data and the screen update. Thus, because of the real time requirement, *Stepwise* is the only viable model. In summary, for ANDOF and NUCLEUSAPP, we employed the *Stepwise* model, for Tri Rose, the *Stepwise* and *Batch* models, and for ENIGMAANDROID, the *Export* model.

The specific workload for each app is determined in a way to try to make each execution run in approximately 30s, keeping a low relative standard deviation as explained before. The

[11] this scenario may have changed since our sample was collected in 2016
[12] Two researchers manually analyzed the source code at GitHub

Table 6 – Results for the modified apps. SD stands for standard deviation.

App	Approach	Time(s)	SD	Energy(J)	SD
anDOF Stepwise	Java	62.79	0.60	32.92	0.07
	Web	250.32	0.70	271.53	0.41
	NDK	6.26	0.17	0.37	0.03
NucleusApp Stepwise	Java	122.40	0.45	6.50	0.30
	Web	122.27	0.24	6.65	0.36
	NDK	122.30	0.18	6.55	0.44
Enigma Export	Java	89.21	2.46	38.13	0.11
	Web	27.78	0.89	12.22	0.06
	NDK	30.11	0.76	13.58	0.06
TriRose Stepwise	Java	27.84	0.03	4.03	0.02
	Web	53.85	0.05	7.78	0.01
	NDK	27.92	0.56	4.14	0.02
TriRose Batch	Java	26.74	0.02	3.95	0.02
	Web	27.49	0.03	4.21	0.02
	NDK	26.75	0.02	3.97	0.02

workload for anDOF was 24×10^5 changes in a scroll that controls the depth of field. For each change, a method is called to recalculate it. The workload for NucleusApp consisted of the geolocation of 100 ecopoints, places where one can dispose of waste cooking oil. For EnigmAndroid we used a randomly generated String with $45,000$ characters. In TriRose, the workload for both *Stepwise* and *Batch* models consisted of drawing 1,500 lines on the screen, since the execution times were more similar.

Table 6 presents the results. In two cases where we employed the *Stepwise* model on hybrid web apps, it degraded performance and boosted energy consumption. For example, the hybrid web version of TriRose using the *Stepwise* model took 92.87% longer to finish than the hybrid NDK version and consumed 87.92% more energy.

This result suggests that unless it is possible to group parts of the work so as to minimize this overhead, building a hybrid web app will not save energy, due to the cost of invoking JavaScript code. On the hybrid NDK apps, we got significant improvements on performance and energy consumption using the *Stepwise* model in AnDOF and got almost identical results to the Java version in TriRose. Invocations to C++ snippets are done using the NDK and incur in minimal overhead for the app as in the cases above. Using the NDK to improve the application may be beneficial even if the developer needs to continually use cross-language functions.

Using the *Batch* model on TriRose, the execution was changed to keep the results of the calculations of the points that were used to draw the curves in a buffer of 11×10^4 positions. We applied this modification to the original native version, the hybrid web app, and the hybrid NDK app. With the *Batch* version of the app we got negligible improvements in performance and energy consumption in the native and hybrid NDK apps. Nonetheless, using the *Batch* model in the hybrid web app made it two times faster and cut its energy consumption almost in half, when compared to the corresponding *Stepwise* version. Furthermore, the difference in performance between the hybrid web app and the native and hybrid NDK apps using the *Batch* model was only 2.77%, with the hybrid web app consuming 6.04% more energy.

We only apply the *Export* to EnigmAndroid. It would not make sense in the context of TriRose because the latter needs to continually update the UI. Updating the graphical interface from JavaScript code in a Java app is non-trivial because Java and JavaScript employ different paradigms for user interface. We also did not use it on anDOF because it creates a potentially unrealistic scenario. The depth of field needs to be informed to the user in real-time. Thus, the aggregated modifications may not be so useful. The results using the *Export* model represent a best case scenario, since the cross-language invocation overhead is almost entirely diluted. In the hybrid approaches, the app runs on Java but the cryptographic processing of the String is made on the other programming languages. In the *Export* model, both hybrid apps improved the performance and energy consumption of the app, with Java version consuming 3.21x more time and 3.12x more energy than the hybrid web app and 2.96x more time and 2.80x more energy than the hybrid NDK app. The hybrid web app was slightly better in performance and energy efficiency than the hybrid NDK app. The latter consumed 11.12% more energy and took 8.39% longer to finish.

Using the *Stepwise* approach on NucleusApp did not improve or deteriorate the performance or energy efficiency of the hybrid apps. That result suggests that hybridization of IO-intensive apps may not result in improvements.

Even though these modifications promoted non-negligible improvements in performance and energy efficiency, they did not require large-scale modifications. For each app, JavaScript, and C++ files had, respectively: anDOF, 160 and 192 LoC; NucleusApp, 16 and 65 LoC;

EnigmAndroid, 173 and 281 LoC; TriRose, 100 and 166 LoC. Changes were relatively simple and represent less than 10% of the total lines of code of each app.

3.3 DISCUSSION

In this section we discuss some of the most important lessons learned from this study. We believe these lessons are useful for both researchers, who can investigate each one in more depth, and practitioners, who can leverage them for their app development activities.

There is no overall winner. Analyzing the benchmarks, there was no better approach in all scenarios. It is possible to state that, in general, JavaScript consumed less energy than Java and C++. It had a lower average energy consumption per benchmark and the JavaScript versions of most benchmarks exhibited the lowest energy consumption when compared to Java and, where applicable, C++ versions. Nevertheless, this result was not universal. For example, in the REVCOMP benchmark, the JavaScript versions had by far the highest energy consumption. In addition, for performance, especially when we look only at the CLBG benchmarks and the four apps, the results were much more mixed, with Java and C++ versions outperforming JavaScript in most cases.

Both energy and performance are important in practice and most developers would be interested in balancing the two. Thus, we choose to look at the weighted energy-delay product (Energy-delay product (EDP)) by Cameron, Ge e Feng (2005). The weighted EDP is calculated using the following equation:

$$ExT^w \tag{3.1}$$

where E represents energy, T time, and W the weight factor. As we choose to focus on the energy, the weight is set to 1. This measure highlights the trade-off between energy and execution time (the lower the better). The boxplot in Figure 7 shows the EDP for all benchmarks across all devices, without outlines. It highlights that some benchmarks perform better in a certain language, e.g., FASTA and REVCOMP in Java and REGEXDNA and REGEXDNAP in JavaScript, whereas others perform worse, e.g., NBODY and BINARYTREES in C++ and REVCOMP in JavaScript. Furthermore, for some benchmarks the difference between the languages is unclear (FANNKUCH and SPECTRAL).

Even though JavaScript usually has lower energy consumption per benchmark, the benefit of running the benchmark faster may overshadow this when looking at an aggregated metric.

When considering all the 50 benchmark-device pairs, JavaScript presented a better EDP on 32 out of 50 pairs when compared to Java, and 29 out of 50 when compared with C++. When comparing Java and C++, the latter had a better EDP on 27 out of 50 pairs. This result suggests that even though there is no overall winner, JavaScript exhibits a good trade-off between energy usage and performance, besides consuming less energy in most cases.

These results show that, in certain scenarios, the different approaches may have significant differences in performance and energy consumption. Having that information, developers can try to improve an app in two ways: (i) by experimenting with different approaches, depending on application requirements; and (ii) by exploring combinations of these approaches and building a hybrid app.

The development approaches differ little across devices, in terms of energy usage. For the set of benchmarks, apps, and devices we analyzed, our measurements indicate that a development approach that consumes less energy in one device is likely to do so in other devices as well. A quick look at the bars of Figure 6 or to the Table 5 highlights this result.

This finding has a clear practical implication. There are thousands of Android smartphone models in the market. Although there are initiatives trying to collect data from a multitude of devices (PEREIRA et al., 2021), it is unfeasible to analyze and test an application on every device. Our results imply that, when trying to optimize the application by using the more energy efficient development approach, optimizing it for a small and diverse subgroup of device models may be sufficient.

Faster != Greener. The main reason for the good performance of some of the Java and C++ benchmarks is parallelism. While all the JavaScript versions run sequentially, the Java and C++ versions of 5 benchmarks (FANNKUCH, FASTAP, KNUCLEOTIDE, REGEXDNAP, and SPECTRAL) are capable of leveraging multicore processors to improve performance. All but one (REGEXDNAP) of these versions outperform the corresponding JavaScript version. However, all these benchmarks, with the exception of FASTAP for Java and KNUCLEOTIDE for C++, consumed more energy. This is consistent with previous work (PINTO; CASTOR; LIU, 2014b; LIMA et al., 2019) that found out that, for programs capable of benefiting from multicore processors, performance is often not a proxy for energy consumption. Other work (COUTO et al., 2014) have shown a high correlation between execution time and energy consumption but also stated that it was not the only factor. Complementarily, even when a benchmark does not leverage parallelism, the relationship between energy and execution time is unclear. For

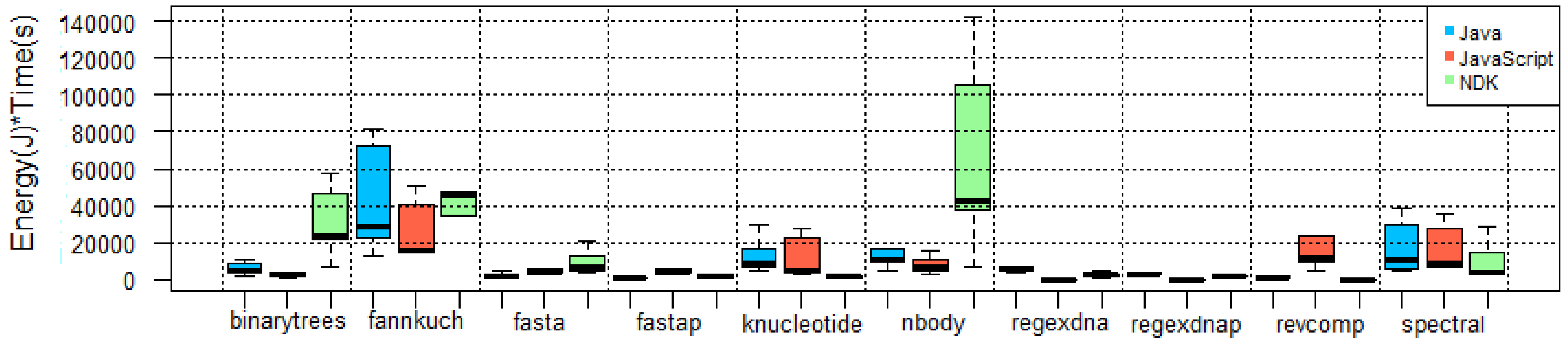


Figure 7 – Energy-delay product (EDP) results of the benchmarks from the Computer Language Benchmark Game (CLBG) across all devices without outlines.

Table 7 – Average data from the division between the values for the sequential and parallel versions of the FASTA and REGEXDNA benchmarks. A value greater than 1 in a cell means that the sequential version had higher execution time, energy consumption, or EDP.

Benchmark	Language	Energy	Speedup	EDP
REGEXDNA	Java	1.20	1.86	2.33
	C++	0.95	1.39	1.32
FASTA	Java	1.37	2.52	3.70
	C++	1.02	3.21	3.35

example, none of the versions of the BINARYTREES benchmark exploits parallelism. Notwithstanding, Java exhibited the best execution time while JavaScript exhibited the lowest energy consumption.

Parallelism may be a good option anyway. Among the selected benchmarks, five (FANNKUCH, FASTAP, SPECTRAL, REGEXDNAP and KNUCLEOTIDE) make heavy usage of parallelism. Two of those benchmarks also had sequential versions (REGEXDNA and FASTA) that we executed on each device and development approach. We compared the results of these two benchmarks with their parallel versions, and the results are summarized in Table 7. From the data, we can see that parallelism is usually better for energy efficiency and performance in Java for REGEXDNA, having a slightly higher energy consumption than the sequential version in C++. The parallel version of FASTA in Java exhibited lower energy consumption while the parallel version in C++ had the best speedup. For all benchmarks, the EDP is better using the parallel version. In these cases, using parallelism is better both for energy consumption and performance, and may be considered an improvement.

NDK is a safer bet to improve performance. Analyzing the modified apps, we noticed that using the hybrid NDK approach improved the application performance in two cases (EnigmAndroid and anDOF). When considering the CLBG benchmarks, C++ exhibited the best performance for most of them (Table 5). Therefore, if performance is of utmost importance and energy is a minor concern, using the NDK can be considered a safer bet. It is interesting to note that the NDK exhibited good performance even in hybrid apps using the *Stepwise* model. But is worth noting that across all benchmarks, eight of them (FANNKUCH, FASTAP, KNUCLEOTIDE, NBODY, REGEXDNA, REGEXDNAP, REVCOMP and SPECTRAL) presented a direct relation between the performance of the Java and C++ versions when compared to JavaScript, i.e., either both approaches are better than JavaScript or both are worse. One theory for that relation is that the type of problem in which Java performed better was also the same type for C++. Although it is important to investigate this phenomenon in other contexts, this result suggests that, in cases where Javascript outperforms Java, it may not be useful to try to boost performance and energy efficiency by exploiting the NDK approach.

3.4 THREATS TO VALIDITY

We observed that the results we obtained for the performance of the CLBG benchmarks differ considerably from those reported on the original website. The difference exists both in terms of the absolute values of the measurements and, in some cases, in terms of which development approach exhibits the best performance. This could be an indication that we committed errors in the design, implementation, or execution of our experiments. It could also indicate that the performance of different development approaches in more powerful machines such as desktops and laptops is not a good proxy for the performance of these approaches in a smartphone. To confirm this, we executed all the versions of the CLBG benchmarks on a desktop computer (Processor Intel i7-4700HQ, 24GB RAM memory using bash on Ubuntu on Windows). Unlike the results we obtained for the mobile devices, the results obtained in this machine were consistent with the ones reported at the CLBG website.

As stated by Cruz et al. (2019), energy aware changes on the code most likely will decrease the software maintainability. This decrease could be even more pronounced if the developer chooses to use multiple programming languages in their project, multiplying the complexity of the project. In case larger portions of the code require changes, developers should be cautious over the possible trade-off on hybridization and maintainability.

Benchmarks do not represent the behavior of an application using JavaScript as the main programming language (RATANAWORABHAN; LIVSHITS; ZORN, 2010; SAHIN; POLLOCK; CLAUSE, 2016), and for that reason it is not possible to extrapolate the results for all applications, since applications are usually much more IO-intensive. Although this is true, benchmarks provide insight on scenarios where the performance gain is measured, by isolating usage pattern behavior. The modifications made in the apps show that even small changes could lead to non-negligible improvements in energy efficiency.

Some of the devices we employed were not in the same version of Android and that may influence the results. We had some preliminary results comparing an earlier version on Zenfone

(5.0) with the current version (6.0.1). Some benchmarks executed faster and used less energy and some executed slower and consumed more energy (the Java version of KNUCLEOTIDE consumed 1.3 times more time and energy in version 5.0 and NQUEENS took 3 times longer to execute in version 6.0.1). However, the relationships between the development approaches did not change across versions. Furthermore, most devices were in the same version (5.1) and our data shows that the relation between the languages remained roughly the same even across Android versions. On Section 2.3 we presented all the major updates regarding energy consumption on Android. As far as we can tell, none of the updates made after this study was finished should change any of the results presented here.

3.5 RELATED WORK

For a long time, the study of the energy consumption of computing systems was targeted mainly at the hardware and operating system levels. Tiwari, Malik e Wolfe (1994) showed, however, that software is also a fundamental part of energy consumption. Since then, several papers have proposed solutions to help software developers in measuring (HINDLE et al., 2014; LI et al., 2013), analyzing (HINDLE, 2012; MANOTAS; POLLOCK; CLAUSE, 2014; PINTO; CASTOR; LIU, 2014b), and optimizing (MANOTAS; POLLOCK; CLAUSE, 2014; COHEN et al., 2012) the energy consumption of the systems they build.

Pathak, Hu e Zhang (2012) proposed the first fine-grained energy profiler to investigate where the energy is spent inside an app. This study revealed that the majority of the energy in apps is spent on I/O operations. In particular, free apps spend a considerable amount of energy due to third-party advertisement modules. Since I/O operations happen without much interaction with methods or functions, it is proposed to group the I/O operations together in a bundle, as a way to minimize the energy consumption of those operations.

Some have attempted to find which Android methods (COUTO et al., 2014), API-calls (LINARES-VÁSQUEZ et al., 2014), or apps (WILKE et al., 2013) are more energy-hungry. Li et al. (2013) developed an accurate approach to find which source code lines are the most battery-draining for Android apps, using a combination of hardware, path profiling, and instrumentation to associate the energy data with the application. An evaluation tool was made to appraise the approach and executed on five apps from the Play Store. This tool was developed to be run on apps that used the Dalvik Virtual Machine, which was discontinued after the fifth Android version.

Nucci et al. (2017) developed an energy profiling tool to assist in the measurement and analysis of the energy consumption of Android applications. It works by installing an apk file containing the application to be tested on an Android device, running it with predefined test cases and using the Android APIs to collect information about energy. Experimental results have shown that its accuracy is comparable to that of an external hardware monitor.

A promising approach to save energy without the need for specialized knowledge is to leverage design diversity (AVIZIENIS; KELLY, 1984), more specifically, the availability of diversely designed implementations of the same abstractions. Recent work has explored the impact of different thread management constructs (PINTO; CASTOR; LIU, 2014b; LIMA et al., 2019), data structures (HASAN et al., 2016; PINTO et al., 2016), API calls (LINARES-VÁSQUEZ et al., 2014), and concurrency control primitives (LIMA et al., 2019). This work complements previous studies by analyzing how the availability of different development approaches can impact energy in Android apps.

The study by Charland e Leroux (2011) compares the native and Web app development approaches. However, it focuses on user interface code, user experience, and performance for remote web apps. In particular, it does not present data on battery consumption or performance of native applications and local web applications.

Kholmatova (2020) conducted a meta-analytical review on impact of programming languages on energy consumption. They found out that they could not find any statistical difference between the energy consumption of C++, C and Java. For their meta review, they used six papers (ABDULSALAM et al., 2014; ABDULSALAM et al., 2015; CHEN; ZONG, 2016; PEREIRA et al., 2017; OLIVEIRA; OLIVEIRA; CASTOR, 2017; COUTO et al., 2017). However, only two out of six had experiments on the mobile environment: Chen e Zong (2016) used three benchmarks to compare the energy consumption of the two programming languages. When running on ART, they did not present any statically significant difference. However, on Dalvik, C/C++ consumed less energy on each of these benchmarks (up to only 4% of the energy consumed by Java), and Oliveira, Oliveira e Castor (2017) presented a statistically significant difference when using C/C++ instead of Java on CPU-intensive applications; This indicates there is a difference between different mobile architectures and between mobile and desktop environments. This could lead to complete different results when analyzing programming language energy consumption.

Some studies have focused on studying the differences between programming languages: Nanz e Furia (2015) have used the Rosetta Code tasks to compare eight different programming languages (C, Go, C#, Java, F#, Haskell, Python, and Ruby) in terms of the run-time performance and memory usage, among other factors. Couto et al. (2017) compared ten different programming languages (C, C#, Fortran, Go, Java, Jruby, Lua, Ocam, Pearl, and Racket) and ranked based on their energy consumption, using the benchmarks from CLBG to execute the experiments. Similarly, Pereira et al. (2021) compared the energy efficiency of different programming languages but on a larger scale. They used the benchmarks from the Computer Language Benchmark Game to compare and rank programming languages based on their energy consumption. The authors analyzed a total of 27 different programming languages. Different from the work presented in this chapter, these works have focused on desktop applications.

3.6 CONCLUSION

This chapter aimed to investigate whether there is a more energy efficient approach for Android app development. It sheds some light on the strengths and weaknesses of each approach, based on experiments with benchmarks and hybridized apps, using Java, Javascript, and C++. We used five devices with different characteristics on two main versions of Android. We found out that there is no clear winner among the development approaches in Android. Each one has its advantages in some scenarios. Our results indicate that JavaScript and C++ could save more energy albeit sometimes being slower than the other Java benchmarks and that app hybridization may be a solution for app optimization, both in performance and energy consumption. Furthermore, for the set of benchmarks, apps, and devices we analyzed, the relationships between the development approaches remained stable for most benchmarks on every device tested. What we can learn from this is that the development approaches seem to have a stronger impact on benchmark energy consumption and performance than the characteristics of each device. Among the two options for hybridization, using the NDK is a safer bet for improving performance, but using a web-based approach may have a better outcome when there are few cross-language invocations.

4 OPTIMIZING JAVA COLLECTIONS

In this chapter, we share our vision of a solution to help investigate the energy behavior of energy variation hotspots within an application and, when possible, make recommendations that can reduce its energy consumption. The solution we propose to support non-specialists to reduce the energy consumption of an application comprises three steps. First, the available alternative solutions are exercised to build execution environment-specific energy consumption profiles (HASAN et al., 2016). These profiles provide a mean to compare the energy footprint of these solutions in an application-independent manner. Second, the application is analyzed to gather information about the selected energy variation hotspots, in particular, to estimate how intensively the system uses them. That step is device-independent. Finally, these two pieces of information are combined to make potentially energy-saving recommendations specific to the application-device pair.

We have created a tool, instantiating our approach, named CT+. Its goal is to optimize the energy consumption of Java collections on desktop and Android applications. Implementing the first step of our approach, it automatically runs multiple micro-benchmarks for 39 different Java collection implementations in an application-independent manner and builds their energy profiles. `List`, `Set`, and `Map` are the three collections targeted by these collection implementations. The latter stem from the Java Collections Framework (25 implementations), Apache Commons Collections (5 implementations), and Eclipse Collections (9 implementations). With data from these micro-benchmarks, it builds the energy consumption profiles. For the second step, an inter-procedural static analysis is performed on the application source code. This analysis collects, for each instantiation of a collection implementation, information such as frequency and location of use, method and variable names, calling methods, etc. The third and final step consists of recommending the most efficient implementation for each energy variation hotspot, considering both the energy consumption of the multiple operations of each collection implementation and the extent to which each collection is used in the application. The recommendations made by CT+are applied automatically to the application's source code.

We evaluated CT+in two studies aiming to answer the following four research questions, separated in two groups:

- **RQ2.1:** To what extent can we improve the energy efficiency of an application by statically replacing Java Collection implementations?

- **RQ2.2:** Are recommendations for Java collections device-independent?

To answer **RQ2.1** and **RQ2.2**, CT+analyzed the source code of 17software systems across 7 devices, making energy saving recommendations for 13 of them. By analyzing the recommendations made by CT+and their effect on the energy consumption of these systems, we concluded that some very popular collection implementations, such as `ArrayList`, `Hashtable` and `HashMap`, have poor energy efficiency.

- **RQ2.3:** How much does the workload size impact the energy efficiency of a Java collection implementation?
- **RQ2.4:** Are recommendations for Java collections profile-independent?

To answer **RQ2.3** and **RQ2.4**, we created 6 different energy profiles, simulating distinct kinds of workloads an application may be subject to. We applied the recommendations made by CT+to 6 systems of the DaCapobenchmark suite. The results show that these different profiles behave very differently, depending on the circumstances. The difference goes up to 6.18$\times$ more energy saved when comparing the best performing profile with the worst one for the same software system.

Overall, 2295recommendations that impacted the energy consumption were made across 17software systems, 12targeting a desktop environment, 2targeting a mobile environment, and 3that work in both scenarios, for a total of 64modified versions. Most of these systems were non-trivial with thousands of lines of code (LoC), such as BioJava with 914kLoC, Cassandra with 466kLoC, and Tomcat with 433kLoC. With no prior knowledge of the application domain or system implementation, CT+made positive recommendations for 13out of the 17systems. It was possible to reduce the energy consumption of software systems up to 16.34% on a desktop environment and 14.78% on a mobile environment, just by replacing collection implementations. For a small number of modified versions (2 out of 64), recommendations made by CT+degraded the energy efficiency, up to 1.21%.

The results of our studies highlight the need to re-assess the adoption of some widely popular, poorly-optimized collection implementations from the Java Collections Framework, such as `ArrayList`, `Hashtable`, and `HashMap`. Recommendations to replace uses of these collection implementations by more efficient alternatives were common in our evaluation. In addition, there was not a single case where `HashMap` was recommended, and just three cases for `Hashtable`, with data suggesting that `Hashtable` is becoming less used by developers.

Furthermore, 89% of the recommendations made by CT+suggested the use of collection implementations not from the JCF. The data related to this work can be found at <https://energycollections.github.io/>.

This chapter is structured as follows: **Section 4.1** briefly describes the importance of Java Collections; **Section 4.2** introduces our approach to help develop more energy efficient software systems; **Section 4.3** demonstrated how we used our proposed approach to develop a recommendation tool, CT+. This tool focus on optimizing applications to use the most energy efficient collections implementation in Java; **Section 4.4** displays the results from our experiments analyzing the energy consumption of different devices and energy profiles; **Section 4.5** report our findings about the energy consumption of Java Collections; **Section 4.6** presents the threats to validity; **Section 4.7** presents the related work, with a particular focus on empirical studies about energy consumption and Java Collections recommendation tools; and **Section 4.8** concludes this chapter.

4.1 JAVA COLLECTIONS

Collections are widely used by developers, in both mobile and desktop environments. They provide easy access to reliable implementations that can reduce the complexity of developing applications. Java collections in particular are usually subdivided in three different APIs: `Lists`, `Maps`, and `Sets`. These categories are divergent in several points but the main factors that distinguish them are: **Lists** are ordered and indexed (with possible duplicates), **Sets** are unordered and do not admit duplicates, and **Maps** are based on key-value pairs and hashing (keys are unique but values can be duplicated).

In Java, each collection's API has multiple implementations. This is expected since there are a number of different algorithms and data structures that can implement the abstract concept of lists, sets and maps. Examples include using dynamically-allocated list nodes and making the list doubly-linked, as in `LinkedList`, or a resizable array, as in `ArrayList`. Although both are implemented differently, they still respect the set of rules of a list and implement the same interface, `List`. As a consequence, it is often possible to change the list implementation being used in a given context by modifying a single line of code, i.e., the line where the collection implementation class is instantiated.

Collection implementations that can be safely used by several concurrent threads are considered to be "thread-safe". This safety usually comes with extra complexity or inferior performance, which might favor the use of "thread-unsafe" collections. In this work, we consider that it is never acceptable to replace the use of a thread-safe collection implementation with a thread-unsafe one. Conversely, although it is possible to replace a thread-unsafe collection implementation using a thread-safe one, this is not efficient in practice (PINTO et al., 2016).

There are many different ways a collection can be implemented, and these diverse implementations can have a non-negligible impact on energy consumption. The usual way to use collections in Java is through the Java Collections Framework (JCF). Yet, previous work (PINTO et al., 2016; HASAN et al., 2016; COSTA et al., 2017) has shown that alternative implementations can have a positive impact on the energy consumption of applications. Based on that, for this research we are looking at collections from three different sources: Java Collections Framework[1], Apache Commons Collections[2], and Eclipse Collections[3].

To get a glimpse at the usage of these alternative implementations in Java projects, in April 2020 we executed a query on GitHub based on the package names of Eclipse Collections (namely `org.apache.commons.collections`) and Apache Common Collections (namely `org.eclipse.collections`). The query results showed that these collections are in widespread use, with 1,276,939 code results for Apache Common Collections and 537,956 for Eclipse Collections.

As expected, JCF collections, both thread-safe and thread-unsafe, are also in widespread use. To investigate the adoption of JCF, we conducted another simple query but, differently from Apache and Eclipse Collections, JCF does not have a package exclusively for collections. As a consequence, the query for its implementations collections were executed individually, using the full package name (e.g., `java.util.ArrayList`). The results suggest that thread-unsafe collections are used more often than thread-safe collections by a fair margin. For example, on the one hand, the most widely used thread-unsafe collection is `ArrayList`, a `List` implementation, with 38,642,021 occurrences in our query. On the other hand, the most widely used thread-safe collection is `Vector`, another `List` implementation, with 5,526,922 occurrences.

We compared our results with our original query, analyzing collection usage in Github projects as of January of 2019 (OLIVEIRA et al., 2019). As shown in Table 8, overall, there was an increase in collection usage for all collection interfaces. The table explicitly separates

[1] <https://docs.oracle.com/javase/8/docs/technotes/guides/collections/>
[2] <https://commons.apache.org/proper/commons-collections/>
[3] <https://www.eclipse.org/collections/>

thread-safe and thread-unsafe JCF collection implementations, since they represent the vast majority of the collection implementations. Nevertheless, the Apache and Eclipse Collections both have thread-safe and thread-unsafe implementations.

Our queries results show that the average increase in the utilization of the collection implementations between January 2019 and April 2020 was 14.83% for thread-unsafe collections and 17.77% for thread-safe. The only two collections with a growth rate of less than 10% were `ArrayList` (9.54% and 3,363,929 new occurrences) and `Hashtable` (4.52% and 90,235 new occurrences).

Because of the extensive adoption of `ArrayList`, the growth rate naturally tends to slow down, since most projects already use it. `Hashtable`, on the other hand, appears to have fallen from grace, with developers opting to use other solutions when they need to use a thread-safe map implementation, like `ConcurrentHashMap` (40.71% growth rate and 455,779 new occurrences). The data from our first query shows `Hashtable` with 78% more occurrences than `ConcurrentHashMap`. However, in our last query, that difference has been reduced to 32%. If the growth rate continues, by the end of year 2021 `ConcurrentHashMap` will have more occurrences than `Hashtable`.

Table 8 – Adoption of collections across Github Java projects. All implementations came from the package `java.util`

Implementation	**Jan. 2019**	**Apr. 2020**	**Growth**
Thread unsafe collections			
`ArrayList`	35,278,092	38,642,021	9.54%
`HashMap`	16,602,391	19,418,572	16.96%
`HashSet`	6,470,505	8,104,539	25.25%
`LinkedList`	3,763,660	4,577,164	21.61%
`LinkedHashMap`	1,470,500	1,953,047	32.82%
`TreeMap`	1,122,886	1,370,672	22.07%
`TreeSet`	950,890	1,174,078	23.47%
`LinkedHashSet`	689,397	944,604	37.02%
Sum of thread unsafe	**66,348,321**	**76,184,697**	**14.83%**
Thread safe collections			
`Vector`	4,731,762	5,526,922	16.80%
`Hashtable`	1,994,173	2,084,408	4.52%
`ConcurrentHashMap`	1,119,704	1,575,483	40.71%
`CopyOnWriteArrayList`	237,541	310,188	30.58%
`CopyOnWriteArraySet`	70,680	94,507	33.71%
`ConcurrentSkipListMap`	39,012	52,394	34.30%
`ConcurrentSkipListSet`	26,826	36,671	36.70%
Sum of thread safe	**8,219,698**	**9,680,573**	**17.77%**
`org.apache.commons.collections`	1,022,778	1,276,939	24.85%
`org.eclipse.collections`	466,394	537,956	15.34%

4.2 OVERVIEW OF THE PROPOSED APPROACH

In this section we propose a novel approach to help developers create energy efficient applications. This approach can be used for general-purpose development, not being restricted to a specific scenario, development environment, device, or application. The proposed approach is organized in three phases: (i) creation of the energy profiles, (ii) collection usage analysis, and (iii) recommendation of source code modifications that have the potential to reduce energy consumption. Figure 8 provides an overview of our approach.

In this section we present a high level overview, with the three phases detailed next. In Section 4.3 we present our instantiation of this approach in the CT+tool, dealing with the optimization of Java collections.

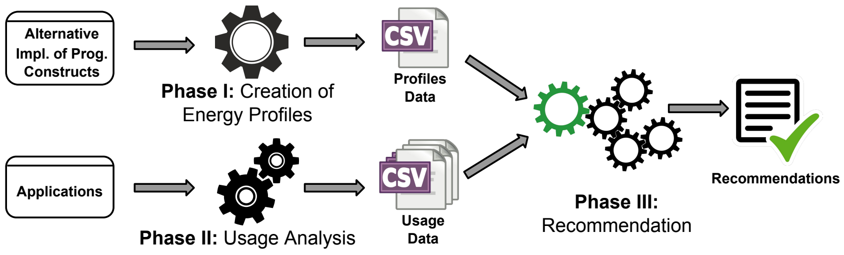

Alternative Impl. of Prog. Constructs
Phase I: Creation of Energy Profiles
CSV
Profiles Data
Applications
Phase II: Usage Analysis
CSV
Usage Data
Phase III: Recommendation
Recommendations

Figure 8 – An overview of our approach. Phase I is application-independent, Phase II is device-independent, and Phase III uses the energy profile and the information about the system under analysis.

Phase I: Creation of Energy Profiles. Here we select a group of programming constructs to analyze and build their energy profiles. This selection determines the energy variation hotspots of the applications that will be analyzed in Phase II. Good choices are constructs that are used intensively and that have alternative implementations. As mentioned before, collections, concurrency control mechanisms, and APIs are examples of potential candidates. Having selected the candidate constructs and their alternative versions, it is necessary to build their energy profiles (HASAN et al., 2016). *The energy profile can be seen as a set of numerical values representing the energy cost of a specific construct, making it possible to compare the energy efficiency of similar constructs under the same circumstances.* The main insight of using energy profiles is that it is possible to order interchangeable pieces of software by their energy consumption, without actually quantifying their energy consumption. This idea can be explored in diverse situations. For example, previous work (WAN et al., 2017; LINARES-VÁSQUEZ et al., 2018) computed energy profiles for the colors that can appear in an OLED screen and employed this information to suggest color schemes for smartphone applications that spend less energy. This improvement could be achieved without the need to precisely measure the amount of energy consumed by each color individually.

Energy profiles can be produced by executing several micro-benchmarks to collect information about the energy behavior of these programming constructs in an application-independent way. This step needs only to be performed once for a given construct, per execution platform. The results can then be reused across multiple software systems employing these constructs. In Section 4.3 we define energy profiles in a more precise manner for the specific context of collection implementations. The energy profiles created in Phase I are used as input to make the recommendations in Phase III.

Phase II: Collection Usage Analysis. This phase extracts information about how the target software systems use the selected programming constructs, for example, usage context and frequency. This information can be extracted either dynamically or statically. In our instantiation of this approach, we relied on a purely static approach. This has the advantage of being platform-independent and not requiring multiple executions of the system under analysis. However, the static approach is more prone to imprecision, since it is not possible to know how often an operation will be executed until the system is actually executed.

Dynamic approaches make it possible to estimate more precisely how intensive energy variation points will be used in realistic scenarios. Notwithstanding, the precision of dynamic approaches depends heavily on the employed workload, specially for complex software systems that make intensive use of multiple resources (CPU, disk, wired network, wireless network, etc.). Dynamic approaches could benefit more UI-focused applications, as these are harder to exercise with static analysis. It could also be useful for cases where part of the logic is outsourced or there's a heavy dependency on non-deterministic factors (e.g., GPS locations or temperature).

The output of this phase is heavily dependent on a number of different factors. Examples may include the kind of construct, line of code where the construct is used, the number of times it is instantiated or invoked, the thread running the programming construct, if the invocation of the programming construct is placed inside a loop, among many others. The data collected from Phase II is used as input to make the recommendation in Phase III.

Phase III: Recommendation. This phase combines the energy profiles and the results of the usage analysis, taking the data produced by Phases I and II as input. Different formulae can be employed in this phase. A straightforward approach is to linearly combine the energy profiles (created in Phase I) with the frequency of use (through the analysis made in Phase II) of the alternative constructs. Each of these combinations will yield an energy consumption number that can be directly compared to determine the most energy-efficient alternative. This is the approach we employed in our experiments. We make it more concrete in the next section. Nonetheless, as with the previous phase, there is ample opportunity to explore different solutions to combine these two pieces of information.

These three phrases summarize our approach to help developers save energy while developing software systems. To be able to offer guidance in as many scenarios as possible, they are open to a number of different instantiations. As an example, the analyses of different parallelism abstractions in Android. There is a number of ways a developer can implement parallelism in their applications (such as `Executors`, `Threads` or `Coroutines`). On Phase I, the tool would need to estimate the energy consumption of each abstraction. On Phase II, the tool would analyze the applications and how they use parallelism abstractions. Finally, on Phase II, the tool would recommend the most energy efficient abstraction for a given scenario. Phase I is application independent while Phase II is execution-environment (device, operating system, runtime system) independent, leaving the possibility for the developer to leverage multiple implementations of each phase.

4.3 INSTANTIATION FOR JAVA COLLECTIONS

In this section, we describe how we have instantiated the proposed approach to work with Java collections. We developed a tool named CT+that implements the three phases explained in the previous section.

CT+analyzes Java programs in desktop and mobile environments, recommending collections that could potentially reduce energy consumption. We organize the presentation of the instantiation in terms of the three phases introduced by Section 4.2.

Phase I: Creation of Energy Profiles. In this phase, we build the energy profiles of the collections. These profiles are based on implementations and operations of three kinds of collections: lists, maps, and sets. We use interchangeable collection implementations for each kind of collection and their operations to create energy profiles that allow these implementations to be compared from an energy consumption standpoint.

The three analyzed collections have differences among them. For example, on lists it is possible to insert or to remove an element at a specific position, differently from a map or a set. To reflect this behaviour, we distinguish operations to insert or remove list elements at the start, middle, or end of the list. We also consider the "default" operation for insertion or removal. For example, for insertions, it is the add method. We iterate over lists using three different approaches: a seeded randomly generated number as index, an explicitly created iterator, and a `for` loop. Table 9 presents a summary of the operations we analyze to build the energy profiles.

The selected operations for List's API cover 59% of all element operations, 42% of Map's and 50% of Sets. Most of the operations not used on this work deal with adding or removing whole data structures from the original implementation, such as `addAll(Collection<? extends E> c)`.

Table 9 – Operations used on each collection.

Operation	Types
Lists	
insertions	start, middle, end, and default
iterations	random, iterator, and loop
removals	object, start, middle, end, and default
Maps	
insertions	default
iteration	iterator and loop
removal	default
Sets	
insertions	default
iteration	loop
removal	default

The definition of a collection implementation, in this case defined as C, can be seen abstractly as a tuple $(N, I, S, o_1, o_2, ..., o_n)$, where N is the name of the collection implementation, e.g., `ArrayList`, `HashMap`, etc., I is the interface of the collection, with $I \in \{List, Set, Map\}$, S is the thread-safety of the collection, with $S \in \{ThreadSafe, NotThreadSafe\}$, and o_i, with $1 \leq i \leq n$ represent the operations of the collection implementation. The following tuple is an example of how `Vector` could be represented:

$$C_v = (Vector, List, ThreadSafe, \texttt{insert.start(e)}, \texttt{insert.end(e)}, ..., o_n)$$

In the previous example, `insert.start(e)` and `insert.end(e)` indicate that these are operations that insert an element in the beginning and at the end of the list, respectively.

The energy profile for a collection implementation is a tuple whose elements are numbers, e.g., energy consumption in joules, that can be used to compare the energy cost of the same operations for different collection implementations under the same execution environment. The energy profile of a collection implementation C is produced by a profiler, a function that, in a given execution environment and under a set of workloads, produces an energy profile:

$$profiler(C, env, w_1, w_2, ..., w_n) = (C, env, e_1, e_2, ..., e_n)$$

where env abstractly represents the execution environment in which the profiler is running (machine, operating system, JVM version). $w_i, 1 \leq i \leq n$, is the workload for the operation

o_i, defined by the function $workload(N, o_i)$, which produces a workload given the name of a collection implementation and one of its operations. Finally, e_i, with $1 \leq i \leq n$, is the energy consumption value for o_i.

Two energy profiles P_a and P_b for two collection implementations C_a and C_b, respectively, can be compared as long as three constraints are satisfied: $C_a.T = C_b.T$, $C_a.S = C_b.S$, and $P_a.env = P_b.env$. In other words, we cannot, for example, compare energy profiles for a list and a set. In the same vein, it is not possible to compare a profile for a thread-safe implementation and one for a non-thread-safe implementation, nor profiles obtained in two different execution environments. We assume that collection implementations whose T and S elements are the same are, from an implementation standpoint, functionally equivalent. For example, adding an element to an `ArrayList` is functionally equivalent to adding that element to a `LinkedList`, although this would not be true for a `Vector`, since the latter is thread-safe. This is true in practice for the vast majority of the collection implementations in the JCF, with very few exceptions (e.g., `WeakHashMap`).

Table 10 lists all implementations analyzed in this work. Creating thread-safe collections based on thread-unsafe collections using the JCF is straightforward: one just needs to use specific `static` methods from the `Collections` class to create synchronized `List`s, `Map`s, and `Set`s. In this work, we labeled those wrapped, thread-safe collections as follows: "Synchronized" + *original collection name*, e.g., `SynchronizedArrayList`.

We build energy profiles by running micro-benchmarks applied to the operations of each collection implementation. Micro-benchmarks are a set of instructions to measure the energy consumption of a specific piece of code by exercising it repeatedly. In our case, a micro-benchmark executes our selected operations a predetermined number of times for every implementation. For example, to collect the energy data, we execute 60,000 times the `ArrayList`'s micro-benchmark to measure the `add(start)` method in one of our devices. That predetermined number is energy-profile dependent and we discuss this in more depth in Section 4.4.2.1.

The executions were made in a specific cycle of operations. We first perform insertions, then we iterate over the whole collection, and finally we remove all elements previously stored in the collection. In cases where more than one type of insertion or removal is necessary, e.g., lists, where it is possible to insert at the start or end, we pair the classified insertions and removals before the initial sequence (e.g., `insert.start(e)` and `remove.start(e)`). The energy consumption was collected throughout each operation. We used this approach to make sure that removals and iterations are measured without the overhead imposed by an insertion

Table 10 – The selected implementations to be used in the experiments. We employed three different sources: Java Collections Framework, Eclipse Collections and Apache Commons Collections

Thread Safety	Implementations	
Lists		
Safe	Vector	CopyOnWriteArrayList
	SynchronizedArrayList	SynchronizedList
	SynchronizedFastList	
Unsafe	ArrayList	LinkedList
	FastList	CursorableLinkedList
	NodeCachingLinkedList	TreeList
Maps		
Safe	Hashtable	ConcurrentHashMap
	ConcurrentSkipListMap	SynchronizedHashMap
	SynchronizedLinkedHashMap	SynchronizedTreeMap
	SynchronizedWeakHashMap	ConcurrentHashMap(EC)
	SynchronizedUnifiedMap	StaticBucketMap
Unsafe	HashMap	LinkedHashMap
	TreeMap	UnifiedMap
	HashedMap	
Sets		
Safe	ConcurrentSkipListSet	CopyWriteArraySet
	SetConcurrentHashMap	SynchronizedHashSet
	SynchronizedLinkedHashSet	SynchronizedTreeSet
	SynchronizedTreeSortedSet	SynchronizedUnifiedSet
Unsafe	HashSet	LinkedHashSet
	TreeSet	TreeSortedSet
	UnifiedSet	

operation.

To execute the micro-benchmarks and build the energy profiles, we developed two different energy profilers, one for the desktop environment and one for the mobile environment. We had to create two different applications because these environments use different methods for gathering energy data and are implemented on different platforms. Nevertheless, we employed the same methodology to collect the energy consumed by each operation on a specific collection implementation. In addition, both profilers were built according to the recommendations of Georges, Buytaert e Eeckhout (2007) for Java performance evaluation. The energy profiles for each collection implementation are used in Phase III to make recommendations.

Phase II: Collection Usage Analysis. CT+employs an inter-procedural data flow static analysis to gather information about the frequency of use and the context in which the collection operations are invoked. For a given program starting point, e.g., the `main` method, it analyzes all the paths in the program method call graph that can be reached from there. This analysis aims to identify calls to collection operations that appear within loops, including loops from different methods. Each time an operation appears on the source code, CT+adds it to the file containing all operations used by that software system. One operation can be counted more than once, depending on its context. For example, the operation "`collection.bar()`" can be called from the method "`foo()`" in different parts of the code; one invocation of "`collection.bar()`" might be inside a loop while another one may not be involved in loops. Each of these operations is counted separately by CT+.

In this phase, among other pieces of information, for each invocation of a collection operation, we collect: class name, collection type, concrete type, calling method, name of the field storing the collection implementation object, invoked collection operation, line of code, whether the invocation appears within a loop, and whether the invocation is performed from within a recursive method.

Taking loops into account is important because operations inside them are usually executed several times and thus consume more energy than ones not invoked within loops. In our implementation, we use the nesting level of the loops as a heuristic to give weights to the operations that are performed within them. Even though there are some approaches to determine loop bounds (*e.g.,* Rodrigues et al. (2014)), these works (1) do not cover many practical loop usage scenarios for languages such as Java, where arrays are allocated dynamically, and (2) typically require program execution. We then opted to use a more conservative approach and only take into account the nesting level of the loops.

Phase III: Recommendation. CT+makes the recommendations based on three different factors about each collection implementation, for each target system: (i) the energy profile information for that collection implementation; (ii) data about occurrences of the collection operations within the target system; and (iii) whether those occurrences appear within loops or not.

More specifically, for each object c that is an instance of a collection implementation C and each path in the program call graph where an operation on c is invoked, considering every collection implementation C' such that $C.T = C'.T$, $C.S = C'.S$, and the profiles for C and

C' were built on the same execution environment, CT+calculates (1) the energy cost of the operations outside loops, (2) the energy of operations inside loops, and then combines these two pieces of information to calculate the energy factor EF according to equation (3), which combines equations (1) and (2).

$$E^{\neg L}(C', c) = \sum_{i=1}^{n} e_i * NL(c.o_i) \tag{4.1}$$

In this formula, n is the number of operations in C, e_i is the i^{th} element of the energy profile of C', notation $c.o_i$ indicates operation o_i from collection implementation C' invoked at object c, NL is a function that yields the number of non-loop occurrences of o_i targeting c for every path in the program call graph.

$$E^{L}(C', c) = \sum_{j=1}^{n} \sum_{d=1}^{m} e_j * ((L_d(c.o_j) + 1)^{d+1} - 1) \tag{4.2}$$

Differently from equation (1), here L_d yields the number of occurrences of o_j applied to c appearing within a loop of nesting level d for every path in the program call graph, and m is the maximum loop depth of the program. The nesting level of a loop affects the energy factor by increasing the exponent which dictates the weight of operations appearing within loops. The "$+1$" in the exponent and in the innermost summation guarantee that operations appearing within a loop always have a greater weight than those that do not. The "-1" within the innermost summation guarantees that operations not appearing within loops (i.e., $L_d(c.o_j) = 0$) get canceled out.

$$EF(C', c) = E^{\neg L}(C', c) + E^{L}(C', c) \tag{4.3}$$

The first factor of each summation is originated from the energy profile created in Phase I (i.e., e_i and e_j) while the second factor comes from data collected in Phase II (i.e., NL and L_d).

CT+makes its recommendation based on the energy factors of the collection implementations. It provides as output an ordered list of collection implementations with better energy footprint than the original collection. Once the implementation is chosen, CT+can automatically refactor the application, within the context of the recommendation, to use the first element of the ordered list.

Implementation. To collect the energy consumption of the different devices, two profilers were created, one for the desktop environment and one for the mobile environment.

The **Desktop Profiler** is a simple Java program that uses the jRAPL (LIU; PINTO; LIU, 2015) library to measure energy consumption on desktop applications. It works on Intel architectures (starting with Sandy Bridge, released in 2011). The **Mobile Energy Profiler** comprises two subsystems: an Android app, responsible for executing the micro-benchmarks for each operation of each collection implementation, and a dashboard application, responsible for collecting and storing the produced data. We used the Android Power Profiler to measure energy consumption on the mobile applications. This loosely limits our profiler implementation since it only works on Android version 5 (released in 2014) or later.

In both environments, the whole application energy consumption data is collected. That means that we analyze how much energy was consumed by the entire application during its execution while running different collection implementations. On mobile devices, the Android Power Profile is used to isolate the energy consumption of the application. On desktop devices, RAPL uses machine specific registers (MSR) to read the stored energy consumption. The energy cost of an operation is calculated using the difference between the energy data on the register before and after its execution. jRAPL includes the execution cost of the underlying system.

The analysis and recommendation aspects of CT+are based on WALA[4], a static analysis library developed by IBM. Since a collection may be thread-safe or not, we employ WALA's built-in type inference and points-to analysis to discover the concrete types of objects, making it possible to recommend collections satisfying the same constraints of thread-safety. This is also useful to support recommendations that account for collection objects being passed as method arguments.

The whole CT+project was developed in Java and has 23kLoc, comprising two energy profilers, the analysis tool, the recommendation tool, and the auxiliary dashboard application for mobile experiments. To help developers and researchers, we also made available in our website a cookbook guiding the developer to use it together with DaCapo.

[4] <http://wala.sourceforge.net/wiki/index.php/Main_Page>

4.4 EVALUATION

In this section, we present the evaluation of the proposed approach. We applied CT+to a number of software systems, running on multiple execution environments. The main objective of this evaluation is to compare the energy consumption of the original versions of these systems with modified versions where the recommendations made by CT+were applied.

The evaluation is divided into two parts. In Section 4.4.1, we examine the efficiency of CT+'s recommendations when considering different execution environments. Then, in Section 4.4.2 we analyze the impact on energy efficiency of using six different strategies to build energy profiles. For this second part, we run all experiments on a laptop.

When executing the benchmarks on the desktop devices, two different versions of the DaCaposuite (BLACKBURN et al., 2006) were used: version 9.12 and development version 19.07. This was the case because since January 15, 2020, Maven's Central Repository no longer supports communication over HTTP[5]. Because of that, the older versions of DaCapothat required the usage of Java Development Kit (JDK) up to version 6 do not work anymore. Using these older versions on newer devices can be challenging, as it requires in-depth knowledge about DaCapo as well as rewriting potentially dozens of configuration files. While executing version 19.07, JDK 8 was used. This newer JDK version is supported by current versions of DaCapoand complies with Maven's new policy. The study presented in Section 4.4.1 uses version 9.12 of DaCapofor two devices and version 19.07 for one. The study presented in Section 4.4.2 uses exclusively the latest version.

Although execution time is not, in general, a proxy to energy consumption (HAO et al., 2013; LI; HALFOND, 2014; PANG et al., 2016; CHOWDHURY et al., 2019), sometimes it can be a good approximation that is more convenient to use. To verify if there was a correlation between the execution time and the energy consumption, we calculate the Spearman Correlation between execution time and energy consumption on the device used in Section 4.4.2, for the systems where CT+recommendations made a statistically significant impact on the energy consumption. We found out that there was a statistically significant difference for 62.86% of the cases. This result suggests that analyzing the energy consumption separately from execution time may still be the more appropriate approach, since it was not a good approximation for energy in more than 1/3 of the cases.

[5] https://central.sonatype.org/articles/2019/Apr/30/http-access-to-repo1mavenorg-and-repomavenapacheorg-is-being-deprecated/

4.4.1 Analyzing different devices

This section describes our experimental environment and results from the study using CT+to analyze the energy efficiency of Java collections on different devices. Overall, seven different devices were used in this experiment: a high-end server, two notebooks, three smartphones, and a tablet.

The remainder of this section is organized as follows. Section 4.4.1.1 describes the methodology of this study, including the seven aforementioned devices and the target software systems of the study; and Section 4.4.1.2 presents the results.

4.4.1.1 Methodology

Our evaluation comprises two different execution environments, **desktop** and **mobile**. These environments differ in terms of the available processing power and memory, use of batteries, and measurement procedure.

Desktop environment. CT+was executed across three different machines on the desktop environment, two notebooks and a high-end server. We labeled the notebooks as **dell** (Dell Inspiron 7000) and **asus**(Asus X555U), and the server as **server**. **dell** has an Intel Core i7-7500U processor with two 2.7GHz cores with four threads, and 16GB of RAM. **asus** has an Intel Core i5-6200U processor with two 2.2GHz physical cores, with four threads and 8GB of RAM. **server** has two-node Intel Xeon E5-2660 v2 processor with 20 2.20GHz physical cores (10 per node) and 20 “virtual” cores[6], and 256GB of RAM. In the experiments, we always execute benchmarks on **dell** and **asus** while they are connected to the power outlet, since we are using it as a desktop machine.

Mobile environment. We executed our tool on three smartphones and a tablet: Samsung Galaxy J7 (**J7**), Samsung Galaxy S8 (**S8**), Motorola G2 (**G2**), and Samsung Galaxy Tab 4 (**Tab4**). Table 11 presents a summary of the devices used in this study in both environments. All experiments were executed while these mobile devices were in discharge mode (not connected to a power outlet), their more typical usage scenario. For all cases, battery charges ranged between 100% and 80%. The latter restriction aims to reduce influence of dynamic voltage and frequency scaling on the measurements. We also report the age of each mobile device at

[6] <https://en.wikipedia.org/wiki/Hyper-threading>

Table 11 – Machines used on the study about devices. *Age* shows how old the device was when we executed the experiments (in years)

Device	Alias	RAM	Chipset	CPU (GHz)	Battery	Age
			Desktop			
Inspiron 7000	**dell**	16GB	i7-7500U	2-core 2.70	N/A	N/A
Server	**server**	256GB	Xeon E5-2660	20-core 2.2	N/A	N/A
Asus X555U	**asus**	8GB	i5-6200U	2-core 2.80	N/A	N/A
			Mobile			
Samsung J7	**J7**	1.5GB	Exynos 7580	8-core 1.5	3000 mAh	3
Samsung S8	**S8**	4GB	Exynos 8895	8-core 2.3'1.7	3000 mAh	1
Motorola G2	**G2**	1GB	Snapdragon 400	4-core 1.2	2070 mAh	4
Samsung Tab4	**Tab4**	1.5GB	Cortex-A7	4-core 1.2	4000 mAh	6

the time the experiments were run, since older batteries tend to exhibit more erratic discharge patterns (LEE; CHON; CHA, 2015). Section 4.6 discusses this further.

When creating the energy profiles for our devices, we chose the same methodology as previous works (HASAN et al., 2016; OLIVEIRA; OLIVEIRA; CASTOR, 2017), although we leverage a larger number of collection implementations (Table 10). We executed the micro-benchmarks, each one representing an operation-collection pair, and calculated their energy consumption. This procedure is repeated 30 times for each micro-benchmark, for each machine. Before collecting the energy data samples, we performed a warmup execution. In the warmup, we executed up to 10% of our workload. By doing this, we minimized Just-In-Time (JIT) noise on the measurements (BARRETT et al., 2017).

As previously explained, for this experiment, two different versions of DaCapo were used. To execute the benchmarks on **dell** and **server**, we used DaCapo version 9.12, and on **asus**, we used version 19.07. Because different versions of DaCapowere used, there are also two distinct versions of TOMCAT. DaCapo version 9.12 uses TOMCAT 6.0.20 (TOMCAT V6for short) while version 19.07 uses TOMCAT 9.0.2 (TOMCAT V9for short).

On **dell**, we analyzed seven desktop-based software systems: BARBECUE, BATTLECRY, JODATIME, TOMCAT V6, TWFBPLAYER, XALAN, and XISEMELE; two mobile-based software systems: FASTSEARCH and PASSWORDGEN; and three libraries that work on both environments: APACHE COMMONS MATH 3.4 (COMMONS MATH for short), GOOGLE GSON, and XSTREAM: These systems were employed in related work on energy profilling (SAHIN; POLLOCK; CLAUSE, 2014; PINTO et al., 2016; HASAN et al., 2016; PEREIRA et al., 2018), and their workloads are available for replication purposes. For **server**, we only ran TOMCAT V6and

Table 12 – Software systems used in the study about devices and where they were executed.

Device	**Selected Software Systems**			
	Environment: Desktop			
dell	Tomcat v6	Xalan	Barbecue	Battlecry
	JodaTime	Twfbplayer	Xisemele	
	Commons Math 3.4	Google Gson	XStream	
asus	Tomcat v9	Biojava	Cassandra	Graphchi
	Kafka	Zxing		
server	Tomcat v6	Xalan		
	Environment: Mobile			
all	FastSearch	PasswordGen		
	Commons Math 3.4	Google Gson	XStream	

Xalan since these are applications one would expect to execute on a high-end server machine.

For **asus**, we executed six systems: Biojava, Cassandra, Graphchi, Kafka, Tomcat v9and Zxing. Xalan was not executed on **asus** because the project seems to be abandoned. As of April 2020, the last update on GitHub's page was in June 2001[7]. Table 12 summarizes the software systems used in this study and the devices in which they were analyzed.

It is possible to tune the workload size on DaCapo's benchmarks and different benchmarks have distinct options for their workloads. Tomcat is a particular application among DaCapo as it has four different workload sizes (small, default, large, and huge). Xalan has three different workload sizes (small, default, and large). The other applications used in this study only have one (default). The advice of DaCapo developers is to use the biggest option available[8].

Each system has a specific workload and routine, defined by Dacapo developers, to follow with the objective of exercising different aspects of their source code. As an example, executing Biojava creates 10 physico-chemical properties of different-sized protein sequences, Tomcat runs a number of web applications, and Graphchi uses the Netflix Prize dataset (BENNETT; LANNING; NETFLIX, 2007) to drive it's engine. These routines are all selected and curated by

[7] <https://github.com/apache/xalan-java>

[8] <http://dacapobench.sourceforge.net/benchmarks.html>

DaCapo developers. For TOMCAT V6and XALAN on the desktop development machines, we used the same workloads sizes (i.e., LARGE) while on **asus**, TOMCAT V9 was executed with two different workloads sizes (i.e., LARGE and HUGE). The number of threads also varied between devices: forty on **server** and four on **dell** and **asus**.

The workloads of systems outside of DaCapo suite were done in a more individualized manner. For four systems (BARBECUE, JODATIME, TWFBPLAYER, XISEMELE) we used unitary tests, following the same methodology as previous work (PEREIRA et al., 2018). On BATTLECRY, we executed a class inside the benchmark designed to test it. On GOOGLE GSON and XSTREAM we tried to exercise each Java primitive using methods inside those systems. With APACHE COMMONS MATH 3.4, we executed multiple statistical functions from its API. As both PASSWORDGEN and FASTSEARCH are utility programs that work like functions, their workloads consist of executing their `main` methods (e.g., generating passwords).

For TOMCAT V6, we could not recommend any implementation from the Eclipse Collections library. This happened because version 9.12 of DaCaporequires the use of Java Development Kit (Java Development Kit (JDK)) up to version 6 to ensure the correct operation of its benchmarks. Unfortunately, the current version of Eclipse Collections is incompatible with JDK 6. For this particular benchmark, our tool still makes recommendations with Java Collections Framework (JCF) and Apache Commons Collections.

Different devices require different workloads to run for enough time for the energy measurement to have expressive values. This adjustment was especially important when running the mobile profiler. Whereas jRAPL (LIU; PINTO; LIU, 2015) is capable of code-level, fine-grained measurement, the Android battery dump collects battery data at the process level. In order to mitigate potential imprecisions, we adjusted the mobile micro-benchmark workload sizes to run for at least 20 seconds.

For the experiments, we collected the results of 30 executions of each software system. When experimenting with thread-safe collections, we used four threads for each operation; with non thread-safe collections, only one thread was used. Since most of our samples are not normally distributed, based on Shapiro-Wilk's normality test (SHAPIRO; WILF, 1965), we used the Wilcoxon-Mann-Whitneytest (WILKS, 2011) to test whether the difference in energy consumption between the original and modified versions of each software system is statistically significant. We did not remove any outliers. We also employed Cliff's Delta (CLIFF, 1993) as a measure of effect size. Wilcoxon-Mann-Whitneytest and Cliff's Delta are non-parametric tests.

4.4.1.2 Study results

We present the results in terms of the desktop and mobile environments. For each one, we first present the energy consumption results and then proceed to discuss the recommendations that were made for each software system. We will only present the energy results, because as mentioned in Section 4.4.1.1 most executions had a designed workload based on the time necessary to execute them. The specific amount of time each system took to be executed can be found at <https://energycollections.github.io/>. In this experiment, across all devices, a total of 831recommendations were made.

Desktop environment. Table 13 summarizes the energy consumption for the desktop environment. The most important column of the table is **Improvement**, which shows how much more energy the original version consumed, when compared to a modified version where CT+'s recommendations have been applied. A positive percentage in this column indicates that the modified version consumes less energy than the original one.

The versions of all the software systems modified according to the recommendations of CT+consumed less energy than the original versions. For five of them, TWFBPLAYER and XISEMELE on **dell**, TOMCAT V9using the HUGE workload, KAFKA, and CASSANDRA on **asus**, the difference between the original and modified versions was not statistically significant. In the case of TOMCAT V9using the LARGE workload, there was a significant difference with a small effect size. Notwithstanding, for the remaining systems, the difference is statistically significant and the effect size is large.

According to Romano et al. (2006), effect size as measured by Cliff's Delta has four different categories: large (≥ 0.474), medium (between 0.474 and 0.33), small (between 0.33 and 0.147), and negligible (≤ 0.147). For example, on XALAN in the **dell**machine, the effect size was 1, which means that every execution of the modified version exhibited lower energy consumption than every execution of the original version.

Among the software systems that only ran in the **dell** machine, JODATIME exhibited the greatest improvement, with the modified version consuming 6.66% less energy than the original one. To make it easier for the reader to focus on the relevant data, this section focused on the results for which $p - value < 0.05$, thus indicating a statistically significant difference, either positive or negative, between the original version and the modified one.

The two software systems that were executed in the **dell** and **server** machines, XALAN

Table 13 – Results for the desktop environment. Energy results are red for the original versions and green for the modified versions. Energy measured in Joules.

System	Improvement	p-value	Mean(J)	Stdev	Effect Size
Development Machine: dell					
Barbecue	4.38%	7.0^{-4}	56.17 53.71	2.70 2.53	0.50
Battlecry	2.78%	1.5^{-3}	67.95 66.06	2.67 3.18	0.48
Google Gson	0.72%	8.0^{-5}	29.93 29.72	0.22 0.16	0.57
Commons Math	1.04%	6.3^{-12}	48.93 48.43	0.29 0.15	0.90
JodaTime	6.66%	$< 2.2^{-16}$	123.02 114.83	2.42 3.50	0.94
Tomcat v6	3.96%	$< 2.2^{-16}$	32.77 31.47	1.02 0.41	0.86
Xalan	4.77%	$< 2.2^{-16}$	107.04 101.93	0.19 0.15	1
Xstream	2.52%	3.12^{-13}	59.97 58.45	0.52 0.49	0.94
Development Machine: asus					
Biojava	0.60%	2.20^{-16}	193.69 192.53	0.11 0.17	1
Graphchi	10.17%	5.79^{-13}	10.90 9.80	0.35 0.42	0.94
Tomcat v9-LARGE	1.04%	2.53^{-3}	74.93 74.16	1.37 1.78	0.31
Zxing	5.84%	2.20^{-16}	85.82 80.80	0.36 0.41	1
Development Machine: server					
Tomcat v6	4.12%	$< 2.2^{-16}$	89.21 85.54	2.99 2.37	0.66
Xalan	5.49%	$< 2.2^{-16}$	242.29 228.98	4.4 7.02	0.86

and TOMCAT V6, exhibited positive results in both scenarios. For XALAN, the modified version consumed 4.77% and 5.49% less energy than the original version in the **dell**and **server**machines, respectively. For TOMCAT V6, the differences were 3.96% and 4.12%, respectively. We found that systems running on **server** consumed more than twice the energy they consumed on **dell**, for the same workload. This can be justified in terms of their differences in processing power. Notwithstanding, the results were consistent across the two machines.

Meanwhile, TOMCAT V9exhibited a different behavior on **asus**. Using the workload size HUGE, the recommendations made by CT+did not result in energy reduction. Nevertheless, while using the same workload as in **dell** and **server**, there was a reduction of 1.04%, an improvement that represents less than 25% the improvement made on the other two devices. Since on **asus**we executed a newer version of TOMCAT, smaller improvements were expected, since developers behind the newer versions of TOMCAT are likely to be more mindful of the collection implementations being used.

For all other systems executed on **asus**, CT+recommendations resulted in a reduction in energy consumption with a large effect size. In particular, GRAPHCHI was the system with the most significant reduction in energy consumption among all systems across all desktop development machines (10.17% of improvement).

Tables 14, 15 and 16 summarize the recommendations for each application on **dell**, **asus**, and **server**, respectively. The first column lists the names of the target systems for the desktop environment. The second column presents the names of collection implementations used in these systems, whereas the third column indicates the collection implementations that CT+recommended using instead. Finally, the fourth column displays the number of times CT+recommended the one in the third column as a replacement to the corresponding collection implementation in the second column. Overall, on the desktop environment, CT+made and applied 724recommendations lead to statistically significant results.

In **dell** and **server** machines XALAN had a significant number of `Hashtable` implementations changed to `ConcurrentHashMap(Eclipse Collections (EC))` (48 and 49 times on **dell** and **server**, respectively). For both machines, we can observe a trend of recommendations to replace well-known collections from the JCF (`Vector`, `ArrayList`, `HashMap`) by alternatives from Eclipse Collections and Apache Commons Collections. For the specific case of XALAN, among the 119 recommendations across the two desktop machines, just three were for JCF collection implementations.

Table 14 – Recommended collection implementations for the **dell**machine and how many times they were recommended. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Development Machine: dell			
Barbecue	HashMap	HashedMap	13
	ArrayList	FastList	8
Battlecry	LinkedList	ArrayList	2
	LinkedList	FastList	2
Commons Math	ArrayList	FastList	112
	HashSet	UnifiedSet	6
	HashMap	HashedMap	9
	HashMap	UnifiedMap	3
	ArrayList	TreeList	3
Google Gson	ArrayList	FastList	12
	HashMap	HashedMap	3
	ConcurrentHashMap	ConcurrentHashMap(EC)	1
JodaTime	ArrayList	FastList	8
	HashMap	HashedMap	7
	ConcurrentHashMap	ConcurrentHashMap(EC)	1
Tomcat v6	Hashtable	ConcurrentHashMap	6
	HashMap	HashedMap	4
	Hashtable	StaticBucketMap	2
	Vector	SynchronizedLinkedList	1
Xalan	Hashtable	ConcurrentHashMap(EC)	48
	ArrayList	FastList	10
	Vector	SynchronizedFastList	3
	ArrayList	NodeCachingLinkedList	1
	HashMap	HashedMap	1
Xstream	HashMap	HashedMap	52
	ArrayList	FastList	21
	HashSet	UnifiedSet	12
	HashMap	UnifiedMap	7
	LinkedList	TreeList	1
	ArrayList	LinkedList	1
	HashSet	TreeSortedSet	1

Table 15 – Recommended collections for **asus**Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Development Machine: asus			
Biojava	HashMap	HashedMap	100
	ArrayList	FastList	37
	LinkedList	ArrayList	2
	TreeSet	TreeSortedSet	2
	ArrayList	NodeCachingLinkedList	1
	HashSet	UnifiedSet	1
	Hashtable	ConcurrentHashMap	1
	Vector	SynchronizedArrayList	1
Graphchi	HashMap	HashedMap	3
	ArrayList	FastList	1
Tomcat v9	HashMap	HashedMap	47
	ArrayList	FastList	9
	ConcurrentHashMap	ConcurrentHashMap(EC)	8
	CopyOnWriteArrayList	SynchronizedArrayList	8
	ConcurrentHashMap	SynchronizedHashMap	5
	ConcurrentHashMap	Hashtable	3
	Hashtable	ConcurrentHashMap(EC)	4
	Hashtable	SynchronizedHashMap	4
	LinkedList	TreeList	2
	LinkedList	FastList	1
	TreeSet	TreeSortedSet	1
	Vector	SynchronizedArrayList	1
Zxing	ArrayList	FastList	3
	HashMap	HashedMap	2

Tomcat v6recommendations differed across these two machines. On **dell**, the tool made thirteen recommendations, seven for collections from the JCF, and six for collections from the Apache Commons Collections. On **server**, there were 60 recommendations, 40 for Apache Commons Collections, and 20 for JCF collections. In particular, there were 68 recommendations to replace `Hashtable`, `HashSet`, or `HashMap` by more energy-efficient alternatives and no recommendation to use any of those. As pointed out in Table 8, these are widely-used collections. We reiterate that Eclipse Collections could not be recommended for Tomcat v6 (Section 4.4.1.1).

Tomcat v9 has a substantial number of modifications on **asus**, with a total of 93 recommendations by CT+. For the sake of comparison, Tomcat v6had 60 recommendations

Table 16 – Recommended collection implementations for the **server**machine and how many times they were recommended. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Development Machine: server			
Tomcat v6	HashMap	HashedMap	39
	Hashtable	ConcurrentHashMap	16
	LinkedList	TreeList	2
	LinkedList	ArrayList	1
	HashSet	LinkedHashSet	1
	Vector	SynchronizedArrayList	1
Xalan	Hashtable	ConcurrentHashMap(EC)	49
	Vector	SynchronizedArrayList	3
	ArrayList	TreeList	2
	HashMap	HashedMap	1
	HashMap	UnifiedMap	1

on **server**and only 13 on **dell**. Being two different versions of TOMCAT, differences in the implementations recommended by CT+were expected. However, two cases show a contrast in this expectation: `Hashtable` and `HashMap` when comparing **asus** and **server**. `Hashtable` was changed 49 times on **server** (26% of the total recommendations for TOMCAT V6) while it was only changed 8 times on **asus** (8.6% of the total for TOMCAT V9), clearly showing a significant difference. On the other hand, the most common recommendation for both **asus** and **server** was to replace `HashMap` by `HashedMap`. That recommendation was made 47 times on **asus** and 39 for **server**, representing 50% and 65% of all recommendations made for TOMCAT on these devices. Although there are considerable differences between the two TOMCAT versions, it seems like, for several cases, changing `HashMap` for more energy efficient implementation still is an effective recommendation.

Across the 724 recommendations made to the desktop systems, 92% (666) of these came from alternative sources to JCF, being 52% (372) from Eclipse Collections and 40% (294) from Apache Commons. Once again it is possible to observe a trend of replacing well-known collections such as `ArrayList`, `Hashtable`, and `HashMap` by more energy-efficient but less-known alternatives.

Mobile environment. Table 17 summarizes the results for the mobile environment. Overall, CT+made 107 recommendations among the analyzed devices with their effectiveness varying strongly. One of the mobile devices used in our experiments (i.e., **Tab4**) did not present any statistically significant difference between the original and the modified versions. A more thorough discussion about this specific device can be found in Section 4.6.

The modified versions of PasswordGen on the **S8** and **J7** devices exhibited significant improvements: the modified versions consumed 4.49% and 14.78% less energy than the original ones, with a large effect size. However, **G2** had no recommendations for this specific system (more on this in Section 4.5).

Google Gson exhibited a significant improvement of 4.79% on the **J7**, with a medium effect size. Nonetheless, the recommendations of CT+yielded a statistically significant but small 0.95% improvement on **S8**.

Commons Math had more inconsistent results. Although the modified version consumed 10.16% less energy than the original version on **S8**, the original versions consumed 1.2% and 0.33% less energy than the modified ones on **G2** and **J7**. Albeit small, these results are statistically significant and the effect size for both cases was negative (medium and large, respectively). This intuitively means that it was more common for executions of the modified versions to exhibit greater energy consumption.

Finally, FastSearch was arguably the most consistent of the software systems on the mobile environment, in the sense that there was no practical difference between original and modified versions. For**J7** and **G2** the results for the modified and original versions did not differ in a statistically significant way. On the **S8**, albeit statistically significant, the difference was small with the modified version consuming just 0.09% less than the original version. It is worth noting that on **S8** the effect size was negative when analyzing FastSearch. That implies that although we observed an decrease in energy consumption in our analysis, when choosing a random element from both groups, there's a 47% chance that the value would be smaller if the element was from the original version.

These results suggest that (i) the energy consumption of different collection implementations varies considerably across mobile devices, and (ii) although the results were not as strong as in the desktop environment, for most cases, the recommendations of CT+ either yielded an improvement or did not have a strong impact on the energy consumption of the software systems.

Table 17 – Results for the mobile environment. Energy results are red for the original versions and green for the modified versions. Energy measured in Joules.

System	Improvement	p-value	Mean(J)	Stdev	Effect Size
Device: S8					
Commons Math	10.16%	1.25^{-8}	92.06 82.70	2.59 9.61	0.86
FastSearch	0.09%	1.67^{-3}	35.06 35.03	3.32 1.78	-0.47
Google Gson	0.95%	6.42^{-4}	16.45 16.29	0.22 0.20	0.40
PasswordGen	4.49%	2.38^{-9}	16.86 16.11	0.41 0.65	0.90
Device: J7					
Commons Math	-0.33%	2^{-4}	23.82 23.90	2.33 2.62	-0.56
Google Gson	4.79%	3.2^{-3}	13.78 13.12	1.59 2.67	0.44
PasswordGen	14.78%	6.44^{-9}	12.83 10.94	0.90 0.76	0.87
Device: G2					
Commons Math	-1.20%	9.0^{-3}	17.22 17.42	0.51 0.14	-0.41

Table 18, 19, and 20 presents the recommendations that CT+made for **S8**, **J7**, and **G2**, respectively (the data from our experiments on **Tab4** can be found on 28) Commons Math running on the **S8**has more recommendations for JCF collection implementations than all the software systems we evaluated on the **dell**machine combined. On the one hand, the only collection recommended by CT+ that is not from the JCF for this software system is `TreeList` from the Apache Commons Collections. On the other hand, it follows the pattern of recommending alternatives to widely popular collections, e.g., it recommends the use of `TreeList` instead of `ArrayList` and `LinkedHashMap` in place of `HashMap`. For the remaining systems, CT+ made few recommendations, 11 for Gson, 2 for PasswordGen, and 5 for FastSearch. Overall, the recommendations only produced a large effect size for Commons Math and PasswordGen. Furthermore, these were the only systems that could achieve energy savings greater than 1% in the **S8**.

Among the 22 recommendations of Commons Math on **J7**, 14 were for Eclipse Collections, and eight were for Apache Commons Collections. In all these cases, CT+ recommended that developers replace `ArrayList` with an alternative implementation. For this specific con-

Table 18 – Recommended collection implementations for **S8**and how many times they were recommended. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Device: S8			
Commons Math	ArrayList	TreeList	8
	HashMap	LinkedHashMap	7
	HashSet	LinkedHashSet	6
	TreeSet	LinkedHashSet	2
	TreeMap	LinkedHashMap	2
	ArrayList	LinkedList	1
Google Gson	ArrayList	FastList	6
	HashMap	LinkedHashMap	3
	ArrayList	TreeList	1
	ConcurrentHashMap	Synch LinkedHashMap	1
PasswordGen	ArrayList	FastList	2
FastSearch	ArrayList	FastList	4
	HashMap	HashedMap	1

Table 19 – Recommended collection implementations for **J7**and how many times they were recommended. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Device: J7			
Commons Math	ArrayList	FastList	14
	ArrayList	NodeCachingLinkedList	5
	ArrayList	TreeList	3
Google Gson	ArrayList	FastList	7
	ArrayList	NodeCachingLinkedList	2
PasswordGen	ArrayList	FastList	5

text, the recommendations did not yield energy savings. CT+ also recommended replacing `ArrayList` by alternatives in the case of GSON and PASSWORDGEN. These substitutions yielded considerable energy savings. The **G2** differed from the others in this study in the sense that only one of the software systems exhibited significant differences between the original and modified versions. Notwithstanding, the trend of CT+ recommending less popular collections as replacements for widely-used ones such as `ArrayList` and `HashMap` can still be observed.

Table 20 – Recommended collection implementations for **G2**and how many times they were recommended. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Device: G2			
Commons Math	HashMap	LinkedHashMap	12
	ArrayList	FastList	8
	ArrayList	TreeList	5
	CopyOnWriteArrayList	Vector	1
	ArrayList	LinkedList	1

4.4.2 Analyzing different profiles

This section describes our experimental environment and results from a study analyzing different energy profiles. Each profile was created simulating a different workload size, in a way to try to analyze how they can affect the energy consumption of Java collections. For this study, only **asus**was used and all the target systems come from the most recent version of the DaCapo benchmark suite, that is, the developer branch of the version 19.07.

The following sections are organized as follows. Section 4.4.2.1 explains the methodology of this study; and Section 4.4.2.2 presents the results of our experiments.

4.4.2.1 Methodology

Through the following experiment, a single device was used, **asus**, described in detail on Table 11. Six target systems present in DaCapo19.07 were used, that is: BIOJAVA, CASSANDRA, GRAPHCHI, KAFKA, TOMCAT V9and ZXING. These systems were executed using JDK 8.

To explore the impact of different energy profiles on the energy-efficiency of Java collection implementations, six energy profiles were created for **asus**, namely `N1`, `N2`, `N4`, `N8`, `N16`, and `N32`. Starting with `N1`(the profile used in Section 4.4.1), created using a specific load size for each API i.e., 15,000 for `Lists`, 18,750 for `Sets`, and 50,000 for `Maps`. We then proceeded to multiply this load size by a factor of two and then used it to create a new profile, with `N2` having two times the load size of `N1`, `N4` having four times, until the maximum of 32 times the load size of `N1` with `N32`. These profiles were created to simulate different workload sizes a collection may face, with the smaller ones representing lightweight applications that do not depend too much on collections and the bigger ones representing data-structure heavy applications that make more intensive usage of collections.

Load sizes smaller than N1 made it unreliable to sample the energy consumption for some of the faster operations, such as removing from the tail of a `LinkedList`. The values of N1 we employed were the lowest loads where it was possible to perform energy measurement, i.e., the results were consistently above zero, with a stable standard deviation, i.e., increasing the load size did not lower it. This phenomenon, in which RAPL is unable to reliably measure energy consumption for small segments of program execution, has been previously reported (HÄHNEL et al., 2012).

For the experiments, we collected the data from 30 executions of each target system, considering the original and all six modified versions. Each instance was executed in a clean machine state, that is, we restarted the notebook to make sure there were no residual traces from the previous execution. All executions were made in Ubuntu 19.04, booting without a graphic user interface. When experimenting with thread-safe collections, we used four threads for each operation; with non thread-safe collections, only one thread was used.

4.4.2.2 Study Results

This section first presents the energy consumption results for the original and modified versions. It then proceeds to discuss the recommendations made for each target system when considering the three different profiles.

In this experiment, considering all six profiles, CT+ made a total of 1711 recommendations.

Figure 9 summarizes the energy consumption of all software systems executed on **asus** in which CT+ made statistically significant recommendations, with the exception of Tomcat v9-HUGE. Out of the six target systems, three were more susceptible to improvements, with high effect size and low p-value across all profiles: BIOJAVA, GRAPHCHI, and ZXING. For two applications, CASSANDRA and KAFKA, CT+was not able to make any impactful recommendations on any of the six different profiles. For all scenarios involving these applications, the energy consumption of the original and modified versions did not differ significantly. As a consequence, we do not report the results for CASSANDRA and KAFKA in the remainder of this section. Instead, we focus on statistically significant results. TOMCAT V9had mixed results, presenting an energy reduction in all profile sizes for the workload LARGE, usually with a small effect size. For the workload size HUGE, TOMCAT V9presented a reduction in energy

Table 21 – Source from the recommendations made to **asus**, sorted by the profile size. The approximated percentage of the total is shown between parenthesis.

		Sources		
Profiles	**Total**	Java Collections	Apache Collections	Eclipse Collections
N1	**247**	26 (10%)	154 (63%)	67 (27%)
N2	**352**	19 (5%)	292 (83%)	41 (12%)
N4	**456**	7 (2%)	116 (25%)	333 (73%)
N8	**173**	38 (22%)	82 (47%)	53 (31%)
N16	**82**	40 (49%)	25 (30%)	17 (21%)
N32	**360**	41 (11%)	125 (34%)	235 (65%)

consumption of **0.36%** only when using **N32** (with a medium effect).

Among the profiles, there was no overall winner. No profile presented the best results among all different software systems and no profile dominated another one, i.e., for every profile, if a target system had lower energy consumption under profile p_1 than profile p_2, there was some other target system that consumed less under profile p_2. TOMCAT V9, on both workloads sizes, consumed less energy using the recommendations made using N32; GRAPHCHI using N2; ZXING using N1; and BIOJAVA using N8. When looking at the profile sizes, it's possible to have a glimpse of a trend of some systems performing better for smaller profiles (the case of GRAPHCHI) and others performing better for bigger ones (such as BIOJAVA and TOMCAT V9).

Table 21 summarises the data about the recommendations made across all different profiles. The total number of implementation changes differs significantly on each profile,with N4having the most with 456 recommendations and N16having the least, with 82. Most of these recommendations changed an implementation from JCF to an alternative collection implementation from Eclipse Collections or Apache Commons Collections. Up to 98% (in the case of N4) of the collections recommended were from alternative sources. On the other hand, N16had the greater number of recommendations within the JCF, for a total of 49%.

Tables 15, 22, 23, 24, 25, and 26 show the specific recommended collections for **asus**among the six different profiles. CT+often recommended alternatives from `ArrayList` across all devices, for a total of 849 implementations of `ArrayList` being changed by CT+, representing 49.6% of all modifications made across all profiles. That is even clearer for two specific profiles, N2and N4, having 76.7% and 64.2% respectively of all their modifications being changes from `ArrayList`. On the other hand, N16had only two `ArrayList` modifications, representing a total

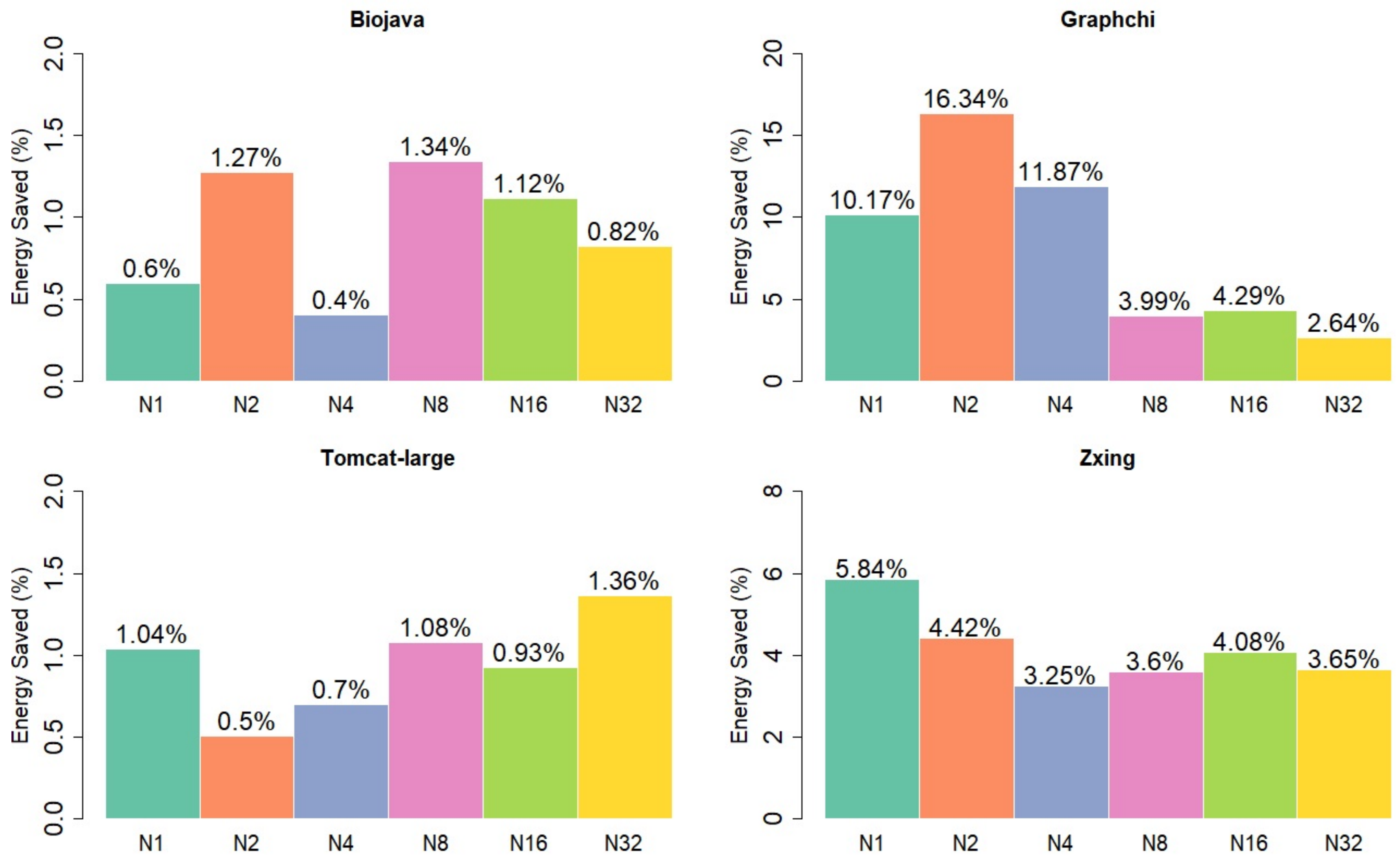


Figure 9 – Percentage of Energy saved by CT+among the different software systems.

of 2.4% of all modifications made on this profile. Surprisingly, that was not the case for N32, which had 210 ArrayList implementations being modified, 52.3% of the total modifications of this profile. The second and third most changed List implementations across all profile sizes were CopyOnWriteArrayList and LinkedList, with 47 and 30 changes respectively.

BIOJAVA showed a special behavior when compared with the other systems. For N4, CT+recommended 291 changes to BIOJAVA while for N16the number of recommendations was only 28, less than 10% the recommendations made for N4. Even so, N16saved more energy than N4(1.12% and 0.4%, respectively), although in both cases the savings were modest. Not only the recommendations between profiles differ in quantity, but also which collections were recommended by CT+. Among all applications tested on **asus**, on profiles N2and N4there was a large number of list implementations that were changed: 283 and 309, respectively. Meanwhile, on profile N16, that number was much smaller: just 15 changes.

Table 22 – Recommended collections for the **asus**on N2. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Profile: N2			
BioJava	ArrayList	NodeCachingLinkedList	179
	HashSet	UnifiedSet	18
	ArrayList	FastList	6
	HashMap	HashedMap	5
	LinkedList	NodeCachingLinkedList	2
	Hashtable	ConcurrentHashMap	1
Graphchi	ArrayList	NodeCachingLinkedList	4
	HashSet	UnifiedSet	1
Tomcat	ArrayList	NodeCachingLinkedList	64
	HashMap	HashedMap	18
	HashSet	UnifiedSet	10
	Hashtable	ConcurrentHashMap	9
	CopyOnWriteArrayList	SynchronizedArrayList	6
	ConcurrentHashMap	ConcurrentHashMap(EC)	5
	CopyOnWriteArrayList	Vector	2
	LinkedList	TreeList	2
	ConcurrentHashMap	SynchronizedHashMap	1
	LinkedList	NodeCachingLinkedList	1
	TreeSet	TreeSortedSet	1
Zxing	ArrayList	NodeCachingLinkedList	16
	ArrayList	TreeList	1

Table 23 – Recommended collections for the **asus**on N4. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Profile: N4			
BioJava	ArrayList	FastList	214
	HashMap	HashedMap	70
	LinkedList	FastList	2
	TreeSet	TreeSortedSet	2
	ArrayList	NodeCachingLinkedList	1
	Hashtable	ConcurrentHashMap	1
	Vector	SynchronizedFastList	1
Graphchi	ArrayList	FastList	7
Tomcat	ArrayList	FastList	69
	HashMap	HashedMap	43
	ConcurrentHashMap	ConcurrentHashMap(EC)	16
	Hashtable	ConcurrentHashMap(EC)	13
	CopyOnWriteArrayList	SynchronizedArrayList	6
	CopyOnWriteArrayList	SynchronizedFastList	2
	LinkedList	TreeList	2
	Vector	SynchronizedFastList	2
	HashSet	UnifiedSet	1
	LinkedList	FastList	1
	TreeSet	TreeSortedSet	1
Zxing	ArrayList	FastList	2

Table 24 – Recommended collections for the **asus**on N8. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
		Profile: N8	
Biojava	HashMap	HashedMap	45
	HashSet	LinkedHashSet	18
	ArrayList	FastList	15
	LinkedList	ArrayList	2
	ArrayList	LinkedList	1
	HashTable	ConcurrentHashMap(EC)	1
Graphchi	ArrayList	FastList	2
	HashSet	LinkedHashSet	1
Tomcat	HashMap	HashedMap	34
	ConcurrentHashMap	ConcurrentHashMap(EC)	19
	HashSet	LinkedHashSet	9
	Hashtable	ConcurrentHashMap(EC)	9
	CopyOnWriteArrayList	SynchronizedArrayList	4
	CopyOnWriteArrayList	SynchronizedFastList	2
	CopyOnWriteArrayList	Vector	2
	LinkedList	TreeList	2
	ArrayList	FastList	1
	HashSet	UnifiedSet	1
	LinkedList	ArrayList	1
Zxing	ArrayList	FastList	3
	ArrayList	TreeList	1

Table 25 – Recommended collections for the **asus**on N16. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
		Profile: N16	
Biojava	HashSet	LinkedHashSet	18
	HashMap	HashedMap	5
	LinkedList	ArrayList	2
	ArrayList	NodeCachingLinkedList	1
	Hashtable	ConcurrentHashMap(EC)	1
	TreeSet	TreeSortedSet	1
Graphchi	HashSet	LinkedHashSet	1
Tomcat	HashMap	HashedMap	16
	HashSet	LinkedHashSet	10
	Hashtable	ConcurrentHashMap(EC)	9
	ConcurrentHashMap	ConcurrentHashMap(EC)	6
	CopyOnWriteArrayList	SynchronizedArrayList	6
	CopyOnWriteArrayList	Vector	2
	LinkedList	TreeList	2
	LinkedList	ArrayList	1
Zxing	ArrayList	TreeList	1

Table 26 – Recommended collections for the **asus**on N32. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
		Profile: N32	
Biojava	ArrayList	FastList	145
	HashMap	HashedMap	80
	HashSet	LinkedHashSet	18
	HashMap	UnifiedMap	3
	LinkedList	FastList	2
	ArrayList	LinkedList	1
	Hashtable	ConcurrentHashMap	1
Graphchi	ArrayList	FastList	2
	HashSet	LinkedHashSet	1
	HashMap	HashedMap	1
Tomcat	ArrayList	FastList	48
	HashMap	HashedMap	40
	ConcurrentHashMap	ConcurrentHashMap(EC)	14
	HashSet	LinkedHashSet	10
	Hashtable	ConcurrentHashMap(EC)	7
	CopyOnWriteArrayList	SynchronizedLinkedList	3
	CopyOnWriteArrayList	SynchronizedArrayList	2
	CopyOnWriteArrayList	Vector	2
	Hashtable	ConcurrentHashMap	2
	LinkedList	TreeList	2
	HashMap	UnifiedMap	1
	LinkedList	ArrayList	1
Zxing	ArrayList	FastList	13
	ArrayList	NodeCachingLinkedList	1
	HashMap	HashedMap	1

When analyzing uses of Set implementations, 35.4% of the recommendations made by CT+for the N16profile suggested using some alternative to HashSet. On the other hand, for N1and N4, only 0.4% and 0.2% of the recommendations involved this collection implementation, respectively. Lastly, HashMap was the Map implementation most often recommended against by CT+. This implementation was frequently changed across all profiles, up to 61.54% on N1, with the solo exception being N2, where its changes represented only 6.53% of the total recommendations. Across all profiles, from all recommendations to change from HashMap, 99.2% were to use HashedMap. Only in 4 cases when analyzing N32, CT+recommended using UnifiedMap as a replacement for HashMap. This suggests that developers should consider using HashedMap as an alternative to HashMap.

As shown in Section 4.4.1 and in other papers (PINTO; CASTOR; LIU, 2014b; PINTO et al., 2016), Hashtable has poor performance and energy efficiency. Therefore, we expected CT+to make multiple recommendations of alternative collection implementations aiming to improve uses of Hashtable. Surprisingly, though, there were only a few such recommendations. More specifically, CT+recommended replacing uses of Hashtable 8 times for N1, 9 times for N2, N8, N16, N32, and 13 times to N4when analyzing TOMCAT V9and a single case for each profile for BIOJAVA. An examination of the source code of the target systems reveals that, differently from TOMCAT V9, where there were 49 instances of uses of Hashtable, the others either used it scarcely (2 cases for BIOJAVA, CASSANDRA, and KAFKA) or did not use it at all (GRAPHCHI and ZXING). This decreased Hashtable usage in more modern software corroborates the results from our queries on GitHub projects using Java collections, presented in Table 8.

To verify if there was any correlation between the execution time and the energy consumption, we executed the Spearman Correlation on the **asus**. We found out that for only 62.86% of the scenarios on the experiment there was a statistically significant the two factors were correlated. This result suggests that analyzing the energy consumption separately from performance may still be the more appropriate approach, since performance is not a good approximation for energy in more than 1/3 of our cases.

4.5 DISCUSSION

This section discusses in more depth the results from the two studies presented in Sections 4.4.1.2 and 4.4.2.2. This section is organized as follows. We start by examining the implementations of JCF, Apache Collections, and Eclipse Collections and how they interact during our experiments. We then proceed by analyzing the energy consumption of the most popular implementations, answering **RQ2.1**, and discussing the importance of analyzing different devices when running experiments on energy efficiency, answering **RQ2.2**. We also discuss how the optimization of the collection implementations change based on the expected workload, answering **RQ2.3** and the impact that different energy profiles have on the expected consumption on an optimized application, answering **RQ2.4**. Finally, we illustrate how some implementations dominate others and explain the employed procedure to optimize the creation of energy profiles.

JCF recommendations. The majority of the CT+recommendations were for collection implementations outside the JCF. Considering only the statistically significant occurrences, out of 724recommendations made in the desktop environment in the first study, only 58suggested the use of JCF collections (8% of the recommendations). When analyzing the data on the study about different energy profiles, out of 1711recommendations, 171came from JCF (10% of the recommendations). This means that overall, in the desktop environment, across all 2188recommendations made, only 9.2% of the CT+recommendations suggested the use of JCF implementations. The contrast is less stark in the mobile environment, where CT+recommended JCF collection implementations in one-third of the cases (36 out of 107 recommendations). However it is worth noting that none of the cases where energy was saved on **J7**used JCF implementations. If we aggregate over all of these recommendations, the JCF was recommended in just 10.4% of the cases.

Popular collections and energy efficiency. Our results indicate that there seem to be more energy-efficient alternatives to some popular collection implementations. In the desktop environment, CT+recommended replacing 184 uses of `Hashtable`, 654 uses of `HashMap`, 138 uses of `HashSet`, 38 uses of `LinkedList`, and 1027 uses of `ArrayList`. Overall, those recommendations amount to 93.2% of all the recommendations in cases where there was a statistically significant difference in energy consumption. This percentage is consistent with the popularity of those JCF collections (Table 8); since they are used often, there will be many recommendations to replace them with alternatives. Some of these commonly used implementations were not recommended by CT+, e.g., `HashMap`, and `HashSet`, while others were very rarely recommended, such as `LinkedList` (3 times), `Hashtable` (3 times), and `ArrayList` (12 times). `Hashtable`, in particular, replaced `ConcurrentHashMap` on `N1`. For as much performance and scalability problems `Hashtable` may have, it seems like it can still perform better than other implementations for a very small number of elements.

Out of the 12 times `ArrayList` was recommended, all of them as a replacement for `LinkedList`, a collection that is not efficient for random accesses. These results, combined with the significant improvements in energy efficiency that could be achieved by following CT+'s recommendations in the desktop environment, suggest that these collections might not be good choices in scenarios where energy efficiency has a high priority.

As pointed out previously, in the mobile environment CT+recommended the use of JCF collections more often. Nevertheless, a similar trend of modifying popular collections can be observed. CT+suggested alternatives to `HashMap` 23 times, to `HashSet` 6 times, and to `ArrayList` 72 times. That amounts to 94.39% of all its recommendations. At the same time, not once did it recommend the use of these collections.

Given the importance of the aforementioned collections, we conducted a more in-depth investigation into why `ArrayList` was replaced an expressive number of times and only rarely recommended. We focus on `ArrayList` because it is arguably the most popular collection implementation in the Java language. Two factors help explain the lack of recommendations in its favor. First, the most common operations in the software systems for list collections are `List.insert(value)` and `List.iteration(random)`. `ArrayList` does not perform these operations well on most devices. In particular, `FastList` was explicitly designed as an alternative to `ArrayList` that performs those operations more efficiently, since it does not support concurrent modification exceptions. As a consequence, `FastList` can "*provide optimized internal iterators which use direct access against the array of items.*"[9]. This kind of direct access is not allowed by `ArrayList`. Second, there are many cases where `ArrayList` is the most efficient alternative, but it is already being used. That is what occurred, for example, for `FastSearch` and `PasswordGen` in the **G2**. In other words, due to the widespread use of this collection implementation, in most cases where it would be the best option, it is already being employed, and thus no benefits can be achieved.

Different devices matter. The recommendations and results varied heavily across devices, even when executing the same application. Although for some specific applications, such as FASTSEARCH, our tool made similar recommendations across devices and those recommendations did not impact energy efficiency, for most software systems, different devices resulted in different recommendations. For instance, CT+recommended ten `ArrayList` instances to be changed to `FastList` and one to `NodeCachingLinkedList` when analyzing XALAN on **dell**. However, for the same system on **server**, it made recommendations for only two instances of `ArrayList` and suggested the use of `TreeList`. In both machines, energy consumption decreased.

In addition, the effectiveness of CT+'s recommendations for the same software systems varied across machines. XSTREAM presents an interesting example. The recommendations made by CT+did not result in a version of the software system that had a statistically significant difference in energy consumption on mobile devices, even if the modified versions

[9] <https://www.eclipse.org/collections/>

consumed less energy. On the other hand, on **dell**, the energy consumption of the modified version exhibited a statistically significant difference (with a p-value of 3.12^{-13}) when compared to the original version. Also, the effect was large (0.94). This difference may be attributed to the number of implementation changes as well as differences between devices. On **dell**, our tool suggested 95 modifications to XSTREAM while the mobile device with most changes, **G2**, only had 41. Those changes also did not target the same implementations: On **dell**, we replaced ArrayList by FastList 21 times and by LinkedList one time. On **G2**, ArrayList was replaced by TreeList just three times. Those devices had different energy profiles and by the number of changes, we noticed that the implementations used on the mobile versions were already optimized for that environment, which was not the case for the desktop environment.

For this topic, results from **asus**were not analyzed because they are not comparable. The only target system used on it and on the other two desktop devices, namely **dell**and **server**, was TOMCAT, but with a different version. Besides that, we used six different profiles on **asus**, producing different recommendations, and making the comparison infeasible. The next sections will give more details about our findings on the different profiles for **asus**.

The best implementation is workload dependent. With the experiments on **asus**, we used six different profiles to try to simulate the different scenarios that the application could be submitted to. The amount of energy saved on different applications was heavily influenced by the profile being used, specially in two cases: GRAPHCHI, with 12.73% of energy was saved using N2but only 2.64% using N32; and BIOJAVA, with 1.34% of energy was saved using N8but only 0.40% using N4. These results indicate that, even though profile creation is an application-independent step of the proposed approach, knowledge about actual usage profiles can be leveraged to produce more useful profiles.

Recommendations applied to TOMCAT V9using the six different profiles on the LARGE workload resulted in a positive impact on energy efficiency with statistical significance. On the other hand, for the HUGE workload, the results did not have statistical significance for five profiles sizes (N1, N2, N4, N8, and N16), having only a positive impact when using the biggest profile size, N32. Comparing the recommendations made to TOMCAT V9-HUGE using N32(Table 26) and using the smaller sized profiles, such as N1(Table 15) and N2(Table 22), we can see that they differ greatly.

Although both LARGE and HUGE were bigger than normal workload sizes, the difference between these two was enough to make CT+unable to recommend better collections implementations to any of the profile sizes with the only exception being N32.

Energy profiles also matter. As the profiles were created to represent different scenarios, a different behavior was expected. In this topic, we take a deeper look at the implementations recommended for these different profiles on **asus**, in particular as replacements to uses of ArrayList and HashSet.

Across all profiles, 90.8% of list modifications were changes from ArrayList to another implementation. In particular, FastList was the implementation of choice by CT+in 68% of the cases (577 cases out of 849). On the other hand, out of 270 changes on N2from ArrayList, only 6 were to FastList. On this particular profile size, the most often used implementation to replace ArrayList was NodeCachingLinkedList, with a total of 263 changes. The main reason behind the substantial difference in recommendations between the different profiles is the distinct operations used by each application. Taking a deeper look at the behavior of those two collections, we noticed that on N2, ArrayList has higher energy consumption in five out of ten operations than FastList and NodeCachingLinkedList. When comparing between themselves, FastList and NodeCachingLinkedList had an even number of operations where they performed better, five each. Even being the best replacement for ArrayList on every other profile size, for N2, NodeCachingLinkedList exhibited better results than FastList.

The modifications from ArrayList on N2were mostly focused on one application: BIOJAVA. On this system, two operations were heavily used and had a greater impact on CT+recommendations List.insert(value) and List.iteration(iterator). On the other hand, NodeCachingLinkedList (on profile N2) consumed less energy than ArrayList for these two operations. In contrast, when looking at profile N8and N16, ArrayList did not have a single implementation that had lower consumption for these two operations. These two are precisely the profiles with the smaller number of modifications from ArrayList: 23 changes on N8and only 2 on N16.

HashSet had a total of 118 modifications across the different profiles on **asus**. Most of these changes replaced it by one of two particular implementations: LinkedHashSet (72%) and UnifiedSet (26%). These changes were not equally divided among the profiles. For smaller profiles, CT+recommended UnifiedSet to replace HashSet on every change on N1, N2and N4. For bigger profiles, CT+recommended LinkedHashSet to replace HashSet on every change on N16and N32, and all but one on N8(the only exception being a recommendation to use UnifiedSet).

A change in the expected scenario (represented here by a change in profile sizes) had a sizable impact on the recommended implementations and in the energy efficiency results. As researchers, we could only faintly foresee what kind of scenario would better represent a normal usage pattern for a specific application. On the other hand, developers would have an easier time figuring it out how much work the most important parts of the application are expected to have. This kind of information could be very valuable for the creation of the energy profile and could improve even more the efficiency of CT+, allowing us to combine different profiles to make the recommendations.

Dominance among collection implementations. Out of the 39possible implementations available to CT+only 20 were recommended. When trying to understand this behavior, we observed that some collection implementations consistently dominate (PETERSON, 2009) others. Given two collection implementations $C_1 = (N, T, S, o_1, o_2, ..., o_n)$ and $C_2 = (N', T, S, o_1, o_2, ..., o_n)$ with energy profiles $(T, env, e_1, e_2, ..., e_n)$ and $(T, env, e'_1, e'_2, ..., e'_n)$, respectively, we say that C_1 dominates C_2 if $e_i < e'_i$ for all $1 \leq i \leq n$. Since every dominated collection implementation has a dominating alternative collection implementation, it will never be recommended by CT+.

Figure 10 depicts dominance relations for the thread-safe `Map` implementations on the **server**machine. Based on this figure, only four thread-safe `Map` implementations could be recommended by CT+on the **server** development machine: `ConcurrentHashMap`, the Eclipse Collections version, `ConcurrentHashMap(EC)`, and the `Synchronized` versions of `LinkedHashMap` and `UnifiedMap`. These are the collections that are not dominated by any other collection. Furthermore, as the figure shows, `Hashtable` is dominated by `ConcurrentHashMap(EC)`, even though `Hashtable` itself also dominates `Synchronized TreeMap`. Therefore, in **server**, instances of `Synchronized TreeMap` and `Hashtable` are never recommended, in favor of `ConcurrentHashMap(EC)`. More specifically, we observed that `Hashtable` was dominated on **dell**, **server**, and on every mobile device that we experimented with. This result, combined with the well-known scalability limitations of this collection (PINTO et al., 2016), and the plethora of more efficient alternatives suggest that it should rarely be used in practice. Implementations such as `ConcurrentSkipListSet`, `Synchronized TreeMap`, and `Synchronized UnifiedMap`, were dominated in three devices.

Among all implementations, `ConcurrentHashMap` shows a particular behavior that is worth mentioning. That implementation was replaced by more efficient alternatives 79 times on the desktop environment, the majority of the time, (i.e, 70 out of 79 cases) for the Eclipse Col-

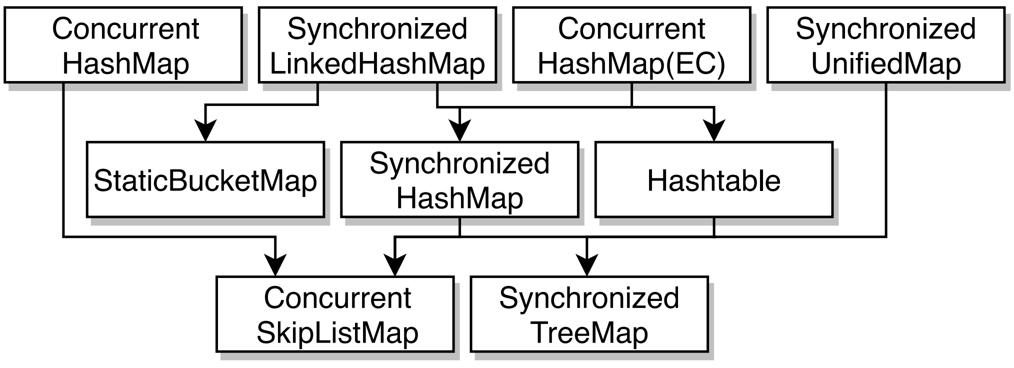


Figure 10 – Order of dominance between the thread-safe Map implementations on **server**. Arrows point from the dominating collection to the dominated one.

lections version (i.e., `ConcurrentHashMap(EC)`). Even so, `ConcurrentHashMap` was also recommended 37 times, every single time replacing `Hashtable`. This illustrates that even if some implementation is usually recommended over another one (e.g., `ConcurrentHashMap(EC)` recommendations over `ConcurrentHashMap`), as long as this implementation is not dominated, there will be cases where the generally worse implementation may still perform better.

Scaling up profile creation. We used two different profilers in this work, one for mobile devices and one for desktop devices, as described in Section 4.3. During our experiments, we noticed that some factors could make it unfeasible to create profiles at a larger scale. If left unchecked, the original process of creating the energy profiles can take a long time, i.e., hours for desktop devices and days for mobile devices. This is due to the enormous variation in the execution time of operations for different collections.

On the one hand, some operations are so fast that it is necessary to increase the number of times they are executed during profiling in order to obtain reliable energy measurements, e.g., insertions at the beginning of a `LinkedList`. On the other hand, some operations are so slow that we need to reduce the number of executions, e.g., updates to `CopyOnWriteArrayList` when the list has many elements. For example, when creating the profile `N16`, the operation `insert(start)` on `LinkedList` would be 554 times faster than on `ArrayList` while `insert(value)` would be 934 times slower on `CopyOnWriteArrayList` than on `Vector`. To address this issue, we employed two strategies, described below.

- *Excluding collections that were dominated.* First we executed each operation for each implementation three times, measured and collected the energy consumption of those operations. In the case where we found one collection implementation exhibiting domination over another, the dominated collection was not included as an option for recommendation. As an example, when creating `N2`on **asus**, there were 13 initial possibilities for Sets. Out of that initial pool, four implementations, that is, `LinkedHashSet`, and the `Synchronized` versions of `UnifiedSet`, `TreeSet`, and `LinkedHashSet`, were excluded because they were dominated by other implementations. In this case, these techniques represent savings of at least 30% of the time to generate the `Set` portion of the profile `N2`, potentially more. Because the dominated implementations would never be recommended by CT+, they can be safely removed from our recommendation pool without reducing CT+capacity of improving the energy efficiency of the application.

- *Using timeouts.* Because of the long time it took to execute some operations on mobile devices, e.g., insertions on instances of `CopyOnWriteArrayList`, very expensive operations were discarded based on the time it took for them to complete. To define which operations should not be measured, a single warmup session was executed, collecting the energy consumption for each operation on each collection implementation. Each operation implementation was executed in sequence and the values for the fastest ones were stored, separated by thread-safety and API (e.g., `insert(start)` for thread-safe `Lists`). Because some of those operations could take a long time to finish, we established a threshold on the number of times that an operation could be slower than the fastest. Any operation slower than that was stopped in the middle of its execution[10].

  We assumed that such a large difference is unlikely to stem from random performance fluctuations. Since CT+needs estimated energy consumption measurements per operation to recommend the implementations, we used the energy consumed by the fastest operation multiplied by our threshold. This approach loses some energy consumption information during profile construction. On the one hand, this means that suboptimal collection implementations may end up being recommended by CT+because we end up underestimating the cost of the very expensive operations. On the other hand, in our experiments, we have observed that scenarios where such collection implementations would thrive, e.g., `CopyOnWriteArrayList` in a scenario where a large collection is subject to many concurrent accesses almost exclusively for reading, were rare. As a consequence, collections in which most of the operations were expensive almost never got recommended.

[10] On our experiments, the threshold was 100 times slower

4.6 THREATS TO VALIDITY

Although we conducted experiments in a number of different devices, we did not use all possible devices available, which is far from feasible. We selected representative devices with very different hardware characteristics (from a mobile phone with 1.5GB of RAM to a server with 256GB of RAM). Second, our findings cannot be generalized to other software applications that use collections. We then chose representative software systems from very different domains (e.g., a XML serializer, a webserver, and mobile apps). Still, the chosen software systems are non-trivial, e.g., TOMCAT has more than 433k lines of code and has been used in multiple studies (PINTO et al., 2016; HASAN et al., 2016; PEREIRA et al., 2018).

Even though we observed an overall good energy savings with our tool, for some software systems it was not possible to reduce the energy consumption reported in other studies. We hypothesize this happens due to the care we took of preventing thread-safety problems due to recommendations that ignore this aspect. We checked that among the recommendations in the study of Pereira et al. (2018) there were cases where a thread-safe collection was replaced by a non-thread-safe one. Similarly, our tool does not guarantee thread-safety when thread-safe collections are performing compounded operations (LIN; DIG, 2015) (e.g., verifying if an item is stored in a collection before adding it). In other words, it does not break thread-safety requirements, but, at the same time, it cannot guarantee thread-safety for non-thread-safe operations. The software construct versions (such as libraries and applications) may influence the recommendations made by CT+.

The Java Development Kit version may have a non-negligible influence on our results. Through this chapter, JDK 6 was used for the study presented in Section 4.4.1 and JDK 8 for Section 4.4.2. Other versions of Java may have different implementations of the collections used on this chapter and this may lead to different results.

Another limitation of this work is the way loops are accounted for in Phase II of the proposed approach (Section 4.3). Estimating loop counts is difficult in a high-level programming language such as Java, where collections are allocated dynamically, usually based on information from outside the system's source code (RODRIGUES et al., 2014). On the one hand, the approach employed in this work is very cheap and takes loop nesting into account. Nevertheless, it is inherently imprecise; in extreme cases, it may consider that loops that are executed millions of times have the same impact on energy consumption as loops that execute just a dozen times. Even though these two situations would be equivalent from an asymptotic perspective, they differ significantly in practice. Exploring different approaches to account for loops, recursion, and API functions that encapsulate repetition, e.g., map/reduce, is left for future work.

The methodology used to collect energy consumption may have an impact on the data presented in this study. The energy measurement was made at the application level for the mobile devices and at the system level for the desktop devices. As explained in Sections 4.4.1.1 and 4.4.2.1, we mitigated the influence of external factors by executing only the application under analysis on mobile devices and executing our experiments on an operating system without a graphic user interface. Nevertheless, the energy data presented in this chapter may differ from other devices or operating systems.

Finally, we did not perform experiments with actual developers, so it is unclear whether developers would face any difficulties while using the tool or whether they would find the recommendations useful.

Results without statistical significance

For some applications, applying the recommendations made by CT+to a target system did not yield a more energy efficient, at least not enough to exhibit a statistically significant difference to the original version. That was the case for two systems on Section 4.4.1 desktop environment (XISEMELE and TWFBPLAYER) and also two on Section 4.4.2 (CASSANDRA and KAFKA). On the mobile environment there were four in this situation: GOOGLE GSON and PASSWORDGEN on **G5**; FASTSEARCH on every device except **S8**; and XSTREAM on every device. The only device where there was no statistically significant difference in the energy consumption of the original and modified versions of the target systems was **Tab4**.

Table 27 shows the results of the experiments on **Tab4**while Table 28 presents the modifications made. Albeit 194 recommendations were made on five different software systems for **Tab4**, none of them had statistically significant results. Unlike other devices from our mobile pool, **Tab4**is a special device, being a tablet and not a smartphone. The idea was to try to investigate a distinct type of device and see if the applications running on it could also be optimized by CT+. Even though Table 27 suggests that the CT+recommendations yielded positive results for most of the apps, the effect sizes were all small or negligible and there was no statistical difference.

We hypothesize that the reason for this result was the very high standard deviation present in the collected samples. Even though the standard deviations we observed for mobile devices (Table 17) was in general much higher than for the desktop devices (Table 13), they were even higher for **Tab4**. Considering the two versions of each of the five apps we have analyzed on **Tab4**, only one version exhibited a standard deviation lower than 10% of the mean energy consumption (the original Google Gson) and the worst-case scenario reached more than 30% (the original PasswordGen), as shown in Table 27.

When investigating the reasons behind this result, we noticed that the battery was discharging at an inconsistent rate, even when in an idle state. We hypothesize that this inconsistency stems from the device's age. For reference, Table 11 shows the age of the devices when executed the experiments on them. On older devices, the energy consumption difference between the original and the modified version seems to be more indistinguishable, that is, the impact made by optimizing the collections were not represented in the final energy consumption. Although these experiments involved only 4 mobile devices, they suggest that battery age can impact energy measurements. Since real-world device usage involves both old and new devices, simply using newer devices is not an appropriate solution. Instead, we recommend that future studies include the **age of the devices (or their batteries)** when reporting experimental results. Future experiments about this could involve measuring the energy consumption on a device using an old battery and then re-execute the experiments, using a brand new battery.

Table 27 – Results for **Tab4**. Energy results are red for the original versions and green for the modified versions. Energy measured in Joules.

System	Improvement	p-value	Mean	Stdev	Effect Size
Device: Tab4					
Commons Math	1.99%	0.39	23.62 23.04	3.31 4.47	0.10
Google Gson	1.58%	0.80	56.01 55.15	5.04 6.77	0.05
Xstream	6.16%	0.24	26.93 25.20	6.48 6.91	0.17
PasswordGen	6.23%	0.24	28.80 27.36	9.65 6.77	0.03
FastSearch	4.44%	0.80	50.11 47.08	7.34 7.49	0.27

Table 28 – Recommended collections for **Tab4**. Implementations from Apache Common Collections are presented in blue and from Eclipse Collections in red.

System	Original	Recommended	# of times
Device: Tab4			
Commons Math	ArrayList	TreeList	50
	ArrayList	NodeCachingLinkedList	9
	HashMap	ConcurrentSkipListMap	8
	ArrayList	FastList	5
	HashSet	LinkedHashSet	5
	TreeSet	LinkedHashSet	2
	HashMap	TreeMap	1
	HashSet	TreeSortedSet	1
FastSearch	ArrayList	TreeList	2
	ArrayList	NodeCachingLinkedList	2
Google Gson	ArrayList	TreeList	9
	ArrayList	NodeCachingLinkedList	3
	HashMap	ConcurrentSkipListMap	3
	ConcurrentHashMap	SynchornizedLikedHashMap	1
PasswordGen	ArrayList	TreeList	8
	ArrayList	FastList	1
Xstream	HashMap	ConcurrentSkipListMap	48
	ArrayList	TreeList	15
	HashSet	LinkedHashSet	8
	HashMap	TreeMap	3
	HashMap	UnifiedMap	3
	ArrayList	NodeCachingLinkedList	2
	HashSet	TreeSortedSet	1
	ArrayList	FastList	1
	LinkedHashMap	UnifiedMap	1
	LinkedHashSet	TreeSortedSet	1
	LinkedList	NodeCachingLinkedList	1

4.7 RELATED WORK

Energy profiling research has been conducted in different contexts, including embedded systems (ŠIMUNIĆ et al., 2000), cloud computing (CHEN et al., 2011), concurrent programming primitives (PINTO; CASTOR; LIU, 2014b; LIMA et al., 2019), neural networks and models (ROMANSKY et al., 2017). These studies share a common finding: simple changes can reduce energy consumption considerably. However, most of these studies do not provide tool support for developers. If interested, developers are still required to have: (1) the infrastructure (software and, eventually, hardware) to conduct the experiments, and (2) in-depth knowledge of low-level implementation details. As a result, non-specialist developers have little chance to apply the findings in real-world scenarios. In contrast, our approach is focused on non-expert developers. They do not have the knowledge neither, time, or tools to understand the energy impact of energy variation hotspots, but still want to reduce energy consumption. With an appropriate design and implementation (Sections 4.2 and 4.3), we believe that our approach can be reused and useful at scale.

In this chapter, we introduce a general idea to save energy during software development and instantiate it in a tool called CT+. We take this into account by organizing related work in terms of empirical studies (which create knowledge about energy efficiency) and recommendation tools (which apply that knowledge).

Empirical studies. In recent years, researchers have empirically analyzed several different aspects of energy consumption on software engineering. Rocha, Castor e Pinto (2019) took a look at the energy behavior of I/O APIs on 22 Java benchmarks and 3 macro-benchmarks. They showed that there is not a single API that can be used on every application independently. Another important aspect of their work was that small modifications on these APIs can result in a better energy performance by already optimized applications. Georgiou e Spinellis (2020) investigated the energy consumption impact of seven different programming languages on inter-process communication, finding out that the implementations on Javascript and Go usually are the most energy efficient. Duarte et al. (2019) developed a framework based on model analysis to study the energy consumption of software systems and then evaluated it experimentally. Among the different usages of their framework, one is to detect refactoring points that could be changed to reduce energy consumption.

Another way to look into the ways developers can save energy is try to optimize their energy choices even before the system development starts, that is, at the design phase. Sahin et al. (2012) made an investigation about the energy impact of 15 different design patterns. Their results show that design patterns could have a high effect on the energy consumption, going from a pattern that consumes 10 times more energy than the original code to another one that only consumes half the energy. In a similar fashion, Cruz e Abreu (2017) analyzed the energy impact of code smells on Android applications. In their work, they present empirical evidence that these anti-patterns increase the energy consumption of mobile applications and should be avoided by developers to help create more energy efficient apps. Lyu et al. (2017) analyzed the usage of local database requests in Android applications, founding out that the most expensive operations are database initialization and write operations. These operations are often used in loops and developers could design them to be bundled to reduce their energy consumption. Chowdhury et al. (2019) studied the impact of different strategies to optimize the energy consumption of the Model-View-Controller architectural pattern (MVC). Two strategies were used to optimize how these types of applications handle data influx: (i) bundling the data updates or (ii) using only the most recent one. These strategies were used to create new versions of the applications. With changes that are almost imperceptible to the human eye and do not impact the user experience, they showed that it is possible to reduce the energy consumption of a MVC application up to 36%.

More specifically, some works have dealt only with recommending more energy-efficient collections. Hasan et al. (2016) compared the energy consumption of collections in Java. They built an energy consumption profile for each collection they analyzed, aiming to answer which implementation of each collection (`Lists`, `Sets`, and `Maps`) consumed less energy. They used that information to manually improve the efficiency of a set of selected applications. Pinto et al. (2016) studied the thread-safe Java Collections on two different desktop machines. The authors found that the cost of each operation varies widely among different implementations of the same collection. For instance, the authors found energy improvements of 66%, when changing to a more energy efficient implementation of `Map`. Saborido et al. (2018) compared two Android-specific collection implementations of `Maps`: `SparseArray` and `ArrayMap`. These implementations were developed to be more efficient than `HashMap`. In summary, `ArrayMap` was considered worse than `HashMap` when optimizing energy consumption and `SparseArray` was considered better when the keys are primitive types. Cruz, Abreu e Rouvignac (2017) developed a refactoring tool called `Leafactor`, capable of statically analyzing and refactor code applying energy-efficient optimizations to Android apps, focusing on removing energy smell from applications. They executed their tool in 140 open-source apps and did a total of 222 code refactors.

Here we investigate the impact of software constructs in energy consumption while also offering a recommendation tool that can be used by developers to save energy. Therefore, this work builds upon knowledge produced by previous studies to make recommendations in an automated manner.

Recommendation tools. Manotas, Pollock e Clause (2014) developed a general purpose framework called SEEDS to guide developers on the laborious work of creating energy-aware software systems. They instantiate the concepts of that framework with the objective of analyzing the consumption of different collection implementations from the JCF. While our proposal uses the concept of energy profile and static analysis to analyze the applications and suggest an implementation of a collection, SEEDS leverages dynamic analysis, executing each different collection for every application and comparing their energy consumption. Furthermore, it did not consider the impact of multithreading and only targeted a desktop environment.

Pereira et al. (2018) implemented an energy-aware tool called jStanley, aiming to recommend the best collection implementation among several over the Java Collections Framework. jStanley was implemented as an Eclipse plugin and worked using experimental results from prior work by the same authors (PEREIRA et al., 2016). It does not account for the impact of loops and it works exclusively in a mobile environment. Furthermore, it does not account for thread-safety.

4.8 CONCLUSION

With this work, we present our vision of a general-purpose approach to aid non-specialist developers to create energy-aware software. This vision was instantiated within a tool to recommend energy-efficient collection implementations. We evaluated two different experiments studies, analyzing the influence of devices and energy profiles on software systems' energy-efficiency using Java collections. Overall, we executed our tool in seven different devices running seventeen different software systems (two mobile, twelve desktop, and three on both environments), and six other energy profiles for a total of 64software versions.

Although in some cases, the recommendations provided did not have a direct impact on energy consumption, our tool was able to reduce energy consumption of some applications up to 16.34%. Overall, CT+made a total of 2295recommendations that lead to a statistically significant impact on energy-efficiency.

Developers trying to increase the energy efficiency of their apps should use alternative sources of collections besides the Java Collections Framework, once our results suggest that some of the most popular collections implementations (e.g., `ArrayList`, `HashMap`, `HashSet`, and `Hashtable`) are often not the most energy-efficient ones. Another important point is trying to estimate the application's workload, since the implementations have different energy behavior while dealing with different quantities of data and developers should try to choose a collection based on the expected work. Finally, optimizing for a single device is not representative of the full spectrum of Android devices. Developers should try to run their tests on different devices and optimize based on their results.

Practitioners trying to use CT+ would want to create an energy profile for better recommendations on a specific device or use one of our energy profiles that matches their hardware. Using non-specific energy profiles could lead to less than optimal recommendations. The software system collection analysis and the recommendations are done entirely by CT+. On our companion website, practitioners can find CT+ repository, the energy profiles used in this work and a cookbook to make it easier to use CT+.

5 THESIS CONCLUSION

Unnecessary energy consumption on mobile devices is a problem that affects developers and end-users. Despite its importance, there is still a notable lack of tools and knowledge to address the shortcomings of green computing development. With this work, we hope to mollify this problem by bringing data and knowledge about various factors that have a significant impact on energy consumption, namely development approaches and Java collections.

In this chapter we summarize our results and the contributions made to the state-of-art. We also present further improvements for the main works presented in this thesis and ideas about future work in the area of green computing.

5.1 SUMMARY

The work presented in this thesis aims to mitigate the **problem** of *the rapid discharge of battery-powered devices*. Although that problem was present as soon as battery-powered devices started to be used, it became a more urgent matter with the rise of smart devices and how people depend on them.

Focusing on this problem and using the concept of energy design diversity, we presented our **hypothesis**, that is, *different solutions can be leveraged to ensure a reduction on energy consumption but without significantly increasing the development complexity*.

To test our hypothesis, two research questions were made: ***RQ1:*** *how can we increase the energy efficiency of a fully functional Android application using an alternative development approach to Java?* and ***RQ2:*** *how can we increase the energy efficiency of a fully functional software system optimizing the Java Collections used by it?*

The two research questions were answered on the Chapter 3 and Chapter 4, using empirical studies and analysis of the energy consumption on the different scenarios. We tried on these chapters to show the importance of the understating of energy design diversity. In both cases, some of the solutions already established were not the most energy efficient ones. By analyzing the energy behavior of the available solutions, we were able to find the optimal solution, reducing the energy consumption without increasing the complexity of the source code. We believe that developers that design their applications using the concept of energy design diversity should produce more energy efficient applications.

5.2 THE CONTRIBUTIONS

With this thesis, we make the following contributions to the field of green computing:

1. Insights on the energy footprint of the development approaches on Android.
2. The importance of experimenting on different devices on mobile.
3. An understanding of factors that influence energy experiments with energy profiles.
4. The viability of energy diversity design as the main concept to improve energy efficiency.
5. A general approach to design energy-efficient applications.
6. Automated energy optimizer refactoring tool.
7. An automatic wireless experiment runner for Android.

Across this thesis, we showed the impact of using the concept of energy design diversity to develop more energy efficient software systems. That was presented in the scenario of development approaches for Android applications and on Java Collections. In both cases, our solutions saved a non-negligible amount of energy across the modified apps. Our results show that developers can save energy by using JavaScript and C++ on CPU-bound applications; using parallelism on specific scenarios, using alternative sources to the Java Collections Framework, and optimizing their application to an expected workload. We believe that there are plenty of opportunities to save energy by developing using the concept of energy design diversity and hope that this thesis fosters that in practitioners.

5.3 FUTURE WORK

Although much has been done in the field of green computing, we believe that there are still some areas that could receive more attention. This section is split in two sub-sections: the first one, Section 5.3.1, lists some direct extensions to the work presented in this thesis; the second one, Section 5.3.2, presents paths where design diversity could be leveraged to improve energy efficiency in new ways.

5.3.1 Refining this thesis

The research work conducted on Chapter 3 could be extended in the following ways:

- **By investigating the reasons behind the different results on execution time and energy consumption of the different approaches.** Although we have empirical evidence that different types of applications can be improved by the different approaches, right now, there is still room for improving our knowledge about the reasons behind these differences.
- **By extending the analysis of Chapter 3 to the Kotlin programming language.** In recent years, Kotlin is becoming more popular[1] and it is fully supported by Google as a development approach for Android. Including Kotlin as another possible approach would enrich the work with one more possible way to save energy.
- **By producing a guide to aid in the selection of the most appropriate development approach.** It may be useful to develop a guide discussing specific scenarios where the developer should choose one approach over the other, if energy efficiency is a pressing concern. It is also possible to make this recommendation having in mind other quality attributes, such as performance, ease of use, and maintainability.

The research work conducted on Chapter 4 could be extended in the following ways:

- **By modifying software systems using CT+ and submitting patches to developers.** CT+ is capable of reducing the energy consumption of software systems on mobile and desktop environments. Applying these modifications on real software systems and submitting these changes as patches would enrich the work.
- **By experimenting with CT+ and developers.** CT+ was made to be fairly simple to use. It can automatically detect inefficiencies, make recommendations of changes for more energy-efficient collections implementations, and apply those recommendations. Nevertheless, until now, CT+ has only been used by researchers. The feedback of developers would greatly help further improve the tool.

[1] <https://alignminds.com/kotlin-fastest-growing-language/>

- **Reduce the time to create an energy profile.** The process of creating an energy profile can be very time-consuming. On our work, we mitigated it (Section 4.5) but there is still room for improvement.

5.3.2 Beyond this thesis

In this section we present three ideas that can lead to further improvements in the energy efficiency of existing software.

5.3.2.1 Cross-platform frameworks

With the increasing use of applications by users, companies that want to develop using native code often have to maintain at least three repositories using three different programming languages to cover most of their possible customers (i.e., Swift for IOS, Java or Kotlin for Android, and Javascript for web). Hybrid development platforms aim to solve this problem, unifying the development language in Javascript and, sometimes, bringing native graphical interface elements in order to improve the interaction with the user.

Early hybrid development frameworks, such as Ionic, needed to use some bridge (e.g., Cordova, PhoneGap) to connect native APIs and functions available in Javascript. One of the problems with this approach was the need to use a `WebView` (i.e., a scaled-down version of a web browser) to execute the code. Because the code was being executed in a `WebView`, the application's performance could be impacted. Another downside of this approach was its lack of support for graphical user interface elements. Modern solutions to the different repositories problem abandoned the concept of hybrid framework and adopted instead what we call cross-platforms frameworks. The difference is that they are not executed on top of a `WebView`. Instead, their code runs on a virtual machine or is translated to native while they are compiled. Some examples of these frameworks are React Native (developed by Facebook), Flutter (developed by Google), Xamarin (developed by Microsoft), and NativeScript (developed by Progress). These frameworks offer native elements and support graphics, making them easier to use by developers, increasing the performance by eliminating the need to use a `WebView`.

Although hybrid and cross-platform frameworks have the same objective, i.e., to reduce the complexity of maintaining three different source code repositories, developers may find it difficult to gather information about their energy footprint. Furthermore, differently from hybrid frameworks that were written in JavaScript, cross-platforms frameworks may have different programming languages (React Native uses JavaScript, Flutter uses Dart and Xamarin uses C#) The different programming languages add another layer of complexity to the problem. These frameworks are used by thousands of projects (in April 2021, we executed a query on GitHub with the name of each of these frameworks. The tag "React Native" returned 24,852 public repositories, while the tag "Flutter" 19,719 and the tag "Ionic" 3,728) but there is a lack of knowledge about the energy cost of using them.

To have an initial understanding of the energy footprint of these hybrid and cross-platform frameworks, we conducted some preliminary experiments, analyzing the differences in energy consumption of Ionic, an hybrid framework, and React Native, a cross platform framework. They are selected as preliminary candidates because they are the most popular choices in their categories[2] and both use JavaScript as the main programming language, also a popular programming language[3].

For our preliminary experiments, we have used the benchmarks from the Computer Language Benchmark Game. Because React Native does not have native parallelism, all benchmarks we used were sequential. The selected benchmarks were: BINARYTREE, FANNKUCH, FASTA, KNUCLEOTIDE, NBODY, PIDIGITS, REGEX, and REVCOMP. All our experiments were executed in a single device (Motorola G5).

Figure 11 shows the results of the execution of these benchmarks on React Native and Ionic. For every benchmark tested, React Native consumed more energy and took more time to execute than Ionic, even though, performance wise, React Native is seen by developers as better [4].

Although these are very preliminary results, they show that there is a difference in energy consumption among cross-platform frameworks and that this difference can be non-intuitive. Because these platforms are open-source (with the exception of Xamarin), researchers could analyze the code looking for points of excessive energy consumption that can be optimized. Understating the energy inefficiencies of these frameworks could lead to improvements on their infrastructure. This could reduce the energy consumption of thousands of apps and help future developers choose the right framework for their project.

[2] <https://insights.stackoverflow.com/survey/2020#most-popular-technologies>
[3] Same as footnote 2
[4] <https://enappd.com/blog/ionic-vs-react-native-which-one-is-better/116/>

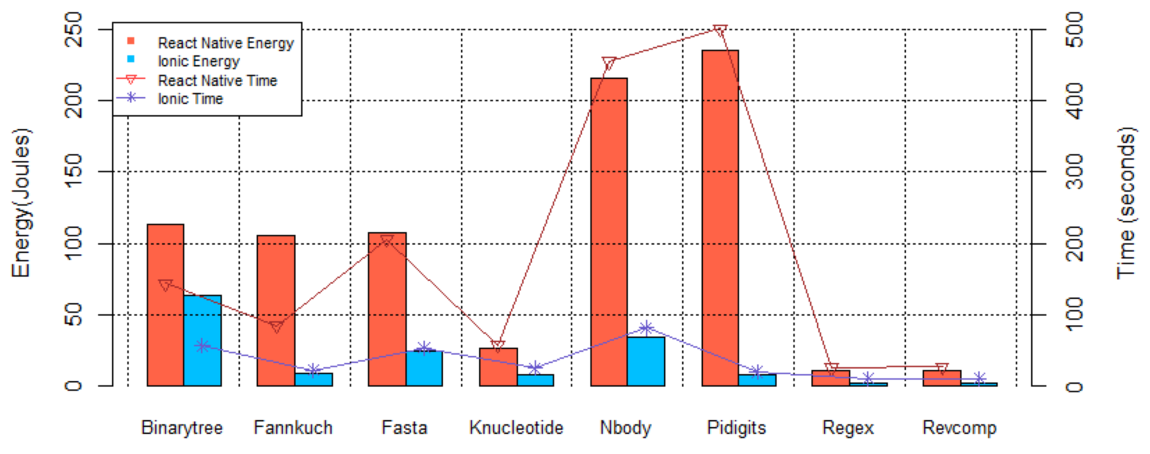


Figure 11 – Computer Language Benchmark Game benchmark results comparing React Native and Ionic

5.3.2.2 *Screen Colors*

Mobile device screens currently have very high pixel densities, sometimes surpassing even TVs and computer monitors that are several times larger than mobile screens. This sharp growth in image quality is accompanied by an equally sharp growth in energy consumption by the device's display. That is a problem because unlike TVs and monitors, smartphones are not plugged into a power outlet all the time. Screens became one of the most energy-hungry factors of a modern device, and even a small increase in energy efficiency can lead to a meaningful increase in battery life over time (TAWALBEH; EARDLEY et al., 2016).

Manufactures mainly use two different types of displays on mobile devices: Liquid Crystal Display (LCD) (Liquid Crystal Display) or OLED (Organic Light-Emitting Diode). Devices with LCD screens draw energy from the battery constantly, regardless of the colors displayed on the screen. This is not the case for OLED screens. Previous work has pointed out that the energy consumed by the device's display is heavily correlated with the screen colors and it's brightness (DONG; ZHONG, 2011; AGOLLI; POLLOCK; CLAUSE, 2017; LINARES-VÁSQUEZ et al., 2018). Given the large amount of energy destined for the screen in mobile devices, several studies have tried to reduce it. Having in mind that OLED's screen consumption is heavily related to colors, some works have used power profilers to reduce energy consumption by changing the colors scheme of browsers (DONG; ZHONG, 2011), mobile apps (DONG; ZHONG, 2012; WAN et al., 2017), and mobile web apps (LI; HALFOND, 2014). As those works are guided to maximize the energy efficiency, focusing on reducing the energy consumption of the application above all things, leaving aside other important factors such as aesthetics and usability.

Linares-VÁsquez et al. (2018) tried to solve the problem of high energy consumption by the incorrect color scheme in a different way. They developed a tool, called GEMMA, to modify applications and reduce the screen energy consumption by changing the color schemes. Unlike the previous solutions, they took into consideration three important factors: (i) consistency of color schemes, (ii) different windows within the same app, and (iii) different time spans spent in these interfaces. GEMMA aims to produce pleasant and consistent color combinations, to provide an appealing GUI based on the original color scheme of the app while still saving energy.

Researchers could investigate deeper in the optimization of colors schemes to Android apps using static analysis. Previous work has looked into several different aspects of static data-flow analysis, particularly into security properties, leak defects, energy consumption, correctness, and performance properties (WU; YANG; ROUNTEV, 2016; ZHANG; LÜ; ERNST, 2012; BANERJEE et al., 2014; PAYET; SPOTO, 2012; FERNANDES; PINTO; CASTOR, 2017)). The possibility of statically analyzing the energy consumption of Android graphical user interfaces is still an open problem.

A possible solution could make use of the Program Analysis Toolkit For Android (also known as GATOR), described by Yang et al. (2018). It uses a new methodology to represent the GUI on Android, using a window transition graph (WTG). The possible UI window sequences and their associated events are represented on the WTG model. The objective would be to model the stack of currently-active windows, the changes to this stack, and the effects of callbacks related to these changes.

This work could have an impact on the energy efficiency of all OLED devices. OLED is a popular technology, with more than 1 billion devices using it, representing more than 30% of all smartphone screens[5]. Optimizing the color pattern of an application can have a significant impact on the energy consumption of the application.

5.3.2.3 Energy Linter

Developing an application with energy consumption in mind may be difficult for a developer. First, because developers may not be familiar with techniques to reduce energy consumption. Second, because it may not be clear when and where these techniques can be applied, since

[5] https://newzoo.com/insights/articles/over-one-billion-smartphones-globally-have-an-oled-screen/#:~:text=Over%20One%20Billion%20Smartphones%20Globally%20Have%20an%20OLED%20Screen,-Bernd%20van%20der

apps with different characteristics (e.g., CPU-bound, network-bound, making intensive use of GPS, etc.) require different solutions. Third, because information about energy efficiency is spread throughout multiple sources, making it difficult to make informed decisions. Furthermore, that information is often imprecise, incomplete, or incorrect (LI; HALFOND, 2014; PINTO; CASTOR; LIU, 2014a).

Recently, a new technology has been gaining traction with developers and users: intelligent virtual assistants or IVA (e.g., Apple's Siri, Amazon's Alexa, Google Assistant, and Microsoft's Cortana). The market for Intelligent Virtual Assistants (IVA) is growing fast, and it is estimated to grow at an annual growth rate of 34.9% between 2016 and 2024[6]. This market involves not only IT applications but also automotive, education, healthcare, and retail[7]. Some recent studies have focused on developing or improving intelligent virtual assistants (GILBERT et al., 2011; JOHNSTON et al., 2014). In the context of aiding developers, Bradley, Fritz e Holmes (2018) has used an IVA to help developers manage tasks while developing applications.

Linters are tools to help developers increase their overall code quality. They work based on static analysis, searching and highlighting snippets of code that could decrease factors related to software quality (e.g., correctness, usability, security, performance, among others). Because of their efficiency, most IDEs (e.g., Android Studio) have a built-in linter to help developers avoid bad practices. Nevertheless, most Android developers do not rely on linters when dealing with performance matters (LINARES-VÁSQUEZ et al., 2015). Some of the reasons for that are: (i) the information the tool provides is not clear enough; (ii) it is hard and time consuming to write linter rules; (iii) developers did not see the real impact the change can make (HABCHI; BLANC; ROUVOY, 2018). Our hypothesis is that, using an IVA could solve most of these problems, helping to communicate the importance of the changes and giving the developers more information about them.

GoaËr (2020) made use of the Android Studio Linter to recommend changes based on the energy consumption, helping developers to reduce the energy consumption of their Android applications. Their methodology was to warn developers every time an energy smell was present in their code. Although this work has a lot of potential, there is still space to improve on the concept, such as: a more general approach, not only for Android applications; use of an IVA to help developers understand the rationale behind the chances; increase the number of energy smells available and allow the user to add new smells.

[6] https://gminsights.com/industry-analysis/intelligent-virtual-assistant-iva-market
[7] https://gminsights.wordpress.com/tag/intelligent-virtual-assistant-market-statistics//

Other works have presented different ways to try to help developers save energy while in the process of creating their apps. Pereira et al. (2020) developed a technique, SPELL, capable of helping developers create more energy-aware applications. Their technique is capable of finding energy inefficient portions of the code and presenting it to the developers. The authors executed an empirical experiment, asking participants to optimize the energy efficiency of a program on three scenarios: without any assistance, using an energy profiler and using SPELL. In all cases, the participants were able to reduce the energy consumption however, SPELL had the best results, with statistical significant difference from the energy profiler, with SPELL performing up to 72% better.

Raising developer awareness about the energy consumption of their apps could greatly increase the energy efficiency of applications across different environments. We believe that an Energy Linter (specially with the help an IVA to increase developers adoption), making use of energy-efficient energy patterns (CRUZ; ABREU, 2019a) could guide non-specialist developers to use energy efficient green computing, greatly reducing the lack of tools and knowledge (PINTO; CASTOR, 2017) that cripples the energy efficiency of applications.

REFERENCES

ABDULSALAM, S.; LAKOMSKI, D.; GU, Q.; JIN, T.; ZONG, Z. Program energy efficiency: The impact of language, compiler and implementation choices. In: *International Green Computing Conference*. [S.l.: s.n.], 2014. p. 1–6.

ABDULSALAM, S.; ZONG, Z.; GU, Q.; QIU, M. Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency. In: *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. [S.l.: s.n.], 2015. p. 1–8.

AGGARWAL, K.; ZHANG, C.; CAMPBELL, J. C.; HINDLE, A.; STROULIA, E. The power of system call traces: predicting the software energy consumption impact of changes. In: *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering, CASCON 2014*. [S.l.]: IBM / ACM, 2014. p. 219–233.

AGOLLI, T.; POLLOCK, L.; CLAUSE, J. 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)*. [S.l.: s.n.], 2017. p. 30–34.

ANDERSEN, L. O. *Program analysis and specialization for the C programming language*. Tese (Doutorado) — University of Cophenhagen, 1994.

ANDRAE, A. Total consumer power consumption forecast. *Nordic Digital Business Summit*, v. 10, 2017.

ANWAR, H. Towards greener android application development. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. [S.l.: s.n.], 2020. p. 170–173.

ANWAR, H.; DEMIRER, B.; PFAHL, D.; SRIRAMA, S. Should energy consumption influence the choice of android third-party http libraries? In: *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. New York, NY, USA: Association for Computing Machinery, 2020. (MOBILESoft '20), p. 87–97. ISBN 9781450379595. Disponível em: <https://doi.org/10.1145/3387905.3392095>.

ARDITO, L.; TORCHIANO, M. Creating and evaluating a software power model for linux single board computers. In: *2018 IEEE/ACM 6th International Workshop on Green And Sustainable Software (GREENS)*. [S.l.: s.n.], 2018. p. 1–8.

ASAFU-ADJAYE, J. The relationship between energy consumption, energy prices and economic growth: time series evidence from asian developing countries. *Energy Economics*, v. 22, n. 6, p. 615 – 625, 2000. ISSN 0140-9883.

AVIZIENIS, A.; KELLY, J. P. J. Fault tolerance by design diversity: Concepts and experiments. *Computer*, v. 17, n. 8, p. 67–80, ago. 1984.

BAGNATO, A.; ROCHETEAU, J. Towards green metrics integration in the measure platform. In: *MeGSuS@ ESEM*. [S.l.: s.n.], 2018. p. 39.

BALDWIN, C. Y.; CLARK, K. B. *Design Rules, Volume 1: The Power of Modularity*. 1. ed. [S.l.]: The MIT Press, 2000. v. 1.

BANERJEE, A.; CHONG, L. K.; CHATTOPADHYAY, S.; ROYCHOUDHURY, A. Detecting energy bugs and hotspots in mobile apps. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 588–598.

BARRETT, E.; BOLZ-TEREICK, C. F.; KILLICK, R.; MOUNT, S.; TRATT, L. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, ACM, New York, NY, USA, v. 1, n. OOPSLA, p. 52:1–52:27, out. 2017. ISSN 2475-1421. Disponível em: <http://doi.acm.org/10.1145/3133876>.

BENNETT, J.; LANNING, S.; NETFLIX, N. The netflix prize. In: *In KDD Cup and Workshop in conjunction with KDD*. [S.l.: s.n.], 2007.

BESSA, T.; GULL, C.; QUINTãO, P.; FRANK, M.; NACIF, J.; PEREIRA, F. M. Q. Jetsonleap: A framework to measure power on a heterogeneous system-on-a-chip device. *Science of Computer Programming*, v. 173, p. 21–36, 2019. ISSN 0167-6423. Brazilian Symposium on Programming Languages (SBLP '15+16). Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642317301776>.

BHATTACHARYA, S.; BASHAR, M.; SRIVASTAVA, A.; SINGH, A. Nomophobia: No mobile phone phobia. *Journal of Family Medicine and Primary Care*, v. 8, p. 1297, 04 2019.

BLACKBURN, S. M.; GARNER, R.; HOFFMANN, C.; KHANG, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIć, D.; VANDRUNEN, T.; DINCKLAGE, D. von; WIEDERMANN, B. The dacapo benchmarks: Java benchmarking development and analysis. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 169–190. ISBN 1-59593-348-4. Disponível em: <http://doi.acm.org/10.1145/1167473.1167488>.

BRADLEY, N. C.; FRITZ, T.; HOLMES, R. Context-aware conversational developer assistants. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 993–1003. ISBN 978-1-4503-5638-1. Disponível em: <http://doi.acm.org/10.1145/3180155.3180238>.

BUCHMANN, A. P.; KOLDEHOFE, B. Complex event processing. *it Inf. Technol.*, v. 51, n. 5, p. 241–242, 2009. Disponível em: <https://doi.org/10.1524/itit.2009.9058>.

C., M.; CHANDRASEKARAN, K.; CHIMALAKONDA, S. Energy diagnosis of android applications: A thematic taxonomy and survey. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 53, n. 6, dez. 2020. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3417986>.

CAMERON, K. W.; GE, R.; FENG, X. High-performance, power-aware distributed computing for scientific applications. *Computer*, v. 38, n. 11, p. 40–47, nov 2005. ISSN 0018-9162.

CHARLAND, A.; LEROUX, B. Mobile application development: Web vs. native. *Commun. ACM*, ACM, New York, NY, USA, v. 54, n. 5, p. 49–53, maio 2011. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1941487.1941504>.

CHEN, Q.; GROSSO, P.; Van Der Veldt, K.; De Laat, C.; HOFMAN, R.; BAL, H. Profiling energy consumption of VMs for green cloud computing. In: *Proceedings - IEEE 9th*

International Conference on Dependable, Autonomic and Secure Computing, DASC 2011. [S.l.: s.n.], 2011. p. 768–775. ISBN 9780769546124.

CHEN, X.; ZONG, Z. Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation. In: *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. [S.l.: s.n.], 2016. p. 485–492.

CHOWDHURY, S.; BORLE, S.; ROMANSKY, S.; HINDLE, A. Greenscaler: Training software energy models with automatic test generation. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 24, n. 4, p. 1649–1692, ago. 2019. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-018-9640-7>.

CHOWDHURY, S.; NARDO, S. D.; HINDLE, A.; JIANG, Z. M. J. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, v. 23, n. 3, p. 1422–1456, Jun 2018. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-017-9545-x>.

CHOWDHURY, S. A.; HINDLE, A. Characterizing energy-aware software projects: Are they different? In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2016. (MSR '16), p. 508–511. ISBN 978-1-4503-4186-8. Disponível em: <http://doi.acm.org/10.1145/2901739.2903494>.

CHOWDHURY, S. A.; HINDLE, A.; KAZMAN, R.; SHUTO, T.; MATSUI, K.; KAMEI, Y. Greenbundle: An empirical study on the energy impact of bundled processing. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2019. p. 1107–1118.

CLIFF, N. Answering ordinal questions with ordinal data using ordinal statistics. *Multivariate Behavioral Research*, v. 31, p. 331–350, 06 1993.

COHEN, M.; ZHU, H. S.; SENEM, E. E.; LIU, Y. D. Energy types. In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. [S.l.: s.n.], 2012. p. 831–850.

CORRAL, L.; GEORGIEV, A. B.; SILLITTI, A.; SUCCI, G. Method Reallocation to Reduce Energy Consumption: An Implementation in Android OS. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2014. (SAC '14), p. 1213–1218. ISBN 978-1-4503-2469-4. Disponível em: <http://doi.acm.org/10.1145/2554850.2555064>.

COSTA, D.; ANDRZEJAK, A.; SEBOEK, J.; LO, D. Empirical study of usage and performance of java collections. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. New York, NY, USA: ACM, 2017. (ICPE '17), p. 389–400. ISBN 978-1-4503-4404-3. Disponível em: <http://doi.acm.org/10.1145/3030207.3030221>.

COUTO, M.; CARÇÃO, T.; CUNHA, J.; FERNANDES, J. P.; SARAIVA, J. Detecting anomalous energy consumption in android applications. In: PEREIRA, F. M. Q. (Ed.). *Programming Languages*. Cham: Springer International Publishing, 2014. p. 77–91. ISBN 978-3-319-11863-5.

COUTO, M.; PEREIRA, R.; RIBEIRO, F.; RUA, R.; SARAIVA, J. a. Towards a green ranking for programming languages. In: *Proceedings of the 21st Brazilian Symposium on Programming Languages*. New York, NY, USA: ACM, 2017. (SBLP 2017), p. 7:1–7:8. ISBN 978-1-4503-5389-2. Disponível em: <http://doi.acm.org/10.1145/3125374.3125382>.

COUTO, M.; SARAIVA, J.; FERNANDES, J. P. Energy refactorings for android in the large and in the wild. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2020. p. 217–228.

CRUZ, L.; ABREU, R. Performance-based Guidelines for Energy Efficient Mobile Applications. In: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. Piscataway, NJ, USA: IEEE Press, 2017. (MOBILESoft '17), p. 46–57. ISBN 978-1-5386-2669-6. Disponível em: <https://doi.org/10.1109/MOBILESoft.2017.19>.

CRUZ, L.; ABREU, R. Catalog of energy patterns for mobile applications. *Empirical Software Engineering*, v. 24, n. 4, p. 2209–2235, Aug 2019. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-019-09682-0>.

CRUZ, L.; ABREU, R. 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)*. [S.l.: s.n.], 2019. p. 101–104.

CRUZ, L.; ABREU, R.; GRUNDY, J.; LI, L.; XIA, X. Do energy-oriented changes hinder maintainability? In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2019. p. 29–40.

CRUZ, L.; ABREU, R.; ROUVIGNAC, J.-N. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. [S.l.: s.n.], 2017. p. 205–206.

DONG, M.; ZHONG, L. Chameleon: A Color-Adaptive Web Browser for Mobile OLED Displays. *CoRR*, abs/1101.1, 2011. Disponível em: <http://arxiv.org/abs/1101.1240>.

DONG, M.; ZHONG, L. Power modeling and optimization for oled displays. *IEEE Transactions on Mobile Computing*, IEEE, v. 11, n. 9, p. 1587–1599, 2012.

DUARTE, L. M.; ALVES, D. da S.; TORESAN, B. R.; MAIA, P. H.; SILVA, D. A model-based framework for the analysis of software energy consumption. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (SBES 2019), p. 67–72. ISBN 9781450376518. Disponível em: <https://doi.org/10.1145/3350768.3353813>.

EMDEN, E. V.; MOONEN, L. Java quality assurance by detecting code smells. In: IEEE. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. [S.l.], 2002. p. 97–106.

FERNANDES, B.; PINTO, G.; CASTOR, F. Assisting non-specialist developers to build energy-efficient software. In: *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*. [S.l.: s.n.], 2017. p. 158–160. ISBN 9781538615898.

FOSSE, T. B. L.; MOTTU, J.-M.; TISI, M.; SUNYÉ, G. Characterizing a source code model with energy measurements. In: *Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS)*. [S.l.: s.n.], 2018.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 2018.

GEORGES, A.; BUYTAERT, D.; EECKHOUT, L. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 42, n. 10, p. 57–76, out. 2007. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1297105.1297033>.

GEORGIOU, S.; SPINELLIS, D. Energy-delay investigation of remote inter-process communication technologies. *Journal of Systems and Software*, v. 162, p. 110506, 2020. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121219302808>.

GILBERT, M.; ARIZMENDI, I.; BOCCHIERI, E.; CASEIRO, D.; GOFFIN, V.; LJOLJE, A.; PHILLIPS, M.; WANG, C.; WILPON, J. G. Your mobile virtual assistant just got smarter! In: *INTERSPEECH*. [S.l.: s.n.], 2011. p. 1101–1104.

GOAËR, O. L. Enforcing green code with android lint. In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. [S.l.: s.n.], 2020. p. 85–90.

GOTTSCHALK, M.; JELSCHEN, J.; WINTER, A. Saving Energy on Mobile Devices by Refactoring. *28th International Conference on Informatics for Environmental Protection (EnviroInfo 2014)*, p. to appear, 2014.

GOTTSCHALK, M.; JOSEFIOK, M.; JELSCHEN, J.; WINTER, A. Removing energy code smells with reengineering services. In: GOLTZ, U.; MAGNOR, M.; APPELRATH, H.-J.; MATTHIES, H. K.; BALKE, W.-T.; WOLF, L. (Ed.). *INFORMATIK 2012*. Bonn: Gesellschaft für Informatik e.V., 2012. p. 441–455.

GRIMMER, M.; RIGGER, M.; STADLER, L.; SCHATZ, R.; MöSSENBöCK, H. An efficient native function interface for java. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. [S.l.]: ACM, 2013. p. 35–44.

HABCHI, S.; BLANC, X.; ROUVOY, R. On adopting linters to deal with performance concerns in android apps. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2018. p. 6–16.

HABCHI, S.; MOHA, N.; ROUVOY, R. Android code smells: From introduction to refactoring. *Journal of Systems and Software*, v. 177, p. 110964, 2021. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121221000613>.

HÄHNEL, M.; DÖBEL, B.; VÖLP, M.; HÄRTIG, H. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 40, n. 3, p. 13–17, jan. 2012. ISSN 0163-5999. Disponível em: <https://doi.org/10.1145/2425248.2425252>.

HANGAL, S.; LAM, M. S. Tracking down software bugs using automatic anomaly detection. In: IEEE. *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. [S.l.], 2002. p. 291–301.

HAO, S.; LI, D.; HALFOND, W. G. J.; GOVINDAN, R. Estimating mobile application energy consumption using program analysis. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 92–101. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486801>.

HASAN, S.; KING, Z.; HAFIZ, M.; SAYAGH, M.; ADAMS, B.; HINDLE, A. Energy Profiles of Java Collections Classes. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16, 9), p. 225–236. ISBN 978-1-4503-3900-1. ISSN 1098-6596. Disponível em: <http://doi.acm.org/10.1145/2884781.2884869>.

HINDLE, A. Green mining: A methodology of relating software change to power consumption. In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. [S.l.: s.n.], 2012. p. 78–87. ISSN 2160-1852.

HINDLE, A.; WILSON, A.; RASMUSSEN, K.; BARLOW, E. J.; CAMPBELL, J. C.; ROMANSKY, S. Greenminer: a hardware based mining software repositories software energy consumption framework. In: *11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. p. 12–21.

HINTEMANN, R.; HINTERHOLZER, S. Energy consumption of data centers worldwide. In: *The 6th International Conference on ICT for Sustainability (ICT4S). Lappeenranta*. [S.l.: s.n.], 2019.

HOQUE, M. A.; SIEKKINEN, M.; KHAN, K. N.; XIAO, Y.; TARKOMA, S. Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 48, n. 3, p. 39:1—-39:40, dec 2015. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/2840723>.

IANNONE, E.; PECORELLI, F.; NUCCI, D. D.; PALOMBA, F.; LUCIA, A. D. Refactoring android-specific energy smells: A plugin for android studio. In: *Proceedings of the 28th International Conference on Program Comprehension*. [S.l.: s.n.], 2020. p. 451–455.

JAGROEP, E.; WERF, J. M. E. M. van der; JANSEN, S.; FERREIRA, M.; VISSER, J. Profiling energy profilers. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2015. (SAC '15), p. 2198–2203. ISBN 9781450331968. Disponível em: <https://doi.org/10.1145/2695664.2695825>.

JOHNSTON, M.; CHEN, J.; EHLEN, P.; JUNG, H.; LIESKE, J.; REDDY, A.; SELFRIDGE, E.; STOYANCHEV, S.; VASILIEFF, B.; WILPON, J. Mva: The multimodal virtual assistant. In: *15th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. [S.l.: s.n.], 2014. p. 257.

KAMBADUR, M.; KIM, M. A. An experimental survey of energy management across the stack. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. New York, NY, USA: ACM, 2014. (OOPSLA '14), p. 329–344. ISBN 978-1-4503-2585-1. Disponível em: <http://doi.acm.org/10.1145/2660193.2660196>.

KHOLMATOVA, Z. 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*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 1693–1695. ISBN 9781450370431. Disponível em: <https://doi.org/10.1145/3368089.3418777>.

KJÆRGAARD, M. B.; BLUNCK, H. Unsupervised power profiling for mobile devices. In: SPRINGER. *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. [S.l.], 2011. p. 138–149.

KWON, Y. W.; TILEVICH, E. Reducing the energy consumption of mobile applications behind the scenes. In: *Proceedings of the 29th IEEE International Conference on Software Maintenance*. Eindhoven, The Netherlands: [s.n.], 2013. p. 170–179.

LEE, J.; CHON, Y.; CHA, H. Evaluating battery aging on mobile devices. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. [S.l.: s.n.], 2015. p. 1–6.

LI, D.; HALFOND, W. G. J. An investigation into energy-saving programming practices for Android smartphone app development. In: *Proceedings of the 3rd International Workshop on Green and Sustainable Software - GREENS 2014*. New York, NY, USA: ACM, 2014. (GREENS 2014), p. 46–53. ISBN 9781450328449. Disponível em: <http://dl.acm.org/citation.cfm?doid=2593743.2593750>.

LI, D.; HALFOND, W. G. J. Optimizing energy of HTTP requests in Android applications. *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile (DeMobile 2015)*, p. 25–28, 2015.

LI, D.; HAO, S.; HALFOND, W. G. J.; GOVINDAN, R. Calculating Source Line Level Energy Information for Android Applications. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2013. (ISSTA 2013), p. 78–89. ISBN 978-1-4503-2159-4. Disponível em: <http://doi.acm.org/10.1145/2483760.2483780>.

LI, D.; LYU, Y.; GUI, J.; HALFOND, W. G. J. Automated Energy Optimization of HTTP Requests for Mobile Applications. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 249–260. ISBN 978-1-4503-3900-1. Disponível em: <http://doi.acm.org/10.1145/2884781.2884867>.

LI, D.; TRAN, A. H.; HALFOND, W. G. J. Making web applications more energy efficient for OLED smartphones. In: *36th International Conference on Software Engineering (ICSE'2014)*. [S.l.]: ACM, 2014. p. 527–538.

LIMA, L. G.; SOARES-NETO, F.; LIEUTHIER, P.; CASTOR, F.; MELFE, G.; FERNANDES, J. P. On haskell and energy efficiency. *Journal of Systems and Software*, v. 149, p. 554–580, 2019. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121218302747>.

LIN, Y.; DIG, D. A study and toolkit of CHECK-THEN-ACT idioms of java concurrent collections. *Softw. Test., Verif. Reliab.*, v. 25, n. 4, p. 397–425, 2015.

LINARES-VÁSQUEZ, M.; BAVOTA, G.; BERNAL-CáRDENAS, C.; OLIVETO, R.; PENTA, M. D.; POSHYVANYK, D. Mining energy-greedy api usage patterns in android apps: An empirical study. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 2–11. ISBN 978-1-4503-2863-0. Disponível em: <http://doi.acm.org/10.1145/2597073.2597085>.

LINARES-VÁSQUEZ, M.; BAVOTA, G.; BERNAL-CáRDENAS, C.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D. Multi-objective optimization of energy consumption of guis in android apps. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing

Machinery, New York, NY, USA, v. 27, n. 3, set. 2018. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3241742>.

LINARES-VÁSQUEZ, M.; VENDOME, C.; LUO, Q.; POSHYVANYK, D. How developers detect and fix performance bottlenecks in android apps. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2015. p. 352–361.

LITTLEWOOD, B.; POPOV, P.; STRIGINI, L. Modeling software design diversity: A review. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 33, n. 2, p. 177–208, jun. 2001. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/384192.384195>.

LIU, K.; PINTO, G.; LIU, D. Data-oriented characterization of application-level energy optimization. In: *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*. [S.l.: s.n.], 2015. (FASE'15).

LIU, Y.; XU, C.; CHEUNG, S.-C.; TERRAGNI, V. Understanding and Detecting Wake Lock Misuses for Android Applications. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016. (FSE 2016), p. 396–409. ISBN 978-1-4503-4218-6. Disponível em: <http://doi.acm.org/10.1145/2950290.2950297>.

LYU, Y.; GUI, J.; WAN, M.; HALFOND, W. G. J. An Empirical Study of Local Database Usage in Android Applications. In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017.

MACEDO, J. A. de; ALOÍSIO, J. A.; GONCALVES, N.; PEREIRA, R.; SARAIVA, J. A. Energy wars - chrome vs. firefox: Which browser is more energy efficient? In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*. New York, NY, USA: Association for Computing Machinery, 2020. (ASE '20), p. 159–165. ISBN 9781450381284. Disponível em: <https://doi.org/10.1145/3417113.3423000>.

MALAVOLTA, I.; CHINNAPPAN, K.; JASMONTAS, L.; GUPTA, S.; SOLTANY, K. A. K. Evaluating the impact of caching on the energy consumption and performance of progressive web apps. In: *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. New York, NY, USA: Association for Computing Machinery, 2020. (MOBILESoft '20), p. 109–119. ISBN 9781450379595. Disponível em: <https://doi.org/10.1145/3387905.3388593>.

MANOTAS, I.; BIRD, C.; ZHANG, R.; SHEPHERD, D.; JASPAN, C.; SADOWSKI, C.; POLLOCK, L.; CLAUSE, J. An empirical study of practitioners' perspectives on green software engineering. In: *ICSE*. [S.l.: s.n.], 2016. p. 237–248. ISBN 978-1-4503-3900-1.

MANOTAS, I.; POLLOCK, L.; CLAUSE, J. Seeds: A software engineer's energy-optimization decision support framework. In: *Proceedings of the 36th International Conference on Software Engineering*. [s.n.], 2014. (ICSE 2014), p. 503–514. ISBN 978-1-4503-2756-5. Disponível em: <http://doi.acm.org/10.1145/2568225.2568297>.

MASANET, E.; SHEHABI, A.; LEI, N.; SMITH, S.; KOOMEY, J. Recalibrating global data center energy-use estimates. *Science*, American Association for the Advancement of Science, v. 367, n. 6481, p. 984–986, 2020. ISSN 0036-8075. Disponível em: <https://science.sciencemag.org/content/367/6481/984>.

MATALONGA, H.; CABRAL, B.; CASTOR, F.; COUTO, M.; PEREIRA, R.; SOUSA, S. a. M. de; FERNANDES, J. a. P. Greenhub farmer: Real-world data for android energy mining. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019. (MSR '19), p. 171–175. Disponível em: <https://doi.org/10.1109/MSR.2019.00034>.

MCINTOSH, A.; HASSAN, S.; HINDLE, A. What can android mobile app developers do about the energy consumption of machine learning? *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 24, n. 2, p. 562–601, abr. 2019. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-018-9629-2>.

MORALES, R.; SABORIDO, R.; KHOMH, F.; CHICANO, F.; ANTONIOL, G. Anti-patterns and the energy efficiency of Android applications. *CoRR*, abs/1610.0, 2016. Disponível em: <http://arxiv.org/abs/1610.05711>.

MOURA, I.; PINTO, G.; EBERT, F.; CASTOR, F. Mining energy-aware commits. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2015. p. 56–67. ISSN 2160-1852.

MYASNIKOV, V.; SARTASOV, S.; SLESAREV, I.; GESSEN, P. Energy consumption measurement frameworks for android os: A systematic literature review. In: *Fifth Conference on Software Engineering and Information Management (SEIM-2020)(full papers)*. [S.l.: s.n.], 2020. p. 18.

NANZ, S.; FURIA, C. A. A comparative study of programming languages in rosetta code. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 778–788. ISBN 978-1-4799-1934-5. Disponível em: <http://dl.acm.org/citation.cfm?id=2818754.2818848>.

NUCCI, D. D.; PALOMBA, F.; PROTA, A.; PANICHELLA, A.; ZAIDMAN, A.; LUCIA, A. D. Software-based energy profiling of Android apps: Simple, efficient and reliable? In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [s.n.], 2017. p. 103–114. ISBN 978-1-5090-5501-2. Disponível em: <http://ieeexplore.ieee.org/document/7884613/>.

OLIVEIRA, W.; OLIVEIRA, R.; CASTOR, F. A Study on the Energy Consumption of Android App Development Approaches. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2017. ISBN 9781538615447.

OLIVEIRA, W.; OLIVEIRA, R.; CASTOR, F.; FERNANDES, B.; PINTO, G. Recommending energy-efficient java collections. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2019. p. 160–170.

OLIVEIRA, W.; OLIVEIRA, R.; CASTOR, F.; PINTO, G.; FERNANDES, J. P. Improving energy-efficiency by recommending java collections. *Empirical Software Engineering*, v. 26, n. 3, p. 55, Apr 2021. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-021-09950-y>.

PALOMBA, F.; Di Nucci, D.; PANICHELLA, A.; ZAIDMAN, A.; De Lucia, A. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, v. 105, p. 43–55, 2019. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584918301678>.

PANG, C.; HINDLE, A.; ADAMS, B.; HASSAN, A. E. What do programmers know about software energy consumption? *IEEE Software*, v. 33, n. 3, p. 83–89, May 2016. ISSN 0740-7459.

PATHAK, A.; HU, Y. C.; ZHANG, M. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. New York, NY, USA: ACM, 2012. (EuroSys '12), p. 29–42. ISBN 978-1-4503-1223-3. Disponível em: <http://doi.acm.org/10.1145/2168836.2168841>.

PATHAK, A.; JINDAL, A.; HU, Y. C.; MIDKIFF, S. P. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2012. (MobiSys '12), p. 267–280. ISBN 978-1-4503-1301-8. Disponível em: <http://doi.acm.org/10.1145/2307636.2307661>.

PAYET, É.; SPOTO, F. Static analysis of android programs. *Information and Software Technology*, Elsevier, v. 54, n. 11, p. 1192–1201, 2012.

PENZENSTADLER, B. Where attention goes, energy flows: Enhancing individual sustainability in software engineering. In: *Proceedings of the 7th International Conference on ICT for Sustainability*. New York, NY, USA: Association for Computing Machinery, 2020. (ICT4S2020), p. 139–146. ISBN 9781450375955. Disponível em: <https://doi.org/10.1145/3401335.3401684>.

PEREIRA, R.; aO, P. S.; CUNHA, J.; SARAIVA, J. jStanley: Placing a Green Thumb on Java Collections. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2018. (ASE 2018), p. 856–859. ISBN 978-1-4503-5937-5. Disponível em: <http://doi.acm.org/10.1145/3238147.3240473>.

PEREIRA, R.; CARçãO, T.; COUTO, M.; CUNHA, J.; FERNANDES, J. P.; SARAIVA, J. Spelling out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software*, v. 161, p. 110463, 2020. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121219302377>.

PEREIRA, R.; COUTO, M.; RIBEIRO, F.; RUA, R.; CUNHA, J.; FERNANDES, J. a. P.; SARAIVA, J. a. Energy efficiency across programming languages: How do energy, time, and memory relate? In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. New York, NY, USA: ACM, 2017. (SLE 2017), p. 256–267. ISBN 978-1-4503-5525-4. Disponível em: <http://doi.acm.org/10.1145/3136014.3136031>.

PEREIRA, R.; COUTO, M.; RIBEIRO, F.; RUA, R.; CUNHA, J.; FERNANDES, J. P.; SARAIVA, J. Ranking programming languages by energy efficiency. *Science of Computer Programming*, v. 205, p. 102609, 2021. ISSN 0167-6423. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642321000022>.

PEREIRA, R.; COUTO, M.; SARAIVA, J.; CUNHA, J.; FERNANDES, J. P. The Influence of the Java Collection Framework on Overall Energy Consumption. In: *Proceedings of the 5th International Workshop on Green and Sustainable Software*. New York, NY, USA: ACM, 2016. (GREENS '16), p. 15–21. ISBN 978-1-4503-4161-5. Disponível em: <http://doi.acm.org/10.1145/2896967.2896968>.

PEREIRA, R.; MATALONGA, H.; COUTO, M.; CASTOR, F.; CABRAL, B.; CARVALHO, P.; SOUSA, S.; FERNANDES, J. Greenhub: a large-scale collaborative dataset to battery consumption analysis of android devices. *Empirical Software Engineering*, v. 26, 05 2021.

PETERSON, M. Decisions under ignorance. In: ______. *An Introduction to Decision Theory*. [S.l.]: Cambridge University Press, 2009. (Cambridge Introductions to Philosophy), p. 40–63.

PETERSON, P. A. H.; SINGH, D.; KAISER, W. J.; REIHER, P. L. Investigating energy and security trade-offs in the classroom with the atom leap testbed. In: *Proceedings of the 4th Conference on Cyber Security Experimentation and Test*. [S.l.: s.n.], 2011. (CSET'11), p. 11–11.

PINTO, G.; CASTOR, F. Energy efficiency: a new concern for application software developers. *Communications of the ACM*, ACM, v. 60, n. 12, p. 68–75, 2017.

PINTO, G.; CASTOR, F.; LIU, Y. Mining questions about software energy consumption. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2014. (MSR 2014), p. 22–31.

PINTO, G.; CASTOR, F.; LIU, Y. D. Understanding energy behaviors of thread management constructs. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. [S.l.: s.n.], 2014. (OOPSLA '14), p. 345–360. ISBN 978-1-4503-2585-1.

PINTO, G.; LIU, K.; CASTOR, F.; LIU, Y. D. A comprehensive study on the energy efficiency of java thread-safe collections. In: *ICSME*. [S.l.: s.n.], 2016.

RATANAWORABHAN, P.; LIVSHITS, B.; ZORN, B. G. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In: *Proceedings of the 2010 USENIX conference on Web application development*. [S.l.: s.n.], 2010. p. 3–3.

ROCHA, G.; CASTOR, F.; PINTO, G. Comprehending energy behaviors of java i/o apis. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2019. p. 1–12.

RODRIGUES, R. E.; ALVES, P.; PEREIRA, F.; GONNORD, L. Real-world loops are easy to predict: a case study. In: *Workshop on Software Termination (WST'14)*. [S.l.: s.n.], 2014.

ROMANO, J.; KROMREY, J. D.; CORAGGIO, J.; SKOWRONEK, J. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the NSSE and other surveys? In: *Annual meeting of the Florida Association of Institutional Research*. [S.l.: s.n.], 2006.

ROMANSKY, S.; BORLE, N. C.; CHOWDHURY, S.; HINDLE, A.; GREINER, R. Deep green: Modelling time-series of software energy consumption. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 273–283.

SABORIDO, R.; ARNAOUDOVA, V. V.; BELTRAME, G.; KHOMH, F.; ANTONIOL, G. *On the impact of sampling frequency on software energy measurements*. [S.l.], 2015.

SABORIDO, R.; MORALES, R.; KHOMH, F.; GUÉHÉNEUC, Y.-G.; ANTONIOL, G. Getting the most from map data structures in Android. *Empirical Software Engineering*, mar 2018. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-018-9607-8>.

SAHIN, C.; CAYCI, F.; GUTIÉRREZ, I. L. M.; CLAUSE, J.; KIAMILEV, F.; POLLOCK, L.; WINBLADH, K. Initial explorations on design pattern energy usage. In: *2012 1st International Workshop on Green and Sustainable Software, GREENS 2012 - Proceedings*. [S.l.: s.n.], 2012. p. 55–61. ISBN 9781467318327.

SAHIN, C.; POLLOCK, L.; CLAUSE, J. How do code refactorings affect energy usage? In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2014. (ESEM '14), p. 36:1–36:10. ISBN 978-1-4503-2774-9.

SAHIN, C.; POLLOCK, L.; CLAUSE, J. From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. *Journal of Systems and Software*, v. 117, p. 307–316, 2016. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121216000893>.

SARAIVA, J. A.; ZONG, Z.; PEREIRA, R. Bringing green software to computer science curriculum. In: *To appear in 26th ACM Conference on Innovation and Technology in Computer Science Education*. [S.l.: s.n.], 2021.

SHAPIRO, S. S.; WILF, M. B. An Analysis of Variance Test for Normality (complete samples). *Biometrika*, v. 52, n. 3-4, p. 591–611, 1965.

SILVA-FILHO, A.; BEZERRA, P.; SILVA, F. Q.; JUNIOR, A.; SANTOS, A. L.; COSTA, P.; MIRANDA, R. Energy-aware technology-based dvfs mechanism for the android operating system. In: IEEE. *Computing System Engineering (SBESC), 2012 Brazilian Symposium on*. [S.l.], 2012. p. 184–187.

SILVA, J. C. R. da; PEREIRA, F. M. Q.; FRANK, M.; GAMATIÉ, A. A compiler-centric infra-structure for whole-board energy measurement on heterogeneous android systems. In: IEEE. *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. [S.l.], 2018. p. 1–8.

ŠIMUNIĆ, T.; BENINI, L.; De Micheli, G.; HANS, M. Source code optimization and profiling of energy consumption in embedded systems. In: *Proceedings of the International Symposium on System Synthesis*. [S.l.: s.n.], 2000. v. 2000-January, p. 193–198. ISBN 0769507654. ISSN 10801820.

SUBRAMANIAM, B.; FENG, W.-c. Towards energy-proportional computing for enterprise-class server workloads. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. [S.l.: s.n.], 2013. (ICPE '13), p. 15–26. ISBN 978-1-4503-1636-1.

TANG, Q.; LYU, H.; HAN, G.; WANG, J.; WANG, K. Partial offloading strategy for mobile edge computing considering mixed overhead of time and energy. *Neural Computing and Applications*, v. 32, n. 19, p. 15383–15397, Oct 2020. ISSN 1433-3058. Disponível em: <https://doi.org/10.1007/s00521-019-04401-8>.

TAWALBEH, M.; EARDLEY, A. et al. Studying the energy consumption in mobile devices. *Procedia Computer Science*, Elsevier, v. 94, p. 183–189, 2016.

THIAGARAJAN, N.; AGGARWAL, G.; NICOARA, A.; BONEH, D.; SINGH, J. P. Who Killed My Battery: Analyzing Mobile Browser Energy Consumption. In: *Proceedings of the 21st international conference on World Wide Web - WWW '12*. New York, NY, USA: ACM, 2012.

(WWW '12), p. 41. ISBN 9781450312295. Disponível em: <http://doi.acm.org/10.1145/2187836.2187843http://dl.acm.org/citation.cfm?doid=2187836.2187843>.

TIWARI, V.; MALIK, S.; WOLFE, A. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, v. 2, p. 437–445, 1994.

WAN, M.; JIN, Y.; LI, D.; GUI, J.; MAHAJAN, S.; HALFOND, W. G. Detecting display energy hotspots in android apps. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 27, n. 6, p. e1635, 2017.

WILKE, C.; PIECHNICK, C.; RICHLY, S.; PüSCHEL, G.; GöTZ, S.; ASSMANN, U. Comparing mobile applications' energy consumption. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2013. (SAC '13), p. 1177–1179. ISBN 978-1-4503-1656-9. Disponível em: <http://doi.acm.org/10.1145/2480362.2480583>.

WILKS, D. S. *Statistical methods in the atmospheric sciences*. Amsterdam; Boston: Elsevier Academic Press, 2011. ISBN 9780123850225 0123850223. Disponível em: <https://www.amazon.com/Statistical-Atmospheric-Sciences-International-Geophysics/dp/0123850223/ref=pd_bxgy_14_img_3?_encoding=UTF8&psc=1&refRID=ESPQQ0R2PB1TP1VJSGCZ>.

WU, H.; YANG, S.; ROUNTEV, A. Static detection of energy defect patterns in android applications. In: *International Conference on Compiler Construction*. [S.l.: s.n.], 2016. p. 185–195.

XU, F.; LIU, Y.; LI, Q.; ZHANG, Y. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In: *nsdi*. [S.l.: s.n.], 2013. v. 13, p. 43–56.

YANG, S.; WU, H.; ZHANG, H.; WANG, Y.; SWAMINATHAN, C.; YAN, D.; ROUNTEV, A. Static window transition graphs for Android. *International Journal of Automated Software Engineering*, p. 1–41, jun. 2018.

ZHANG, S.; LÜ, H.; ERNST, M. D. Finding errors in multithreaded gui applications. In: ACM. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. [S.l.], 2012. p. 243–253.

ZIMMERLE, C.; OLIVEIRA, W.; GAMA, K.; CASTOR, F. Reactive-based complex event processing: An overview and energy consumption analysis of cep.js. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (SBES 2019), p. 84–93. ISBN 9781450376518. Disponível em: <https://doi.org/10.1145/3350768.3352492>.