Genetic Improvement of Software for Energy Efficiency in Noisy and Fragmented Eco-Systems

Mahmoud Abdulwahab Bokhari

Optimisation and Logistics Group
School of Computer Science
University of Adelaide

A dissertation submitted for the degree of
Doctor of Philosophy
In the subject of
Computer Science

Supervisor:
Dr. Markus Wagner
Co-Supervisor:
Dr. Bradley Alexander

December 7, 2020

Contents

List of Figures

List of Tables

Genetic Improvement of Software for Energy Efficiency in Noisy and Fragmented Eco-Systems

Abstract

Software has made its way to every aspect of our daily life. Users of smart devices expect almost continuous availability and uninterrupted service. However, such devices operate on restricted energy resources. As energy efficiency of software is relatively a new concern for software practitioners, there is a lack of knowledge and tools to support the development of energy efficient software. Optimising the energy consumption of software requires measuring or estimating its energy use and then optimising it. Generalised models of energy behaviour suffer from heterogeneous and fragmented eco-systems (i.e. diverse hardware and operating systems). The nature of such optimisation environments favours *in-vivo* optimisation which provides the ground-truth for energy behaviour of an application on a given platform. One key challenge in *in-vivo* energy optimisation is noisy energy readings. This is because complete isolation of the effects of software optimisation is simply infeasible, owing to random and systematic noise from the platform.

In this dissertation we explore *in-vivo* optimisation using Genetic Improvement of Software (GI) for energy efficiency in noisy and fragmented eco-systems.

First, we document expected and unexpected technical challenges and their solutions when conducting energy optimisation experiments. This can be used as guidelines for software practitioners when conducting energy related experiments.

Second, we demonstrate the technical feasibility of in-vivo energy optimisation using GI on smart devices. We implement a new approach for mitigating noisy readings based on simple code rewrite.

Third, we propose a new conceptual framework to determine the minimum number of samples required to show significant differences between software variants competing in tournaments. We demonstrate that the number of samples can vary drastically between different platforms as well as from one point of time to another within a single platform. It is crucial to take into consideration these observations when optimising in the wild or across several devices in a control environment.

Finally, we implement a new validation approach for energy optimisation experiments. Through experiments, we demonstrate that the current validation approaches can mislead software practitioners to draw wrong conclusions. Our approach outperforms the current validation techniques in terms of specificity and sensitivity in distinguishing differences between validation solutions.

Declaration of Authorship

I acknowledge the support I have received for my research through the provision of the Saudi Cultural Mission and Taibah University Scholarship program.

Mahmoud Abdulwahab Bokhari

August 2020

Acknowledgements

In the name of Allah the most Gracious and the most Merciful, I first thank Allah for enabling me to finish my studies and for all the blessings I receive, praise be to Allah.

I would like to express my sincere and deepest appreciation to my principle supervisor Dr. Markus Wagner for the endless support, dedication, professionalism and patience throughout the last several years. Like the best leaders, he enlightened and advised me to conduct world-class research in my field. I still remember our extended meetings where he always challenged me to polish my ideas and to teach me how to be a great researcher. His challenges were the most significant factors in leading me on this journey and pushing me to go further than I expected. From MSc to a PhD, I hope these are just the initial collaborations on a long road ahead.

I would also like to express my gratitude to my co-supervisor Dr. Bradley Alexander for all helpful and insightful comments, ideas and collaborations throughout my PhD degree. Through his wisdom, I learned how to see the glass as half full whenever I encountered a new obstacle that got me at my wits' end during this winding journey. My gratitude also goes to my research collaborators Yuanzhong Xia, Younis Altoma, Bo Zhou, Bobby Bruce and Lujun Weng for their support in various phases of this research journey.

As it is said "keep the best for the last", I am extremely grateful to my family for a lifetime of love and support. From the bottom of my heart, I thank them for willingly listening as I ramble incomprehensibly about the problems I came across in my journey. All have contributed to where I am today, and each have given me motivation at different points of times to continue pushing through when it was needed. Even before I started this journey, when I was considering the topic of my PhD thesis, my sister Saliha advised to take the most recent and challenging one.

To my beloved family.

Chapter 1

Introduction

"Take up good deeds only as much as you are able, for the best deeds are those done regularly even if they are few."

Muhammad peace be upon him

1.1 Introduction

Software has assumed a core role in our daily lives. Software applications influence almost every field including mobile phones, medical treatments, and education. Software is embedded in almost all electronic devices from here on Earth up to space. The ubiquity and importance of software means that well-defined methodologies for its development must continually improve in order to avoid growing negative impacts from unreliable software.

At the beginning of the modern computing era, computer programs were simple in terms of the tasks they were able to perform. For example, the first computer in 1944 weighed 30 tons and consumed 174 kW (237 horsepower), but it could only execute programs that have no more than 5000 additions or 357 multiplications[1] [170].

In the following two decades, much effort was made to improve the state of computer hardware (HW), and more powerful devices emerged. In contrast to the fast improvement in hardware platforms, creating programs remained more a craft than a scientific or engineering discipline. In the early days of software development little effort was dedicated to investigating how to develop software that could leverage improving hardware capabilities.

[1]It was shut-down as the US Army could not justify its running costs.

Although badly constructed code can function, it can bring a development organisation to its knees by spending tremendous effort and countless hours to patch it and to alleviate its effects. Consequently, computer programming was in a state of crisis; projects failed or were delayed, running over budget, and delivered with low quality and inefficient computer programs. This situation led the NATO science committee to hold a conference that included international experts on computer software in 1968, and the terms: *Software Engineering* (SE) and *Software Crisis* were coined. The attendees concluded that in order to overcome the current crisis, new principles and methodologies are needed for software development.

While software engineering models have evolved and diversified over the decades software development processes still incorporate these core phases in various forms. Among these software construction can be considered the most vital phase [106]. This is because software construction is where the delivered code is first produced. Methodologies that assist in this phase can improve software practitioners' productivity enormously - freeing more times for further design and development.

In order to construct a successful software solution, one must clearly define:

1. what the software should do, in terms of its input and output, and

2. specify other required global constraints on how the software should behave.

The first point above defines the functional properties of software and represents the main goals of software from its users perspective such as an email client that sends and receives emails. The second point above defines the non-functional properties. These non-functional properties are sometimes called software quality attributes, which govern the quality of the service provided by the software product, for example, how secure and efficient the email client is in sending and fetching emails. These two sets of properties are both essential to project outcomes regardless of the nature of the project.

Software efficiency is one of the main quality attributes of software [73]. It refers to software performance and effectiveness relative to the amount of resources used under specific conditions. Examples of resources include CPU and memory utilisation, network bandwidth and energy consumption. Software efficiency is driven by several factors such as software requirements, architecture and design, operating system interactions, hardware utilisation and code tuning. Making decisions that accommodate these factors affect the software construction process [106].

As computer HW components and applications evolve, the complexity of software and its construction increases. On this point, Edgar W. Dijkstra stated in 1979:

"The machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem .... To put it in another way, as the power of available machines grew by a factor of more than a thousand, society's ambitions to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means" [41].

The evolution of HW diversified in the current millennium from the expansion in the use of data centres, warehouse-scale and super-computers, to the ubiquitous (anywhere and anytime) computing via smart devices and things. In turn, the specifications of smart devices have evolved dramatically from being simple hand-sets to powerful and sophisticated devices housing quad CPUs, up to 16 gigabytes of main memory, and 512 gigabytes of storage. This evolution is a consequence of rapid increase in their use and through them rapidly becoming an integral part of our lives. In 2018 Cisco reported that the total number of smart devices exceeded five billion [33] worldwide. Interestingly, a survey conducted by Boston Consulting Group reported that people are willing to sacrifice traditional needs rather than the use of mobile devices [53]. More than 60% of the participants would stop dining out for a year, 50% of them would put off going on vacation, and 46% would give up a day off per week rather than losing their smartphones. Such powerful devices and increased intensity of usage demands more electrical power consumption in order to satisfy users' needs.

In addition, Wirth's law states that Software is getting slower more rapidly than hardware becomes faster. There are several reasons of software complexity hurting its efficiency. This can happen by run-time bloating which refers to a general situation where a task in software is accomplished by an inefficient technique although there exist other efficient and easy-to-implement methods [174]. Software bloating is mainly due to the availability of more powerful HW resources which relaxes the effort of software practitioners to create efficient code. For instance, Java collection interface implementations have different energy usages. Each implementation can be more or less optimal in different contexts [123], and therefore developers may not choose the right implementation for their software. In general, over-utilisation of resources due to software bloating can result in higher power consumption.

The energy efficiency of software is relatively a new concern for software practitioners. This is mainly due to the new paradigms of mobile and ubiquitous computing introduced by the revolution in smart devices whose usefulness depends on restricted battery resources.

Unfortunately energy consumption of software is, in general, harder to measure than other non-functional software properties such as: time, memory and network usage. Moreover, according to a survey by Pang et al. [118], only 14% of developers considered energy consumption as a requirement and only 12% know how to measure software energy use. Besides the lack of knowledge, Manotas et al. [105], Pinto and Castor [127] observed that there is a lack of support tools to help developers solve issues related to software energy estimation and optimisation.

There is a clear need for better methods to measure and improve energy consumption of software on mobile platforms. To be maximally useful such methods need to automatically operate on diverse platforms without the need for specialised hardware.

In this thesis, we explore the *in-vivo* optimisation on smart devices using Genetic Improvement of Software (GI), a Search-Based Software Engineering (SBSE) technique, to automatically generate energy efficient software variants of an existing software.

1.2 Motivation

Arguably, lithium-ion batteries are the fastest growing and most promising battery technology at the present time; in just the time between 2010 to 2017 there were at least 119,188 publications on batteries [98]. However, the current advancing technologies and use profiles in ubiquitous computing are placing increasing demands on lithium-ion chemistry. Users of mobile devices expect almost continuous availability and uninterrupted service. Moreover, ageing is also a concern with lithium-ion batteries because the capacity deteriorates over time. This is because recharging cycles, induced by high energy consumption, cause irreversible capacity loss [98].

At the present time, it seems that engineers are incapable of increasing the amount of energy created by the chemical reactions in relatively small size batteries. Currently, enlarging the battery size is the primary way to increase its capacity. This conflicts with the evolution of smart devices, where lighter and thinner devices are more desirable. It is worth mentioning that nanotechnology is being explored to build batteries that are able to generate ten times the amount of energy of the current lithium-ion batteries [30]. To the best of our knowledge, the mobile industry has yet to adopt this new technology.

Smart devices are deployed with fixed policies that respond to the system state to save energy. For example, Android devices have CPU governors (e.g. interactive governor) that implement algorithms to determine CPU frequencies to be used under different situations. To save energy, the interactive governor quickly increases the frequency when it detects users' interactions and reduces the frequency when there is no interactions. However, there is currently no system that customises applications or system policies for each device and its regime of use.

Current work in this field attempts to profile and improve the energy use of software according to user behaviours and third-party applications [28, 112, 144]. However, these solutions do not consider the fact that smart devices eco-systems are diverse. The Android eco-system in particular is fragmented and heterogeneous, and third-party applications exhibit different energy needs and behaviours depending on the used HW configuration. In addition, previous work does not take into account that smart devices are evolving at rapid speed in terms of HW specifications and OS version[2]. This fragmentation necessitates device-dependent energy models and optimisation. If acted upon, these issues would require software developers to manually tune their applications to improve energy efficiency on each individual platform, which is clearly impractical.

In addition to the problem of diverse platforms, there is a lack of knowledge among developers of how to optimise applications for energy [105, 118, 127, 128]. For example, Pang et al. [118] observed in their survey that only 10% of developers correctly ranked mobile phones' HW components according to their energy use and only 3% of the participants identified the misuse of sensors such as GPS can cause high energy use of applications. Moreover, Manotas et al. [105] reported that compared to other performance bottlenecks, 61% of developers agree that energy issues are more difficult to discover, and 55% of the participants agree it is more difficult to diagnose. Furthermore, the authors above pointed out that there is a lack of support and tools that help facilitate energy-aware software development [105, 127].

A human or automated software optimiser needs to precisely measure the energy consumption in order to optimise it[3]. Software practitioners can use estimation models, external or internal HW meters for energy estimation and measurement. However, models abstract away the real behaviour of software and its host OS. It is also impractical to create a model for each platform. Additionally, capturing the whole required energy behaviour of software and its hosting OS in one model is infeasible.

Although external meters are accurate and precise, they are expensive and hard to set up because they require special skills for probing a device. In contrast, internal meters are built-in devices and energy reading can be obtained using APIs or by accessing the file system in worst-case scenario. They also have acceptable precision [15]. Therefore, they can be utilised for conducting the energy optimisation on each device (i.e. *in-vivo*).

[2]This will be discussed in detail in the Background section.

[3]"If you can't measure it, you can't improve it."– Peter Drucker.

Having internal meters in smart devices helps solve the problem of improving energy consumption of software. The next step is either to manually tune applications for energy usage per platform, which is infeasible, or to find a sound approach for automatically generating greener software variants without drastically changing their functionality, regardless of the hosting platform.

Fortunately, *in-vivo* Genetic Improvement (GI) of software non-functional properties can help to automatically generate customised software variants out of an existing software provided there is a precise method to measure each variant's quality (i.e. fitness function). In essence, the use of GI helps to preserve the functionality of the targeted application through regression testing. At the same time, it automatically generates customised variants of the target application improving its non-functional requirements such as energy. Admittedly, GI may affect software functionality, nevertheless, the majority (80%) of software engineers who work in energy-constrained systems are willing to sacrifice some requirements for energy improvements [105].

In this thesis, we plan to use the built-in meters to evaluate solutions created by GI. This constitutes *in-vivo* optimisation using GI. By the means of *in-vivo* optimisation using GI, we envision a future where software developers are only concerned with what should their applications do and leave the burden of how they should behave on different smart devices to *in-vivo* optimisation using GI.

1.3 Thesis Structure and Contributions

This thesis is the first published work to explore, in detail, the automatic *in-vivo* optimisation of energy used by applications on mobile platforms. In this section we outline the contribution of this thesis.

Chapter 2: *Background.* In this chapter we provide the required background information to facilitate reading this thesis. The chapter presents the architecture of Android operating system and discusses the problem of fragmentation in Android eco-system. It also presents an overview of optimisation processes that use meta-heuristic techniques, the Search-Based Software Engineering and Genetic Improvement of Software.

Chapter 3: *Deep Parameter Optimisation on Android Smartphones for Energy Minimisation - A Tale of Woe and a Proof-of-Concept.* With power demands of mobile devices rising, it is becoming increasingly important to make mobile software applications more energy efficient. Unfortunately, mobile platforms are diverse and very complex which makes energy behaviours difficult to model. This complexity presents challenges to the effectiveness of offline optimisation of mobile applications. In this chapter, we outline solutions to expected and unexpected technical challenges that arise when creating test-beds for energy optimisation experiments. We also demonstrate that it is possible to automatically optimise an application for energy on a mobile device by evaluating energy consumption *in-vivo*. In contrast to previous work, we use only the devices own internal meter. Our approach involves many technical challenges but represents a realistic path toward learning hardware specific energy models for program code features.

Chapter 4: *In-vivo and Offline Optimisation of Energy Use in the Presence of Small Energy Signals - A Case Study on a Popular Android Library.* Energy demands of applications on mobile platforms are increasing. As a result, there has been a growing interest in optimising their energy efficiency. As mobile platforms are fast-changing, diverse and complex, the optimisation of energy use is a non-trivial task. To date, most energy optimisation methods either use models or external meters to estimate energy use. Unfortunately, it becomes hard to build widely applicable energy models, and external meters are neither cheap nor easy to set up. To address this issue, we run application variants *in-vivo* on the phone and use a precise internal battery monitor to measure energy use. We describe a methodology for optimising a target application *in-vivo* and with application-specific models derived from the devices own internal meter based on jiffies and lines of code. To address the problem of noise in energy reading we propose a simple approach based on code rewrites. We demonstrate that this process produces a significant improvement in energy efficiency with limited loss of accuracy.

Chapter 5: *Mind the Gap - A Distributed Framework for Enabling Energy Optimisation on Modern Smart-Phones in the Presence of Noise, Drift, and Statistical Insignicance.* Smartphones are becoming essential to people's everyday lives. Due to the limited battery capacity of smartphones, researchers and developers are increasingly interested in the energy efficiency of these devices and the software applications that run on them. In the most basic setting, a developer might be interested in knowing which of two program variants might consume more energy, whether this is for use in regression testing or for use in full-scale evolutionary optimisation. To perform such comparisons (tournaments) reliably, we need a model of the number of trials needed to discern between two variants to a desired level of statistical significance. To enable this, we present a conceptual framework based on tournaments which we use to compare a range of test workloads on different combinations of phones and operating systems. We also use it to determine the minimum number of samples required to show a statistical significance when comparing between two variants in evolutionary tournaments. Our results quantify the number of trials required to resolve different variants to different levels of fidelity on a range of platforms.

Chapter 6: *Towards Rigorous Validation of Energy Optimisation Experiments.* The optimisation of software energy consumption is of growing importance across all scales of modern computing, i.e., from embedded systems to data-centres. Practitioners in the field of Search-Based Software Engineering and Genetic Improvement of Software acknowledge that optimising software energy consumption is difficult due to noisy and expensive fitness evaluations. However, it is apparent from results to date that more progress needs to be made in rigorously validating optimisation results. This problem is pressing because modern computing platforms have highly complex and variable behaviour with respect to energy consumption. To compare solutions fairly we propose in this chapter a new validation approach called R3-VALIDATION which exercises software variants in a rotated-round-robin order. Using a case study, we present an in-depth analysis of the impacts of changing system states on software energy usage, and we show how R3-VALIDATION mitigates these. We compare it with current validation approaches and show that it aligns better with actual platform behaviour.

Chapter 2

Background

"Be fearless in the pursuit of what sets your soul on fire"

Jennifer Lee

2.1 Introduction

In this chapter we provide the required background information to facilitate reading this thesis. We first present and discuss the architecture of Android operating system (OS) and the problem of fragmentation in its eco-system in Section 2.2. Then, we briefly review optimisation processes that use meta-heuristic techniques in Section 2.3. Next, we provide an overview of the Search-Based Software Engineering (SBSE) in Section 2.4, followed by an overview of Genetic Improvement of Software (GI) (which combines optimisation and SBSE) in Section 2.5. We wrap up the chapter in Section 2.6.

2.2 Android OS

In this thesis, we use Google Android operating system as the main platform for our experiments. Android is an open-source embedded operating system (OS) that runs on a diverse set of mobile platforms. It has evolved over the years and rapidly become the dominant mobile platform across the globe. As of June 2020, it holds over 80% of the global mobile OS marketshare[1] in terms of installed instances.

[1]https://tinyurl.com/y9eg9occ accessed in June 2020.

Figure 2.1 illustrates the main components of the Android Architecture. Android is based on a modified Linux kernel to support the special needs of pervasive computing. For example, Google introduced power wakelocks in the Linux kernel to improve the energy consumption and network usage of running background processes. In addition, the memory management system in Android kills processes with low priority as the available memory becomes low.

In order to facilitate the portability of Android on a variety of hardware (HW) devices, Google has abstracted the physical HW through the hardware abstraction layer (HAL). HAL provides interfaces to expose HW capabilities to running applications. It consists of modules for each HW component such as the Bluetooth module.

Android applications are compiled into Dalvik bytecode format (.dex files), which are packaged to create the Android Application Package (.apk file). The dex bytecode is designed for Android to run with a minimal memory footprint. This bytecode is dynamically translated into native code and executed by Dalvik Virtual Machine (DVM) or Android Runtime (ART). DVM uses Just-in-Time compilation process and is used in devices running Android 4 and earlier versions. In contrast, ART uses an Ahead-of-Time (AOT) compilation process to improve the runtime performance.

In addition, native libraries written in C and C++ are used to run core system components and services such as ART and HAL. Android applications can also access these using Android framework. For example, developers can access Webkit and OpenGL using their APIs to render web contents and add 2D and 3D graphics support to their applications, respectively.

All Android features are available to developers through APIs built using the Java language which is called Android APIs or framework. The framework contains a set of services that forms the environment in which Android applications run. For example, content providers allow application to save and share data with other applications, and the location manager provides location data and services to the running applications.

Android systems come with system apps that are shipped with every Android device to provide the main features and services to the end users such as making calls, internet browsing and messaging. Device manufacturers also provide their own system apps. Furthermore, they develop their own software overlay (a system app) which sits on top of the system, which are sometimes called skins. Examples of such skins include Oxygen OS, One UI and Sense developed by OnePlus, Samsung and HTC, respectively. It is worth mentioning that these skins are not just simply UI themes. For instance, they can interact with the native libraries and the kernel.

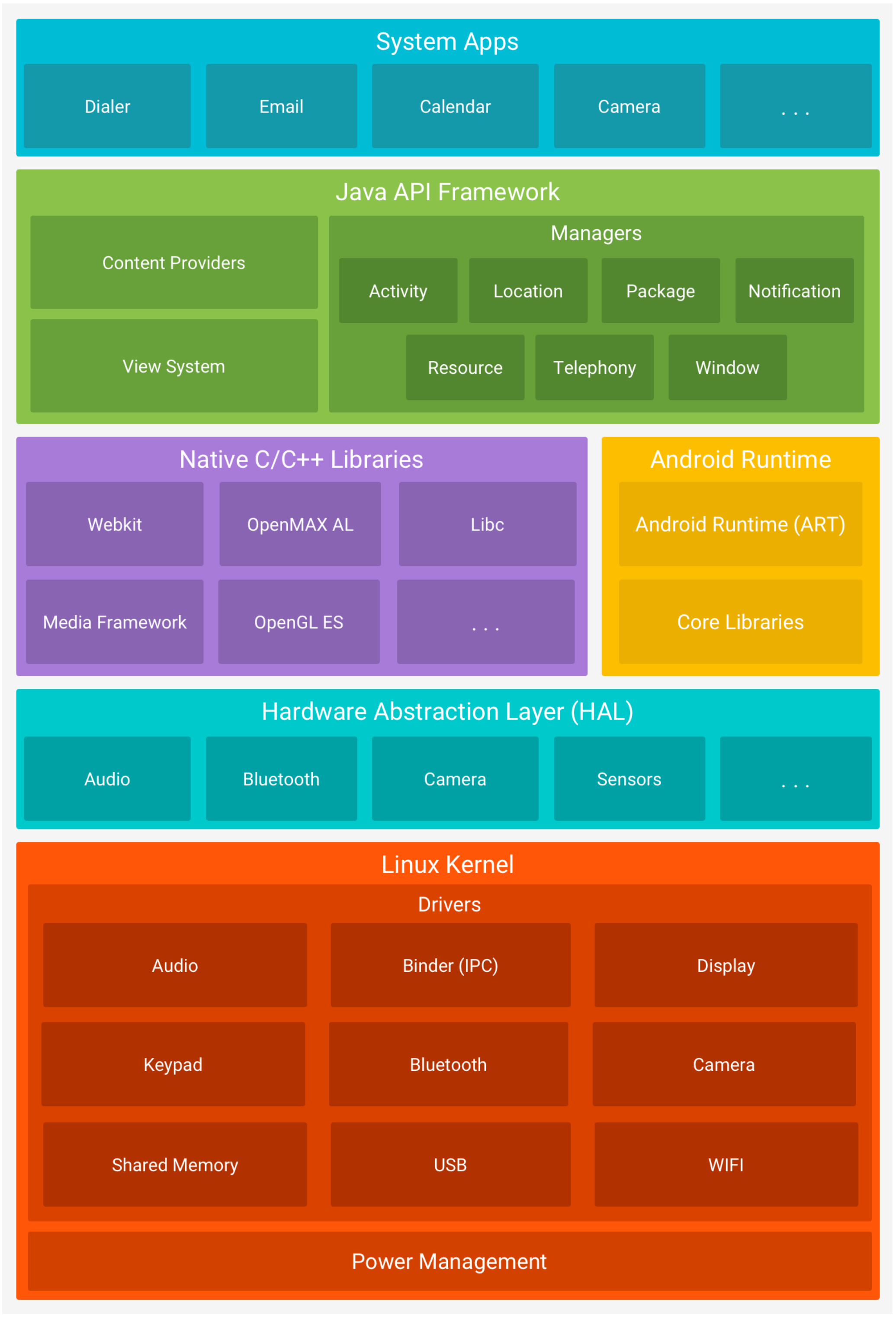

FIGURE 2.1. The architecture of Android, the diagram reproduced from Android developer official website [51]

2.2.1 Android Fragmentation

The need for customised energy estimation models and *in-vivo* optimisation is driven by the heterogeneous and fragmented nature of the Android eco-system. In order to achieve a fast product launch, Android provides devices vendors with a flexible and an open software infrastructure, however, this flexibility complicates the task of developing reliable software and affects the software performance. This problem is induced by Android heterogeneity which arises from the need to support a multitude of device models with diverse HW and software specifications. As of writing this thesis, Android OS has reached 15 different releases with 10 major versions [142]. Moreover, in 2015 the estimated number of distinct device models in use was 24,000 devices [148], and a recent report indicated that the number of different device models increases by 20% annually [140]. Android practitioners and end users have to cope with this very high level of diversity.

The problem of Android fragmentation and heterogeneity has been studied from different angles in literature. Wei et al. reported that 50% of the fragmentation related issues are due to problematic implementations of various HW drivers [165]. Device vendors are responsible to equip their devices with proper drivers through implementing the Android HAL interfaces. Although this improves the portability of Android, it causes compatibility issues. This is because different drivers on various device models can make applications behave differently. For example, some devices such as Samsung Galaxy S5 and Motorola Defy have different implementations of the proximity sensor drivers. The sensor can measure small distances between the device and its surrounding objects such as user's cheek. In those device models, the maximum distance was not properly set which caused undesirable app behaviours [116]

Besides problematic implementations of HW drivers, device manufacturers apply special customisations to Android OS which exacerbates the fragmentation problem. Such customisation ranges from software skins (mentioned above) to new features in the system. Wei et al. [165] categorised system feature related customisation into three types. Functionality modifications in which vendors modify the Android OS in order to support the new customisation. In addition, they often add new features that require modifications to the system which are called functionality augmentation. For example, the split-screen feature to view different apps was first introduced by Samsung in Galaxy Note 2. The third type of modification is functionality removal where device manufacturers may remove some parts of the system.

Such modifications add much effort on developing reliable software and sometimes lead to compatibility issues [103, 165], performance issues [102] and security vulnerabilities [172].

For instance, a range of HTC devices could expose 802.1X Wi-Fi credentials to any application with WI-Fi permissions [36]. This can be achieved simply by using the `.toString()` method of the `WifiConfiguration` class [75]. In addition, a recent bug was revealed in some devices such Samsung and Google Pixel 4 XL when using a specific picture as a wallpaper. The problem resided in the code responsible for converting pixel colours from the RGB space to the sRGB spaces [130]. The bug crashes the affected devices and causes them to enter a boot-loop. It is worth mentioning that some old devices did not suffer from the bug which suggests that they do not support the conversion from RGB to sRGB spaces.

Fragmentation and heterogeneity in Android eco-system can also delay updates and upgrades and can also lead to reliability issues. After Google releases a new version of Android or an update patch, device vendors have to add HW support to the new release and customise it to their needs in order to provide end users with a reliable customised OS. Han et al. analysed the bug reports of HTC and Motorola devices running Android 2 [54]. They reported how the problem of fragmentation manifests when the OS is updated through compatibility and portability issues in those distinct product-lines. For example, 43 bug reports associated with upgrading Android 2.1 to Android 2.2 on HTC Dream. In addition, Pathak et al. [120] found that 20% of overall energy-related bugs in Android occur after an OS update in several device models such as Samsung Galaxy S and Nexus One.

In addition, the evolution of Android APIs can indirectly degrade the quality of Android apps. Currently, there are 29 API levels in Android universe and it is increasing [1, 108]. Android API fragmentation arises due to the changes in the underlying API by different service providers and vendors. The work by McDonnell et al. [108] observed a strong correlation between Google API updates and bug-fix occurrence in open source applications. These bugs occurred due to new updates that include deprecating or modifying existing API, or introducing new ones. For example, applications may invoke new methods that do not exist on un-updated devices [165]. Issues caused by API fragmentation can also have severe impacts on users ratings [100].

We also observed that some device's vendors do not comply with standard Android APIs. For example, although Android APIs provide the `BatteryManager` class to obtain battery stats such as remaining capacity, it is not accessible on Motorola Nexus 6 device through the designated APIs. To retrieve these statistics we read them directly from a system file that is updated by the battery fuel gauge. This issue is also found on other devices[2]. In this thesis, we demonstrate how the fragmentation and heterogeneity of Android eco-system imposes challenges to the optimisation of the energy use of software as different platforms (combinations of HW and OS versions) can exhibit different energy usage behaviour. These challenges motivate the use of *in-vivo* optimisation as a way to modify applications for their particular environment.

2.3 Optimisation Using Meta-heuristic Techniques

Optimisation is a process of finding a set of solutions that minimises or maximises an objective function of a problem. The problem can have one or several objective functions. For a given problem, an objective function can have local optimum values as well as a global optimum value. When there are multiple objective functions, conflicts may occur between them. For example, one may want to find a set of software tests that increases the test coverage and decreases the total test execution time.In this thesis, we optimise software by finding new solutions that minimise its energy consumption and also preserve the quality of its outputs to some extent (i.e. minimise the error in the software output).

Meta-heuristic algorithms are often used to search for optimised solutions. Such techniques can be categorised into trajectory-based and population-based techniques. Trajectory-based techniques include Hill Climbing (HC) [44], Simulated Annealing (SA) [83] and Tabu Search (TS) [49] algorithms. Population-based techniques include, evolutionary strategies (ES) [139], particle swarm optimisation (PSO) [43], differential evolution (DE) [155] and genetic algorithm (GA) [65].

In this thesis, we use Evolutionary Algorithms (EAs) to find fitter software variants in terms of energy consumption. EAs are a global search technique and are members of the family of bio-inspired algorithms. EAs mimic fundamental aspects of Darwin's evolutionary theory incorporating variation and selection. We favour EAs because they are *anytime* algorithms that return at least one solution when interrupted. This is a key feature since we conduct our experiments on smart-devices that run on restricted battery resources. In addition, EAs can be considered lightweight frameworks that do not require intensive resources to run. This is crucial as smart devices have restricted resources unlike personal computers and servers. Furthermore, they do not require detailed domain knowledge in order to find promising solutions.

[2]Examples can be found in https://stackoverflow.com/questions/27998034/total-battery-capacity-in-mah-of-device-programmatically.

EAs explore the search space through random perturbations and create a set of candidate solutions which is called a population. Individuals within a population are evolved using a combination of crossover and mutation operators to create a new generation. Crossover is a process of mating two parent solutions to produce new offspring by mixing the parents genetic materials. A mutation occurs on individual solutions by altering their genetic materials at random introducing new genetic materials. Individuals for the new generation are chosen using a survival selection mechanism. It normally prefers elite individuals (i.e. fitter), nonetheless, selection does not prevent the survival of unfit solutions. Such preservation diversifies the new generation.

2.4 Search-Based Software Engineering

In software engineering field there are often tedious and potentially expensive tasks involved in the development of reliable and efficient software. As a result, software practitioners try to automate such tasks. Software automation can significantly reduce the development and cost of software since it would save human effort and resources.

In Search-Based Software Engineering (SBSE), traditional SE problems are solved by optimisation algorithms. Such problems involve a trade-off between conflicting and competing objectives. In addition, SBSE problems have a very large solution space that traditional SE methodologies cannot effectively explore. An example of such problems is regression testing which is the process of retesting a system after applying some modifications to it to gain confidence that these recent changes have not interfered with its functionalities [175]. A software practitioner (Sam) may try to re-run all existing test cases to ensure the system behaves as it is intended to, however, as the system evolves the number of test cases grows and it might get to a point where executing the whole test suite after every modification becomes prohibitively expensive and infeasible. Therefore, Sam wants to be efficient in finding the smallest set of test cases that increases the line of code (LOC) coverage and/or decreases the test execution time. This problem is known as test suite minimisation problem in the SBSE testing literature [175].

In order to solve SE problems using SBSE techniques, the problems have to be reformulated into search problems. The main difference between the candidate solutions in a solution space is the extent to which they can solve the problem at hand (i.e. their quality). The search process is therefore guided by a fitness function by which each solution is evaluated. Solutions are created using different techniques depending on the search methodology used to solve the problem. Getting back to our example, Sam can use either or both of the LOC coverage and execution time as fitness functions. In addition, all possible subsets of the main test suite form the search space. Sam now needs a software that checks the fitness values for each subset (brute force), keeps picking randomly till satisfaction (random search), picks randomly and then applies some changes and keeps the best found subset (HC), or uses some meta-heuristic search algorithms such as GA that explores the space of test subsets until an optimised solution is found.

It is noteworthy that SBSE methods have been shown to perform similarly to humans and sometimes even better. For example, the work of de Souza et al. [37] reviewed several SBSE works on the next release problem, work-group formation problem and test case selection problems. They then compared the solutions of those works with results obtained by professional programmers and senior students. They found that machine-generated solutions are generally more consistent. In addition, Weimer et al. [166] and Le Goues et al. [92] developed tools for fixing software bugs that received the gold and bronze Hummies awards in 2009 and 2012, respectively. The Hummies competition is for evolutionary computation techniques that produce human-competitive results. Furthermore, Barr et al. [7] received the gold Hummies award in 2016 for transplanting software features from one donor software into another host software codebase. These award-wining works are based on a technique used in SBSE called genetic improvement of software.

2.5 Genetic Improvement of Software

One of the SBSE techniques that has been growing significantly recently is Genetic Improvement (GI) of software, which utilises meta-heuristic search algorithms to automatically improve existing software [125]. The improvement can be on either or both of its functional and non-functional properties. One of the earliest example of GI is the work carried out by Walsh and Ryan [164] which optimised the run-time of existing sequential software by converting them automatically into parallel software.

2.5.1 Software Functional Properties

GI has been used to fix buggy functionalities [92, 132, 133, 166, 167]. One of the most famous GI tools is called GenProg, developed by Weimer et al. [166]. It utilised unit tests to locate bugs in software, and to verify program correctness after applying genetic operators to fix the bug. It modifies the target software by deleting a statement, inserting a statement, or swapping between statements. This shows that the source code of software is often contains the required materials to fix its covered bugs [47]. GenProg was evaluated on eight real-world applications containing 105 bugs in total. It successfully created bug patches for 55 bugs [92]. Qi et al. [133] demonstrated through their Kali tool that only the delete operator is sufficient to generate at least as many bug patches as those produced by [91, 132, 166, 167].

The above bug-fixing techniques essentially require a sufficient test suite to be ready by developers in order to evolve software patches that fix the found bugs. On the other hand, Haraldsson et al. [59] capture real user input data that causes a failure in a live rehabilitation system during the day. Then at nights, it creates test cases using the recorded inputs and adds them in a test suite if they successfully recreate the same failure the system exhibited during the day. GI then runs on the system and modifies it using similar operators to GenProg until all test cases pass, or the next working day starts. The approach generates a report with the new edits for the developer team of the system for verification and implementation purposes. It successfully resolved 22 bugs over a 6 month of trial.

In addition, Kim et al. [82] developed a patch generation approach that utilises fix patterns learned from developer-written patches (i.e. fix templates). Their approach localises the faulty code segments and the code around it, and analyses its Abstract Syntax Tree (AST). GP then is used to create patches using the templates to fix the code. Kim et al. evaluated their approach PAR on six case studies containing 119 bugs in total and compared it with GenProg. PAR fixed 27 bugs whereas GenProg resolved only 16 bugs. Tan et al. [157] proposed Droix framework for fixing bugs in Android applications. The authors studied a set of Android crash reports from Github repositories and identified 8 operators that are frequently used by Android developers to fix bugs. Droix uses the Android Debugging Bridge (ADB) and Monkeyrunner framework to construct GUI test scripts. The framework then utilises these scripts to uncover bugs within the app and to check its correctness after applying the patches. Additionally, the Droix framework uses the stack trace information to localise the bug location. Droix implements $(\mu + \lambda)$ evolutionary algorithm to evolve new variants of the buggy app, by applying one of the eight operators in the faulty location within the app.

Besides improving software functionalities by fixing bugs, GI can be used for adding new functionality to an existing software. Barr et al. developed a framework that successfully auto-transplanted new functionalities into host software which were taken from donor software [7]. The framework uses program analysis to identify an organ (code segment) in the donor source code whose functionality is a desirable feature to transplant into another host software. It then uses genetic programming GP to reduce the selected organ by implementing observational program slicing [10], and to transform the organ to suit the host software. It also uses unit testing to validate that the organ exhibits the required behaviours before and after transplantation. The technique was evaluated by transplanting features from five donor software into three hosts, and four features were successfully transplanted. One of these operations was to transplant a new encoding format (H.264) to the VLC media player.

Harman et al. also developed a new technique to evolve a new functionality in existing software. The technique is known as grow and graft. It grows the desired functionality and then grafts it into the target software [61]. The technique requires developers to suggest some useful libraries and APIs to evolve the new functionality. Then using GP, it grows the new code segments required to implement the desired functionality in isolation. The grafting phase, which is also based on GP, involves finding one or more valid insertion points and determining how the grown code can be inserted in the target software. The technique was successfully used to grow new features in two applications. It was used to add translating text from English to Korean and Portuguese feature into Pidgin software.

2.5.2 Software Non-Functional Properties

The optimisation of software run-time has caught the attention of the GI community more than the other non-functional properties. This is due to the ease of measuring software run-time compared to other non-functional properties such as memory use and energy consumption. Some works improved the target software by converting some part of them from sequential to parallel computing paradigm [84, 85, 87, 164] while others opted to generate the improved software variants without such conversion [23, 34, 124, 171]. In general, these works create new evolved variant by converting the target code into BNF grammar [84, 85, 87, 124], by syntax tree [34, 164], or by exposing hidden parameters within the code such as constants [23, 171]. Evaluating the new variants is done by running regression test suites.

GI has been also used to enhance an existing trade-off between execution time and output accuracy in software. Langdon and Harman [86] used in their experiments a DNA sequencing system called Bowtie2, which trades-off accuracy to improve the run-time since DNA sequencing is a time-consuming task for large data-sets. They represented the target software using BNF grammar and then used GP to create new software variants. Prior to the optimisation process, their technique runs a sensitivity analysis on the target software to identify the most-used parts of the system. This helps identifying resource hungry code. In addition, targeting only those parts reduces the search space of the evolutionary process. Their approach managed to evolve a new system variant that is 70 times faster than the original system. In addition, it was tested using a large data-set of DNA sequencing. It improved the results in 9% of the cases and reported identical results on the rest. It is worth mentioning that this work did not bring the accuracy of the results into consideration, yet accuracy was preserved and improved. This shows that GI can obtain promising results in both objective dimensions.

In contrast, other work used GI to trade-off between the run-time and the results of the target software. Sidiroglou-Douskos et al. [145] used a multi-objective approach to minimise the run-time of a shader software by allowing some degradation in the quality of the output graphics. Their approach used GP to modify the code of the shader application which was represented as an AST. They managed to create a variant that produces graphics with nearly unnoticeable errors and reduces the execution time by roughly 77%. Similarly, Bruce et al. [23] explored the trade-off between the execution time and the functionality of face detection in the Viola-Jones algorithm in the computer vision library OpenCV. To create new software variants, they exposed and tuned integer constants in the code. Their work achieved up to 48% of run-time improvement while reducing the accuracy of face detection by about 2%.

Wu et al. [171] explored the trade-off between software memory consumption and run-time using GI. They developed a tool that exposes hidden parameters (e.g constant, arithmetic and relational operators) in the code and tune them to find efficient software variants. The tool applies mutation operators to automatically create software variants. It also uses a regression test suite to evaluate the created new solutions. The authors tested their approach using the memory management library dlmalloc for C programs and four subjects to demonstrate their approach. The best results obtained was 12% reduction in the execution time and 20% reduction in the memory consumption. They observed that the default configurations of dlmalloc are not optimal for the four case studies. In addition, their results shows that decreasing the run-time can be achieved at the cost of increasing the memory consumption.

We cover the related work that uses GI to optimise the energy consumption of software in the upcoming chapters. It is worth mentioning that no related work has carried out *in-vivo* optimisation on smart devices.

2.6 Summary

In summary, the smart devices eco-system is diverse and the Android eco-system in particular suffers from fragmentation and heterogeneity in all layers of its architecture. Such issues necessitate tuning applications to improve their energy consumption per platform, which is impractical to do manually. To achieve good performance for each platform it is essential to utilise an automated approach to solve these issues. This can be achieved by the use of the GI which is an SBSE technique that automatically creates new customised software variants out of an existing software product. The new variants can improve the energy consumption, memory use, run-time, or the functionality of the original software.

Chapter 3

Deep Parameter Optimisation on Android Smartphones for Energy Minimisation – A Tale of Woe and a Proof-of-Concept

“I have not failed. I’ve just found 10,000 ways that won’t work”

Thomas Edison

3.1 Introduction

In this chapter, we report on the solutions for the technical challenges that arise when setting up a test-bed for *in-vivo* energy optimisation experiments. This chapter has been published in the third edition of the International Workshop on Genetic Improvement at GECCO in 2017 [14].

In recent years there has been growing interest in improving software systems’ energy efficiency. This is partially due to environmental concerns (the majority of the world’s electricity consumption is still derived from polluting fossil fuels [2]) but also usability issues present in battery-constrained mobile devices such as smartphones. The sale of smartphone devices has exceeded that of personal computers [27] with the average user spending 30 hours a month on mobile applications [160]. These applications consume the smartphone’s limited energy reserves which can leave users with a drained battery at inopportune moments. A survey of mobile application complaints found that having resource intensive features had a larger negative impact on an application's rating than uninteresting content or a poorly designed interface [81].

It is known that mobile applications may be refactored to consume less energy and, thereby, increase battery life [111, 128]. However, a study by Pang et al. [118] showed that developers typically lack the necessary knowledge to make software more energy efficient. While education of developers may be a solution to this problem, a more cost-effective approach is to automatically refactor software to a more energy efficient state, entirely removing the need for developer intervention. Previous studies have shown that this is possible when given reasonable assumptions about an application's end-use such as the likely input data [22] or network usage [94].

Small improvements in energy efficiency are achievable without changing the functionality of a mobile application. Semantics preserving changes to design pattern implementation [115] and the resolution of energy bugs (instances when smartphones are unnecessarily left in high energy states) [6] have both been shown effective at reducing energy consumption. There are, however, limits to how much energy can be saved without sacrificing functionality [24]. Fortunately the majority (80%) of software engineers who work in energy-constrained systems are willing to sacrifice some requirements for reduced energy consumption [105]. Allowing a degradation in quality to fulfil non-functional requirements is part of an emerging field known as *approximate computing* [55]. Examples of approximate computing include Li et al.'s attempt to reduce the energy consumption of Android applications by decreasing the quality of visual interfaces [96] and Sitthi et al.'s work on reducing shader execution time by permitting faults in the graphics they produce [151].

In this chapter we present the challenges encountered, and solved, in performing automatic optimisation of energy consumption by a software library for mobile devices. The target software library, *Rebound*, is a Java physics library that models spring dynamics. It is used by popular Android applications like Evernote, Slingshot, LinkedIn and Facebook. During our optimisation, we allow for an approximation of the intended output, with the goal of finding a set of configurations that represents trade-offs between energy consumption and faithfulness to physically correct animations.

This chapter is structured as follows. We introduce the target software of this study in Section 3.2, and describe our approach to optimising it by means of deep parameter optimisation in Section 3.3. We present our used target hardware in Section 3.4. Strongly related to this is our subsequent tale of woe in Section 3.5, which reports on some of rabbit-holes that we went down when using a modern smartphone with a modern operating system

(Android 6) as a testbed. A proof-of-concept evolutionary run is shown in Section 3.6, before we conclude this article with a summary in Section 3.7.

3.2 Target Software: Java physics library Rebound

In this section, we first lay out our requirements that target applications need to satisfy for our subsequent optimisation. Then, we introduce our chosen application, characterise its test cases and define how we measure the impact of optimisation on the application's behaviour.

Our requirements for target applications are as follows:

- *R1*: open-source, so that we can modify them;
- *R2*: widely used, for maximum impact;
- *R3*: contain at least one energy hog, for potential room for improvement;
- *R4*: compilable in under a minute on a regular computer;
- *R5*: provide tests that allow for gradual deviations from the targets.

Interestingly, many open source applications do not satisfy requirement *R5*, as tests tend to focus on functional property checks such as data extraction from files, listening to events, application of ciphers, user interface tests, and so on. Additionally, two noteworthy groups of applications are internet browsers and media players, however, they are not considered here as their compilation requires many resources and some of the media decoders are implemented in hardware, which means testing on different platforms is not straightforward.

Following a comprehensive search for applications that satisfy all requirements, we use Rebound[1] in this study. Rebound is a Java library that models spring dynamics. The spring models in Rebound can be used to create animations that feel natural by introducing real world physics to applications. For example, in complex components like pagers, toggles, and scrollers. Major apps that use Rebound include Evernote, Slingshot, LinkedIn, and Facebook Home.

[1]Rebound Spring Animations for Android: `http://facebook.github.io/rebound/`, accessed 19 Feb 2020

The target for our optimisation is the `Spring` class in the `com.facebook.rebound` package [2]. This class implements a classical spring model using Hooke's law with configurable friction and tension. Inside this class, the `advance` function is responsible for the physics simulation based on `SOLVER_TIMESTEP_SEC` sized chunks. The computations include, among others, Euler integrations and calculations of derivatives. Interestingly, some level of performance optimisation has already been done, as evidenced by the source comment "The math is inlined inside the loop since it made a huge performance impact when there are several springs being advanced."

Rebound comes with 44 test cases. These tests vary significantly in nature. For example, some tests check if the ID of a spring is set correctly, and if listeners work as intended. Most importantly for us are the tests that perform the actual physics calculations. These are (i) relatively time consuming and (ii) deviations from the exact results may be acceptable if energy consumption is decreased as a result of a configuration change.

In general, for a set of n tests with test oracles $T_1, \ldots, T_n$ and corresponding observed outputs $O_1, \ldots, O_n$, we measure the quality in three ways:

1. *M1*: How many tests are passed, as determined by `assertEquals(`$\mathtt{T}_i$`,`$\mathtt{O}_i$`)`, $1 \leq i \leq n$?

2. *M2*: If an array is to be produced, what is the average per-element deviation? Variations here can result in unrealistic looking animations.

3. *M3*: If an array is to be produced, what is the average array length deviation? Variations here can result in too long/short animations.

Note that if for test i the output arrays of T_i and O_i differ in length, then *M2* considers only the first $\min(|T_i|, |O_i|)$ fields.

Interestingly, the original test cases do not result in a *M2* quality of 0, but in a tiny non-zero value. This is due to tests not resulting in exactly the spring speed and position values provided in the test oracle. To address this, we adjust the test oracles based on the actual output of the code on the device.

Under the above testing regime the original code has the following outcome:

1. *M1*: all 44 tests are passed;

2. *M2*: the average deviation from the values provided by the oracle is zero;

3. *M3*: the average deviation from the oracle's array lengths is zero.

[2]The selection criteria is discussed in Section 3.3.

3.3 Deep Parameter Optimisation

Deep parameter optimisation [23, 171] is a genetic improvement technique [125] where the variation operations to be applied to a target application are done so by toggling deep parameters found within its source code.

What constitutes a 'deep parameter' is anything within code, not previously exposed to the user, which may exist in multiple known forms while preserving some fundamental functionalities. For example, in Java, when a developer wishes to use a collection they must declare which subclass of the abstract superclass `java.util.Collection` to implement (e.g. `java.util.ArrayList`, `java.util.HashSet`, etc.). Each of these subclasses consume different amounts of energy depending on how they are used [104, 123]. In the vast majority of cases the developer will choose an implementation based on his own intuition or preferences; utilising little information on how this may affect the software system's performance. In essence, the developer hard-codes these parameters. In deep parameter optimisation we expose these parameters to be toggled and then optimised using a search-based approach [60].

Within this investigation (as in previous investigations using deep parameter optimisation to reduce energy consumption [23]) we expose integer and double constants. To do so we start by replacing integer or double constants with placeholders. These placeholders are calls to read that placeholder's value from a configuration file. In most genetic improvement research, modifications to the source-code require recompilation before evaluation. This can be costly – within this investigation, recompilation carries a penalty of 20-30 seconds. With this setup the configuration file, which we may conceptualise as the exposed parameters, can be modified any number of times without recompilation. The configuration file is read once per execution and thus incurs a fixed energy overhead though, since this is constant across any and all evaluations this does not effect the impact of our results.

While one could begin tuning the parameters at this stage, it would be inefficient. Previous research on deep parameter optimisation has shown that the majority of exposed parameters are not worth optimising [23, 171]. We categorise deep parameters as falling within three categories – those too insensitive (large changes have no effect on the target property/properties), those too sensitive (small changes break hard constraints), and those worth optimising. In this investigation we start by profiling our target application, Rebound, and selecting only those files which consume large amounts of energy for optimisation. In our case we find that most calculations are performed in just one Java class, `Spring`. For example, the previously mentioned `advance` method, which performs the physics calculations, is the second-most called method (9406 times).[3] The most frequently called method is `isAtRest` (20340 times, also in `Spring`), which performs a rather simple check. All other methods are computationally uninteresting, and only called a few dozen times, if at all. We therefore choose to target this class exclusively as the remainder are unlikely to contain parameters that are worth optimising (i.e. they are too insensitive).

Before exposing the parameters within `Spring` we replace all instances of `{variable}++` with `{variable}+=1` and all instances of `{variable}--` with `{variable}-=1` as this improves the search-space by giving us more parameters to target for optimisation. Once this is done we expose 38 parameters from `Spring`. We then, for each parameter, increment its value by 1 if it is an integer or by 10% if a double (all doubles are non-zero). If this results in the program crashing when run we tag this parameter as unmodifiable as it is too sensitive. Otherwise we multiply its value by 10 (after the incrementation). If this results in a program in which energy consumption exists within the 95% confidence interval of the unmodified application's energy consumption (determined by running the Rebound 100 times, measuring its energy each time) we tag the parameter as unmodifiable as it is too insensitive. After this has been done for each parameter we are left with a set of parameters which we have not tagged as unmodifiable at any stage. These are the target parameters for deep parameter optimisation. In the case of `Spring`, we are left with 19 of the original 38 parameters.

With these parameters we may toggle them, thereby altering the software. These alterations may reduce loop iterations [122], or disable certain costly branches [171] to reduce energy consumption. Given we permit output approximation, it is likely trade-offs can be found to reduce energy consumption at the expense of output quality [24]. Though, at this abstract level, the problem representation is simply an n-tuple of numbers which may be assigned a fitness value for each objective producible for any given variant. Given this, it is an ideal candidate for optimisation using a genetic algorithm [153]. As we wish to optimise for multiple objectives (see Section 3.2) we utilise NSGA-II [38], a genetic algorithm designed for multi-objective search, as implemented by the MOEA Framework.[4]

We limit any parameter to have a minimum value of 0 and a maximum value of 64 (the parameters, when exposed, have initial values between 0 and 6). With a population size of $\mu = 20$, we seed the initial generation with the original solution (i.e. the parameters as exposed from the initial, unmodified application) and those close to the initial solution in

[3]Determined by Corbertura 2.1.1, available at `http://cobertura.github.io/cobertura/,accessed19Feb2020`. The total class/line/branch coverage is 40%/61%/61%.

[4]MOEA Framework version 2.9 available at `http://moeaframework.org`, accessed 19 Feb 2020. We leave all variation operators and variation probabilities at their standard values.

the search space by iterating through the parameters and generating variants equal to the original but with one variable being incremented by 1 in order to introduce some initial diversity around the official parameter choice.

3.4 Target Hardware: Android Smartphones

3.4.1 Hardware Platform

Modern mobile phones are equipped with battery fuel gauge chips that report the voltage, current and remaining energy within the battery [4]. The target devices for our experiments are the HTC Nexus 9 and the Motorola Nexus 6. Both are equipped with the Maxim MAX17050 fuel gauge chip that compensates measurements for temperature, battery age and load [72].

For the optimisation of energy consumption, we solely rely on the energy readings as provided by the battery chip. This is in contrast to some existing work (e.g., [63]) which relies on external meters. A brief practical characterisation of the internal battery meters on the Nexus 6 and Nexus 9 is included in [15]. There, we outline results for validating the precision of the internal meters under various workloads. The internal meters are deemed sufficiently precise if the experiments are long. Also, based on our experience with external meters, their setup can come with unexpected electronic challenges. For example, we observed voltage drops and system crashes due to cheap alligator clips and corroding copper strips.

3.4.2 Software Framework

Our software framework includes a data logger, hardware component controller and battery monitor. The data logger samples hardware settings and utilisation data such as CPU frequency and load, screen brightness and network traffic. The controller's main job is to create test scenarios. It activates, deactivates and applies workloads on hardware components. For example, while profiling the screen, it changes and fixes its brightness, as well as it turns off other components and fixes the CPU frequency.

The battery monitor records the power consumption data such as the remaining energy, voltage and drawn current during each test session. Accessing the battery chip's values can be done through the battery API, such as Android's BatteryManager class. This API broadcasts these values with a frequency of 4Hz.

3.5 Getting the Experimental Setup Right - A Tale of Woe

Here, we report the various encountered challenges. The discussion is divided into expected challenges and unexpected challenges. We report on this because genetic improvement of energy usage is only as reliable as the measurements [58].

3.5.1 Expected Challenges

System Behaviour

It is important to note that both considered devices are complex with many communication interfaces, controller chips, and multiple CPU cores, where much of the device behaviour is controlled by the operating system. The operating system, Android 6.0.1, is complex, with system and user processes running in parallel with elaborate power consumption management in place.

To minimise the noise from these complex systems and to maximise the relative strength of the energy signal from our experiments it is important to reduce the energy footprint produced by background processes. To this end we deactivated communication interfaces using the flight mode. This prevents processes from transmitting data, which has proven to substantially impact energy consumption, even when occurring in short bursts [13].

We also put the display in sleep mode, which reduces the power drawn for the display light and GPU. Turning the screen off allows the system to enter the so-called Doze-mode [5], which was introduced in Android 6, and which deactivates a number of system services that would otherwise inject noise into the energy signature of the experiments.

Another potential source of noise is the system dynamically adjusting CPU speed according the current workload. To avoid this issue we fix the speed of all cores to the same value.

Sampling-Frequency-Induced Error

Some sampling error is induced by the fact that the battery fuel gauge can be sampled with a maximum frequency of 250ms. This means that some noise is introduced by gaps between the start and finishing time of the measured process and the time the battery is sampled. We minimise the impact of this low sampling frequency by running each individual 20 consecutive times and recording the fuel-gauge samples only before and after these runs. The use of 20 consecutive runs also serves to increase the measured magnitude of energy use which further reduces the impact of random noise in the fuel-gauge. Finally the large number of runs also smooths out random variations in runs caused by sporadic drains on energy caused by system processes.

Note that the expected causes of measurement error above can be accounted for by reasonably standard approaches to sampling and controlling the runtime environment. Next we describe some unexpected challenges which require interventions which are specific to this domain.

3.5.2 Unexpected Challenges

System Behaviour

There are a number of unanticipated challenges presented by system behaviour. One unexpected interaction stemming from having the display in sleep mode is that the system will go into Doze-mode after the experiment starts. Doze-mode impacts the experiment by suspending and rescheduling background processes including those we use to log data and run the test suite of Rebound library. As a consequence, the test execution time increases drastically from seconds to hours in some cases. While the existence of Doze-Mode has benefits, it can be problematic in settings such as these experiments. To counter this problem we use partial wakelocks which prevents the system from suspending our processes. In addition, temporarily activating the display after each generation of the evolutionary process allows our framework to run Rebound's test cases normally. Needless to say, while the screen is on, the test execution is suspended.

Android Debug Bridge

Android Debug Bridge (ADB) is a command-line tool by which developers can communicate with Android devices [149]. It supports a variety of actions such as copying files to and from the device and installing/un-installing applications. ADB consists of a client to initiate commands from the development machine, a daemon to run execute command on the Android device, and a server to manage the communication between the client and the daemon.

In our framework, ADB is used to install code on target devices, start an experiment and retrieve the results. In early experiments the Rebound library was compiled, transferred to, and installed on the device for every change in its parameters. Unfortunately, these operations took up to a minute to complete, making iterative search impractically slow. Moreover, app transfer and installation could fail due to ADB server instability. Failure modes included the connected device going off-line and interference with the communication port by other Android services.

To reduce deployment time the Rebound library was modified to read its code parameters from a configuration file - thus avoiding the need to compile and transfer the application. To address the ADB connectivity issue a programmable USB hub was configured to automatically drop and restore the link to the device whenever the device found to be offline. The framework is also configured to poll for devices using `adb devices` and to restart the ADB server in the case of interference on the communication port.

Temperature

Temperature variations during experimental runs produced an unexpected source of systematic noise in our experiments. In preliminary testing we observed that the battery temperature increased after many successive runs of Rebound. This temperature increase was observed to increase the fuel consumption of the same program when run repeatedly over time. Figure 3.1 shows how the rise of battery temperature correlates with increasing energy consumption on the same program run 1000 consecutive times (100 trials of ten Rebound runs each) on the Nexus 6 device.

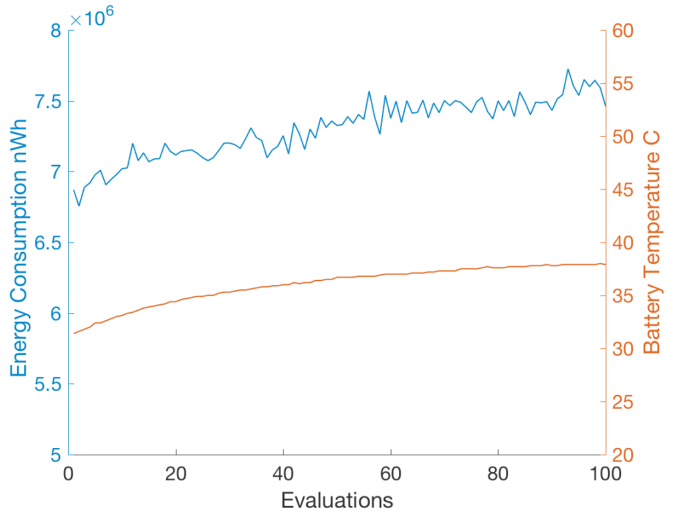

FIGURE 3.1. Battery temperature (orange) and average energy consumption (blue) over 100 consecutive trials of 10 runs each on the same benchmark.

This trend of increasing energy consumption for the same code over time is extremely problematic in the context of an evolutionary run using NSGA-II. This is because the very first variant of the program will be on the Pareto frontier in the dimension of energy consumption. If the energy consumption increases as a function of temperature, and the temperature increases as a function of the number of evaluations, then it becomes progressively more difficult for genuinely improved individuals to dominate the starting program variant.

To see if it was possible to learn the relationship between temperature and power consumption – and therefore compensate for it – we collected energy consumption for the same Rebound benchmark in varying temperatures.[5] Figure 3.2 shows a scatter plot relating temperature to energy readings from the same benchmark. As can be seen, there is a general upward trend in the data.

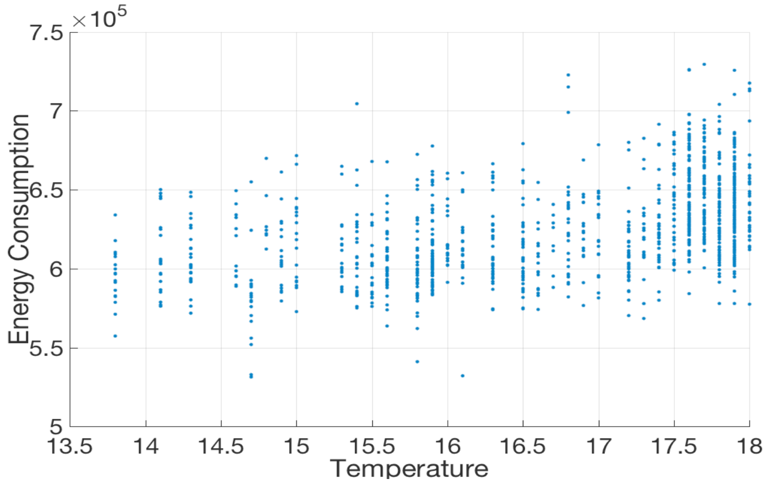

FIGURE 3.2. Scatter plot of temperature vs. energy consumption on the same benchmark.

We then ran non-linear regression on the resulting data using the GPTIPS2 [141] symbolic regression package. The learned function is: $e = 357t^2 - 6180.0t + 608000$ where e is the energy consumption of the benchmark in nWh and t is temperature in Celsius degrees. The actual-vs-predicted curves for the sorted temperature data is given in Figure 3.3.

[5]The experimental rig was placed in the refrigerator to extract some cooler readings.

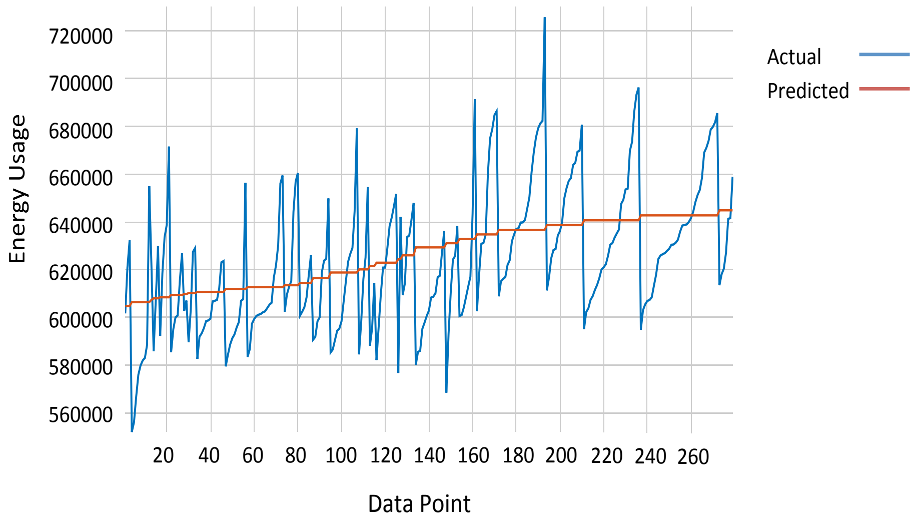

FIGURE 3.3. Actual energy consumption (blue) vs. predicted energy consumption (orange) as a function of temperature.

Note that the data is sorted by temperature from left-to-right. We observe that there is large variance of energy consumption at each temperature level. Moreover, the variance seems to get larger as temperatures rises. This noisy relationship is borne out if we sort actual versus predicted energy consumption by error level, as shown in Figure 3.4.

Figure 3.4 shows that the distribution of energy consumption is bi-modal and the shape of the predictor function appears to be influenced by this. These systematic variances seem to indicate that there may be two populations of energy sensor readings from the Nexus 6.

To avoid this problem we switched to the Nexus 9 – which exhibited less systematic variation in energy readings. However, given the difficulty in determining an accurate relationship between temperature and energy consumption, we decided instead to setup a test rig to control temperature by using a desk-fan coupled with sufficient egress for airflow around the device. This new rig is pictured in Figure 3.5.[6]

[6]For a video demonstrating an earlier setup, see https://www.youtube.com/watch?v=xeeFz2GLFdU

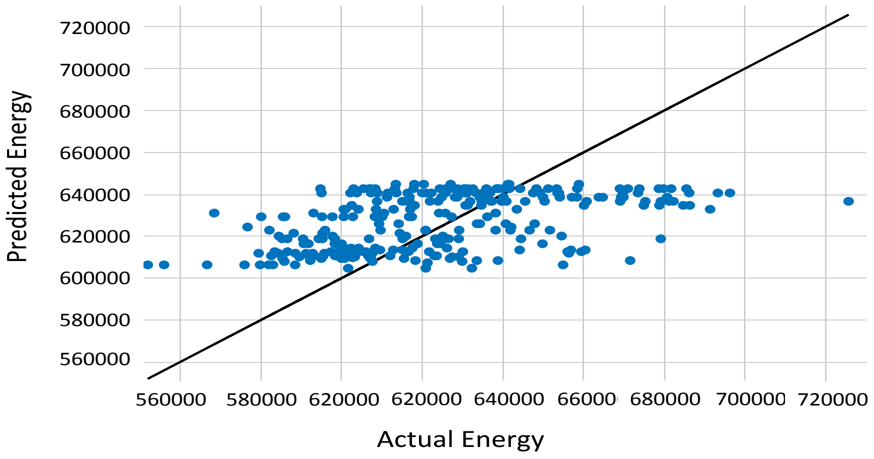

FIGURE 3.4. Fit of the regressed temperature function. As can be seen, the errors are bi-modal and the fitting function appears to be influenced by this.

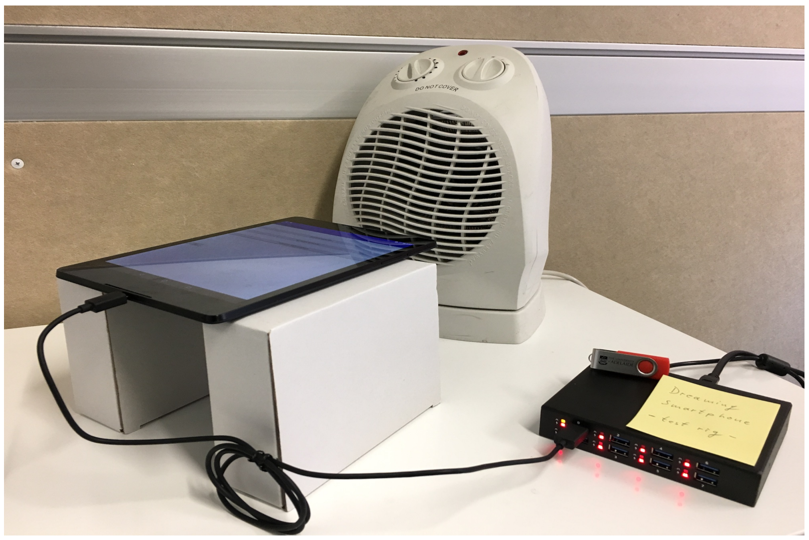

FIGURE 3.5. Test rig to limit variations in temperature in the Nexus 9 platform.

Processor Throttling

The test rig described above moderated rises in temperature but not enough to prevent a hardware CPU governor in the device from triggering reductions in processor speed. The effect of this fail-safe is shown in Figure 3.6.

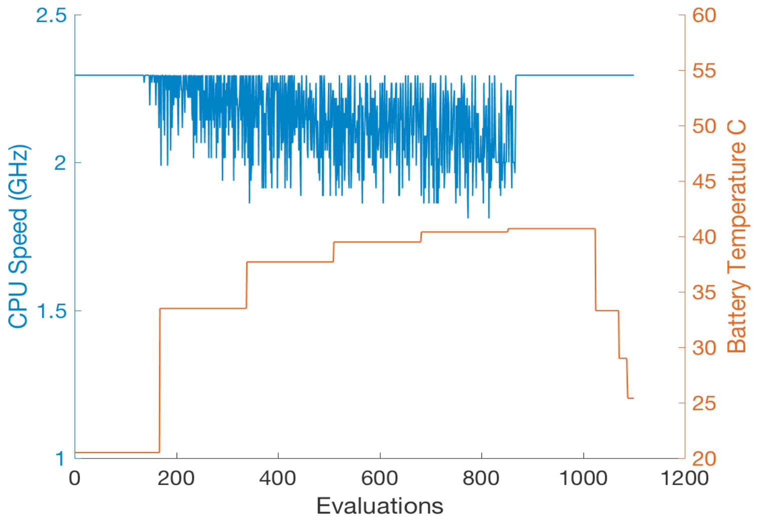

FIGURE 3.6. Battery temperature vs. processor speed over consecutive runs. As battery temperature rises the processor speed is throttled.

The figure seems to indicate that throttling is activated by steep rises in temperature rather than high absolute values in temperature. The throttling also seems to have the desired protective effect in slowing rises in temperature. This throttling has the potential to impact on the energy consumption of benchmarks. Figure 3.7 traces processor speed and energy consumption.

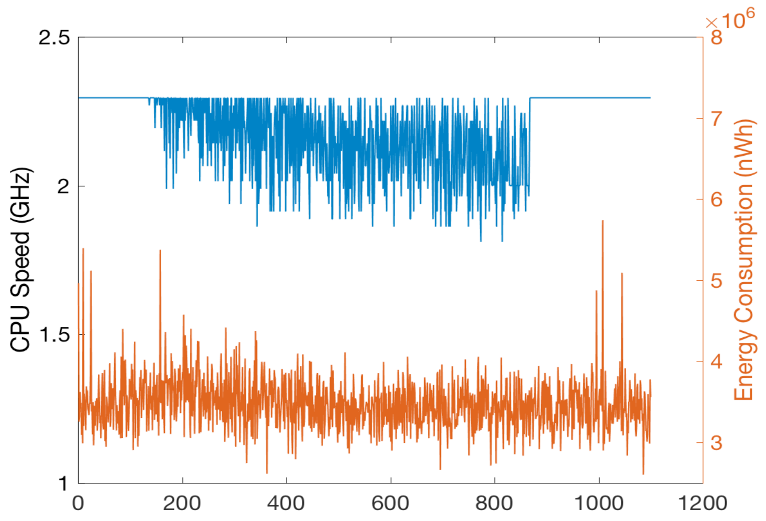

FIGURE 3.7. Processor-speed (blue) vs. energy consumption (orange) during an evolutionary run.

The lower CPU speed leads to longer execution times on the Rebound benchmark but appears to lead to slightly less energy consumption. There also seems to be less variation in execution time at lower processor speeds.

Informed by these results, we throttled processor speeds using the system governor. We found a speed of 1.428GHz allows the benchmark to run in a tolerable time-frame whilst reducing measurement variability and temperature increases.

Log Files and Memory

Further experiments with reduced processor speed revealed that per-run energy consumption still increased over multiple runs – albeit at a slower rate than before. Preliminary investigations revealed that the increased energy consumption seemed to correlate with the size of logs and the size of the memory footprint of the Java test-harness used to run each generation. We re-designed the logging procedures to reduce the memory footprint of the generational log to 500kB. This reduction removed some of the variability energy consumption but an increase in energy consumption over multiple runs persisted. For further improvement we attempted to reduce the memory footprint of the test harness by calling the Java garbage collector (GC) every 250ms. This reduced overall energy consumption level and growth. As an illustration of the impact of calling the GC Figure 3.8 shows the sorted energy consumption data for a sequence of 100 runs with (orange) and without (blue) garbage collection.

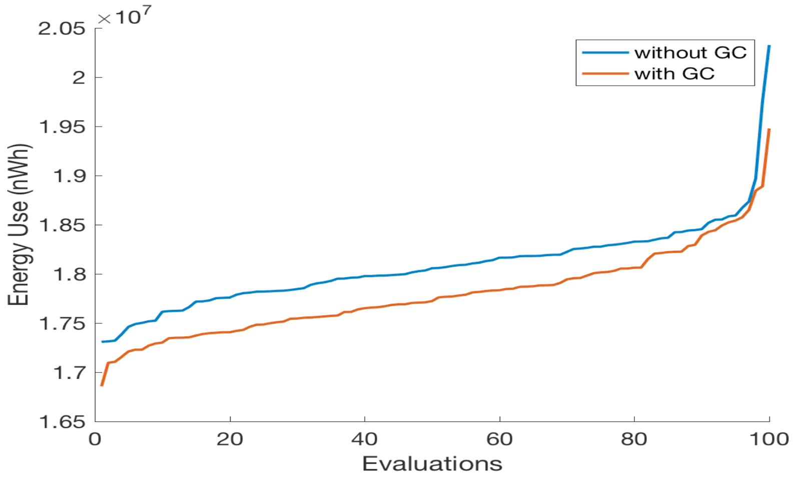

FIGURE 3.8. Sorted energy-use data for 100 runs with calls to the GC (orange) and without calls to the GC (blue).

After checking that both the with-GC and without-GC data were normally distributed (with a one-sample Kolmogorov-Smirnov test) we applied a t-test and confirmed the with-GC runs were significantly less than the without GC runs with $p \ll 0.001$. After further experiments it was determined that the same effect could be achieved by calling GC after each generation.

Informed by explorations described above, the experiments described in the next section were run with a processor speed 1.428GHz, using the physical setup shown in Figure 3.5.

3.6 Proof-of-Concept

In the following, we report on our actual experiments performing deep parameter optimisation of a Java physics library as described in the previous sections.

3.6.1 Evaluation times

Due to the communication and framework overheads, our evaluations are relatively time-consuming. ADB reports 20 Rebound runs have a total runtime of approximately 30 seconds. However, the initialisation of the individual runs and eventual clean-up by the framework results in a total time of approximately 50 seconds per effective evaluation of a Rebound configuration on the device. Also, in order to allow for variation in Rebound's runtime, and to account for variations in logfile write times, sleeping threads, and other Android peculiarities, we set the time-out per configuration evaluation to two minutes.

As running tests consumes energy, we recharge the device after each generation. Interestingly, different USB cables result in different charging currents and thus in different charging times needed. For example, one cable's charging current is approximately 0.8A (as reported by the USB hub's software), whereas a "better" cable allows for 1.4A. Based on preliminary experiments, we have conservatively chosen a charging time of 20 minutes between generations.

The above means that a single generation of $\mu = 20$ solutions takes approximately 1 hour on the device. While this seems costly, it is a big jump over some existing work, where a single configuration is evaluated by measuring the time needed for the device to run dry from 100% charge to 0% charge – which typically takes hours [177].

Note that, as a welcome side-effect of our recharging strategy between generations, we can keep the battery close to its maximum charge. Within each generation, we observed only very minor drops in the battery voltage, for example, from 4,214V to 4,201V over a duration of approximately 15 minutes. This greatly reduces potential measurement drift by the fuel gauge chip.

3.6.2 Results

In the following, we report on the results of a successful experiment, which ran for 17 generations. The first objective was the minimisation of energy, and the second objective was the minimisation of deviation from the oracle's values. We also recorded the array length deviations *M3*, however, we only observed two cases: either the length deviation was zero, or the entire test timed out.

After removing duplicates and timed-out solutions, 55 evolved Rebound configurations remain that are different from the original one. Figures 3.9 show these in the objective space, with the original configuration highlighted. Significantly, we can see solutions that consumed less energy at the cost of a decreased accuracy. In total, the 56 solutions achieve

43 different levels of accuracy. Their overall distribution of energy consumption covers quite a range, with the overall average being 37.5mWh and a standard deviation of 1.9mWh.

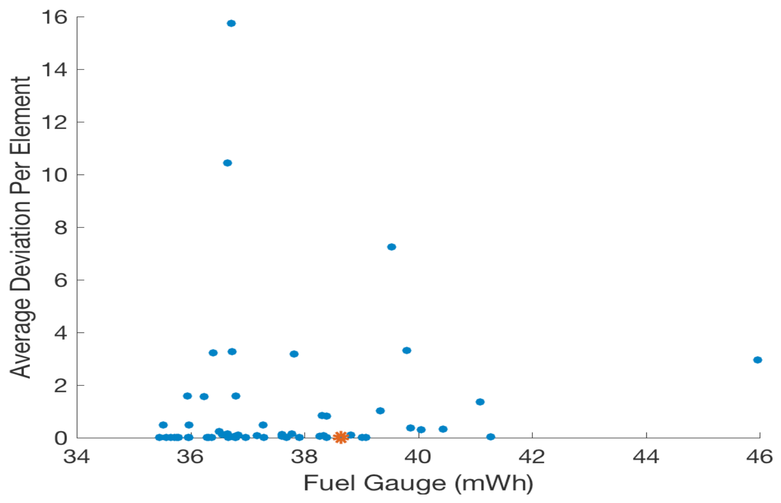

FIGURE 3.9. Scatter plot of energy vs. accuracy. Shown are the evolved configurations (blue) and the original one (orange).

In terms of code features, it is difficult to gain consistent insights from the produced configurations, as we have 19 decision variables, 44 test cases with physics simulations, and noise in energy and time measurements. Out of the 19 variables, nine have direct influence on the mathematics involved in the spring simulation. When these are changed, then the resulting calculations deviate from the oracle. Additional experiments are necessary to further reduce the noise in the energy and time measurements to better understand the influence of certain parameters on the algorithms behind the simulations and thus on the observed energy consumption.

Given the amount of technical difficulties described here, one might wonder if we could use a proxy function instead on mobile devices that do not have a dedicated battery chip installed. As one would expect, Rebound's energy consumption and runtime are correlated in our experiment, as we can see in Figure 3.10. However, this is just a moderate, positive correlation with a correlation coefficient $r = 0.7382$.

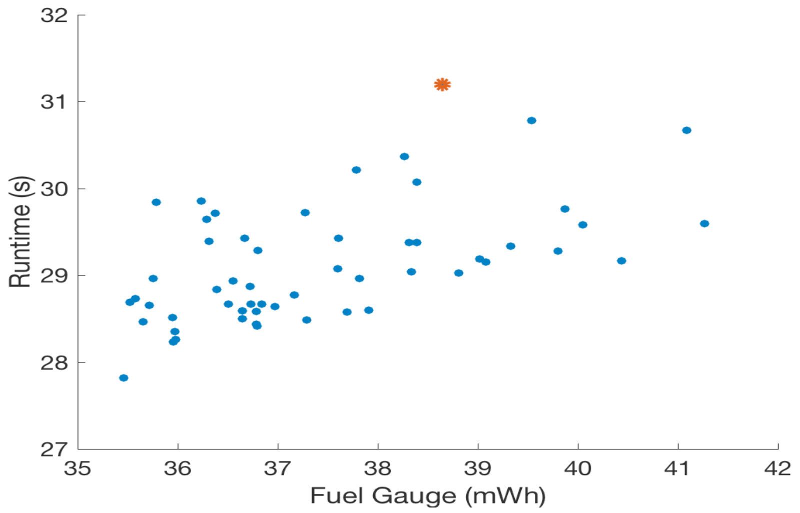

FIGURE 3.10. Scatter plot of runtime vs. energy. Shown are the evolved configurations (blue) and the original one (orange). One configuration at about (46mWh,40s) is not shown.

Other data sources that might be able to serve as proxies for energy consumption are Android's kernel-based estimations (which are based around imprecise and static power profiles for the entire device) and ADB's own reports (which are also based on power profiles while considering time slices allocated to tasks and the drop in battery percentage during running that task).[7] We show the measurements for the 56 configurations in Figure 3.11. While the kernel's estimates are better ignored due to extreme noise, ADB's correlation with runtime is comparable to the one of the battery chip. Interestingly, the measurements of ADB and the chip are not very highly correlated ($r = 0.758$), but this might again simply be the consequence of noise in the measurements. A limiting factor of ADB's estimates when used for short experiments is its low resolution, which discretises the objective space unnecessarily[8] and thus can pose challenges for optimisation algorithms.

[7]The resolution of the kernel's and ADB's estimates are 0.5mAh and 0.1mAh.

[8]ADB/kernel: 14/33 different charge estimation values for the 56 configurations.

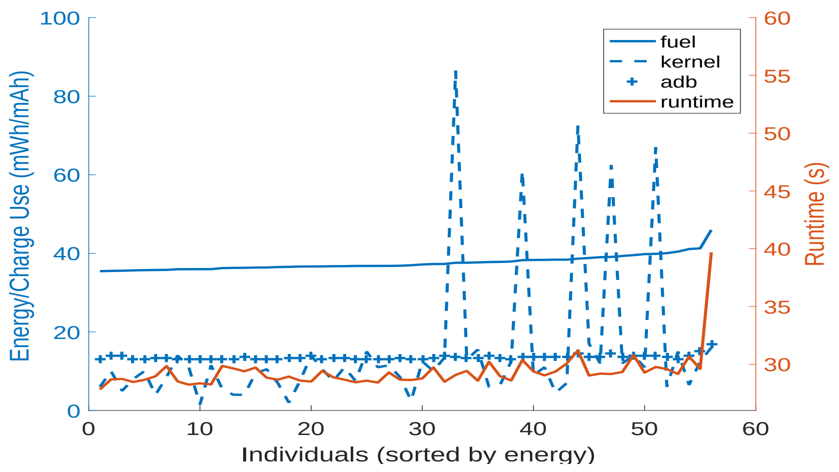

FIGURE 3.11. Correlation of energy with charge measurements and runtime. *fuel* is measured in mWh (correlation with runtime $r = 0.7382$), *kernel* and *ADB* is in mAh ($r = 0.1479$, $r = 0.7727$).

3.7 Conclusions

As shown in this chapter, the evolution of software configurations with modern mobile devices in-the-loop is feasible. However, our path to achieving this was littered with smaller and larger stumbling blocks, with some visible from afar and some only upon close scrutiny of results.

While there is a good chance for some of the future consumption models to be transferable across a range of mobile device models, we expect a degree of device-dependency of the models due to different CPU architectures in the over 24,000 different Android phones available [148]. This means that discovering accurate models will have to be on a per-platform basis. If we may express a wish, then we would love for smartphone manufacturers to spend an extra \$1 on an accurate internal battery chip.[9] With this hardware

[9]Price per MAXIM17050 was USD 1.51 in 2013, see `http://tinyurl.com/maxim17050press2013`, accessed on 24 March 2017.

support, the in-vivo energy modelling and evolution of customer specific software configurations can become a standard feature. In the next chapter we learn simple energy models and use them as fitness function besides the in-vivo optimisation.

Smartphones will continue to evolve both in hardware and software in order to improve the user experience. However, this is likely to result in challenges for researchers interested in creating relatively noise-free test beds that are to support the automated optimisation of non-functional properties. Although the simple approach used in this chapter for the fitness evaluation minimised the sampling error induced by low frequency sampling rate, the obtained measurements are still noisy. This is because executing the test suite comes with an overhead that is up to 600% of the whole test execution in some cases [9]. In the next chapter we present a new approach to mitigate noise induced by the test overhead.

Chapter 4

In-vivo and Offline Optimisation of Energy Use in the Presence of Small Energy Signals – A Case Study on a Popular Android Library

"In dwelling, live close to the ground. In thinking, keep to the simple. In conflict, be fair and generous. In governing, don't try to control. In work, do what you enjoy. In family life, be completely present."

Lao Tzu

4.1 Introduction

In the previous chapter, we have presented solutions to the expected and unexpected technical challenges when optimising energy consumption of software *in-vivo*. We also have conducted a proof-of-concept experiment to show that it is possible to carry out *in-vivo* optimisation on smart devices. In this chapter, we implement a new method to overcome noisy energy measurements obtained from smart devices' internal fuel gauges.

We then create energy estimation models based on CPU utilisation and executed lines of code. We also run *in-vivo* and model-based optimisation experiments. We validate our results on the target platform (i.e. used for optimisation) and on a different platform. This chapter has been published in the fifteenth edition of the EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services in 2018 [16]. We have extended the paper with Sections 4.3.1 and 4.5.3.

Energy demands on mobile platforms are increasing as users spend more time on their devices [160] and their applications become more powerful. In response, hardware manufacturers have given a very high priority to improving battery capacity [161] and operating system vendors have started to ration energy-hungry resources [70]. Unfortunately, it is still the case that mobile application vendors can produce applications that use too much energy [101]. A cause of this problem is a lack of developers' skills in optimising applications for energy [118]. Search-based software engineering (SBSE) can help address this problem through automated search for energy-efficient variants of mobile software [14, 96].

Current work in automated optimisation of energy use has employed models derived from external meters to drive search [21, 22, 26, 96, 138]. However, as operating system behaviour becomes more complex and platforms become more diverse, such models are becoming less generally applicable [42]. An alternative approach to energy evaluation is to test application variants for energy use *in-vivo*, i.e., on the device itself [14] using the device's internal meter.

Contributions. In this chapter we use measurements from the device's own internal meter that, for the first time:

1. build energy models that are subsequently used for successful optimisation of energy use of a CPU-bound application and, alternatively,

2. successfully guide the same optimisation task using direct sampling of the internal meter *during* optimisation.

By using the internal meter of the phone we make progress toward optimisation processes customised to each platform and its current software environment. As hardware platforms and software configurations become more complex and diverse we envisage the ability to create custom models in this way will become increasingly important.

We also demonstrate that it is possible to overcome the relatively low accuracy and resolution of the internal meter through:

1. simple rewrites to the code of the source application to amplify the signal the code produces whilst running in its test harness, and

2. by performing an initial in-vivo sensitivity analysis to identify the most promising targets for optimisation within the application code.

Through these measures we demonstrate that it is possible to significantly reduce the energy use of Rebound: a short-running physics library, for animating GUI interfaces that is installed on over 1 billion devices worldwide.

The rest of the chapter is structured as follows. After putting our work into the context of existing work in the next section, we present our experimental methodology and describe our preliminary experiments in Section 4.3. Section 4.4 describes the energy optimisation experiments. The experimental results and validation is presented in Section 4.5. Finally, we present our conclusions in Section 4.6.

4.2 Related Work

Related work can be divided into work that builds energy models for CPUs on mobile devices and work that performs automatic energy optimisation on code. In terms of CPU energy models, many works have used an external meter to help derive energy models [32, 63, 74, 121]. Such meters, while accurate, are increasingly difficult and expensive to set up, among other, because batteries are often non-removable nowadays and because batteries and devices communicate – this is difficult to mimic if all one has is an external power meter that cannot cover the communication protocols of various battery manufacturers.[1] In contrast, this work uses the easy-to-access internal meter (a specialised battery fuel gauge chip with various compensations) for both model-building and in-vivo optimisation. Loosely related here are specialised profilers such as Trepn [71] for Qualcomm processors have their place, however, it is not trivial to integrate their results into the model-building process for arbitrary code involving entire smartphones. In contrast to these approaches, we use the battery's internal meter to obtain the energy readings for the modeling building. This Maxim MAX17050 fuel gauge chip compensates measurements for temperature, battery age and load [72] and it is an adequate substitute of an external meter if the measurement periods are sufficiently long [15].

[1]In our preliminary testing with external meters, our modern devices would either not start or they would shut down abruptly when we would leave the battery communication pin(s) unconnected.

In addition, the code instrumentation procedure used for modelling in this paper is simpler and less labour intensive than the techniques in [56, 57, 95]. For example, the *vlens* tool [95] calculates energy use of apps at source line level using an external meter readings combined with program analysis and statistical modelling. Another example is the *elens* tool [57], which is a technique based on program analysis and the Software Energy Environment Profiler (SEEP) that estimates energy usage of Android APIs. SEEP is a labour intensive and infeasible to maintain, as there are thousands of APIs in Android SDK, and they evolve rapidly at rate of 115 API updates per month [108]. In addition, the work in [32, 150] measures the energy use of instructions at microprocessor level. Their results show that the variation in energy use between different instructions is relatively small. We make this one of our assumptions, but validate the trade-off configurations in the end on the device nevertheless. Our approach is similar to the work of [42, 176], which uses a battery monitor unit to measure energy usage and correlates it with CPU utilisation. Our work also extends the modelling process to the relationship between energy and lines of code (LOC).

There are several studies that use SBSE to improve the energy efficiency of software on desktop platforms [22, 26, 138]. All of that work builds the models from external meter readings and uses a single-objective technique. In contrast to this, our research targets mobile devices, which have been shown to be a hostile environment for such experiments [14].

In terms of portable devices, [24] applied a multi-objective optimisation technique trading off energy consumption (measured externally) on a Raspberry Pi, and [101] utilised an optimisation approach to trade off energy use of an OLED screen (model-based) against a measure of user-experience. We utilise both generated models and live battery readings to discover energy-accuracy trade-offs both in-vivo and off-line.

4.3 Methodology

This section outlines the setup of the experiments described in this work. In the following we describe, in turn: the animation function in the target application that we use to demonstrate our approach; the evolutionary search framework used; the fitness function; the methodology for defining the search space; and the initialisation process for the search.

4.3.1 The Advance Function

Our case study in this chapter is Rebound library presented in Chapter 3. Now we describe how a spring animation is created using Rebound. The `advance` function, which is responsible for creating animations, has two termination conditions. At the start of the function an `if-statement` checks whether a spring is at rest or was at rest in case of the current function call is simply a subsequent call of an animation job. The second termination condition is a `while-loop` guard which checks the time-step of the current animation, since animation is a function of time.

The `while-loop` branch keeps animating the spring by iteratively approximating its next position and velocity starting from initial values. Initially the spring is in a stationary state where its velocity is zero and its position is the equilibrium position (at-rest position). The next state of the spring is unknown and is a function of time. The update code for the spring runs inside a `while-loop` the next state is approximated using the Runge-Kutta RK4 numerical analysis method which is a family of implicit and explicit iterative methods [135]. RK4 is used to accurately approximate solutions of initial-value ordinary differential equations [64]. This method requires specifying the initial conditions which are the initial time t_0 and the spring values position and velocity y_0 for each. At the beginning we know these values t=0, and spring values are set to defaults. However, the next spring values y_{n+1} at t_n and are calculated using the Equation (4.1).

Next, we describe the equation in a geometric terms – we refrain from describing it in detail mathematically. The mathematical explanation can be found in [64]. Equation (4.1) is simply a weighted mean of four intervals k_1 to k_4 added to the current state y_n. As can be seen, two middle intervals k_2 and k_3 are given more weight (1/3) than the outer points. They are twice as heavy as k_1 and k_4.

$$y_{n+1} = y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) [135] \tag{4.1}$$

where,

$k_1 = h\ f\left(t_n, y_n\right)$, $k_2 = h\ f\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$,

$k_3 = h\ f\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$, and $k_4 = h\ f\left(t_n + h, y_n + k_3\right)$.

k_1 is the slope at the beginning of the step-size h which is the left-end interval (t_n, y_n). Simply, this term is Euler's prediction of a jump (by h) from the current state y_n. Based on k_1, the second slope k_2 is calculated using half of the step-size (i.e $t_n + h/2$). This means f is evaluated at a midpoint between the current state and the next state. Hence, it gives us an estimate of the desired curve (position or velocity) slope at this midpoint. Similarly, the third interval k_3 is computed using a halfway jump $(t_n + h/2)$, but this time based on k_2. At the end, k_3 is used to step all the way across the step-size which is the right-end interval (i.e. $t_n + h$).

After the `while-loop` finishes this integration step (i.e. finding the next position and velocity), the spring's velocity is evaluated to find whether it is at rest. This is done by checking its current velocity against a rest speed and displacement from rest thresholds `RestSpeedThreshold` and `DisplacementFromRestThreshold`, respectively. Interestingly, both are set by the Rebound developers to 0.005 with no clarification in the source code comments.

As can be observed, the computation in the `while-loop` is an intensive task that is required with every animation job. It is unknown what could happen to the animation quality when changing the values of the coefficients in the Equation (4.1). Indeed, maximising them would increase the movement of the spring, however, it definitely deteriorates the accuracy of the calculations. It is worth mentioning these coefficients are only a small subset of the deep parameters hidden in Rebound's `Spring` class source code which will be discussed in subsequent sections.

4.3.2 Evolutionary Framework

In our main experiments we use NSGA-III [39] to optimise two objectives: energy use by Rebound and the deviation of the output of the modified application from the original. We want to explore the configuration space to find energy efficient variants that have minimal effects on the animation quality. Individuals in the search space are variants of the Rebound application. These variants are created during search using deep parameter optimisation (DPO) [23, 171]. DPO is a genetic improvement technique [125] where variants are produced by mutating constants (deep parameters) found within its source code, and which are typically not accessible to a user. These deep parameters are exposed to the search process by automatically lifting them to be encoded as explicit constants.

For Rebound, our framework exposes integer and double constants within the source code. This starts by replacing those constants with placeholders. The placeholders are calls to read each placeholder's value from a configuration file. In most genetic improvement research, modifications to the source-code require recompilation before evaluation. This can be costly because recompilation in our case carries a penalty of 20-30 seconds. Moreover, transferring and installing Rebound takes up to 2 minutes. Encoding individuals as configuration files for the exposed parameters eliminates the cost of recompiling Rebound variants. The configuration file is read once per execution and thus incurs a fixed energy overhead though. As the file size remains stable, we assume this read overhead to be constant across any and all evaluations, and therefore it does not effect the outcomes of the search process.

4.3.3 Fitness Functions

Two fitness functions are used in this work: the *energy* used by an individual program variant and the *accuracy* of that program variant.

Fitness: Energy

In our experiments, we utilise three alternative methods to measure fitness: in-vivo measurement; on-phone measurement of CPU utilisation (Jiffies); and a Lines-of-code (LOC) proxy for energy use running on a computer. In all cases the proxies model CPU-usage.

In-Vivo Energy Measurement. One way to measure the energy use of a program variant is to perform experiments on a working platform – in-vivo – and sample the internal battery meter before and after the trial run. In our experiments our target platform is the HTC Nexus 9 running the Android 6 operating system. The special feature of this device is that it is equipped with the Maxim MAX17050 fuel gauge chip that compensates measurements for temperature, battery age and load [72], which provides an adequate substitute of an external meter if the measurement periods are sufficiently long [15]. Android 6 is used in our test-bed as its market share of Android flavours was 32% and 25% for the second half of 2017 and first half of 2018, respectively[2].

In our experiments we use a version of the methodology described in Chapter 3 to load a set of program variants to the phone via the Android Debugging Interface version 1.0.36 (ADB), and then cycle through the variants, sampling the internal meter before and after, and finally disconnecting ADB and charging the phone for the next generation. The internal meter is accessed through Android's BatteryManager. This API broadcasts these values with a frequency of 4Hz. The precision of this approach is significantly higher than ADB's own energy estimates which are based on rough and uncompensated system models as we showed in Chapter 3.

In running the optimisation process a great deal of care is required to avoid systematic noise induced by dynamic CPU speeds, effects of heat, non-linearity in battery response, overheads of memory logs, garbage collection, communication and UI devices, and sleep modes. For the detailed description, we refer the interested reader to [14].

Dealing with small energy signals. One problem specific to Rebound is test-harness overhead. As it is pointed out in [88], test duration is inversely proportional to smallest

[2]Statista – the Portal for Statistics: `https://www.statista.com/`, accessed 8 October 2018.

detectable impact. The easiest way to increase test duration is to repeat the whole test case several times to increase the energy signal. However, the test harness overhead for Rebound is bigger than the run-time of the software being optimised – by a factor of four-to-one. One can alleviate the overall overhead by repeating only the core application or function under test multiple times within each test case to ensure the test run spends a greater percentage of time running the program under mutation. Unfortunately, for Rebound this approach is not effective because the overhead in running each test case, e.g. state changes in the JVM caused by multiple runs and setting up listeners, is still large compared to the target code. In these cases, even with repetition, the code that is not subject to optimisation dominates the code being optimised. In addition, in memory constrained environments such as smart-phones, the impact of just re-running the tests fills up the application's allocated heap portion, which causes frequent Garbage Collector (GC) invocations to free the allocated memory. This notably increases the test time as well as it adds more noise to the collected energy signal [9, 68]. We address this problem in our experiments by instrumenting the code to be optimised with dummy loops after each line of code. This simple approach serves to amplify the effect of any change to the parameters of the original code. In a setting where code structure is being optimised this technique can be applied automatically with a tool such as JavaParser.[3]

Note that, by using dummy loops, we make the assumption that all lines of code are equal in terms of usage, however, different instructions can have small differences in energy consumption [150, 154]. However, in this setting, where the target code exclusively uses CPU and RAM, the assumption of uniform usage doesn't adversely affect search.

Of course, final validation of the optimised configuration using the non-amplified code is required and this validation is presented later.

CPU Utilisation Model. To build the CPU utilisation model for the Nexus 9, we used the amplified version of Rebound used for the *in-vivo* energy measurement mentioned above. We also created a set of 15 different configurations of Rebound, each of which has a different CPU workload. The creation mechanism is discussed in section 4.3.4. Each run is repeated eleven times at every supported CPU frequency. This re-sampling averages out measurements' noise and the odd number is to permit choosing a real middle measurement instead of averaging the two middle measurements to compute the median. During the run, the system statistics are accessed for the system software clock expressed in *jiffy counts* as a measure of CPU utilisation. In these experiments, the precautions described

[3]JavaParser, `https://javaparser.org/`, accessed on 10 May 2018.

in [14] were taken to minimise the effects of temperature, memory-consumption, file-system overhead, hardware and software governors, and other peripheral devices.

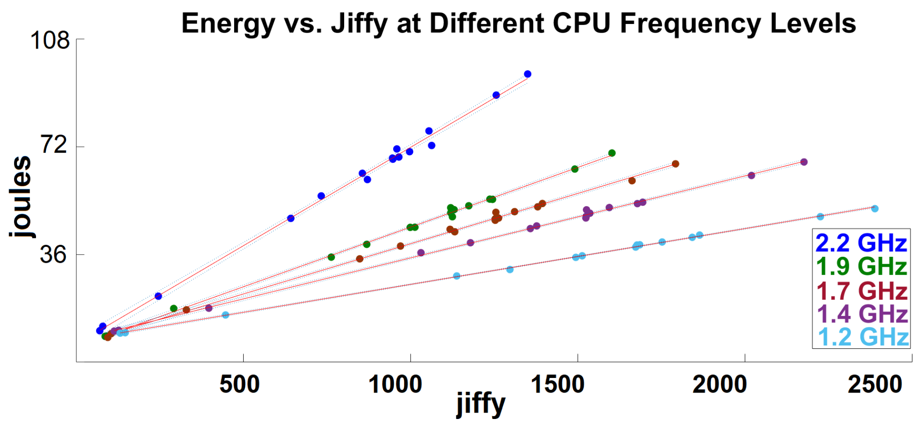

FIGURE 4.1. Energy estimation using jiffies at different CPU levels.

Figure 4.1 shows the mapping of jiffies to energy use, for different CPU frequency levels. As can be seen, the correlation between CPU utilisation expressed in jiffies and energy use is linear. While this finding for Rebound on our hardware aligns with other works in the literature [46, 163, 176], such relationships do not, by any means, hold in all settings and can even vary across devices [42, 107].

Lines-of-Code Model. As another basis for comparison, we use the number of executed lines-of-code (LOC), to estimate the energy consumption. In this model we use experimental data from profiling the CPU at 1.4 GHz. It can be observed from Figure 4.2, the energy consumption linearly correlates with the executed LOC for Rebound. The R^2 of the model is 0.99, indicating a strong correlation. Moreover, the computed mean absolute percentage error is less than 1%.

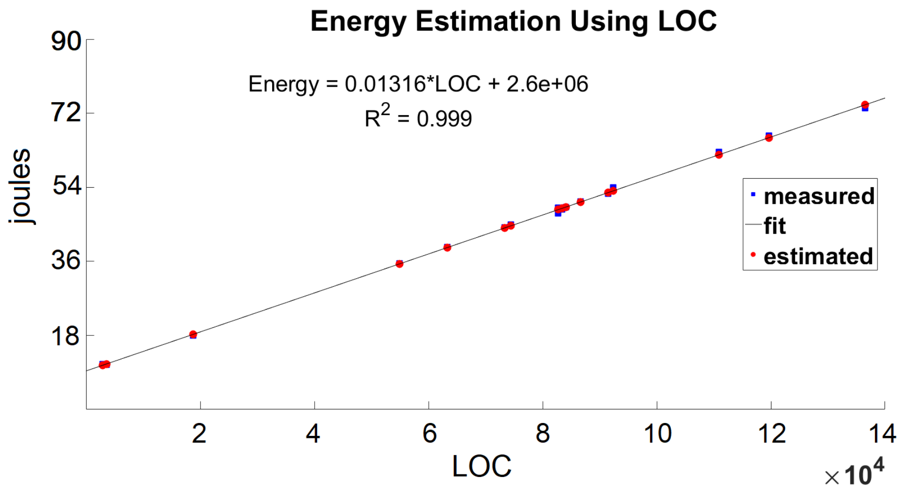

FIGURE 4.2. Measured energy vs. estimated using LOC at CPU frequency of 1.4 GHz.

Fitness: Accuracy

The second dimension of the fitness function is the accuracy of Rebound's output, which is calculated using the mean absolute error (MAE) in Equation (4.2). When the Rebound library is run, it produces for each test case a single-dimensional trace $result_i$ of spring positions and speed depending on the test case i. In this work, the accuracy is determined by comparing the trace produced by the Rebound variant with the original Rebound $oracle_i$ for each test case i. Variants whose traces closely track the original trace receive high accuracy. Major deviations from this trace receive low accuracy.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |result_i - oracle_i| \tag{4.2}$$

4.3.4 Refining the Search Space

The search space for the optimisation of Rebound are constants lifted from the code for the purpose of deep parameter optimisation. There are dozens of such parameters, which is an impractically large number to optimise with a quite limited number of function evaluations. To reduce the number of parameters, we conduct a sensitivity analysis to isolate the parameters to which the energy consumption of Rebound is most sensitive.

The class `Spring` contains 24 parameters. To check their impact on energy use, each parameter is multiplied by $(10)^x$, where x is an integer in the range $[-3, 3]$. The test is then executed 11 times for each parameter's setting. One test run takes about 15 seconds.

The parameters fall into three categories with respect to tests. First, *sensitive* parameters are those where the applied changes induce a significant change in energy use — these are worth optimising. Second, *insensitive* parameters create little change in energy consumption. Third, *too-sensitive* parameters cause a timeout in response to changes in parameter values. Within the category of *sensitive* parameters, alterations to values may reduce loop iterations [122], or disable certain costly branches [171] to reduce energy usage. Because we permit deviation from the test oracles, it is likely that trade-offs can be found to minimise the consumed energy at the expense of test quality [24]. For the purposes of optimisation, the sensitive parameters are represented by an n-tuple of numbers to form a solution. A fitness value for the objectives of energy and accuracy is assigned to each solution.

Among the 24 parameters in `Spring`, nine parameters can be classified as sensitive. Since the number of evaluations is limited in our experiments due to the expensive fitness functions on hands (i.e. *in-vivo* and jiffy-based model), we furthermore select from these nine only those parameters that reduce the overall CPU utilisation (i.e. CPU jiffies) by at least 20%. Table 4.1 shows the selected five parameters, the number of jiffies required to run the test suite and the reduction percentage. Interestingly, the impact on CPU use starts to appear only after at least two magnitudes of change in all parameters, except for `Spring_DOUBLE_24_1` which dramatically decreases the number of jiffies (reduction by 91%). These findings conform with previous research on deep parameter optimisation, which indicate that the majority of exposed parameters are not worth optimising [23, 171].

The parameters are as follows: the maximum amount of time to simulate per physics iteration in seconds is `Spring_DOUBLE_24_1`, the fixed time- step/chunk to use in the simulation is `Spring_DOUBLE_26_1`, the at-rest speed threshold is `Spring_DOUBLE_46_1`, the numerator of the weighted sum in Equation (4.1) is `Spring_DOUBLE_377_1`, and the coefficients of k_2 and k_3 are represented by `Spring_DOUBLE_377_3`. It is worth noting that the parts of the equation $2k_2 + 2k_3$ are represented as $2(k_2 + k_3)$ in the source code (based on the distributive property of multiplication over addition). Therefore, `Spring_DOUBLE_377_3` represents the coefficients of both k_2 and k_3.

Parameter Name	Multiplication Factor	Jiffy	Reduction %
Original	n/a	1524	n/a
`Spring_DOUBLE_26_1`	1000	116	92%
`Spring_DOUBLE_24_1`	0.001	128	91%
`Spring_DOUBLE_26_1`	10	397	74%
`Spring_DOUBLE_377_3`	1000	1000	34%
`Spring_DOUBLE_46_1`	1000	1034	32%
`Spring_DOUBLE_377_1`	1000	1034	32%
`Spring_DOUBLE_377_3`	100	1048	31%
`Spring_DOUBLE_377_1`	100	1049	31%
`Spring_DOUBLE_46_1`	100	1181	23%

TABLE 4.1. The five selected parameters after the sensitivity analysis. The parameters' names reflect their properties. For example, `Spring_DOUBLE_377_3` is the third floating-point number used in line 377 of the Spring class.

Figure 4.3 shows a comparison between the original configuration (default values) and the selected parameters for optimisation. As can be seen, the search space is non-monotonic. For example, changes to `Spring_DOUBLE_26_1` can drastically improve the energy efficiency after being multiplied by 10 and 1000, however, multiplying it by 100 (and by quite a few other numbers not listed here) results in timeouts.

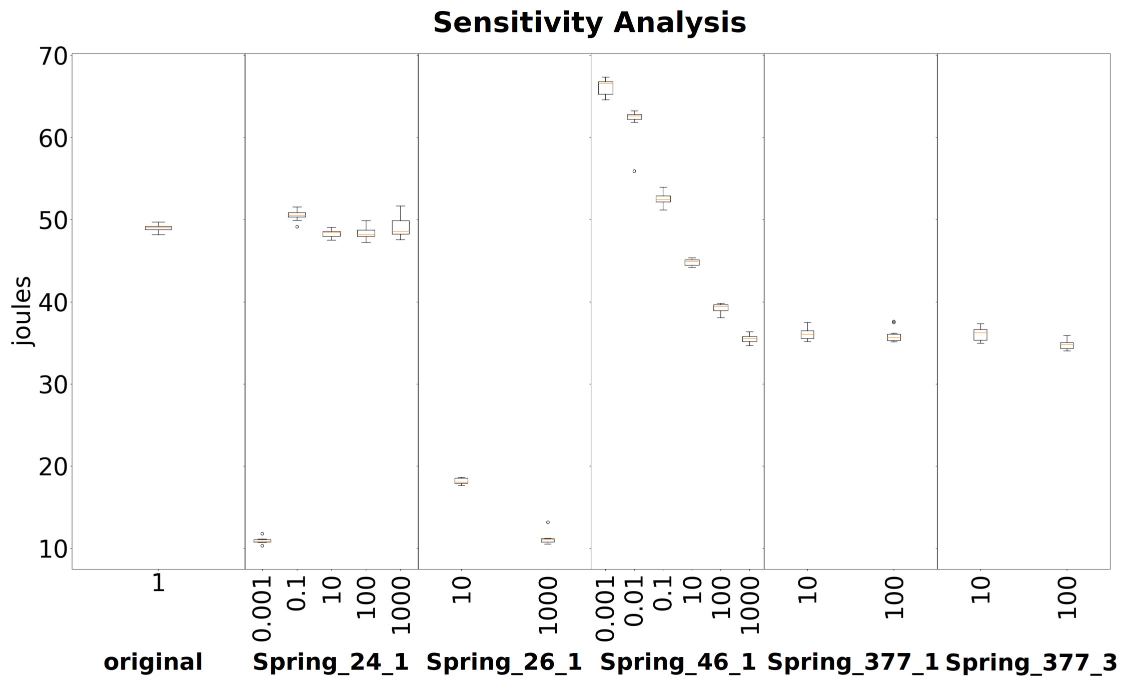

FIGURE 4.3. Comparison of the original configuration with modifications of the sensitive parameters. The numbers 0.1 to 1000 denote the factors by which the original value of the parameter is multiplied. Missing factors produced invalid solutions. The type of each parameter was removed from the plot.

4.3.5 Initialisation

While we could create the initial population (i.e., the initial set of program configurations) by generating random solutions, we attempt to maximise diversity by sampling both energy-hungry and energy-frugal values for parameter settings.

As with the sensitivity analysis, we base the seed population on the original program configuration, and then multiply selected parameters by factors of the form 10^x. The exponent here is randomly drawn from a Gaussian distribution with $\sigma = 3$, to allow us to cover an even greater space than the original sensitivity analysis.

Individuals are generated until the initial population is seeded with $\mu = 25$ valid parameter vectors. We limit the perturbations to only two parameters (dimensions) per solution, to reduce the number of timed-out (invalid) solutions generated: we found that one and two dimensional changes in one solution require more than 50 trials while more changes take up to 120 trials.

Figure 4.4 shows the resulting initial population. The x-axis presents the seeds (solutions), and each seed contains five data points representing its decision variables (selected parameter). The amount of modification to the original value (ratio) of each parameter is presented on the y-axis. For example, the parameter `Spring_DOUBLE_46_1` shown in green in seed number 19 is 10,000 times bigger than its original value (0.005).

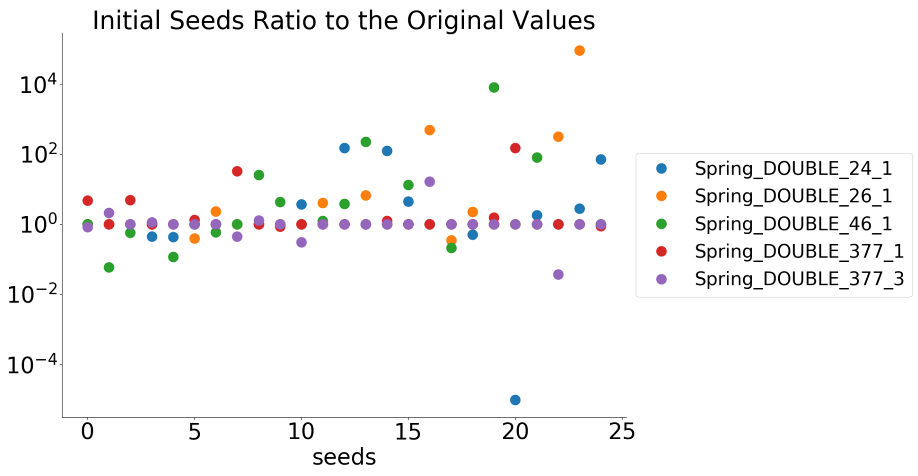

FIGURE 4.4. The amount of modification applied to each parameter in the seeded first generation of $\mu = 25$ solutions. Only two parameters are altered per solution (two dimensions).

4.4 Experiments

As mentioned previously, we use NSGA-III [38], a genetic algorithm designed for multi-objective search, as implemented by the MOEA Framework.[4] Three experiments are conducted where energy use obtained by the internal meter, and via our jiffy and LOC models. The *in-vivo* experiments were conducted on Nexus 9 tablet running Android 6, whereas

[4]MOEA Framework version 2.12 available at `http://moeaframework.org`, accessed 10 May 2018. We leave all variation operators and variation probabilities at their standard values.

the off-line experiment (using the LOC model) was performed on a Windows 10 machine with 16GB memory and Intel i7-6700 CPU clocked at 3.4GHz.

Note that we use the algorithm purely for the purpose of navigating the trade-off space, and as part of this study to optimise small energy signals. For a current overview of multi-objective algorithms, we refer the interested reader to [31, 93].

Any parameter is constrained to values between 0 and 10000; their original values are between 0 and 6. With a population size of $\mu = 25$, we seed the initial generation using the steps described above. With this setup we run for 1250 evaluations. Since two of the experiments run *in-vivo* (optimisation based on internal meter readings and the jiffy model), we use settings described in Chapter 3 to overcome Android's challenges for energy optimisation experiments. In addition, we limit the generation size to avoid any variation in voltage, as this is a real energy-based experiment. In case of having a generation full of invalid solutions (worst case scenario), the timeouts are responsible for longer runtimes, and thus for longer discharge phases, voltage drops, and less reliable readings.

4.5 Results

In this section we discuss the results of our optimisation experiments. We start by showing the solutions in the objective space: energy and accuracy (i.e. their deviations from the original behaviour), we then demonstrate how the solutions found in the Pareto fronts are distributed in the decision space, followed by the validation of a subset of the improved solutions.

4.5.1 Solutions in the Objective Space

Figure 4.5 shows the evaluated solutions in black, initial population are black circled, as well as the Pareto front obtained by relying on the internal meter are red circled. Solutions for later validation marked as solid red circles and the original configuration is in light green. The x-axis shows the energy use in joules and the y-axis shows the accuracy of the solutions on log scale.

As can be seen, there are ten non-dominated solutions. In terms of energy efficiency, the best solution found uses 8.7 joules in the raw energy optimisation. The optimiser took 991 fitness evaluations to find the best solution (in terms of energy consumption) among all the experiments. This compares with 48 joules used by the original configuration. On the other hand, its accuracy deviates by 1.2 on overall, and it passes only one test, making it the worst solution on the front in terms of deviation. It has a huge impact on the quality of the test.

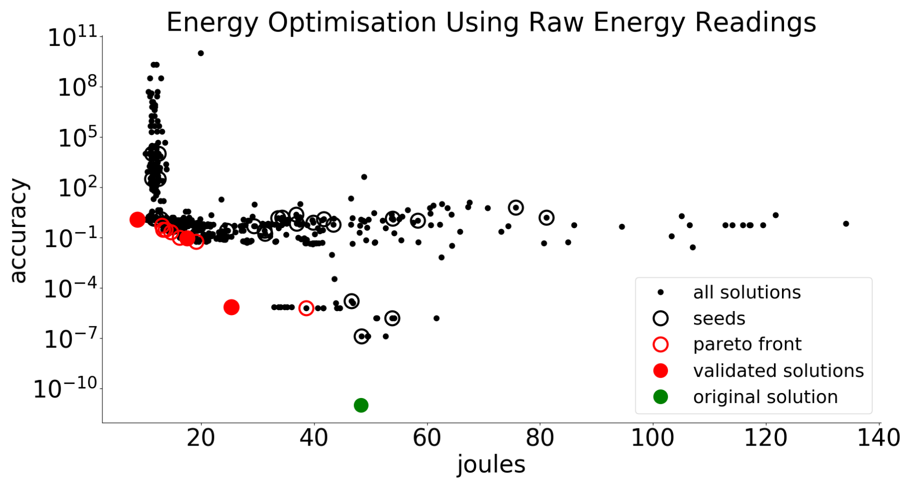

FIGURE 4.5. The results of the optimisation experiment using raw energy readings from the internal meter.

As an example, let us investigate a particular solution, and how changes affect the energy consumption. In Figure 4.5, the solution with the second-lowest energy consumption (second red dot from the left) consumes 16.2 joules and has an acceptable deviation from the test oracle. Although, it fails to pass five test cases, its accuracy is relatively low with MAE of 0.09.

Figure 4.6 shows the results of testing the `Spring` positions/steps while being in motion on all six tests. As can be seen, after evolving the new values of the solution's parameters, the deviation is very small and might not even be noticeable by a user.

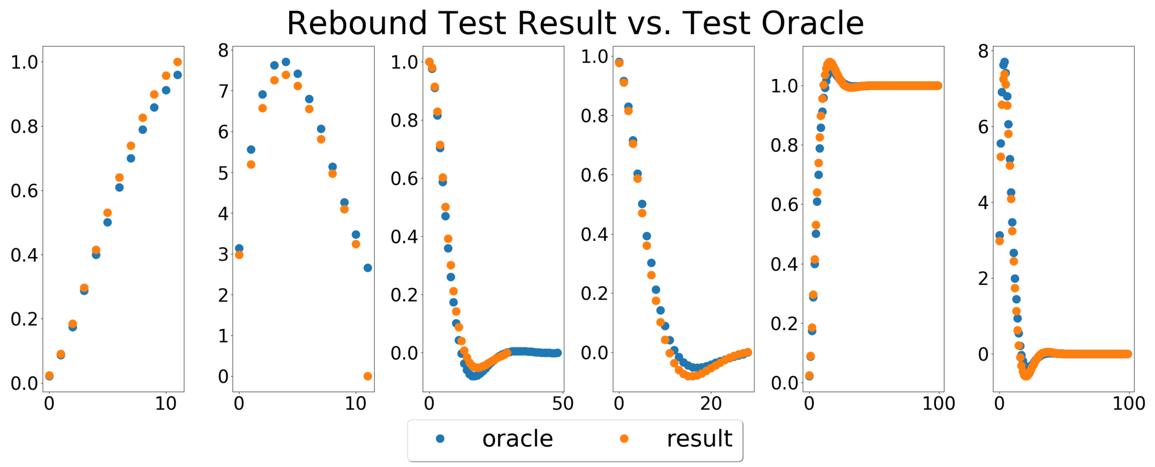

FIGURE 4.6. Spring's expected result (test oracle, blue) vs. the actual result (orange) of the second marked solution from raw energy optimisation experiment.

Such a deviation in each test is beyond what humans can perceive. This is because the animation resulted from the improved solutions differ from the original animation by less than 0.1 second, moreover, the animation duration is less than a second which is less than the threshold for a user to notice the delay i.e. sluggish animations [114]. We conjecture that Rebound's developers might not be aware of such an energy improvement at the expense of slight deviations from the functional requirement.

In Figures 4.7 and 4.8 which follow the same encoding as in Figure 4.5, we show the fronts of the optimisation runs using the energy models. In the extreme case, the energy use was reduced to 10.6 joules and 10.8 joules by utilising the LOC and jiffy model-based optimisation, respectively.

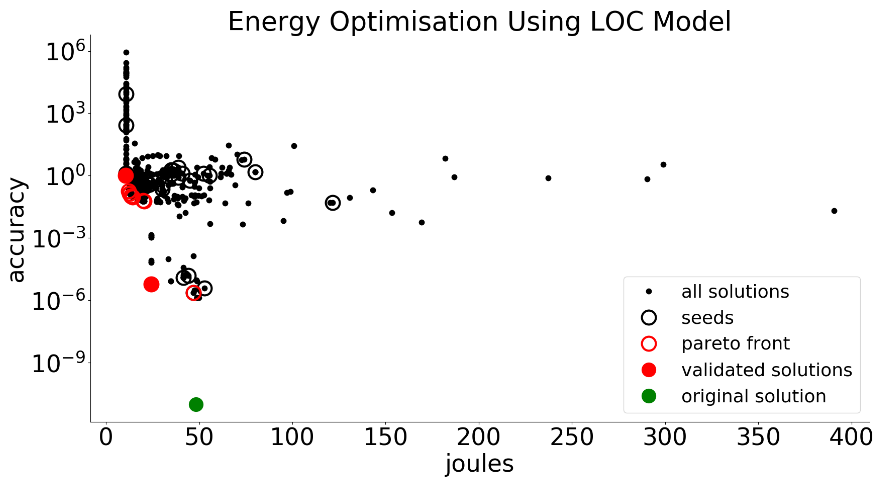

FIGURE 4.7. The results of the optimisation experiment using the LOC model.

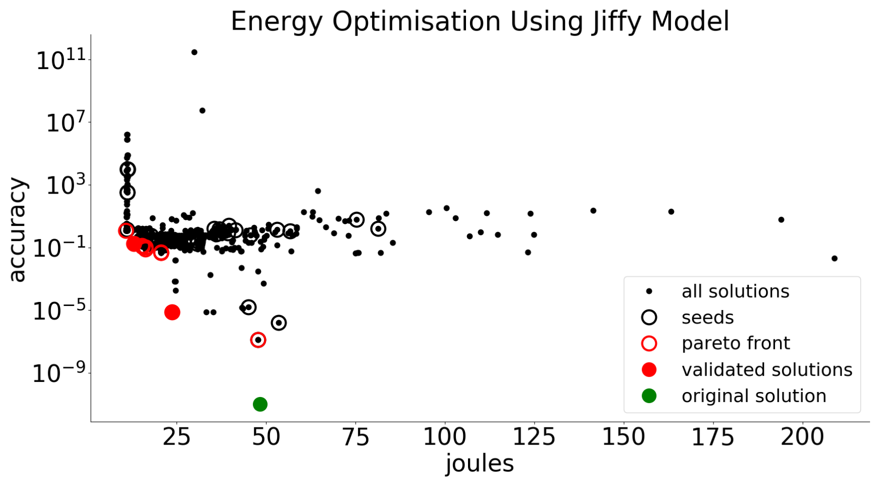

FIGURE 4.8. The results of the optimisation experiment using the jiffy model.

For a quick check of the diversity along the two objectives, we use the coefficient of variation (CV). For the optimisation based on raw readings, the CV values are 56% and

134% for energy use and accuracy. The CV values for the fronts resulting from the model-based optimisation are comparable: for energy/oracle deviation they are 55%/208% and 61%/169% for the jiffy and the LOC models, respectively.

4.5.2 Solution Distributions

Figure 4.9 illustrates the actual solutions of the Pareto fronts in the three experiments. To allow for an easy comparison with the original configuration, and in line with previous visualisations, we provide the solutions as factors applied to the original configuration. These Pareto-optimal solutions are sorted from left to right in increasing order based on their increase in energy consumption (and thus in decreasing order of test deviation).

Quiet surprisingly, the solutions of the three fronts are relatively similar. For example, DOUBLE_46_1 (green) is almost always increased by about five orders of magnitude. The parameter represents the rest-speed threshold at which the `Spring` is determined to be at rest. Also, DOUBLE_24_1 (blue) is modified by two to four orders of magnitude, and the others often just comparatively little. To us, this is additional evidence that (1) the developed models are reasonably consistent with the real world, and (2) the results are reproducible, even when using a different model or the real device.

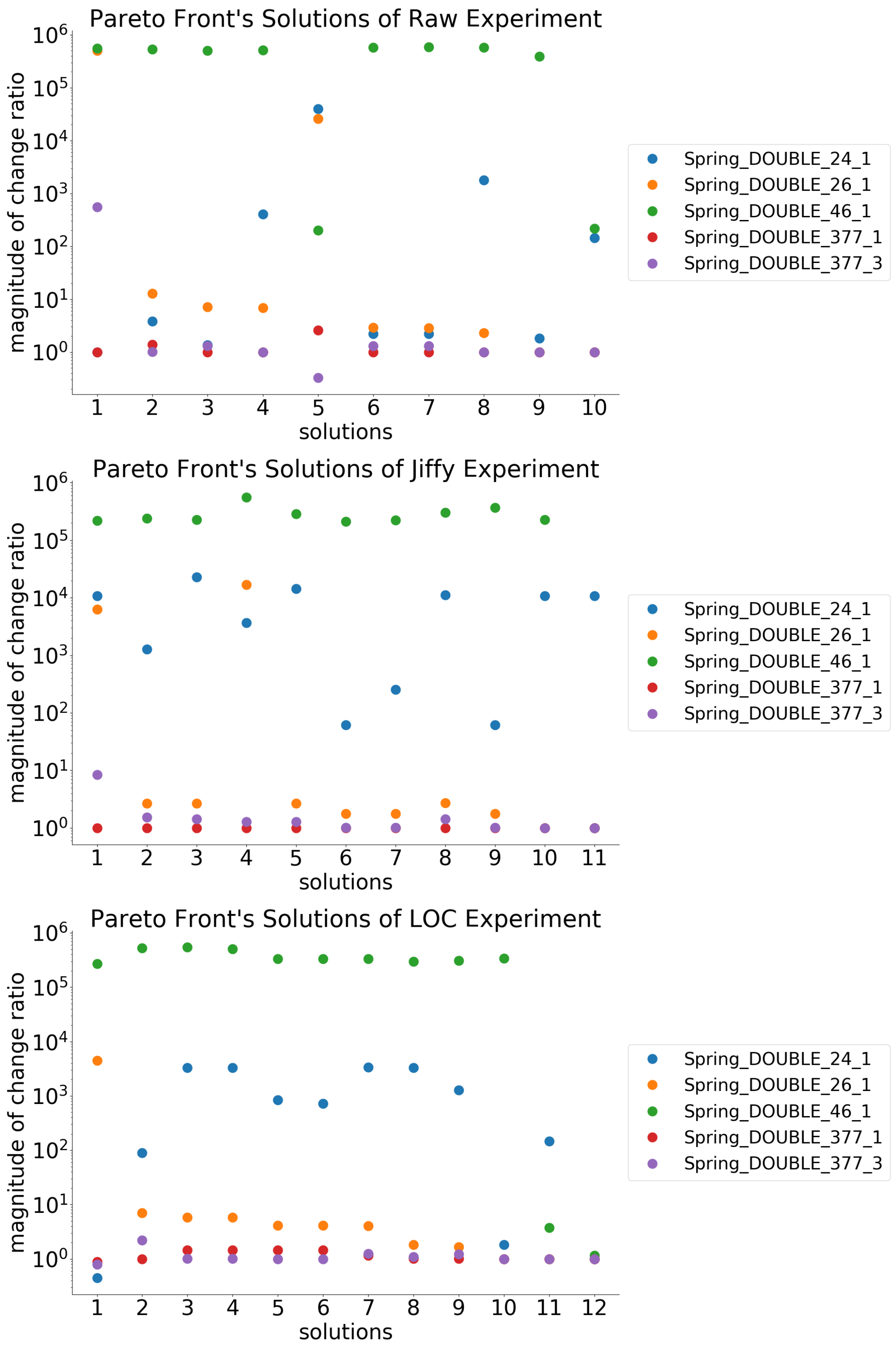

FIGURE 4.9. Distributions of Pareto-optimal solutions in the search space.

4.5.3 Source Code Insights

Previously, we described the process of creating animations using the `advance` method in `Spring` class. Now, we discuss the selected parameters' effects on the energy use and the accuracy of the resulting animation. In order to decrease the energy use of the desired animation, one has to reduce the number of iterations in the `advance`'s `while-loop`. This is a tricky task since it depends on the `while-loop`'s guard as well as, the position and the velocity of the animated springs[5], which are computed by Equation (4.1).

Besides, the visual analysis in Figure 4.9, we analyse our results by measuring the correlation between the selected parameters, the energy consumption of Rebound and its animation accuracy, measured as *MAE*, to confirm expected dependencies or to discover unexpected correlations. We use the Spearman correlation coefficient (r_s) [152], which is a non-parametric measure of the strength and direction of the relationship that exists between two variables. The coefficient value can be interpreted as perfect ($r_s = 1$), very strong ($1 > r_s >= 0.8$), strong ($0.8 > r_s >= 0.7$), moderate ($0.7 > r_s >= 0.4$), low ($0.4 > r_s >= 0.1$), negligible ($0.1 > r_s > 0$), or no relationship ($r_s = 0$) [117]. The sign of the coefficient determines whether the relationship between the two parameters is positive or inverse. Figure 4.10 shows the energy use of Rebound has a very strong reverse relationship with its animation accuracy.

The `while-loop` is guarded by the time-step `Spring_DOUBLE_26_1` used in the integration (see Equation (4.1)). Our results indicates that the time-step has a very strong inverse correlation with the energy consumption of Rebound with r_s of -0.88. This indicates the energy consumption decreases as the time-step increases. This is because bigger steps result in dividing the overall animations into a smaller number of tasks, which requires fewer calls to the `advance` function. In contrast, it decreases the accuracy of the animation (i.e. it increases the MAE) as it grows. This is due to the fact the RK4 method approximates the next position/velocity for four subsequent intervals each of which depends on the prior. Hence, *bigger* time-steps increase the difference/distance between the intervals. As a consequence, the difference between the tangent line needed and the position/velocity curves increases. It is worth mentioning that the trade-off between the computation overhead of RK4 and its accuracy depends on the chosen step-size, and has been studied in the numerical analysis literature [129]. Our results show its impacts on energy use when it is used in creating animations.

[5]one GUI animation job requires several spring objects

In addition, the threshold that determines when a spring is at a rest state `Spring_DOUBLE_46_1` is of great importance to reduce the number of iterations and consequently the energy use. As can be seen in Figure 4.9, there is a notable increase in its value among the majority of the solutions. A spring's velocity initial value is zero, and as the spring is animated, its velocity increases or decreases depending on the spring state (i.e. stretching or compressing). The first termination condition checks whether the spring is at rest by comparing the absolute value of the current velocity against `Spring_DOUBLE_46_1`, if it is less than or equal to the threshold, the animation task stops. This means the larger the threshold is the sooner the spring stops and therefore the computation overhead is reduced. Consequently, the quality of the animation degrades. This causes the look-and-feel of the animation changes from smoothly slowing down to a sudden stop[6].

What is worse, increasing both of `Spring_DOUBLE_26_1` and `Spring_DOUBLE_46_1` as in *raw 1* results in very sluggish animations. For example, we tested the solution in a demo app where a list view is scrolled up and down. When scrolling up and down, the list seemed rigid and hardly moving. On the other hand, the other solutions (such as *raw 2*)[7], where a dramatic difference exists between these two constants, have a smoother stopping effect[8].

Interestingly, breaking some constraints in the original code semantics still produces acceptable behaviours (i.e. animations). The maximum amount of time to simulate a spring per advance call `Spring_DOUBLE_24_1` is set by the `Spring` developers and its main purpose is to make sure that when creating an animation, its duration does not exceed that threshold. For example, although *solution 2* in the raw front in Figure 4.9 breaks this constraint, its animation traces slightly deviates from the original configuration as it is shown in Figure 4.6. This is because regardless of the maximum threshold, the spring stops whenever its speed reaches `Spring_DOUBLE_46_1` threshold, and therefore such a condition is ignored in our example. As a result of such constraints the correlation between the three constants `Spring_DOUBLE_24_1`, `Spring_DOUBLE_26_1` and `Spring_DOUBLE_46_1` is not straight forward. Our results show that developers might be not aware of the interactions of the hidden parameters and their impacts on the non-functional properties of software[9].

[6]The original behaviour can be viewed at https://youtu.be/LgBHzGBumSE.

[7]The behaviour of *raw 2* can be viewed at https://youtu.be/DdxyFonqgtM

[8]The behaviour of *raw 1* can be viewed at https://youtu.be/AI9eKPgfnsA.

[9]other solutions such as solution 1 in LOC with the same observation have been tested on demo apps and confirms such findings

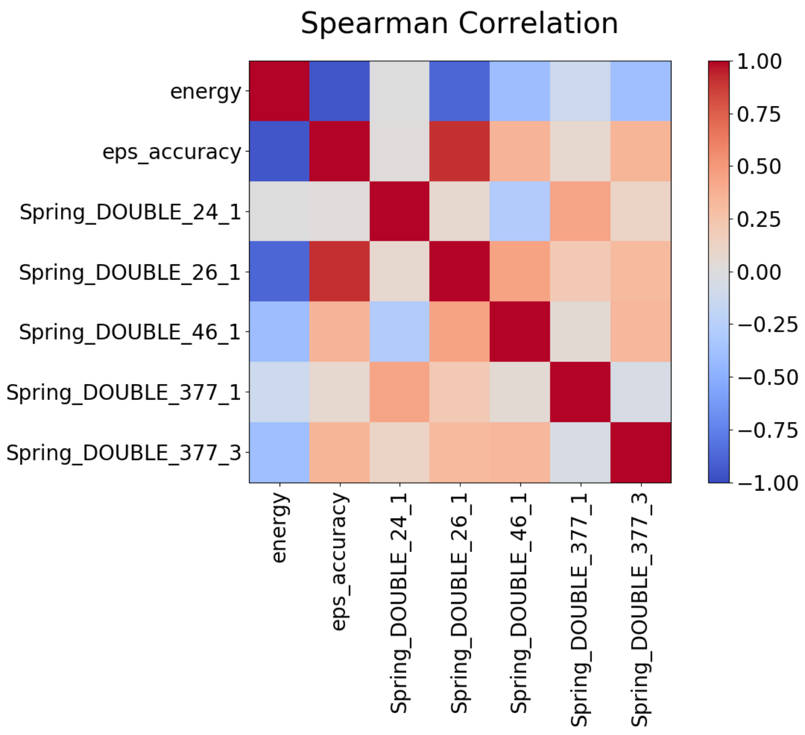

FIGURE 4.10. Spearman correlation coefficient.

Another interesting observation is violating the concept of the weighted mean in Equation (4.1) results in minimising the energy consumption of Rebound by reducing the number of iterations required to create spring animations. This is depicted in the increase found in the parameters `Spring_DOUBLE_377_1` and `Spring_DOUBLE_377_3`. Both parameters have a low to moderate inverse correlation with the energy use with Spearman r_s of -0.12 and -0.38, respectively. This indicates as their value increases the energy use decreases. This is because such an increase speeds up the spring motion towards reaching the at-rest threshold `Spring_DOUBLE_46_1` which, consequently, push-forward the moment at which the spring stops. Indeed, this happens at the cost of the quality of the animation.

4.5.4 Pareto Front Validation

Next, we validate solutions found on the three Pareto fronts. The validation process consists of removing the dummy for-loops, and running the test suite by repeating the actual call to the `advance` function 1000 times. During the test run, the energy is measured by the internal meter. The test run is repeated 31 times for each solution.

Figure 4.11 shows a box-plot of the results of the validating the marked solutions from Figures 4.5, 4.7 and 4.8, and a comparison with running the default values of the parameters with the same settings. It shows the lower and upper quartiles, the median (orange vertical line), the mean (green triangle), the minimum and maximum of the result. As can be seen, all of the optimised variants have a significant difference compared to the original settings. This indicates the feasibility of using energy models as a fitness function. Despite the battery's internal meter being less accurate than expensive external meters, the results demonstrate that it can be used for optimising energy efficiency *in-vivo*. This is because these types of internal meters are precise [15] and therefore can be used to rank solutions in terms of energy use. Also, this does not require developers to have a special skills to obtain energy readings.

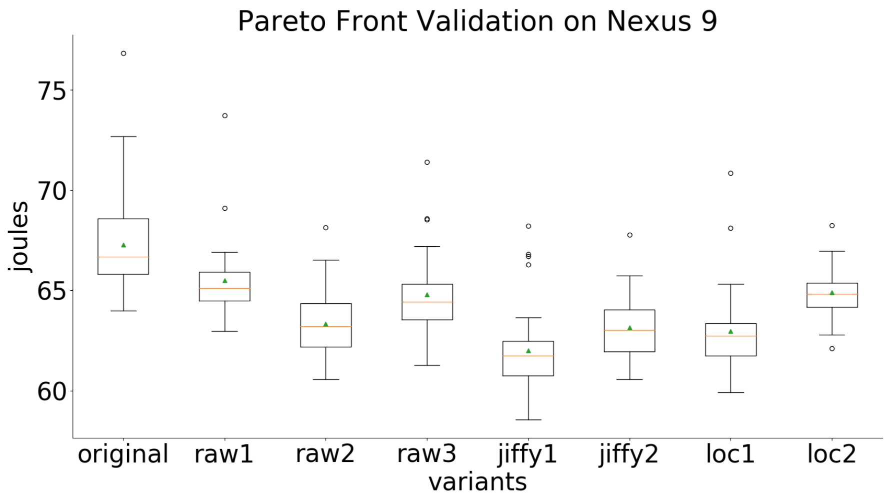

FIGURE 4.11. On-device validation of highlighted solutions from Figures 4.5, 4.7 and 4.8; this is the non-amplified code. The median of the test overhead is 44 joules.

In order to better quantify our improvements, we measured the test framework's overhead by running the test suite with an empty advance function. It was found that the overhead amounts to 66% of the default configuration runs (based on the median of each set of the 31 runs of each setup). After deducting it from the results in Figure 4.11, the actual improvements in the energy efficiency of running Rebound's test cases using the found configurations range between 7-22% (based on the medians) across the seven shown solutions. We conjecture that Rebound's developers might not be aware of such an energy improvement at the expense of a slight deviation from the functional requirement.

Finally, we use a Nexus 6 phone running Android 6 to check if the evolved solutions can improve the energy efficiency on another device other than the Nexus 9. In terms of hardware specifications, both devices are drastically different. For example, The Nexus 6 is powered by a 2.7 GHz quad-core Snapdragon 805 processor with 3 GB of RAM, where as the Nexus 9 has 2 GB of memory and its system chip is NVIDIA Tegra K1 with a 2.3 GHz dual-core Denver CPU.

Figure 4.12 shows the results of validating the marked solutions from the optimisation experiments and the original configuration on Nexus 6 using the same settings mentioned earlier in this section. As can be seen, interestingly, the overall trend is similar to the results found on the optimisation platform (i.e. Nexus 9) though the two devices have different hardware specifications. In addition, the jiffy 1 variant still uses the lowest amount of charge among the validated solutions.

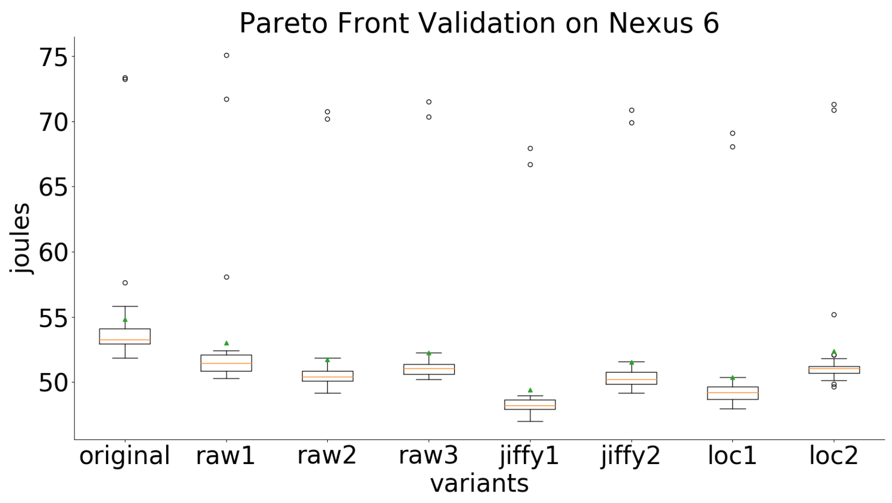

FIGURE 4.12. Validation of solutions marked solutions from the optimisation experiments on Nexus 6.

4.5.5 Statistical Analysis

We complement our validation study with statistical analyses due to the noisy environment in which the measurements were taken. This is important to help us to draw valid conclusions and to show that the found solutions have statistically significant improvements over the original configuration in terms of energy consumption.

Normality Test We use a rigorous statistical technique to study whether the energy use of the validated solutions are normally distributed. This is essential to determine which statistical test to use in order to show whether there is a significant difference between the original configuration and the improved solutions. The statistical technique is the Shapiro-Wilk test [143] which tests if a sample comes from a normal distribution with unknown mean and variance, against the alternative that it does not come from a normal distribution. This test has been rigorously evaluated in the literature [8, 80, 109], and found to be more conservative than other statistical tests for normality such as Kolmogorov-Smimov and Lilliefors tests.

Interestingly, the test on the validation result obtained from Nexus 9 revealed that five out of the eight configurations are non-normally distributed with 95% confidence level. The normally distributed solutions are *raw 2, jiffy 2 and loc 2*. It is worth mentioning that using Kolmogorov-Smimov test showed that all configurations have non-normal distributions.

Statistical Difference To determine whether the difference between the amount of energy consumed by the original version and evolved variants of each subject is statistically significant, we use the right-tailed Wilcoxon rank-sum test [169] which is also known as Mann-Whitney test. This conservative non-parametric test does not make any assumptions about the distribution of the data sets. The alternative hypothesis states that the median of the original configuration is greater than the median of the improved variant. We chose Mann-Whitney-Wilcoxon test due to the observation of having non-normal distribution in most of our data sets.

Table 4.2 presents the results (p-value) of the conducted statistical test on the validation results from Nexus 9. As can be seen, all obtained p-value for each test are less than 0.001 (i.e. the required confidence is 99.99%), indicating the acceptance of the alternative hypothesis with high confidence. For instance, the p-values for the solutions *raw 1* and *jiffy 1* in comparison with the original configuration are $6.3 \cdot 10^{-5}$ and $1.4 \cdot 10^{-9}$, which we consider to be highly significant.

In addition, running the right-tailed Wilcoxon rank-sum test on Nexus 6 data indicates the difference (to the original energy usage) is statistically significant. For example, the (p-value is $1.41 \cdot 10^{-9}$) when *raw 2* is compared to the original configuration.

Effect Size Although the rank-sum test can examine the existence of a difference between the energy consumption of the original configuration and the optimised solutions, it cannot quantify the magnitude of such differences. We therefore use the Vargha and Delaney $\hat{A}_{12}$ effect size to measure the approximate differences between the original configuration energy use and the optimised solutions. This measure is non-parametric and calculates the proportional difference between two data sets [162]. It quantifies the difference in four ranges/thresholds for interpreting the effect size: 0.5 means no difference; up to 0.56 indicates a small difference; up to 0.64 indicates medium effect; over 0.71 is a large difference. This approach calculates the expected probability that solution 1 consumes less energy than solution 2. For instance, if $\hat{A}_{12} = 0.8$, then solution 1 is expected to consume less than solution 2 in 80% of the time.

Table 4.2 shows the effect size of the differences between each optimisation solution and the original configuration computed using Vargha and Delaney. Interestingly, the difference is large in all instances. For example, the minimum difference is between the original configuration and *raw 1* where $\hat{A}_{12}$ is equal to 0.78.

	p-value	effect size $\hat{A}_{1n}$	magnitude
original vs. raw 1	6.3e-05	0.78	
original vs. raw 2	9.4e-10	0.94	
original vs. raw 3	1.5e-06	0.84	
original vs. jiffy 1	1.4e-09	0.93	large
original vs. jiffy 2	1.7e-10	0.96	
original vs. LOC 1	1.2e-09	0.93	
original vs. LOC 2	5.3e-07	0.86	

TABLE 4.2. The results of the statistical analysis of the validation experiment conducted on Nexus 9. The p-value obtained using Mann-Whitney-Wilcoxon test and the effect size was calculated using Vargha and Delaney.

4.6 Conclusions

The optimisation of non-functional properties of applications is of increasing interest: while developers generally lack the skill, search-based software engineering can assist with an automated approach. When it comes to minimising the consumption of energy, one has to deal with noisy sensors, huge and complex search spaces, and long evaluation times.

In this chapter, we demonstrated that it is possible to detect small changes in energy consumption using code rewriting. This was required to explore the configuration space of an Android physics library. To speed up the optimisation process, we created models based on run-time and lines-of-code, which were sufficiently precise to guide the optimisation. The former still requires variants to be run on the device, however, it no longer requires us to connect and disconnect the device for configuration evaluations and recharging which drastically reduces the experiment run-time. The latter model can run entirely on the computer and thus is significantly faster than the other two optimisation approaches.

The results show that substantial energy savings of up to 22% can be achieved for our target application (after deducting the test overhead), at comparatively little deviation from the functional requirement. Such small changes do not drastically affect the animation (i.e. the use experience).

In the upcoming chapter, we proceed exploring ways to deal with the problem of noisy and expensive fitness function using a different approach. We base our methodology on re-sampling and statistical tests for finding when to stop re-sampling. We also, uncover more interesting hidden problems waiting to be solved, but this time among different smartphones.

Chapter 5

Mind the Gap – a Distributed Framework for Enabling Energy Optimisation on Modern Smart-Phones in the Presence of Noise, Drift, and Statistical Insignicance

"In our lust for measurement, we frequently measure that which we can rather than that which we wish to measure... and forget that there is a difference."

George Udny Yule

5.1 Introduction

In the previous chapter, we have implemented a new approach to alleviate the severity of induced noise in energy readings, and created energy estimation models using executed line of code and CPU utilisation. Then, we have conducted in-vivo and model-based optimisation experiments, and validated our results on the platform used for optimisation and on another platform. In this chapter, we characterise several platforms for energy related experiments and present a new conceptual framework for determining the minimum number of samples required to establish a statistical significant difference between two competing software variants. This chapter has been published in the IEEE Congress on Evolutionary Computation in 2019 [18]

We implement a new method to overcome noisy energy measurements obtained from smart devices' internal fuel gauges. We then create energy estimation models based on the CPU utilisation and line of code. We also run *in-vivo* and model-based optimisation experiments. We validate our results on the target platform (i.e. used for optimisation) and on a different platform.

In order to optimise software energy consumption, the optimiser needs energy measurements. The gold standard for obtaining energy measurements is external meters. These directly and accurately measure the electrical current drawn by smartphones. They can sample at up to 125kHz with only 0.7% estimation error [136]. External meters have been successfully applied to energy optimisation on mobile devices [24, 111, 179]. However, as meter accuracy increases, the price of the meter also increases. Moreover, setting up an external meter involves a complicated process, requiring specialised technical skills since practitioners need to open the device to attach the meter to its battery [62]. This factor makes it infeasible for most software developers to use high-standard external meters.

Another measurement solution is the use of energy models. There are several techniques under this umbrella. Models can estimate the energy use by utilising the hardware (HW) and software components' counters [76, 177]. Other works extend this methodology to include the different states induced by HW components and system calls [121, 176]. The last category utilises the executed line of codes (LOC) for energy estimation [57, 95]. Energy models are less accurate compared to the hardware meter but more convenient and accessible to practitioners.

However, the dilemma here is which model to use? Energy models based on HW utilisation do not capture all factors that contribute to the energy consumption. For instance, these models do not consider the states of network interface controllers (NICs) during data transmission. State and system call models cannot be easily applied when HW manufactures do not publish the relevant information about their products [121]. LOC based models, while conceptually simple, are difficult to maintain. There are thousands of APIs in the Android SDK, and they evolve rapidly at rate of 115 API updates per month [108]. Moreover, the energy use profile of software is very platform dependent and can vary in complex, state-dependent and even individual-device-dependent ways that are very challenging to model [24, 62, 66].

A solution that alleviates these issues is to utilise the internal meter of smart-devices and compare the energy consumption of software variants *in-vivo* as has been shown in the previous chapters. In contrast to external meters, internal meters do not require advanced electrical engineering skills and are accessible to software practitioners. Internal meters are less accurate compared to external meters, however, they can be precise [15, 42] and can be an adequate substitute to obtain real measurements for optimisation purposes, if the measurement periods are sufficiently long [15]. Battery readings from the internal meter are easy to access through the battery APIs, such as BatteryManager in the Android SDK [50]. In addition, internal meters do not suffer from the inherent inaccuracy of models – they show what software variants actually use at a particular moment.

A common approach to coping with noise in fitness evaluations is re-sampling [25, 29, 40, 45, 69, 110, 119, 126, 146, 147, 156, 178]. When an optimisation process uses re-sampling, variants are run multiple times with the number of runs (samples) determined by some function of an estimation of system noise. In work to date, re-sampling strategies have assumed that noise can be characterised by an underlying normal distribution.

Contributions. In this chapter we present a conceptual framework based on tournaments which we use to compare a range of test workloads on different combinations of phones and operating systems. We explore the problem of running a tournament to compare the energy use of two variants of a software application on a mobile platform by using that platform's internal meter. We explore the distributions of measured energy produced by known workloads on different platforms in different battery states. We demonstrate that these distributions are often complex and, none so far, conform to normal distributions. From these observed distributions we then infer the minimum number of consecutive samples required to rank two variants to a given degree of confidence at a given battery state. We show that the number of samples required not only depends on the difference between the running times of each variant, but also on the battery state, and, very importantly, the HW/OS combination on the platform. We use this information to propose a number of re-samples for each platform and battery state. Finally, we run a simple, proof-of-concept optimisation experiment on a simple benchmark to compare our re-sampling strategy to two other published strategies. These early experiments produce promising indications of the potential effectiveness of our approach.

The rest of the chapter is structured as follows. After putting our work into the context of existing work in the next section, we discuss an example to motivate the problem in Section 5.3. We then present our framework in Section 5.4, and describe our methodology and its application on a case study in Section 5.5. Section 5.6 describes the energy optimisation experiment and its results. Finally, we present our conclusions in Section 5.7.

5.2 Related work on re-sampling

Optimisation processes for real systems have to be robust to noise in fitness evaluations of individuals. There are three basic approaches to coping with noise in evolutionary optimisation. The first approach is implicit-averaging [58]. Under this approach, we simply rely on a large population of individual variants to contain similar individuals. When very similar individuals are evaluated, we are approximating a re-sampling [134, 146]. One drawback of this approach is the requirement for large populations which are not easily supported when fitness evaluations are expensive.

A second, more direct, approach is simply to re-sample a fixed number of times according to some characterisation of noise [134]. A drawback of this approach is that it is not able to adapt the re-sampling rate according to circumstances as an optimisation run progresses. For example, changes in system state during optimisation may induce more or less noise [13].

The third strategy, dynamic sampling (DS) addresses this problem by allowing the re-sampling strategy to adapt during optimisation. A simple DS technique is time-based sampling (TBS) [45, 147]. Under TBS the re-sampling rate increases as optimisation progresses based on the observation that individuals in a population approach similar levels of fitness in later generations. Confidence-based DS [156] determines the current re-sampling rate based on a Welch confidence interval test [90]. Under this scheme individuals are re-sampled a minimum number of times to calculate a value for mean and standard deviation of an assumed underlying normal distribution. A Welch test is then applied to these parameters and if a significant difference (to a user-specified level) is not found then the individuals are re-sampled. This re-sampling is repeated until a significant difference is detected or a predefined maximum number of samples is reached. Cantu-Pazcite [29] used a one-sided t-test to decide between two candidate solutions in a tournament. After the initial sampling, the solution with the highest variance is re-sampled one more time and the t-test is run to check for a significant statistical difference. As with Confidence-based DS this sampling is repeated until the desired p-value is reached or the maximum sample number is reached. Other work [69, 110], determines the number of re-samples using the difference between the solutions' fitness value δ, the environmental noise σ and known properties of Gaussian distributions. In Gaussian distributions, 95.4% of samples fall in an interval whose width is 4σ. In case the distance between the two solutions is greater than 2σ, it is likely that the fitter solution is truly the best performing in the tournament. On the other hand, having a small δ means the two distributions overlap and therefore more samples are required. This is done by calculating the normalised measure of the overlap v and the critical value of Gaussian distribution with a confidence level of 0.95.

Di Pietro et al. [40, 126] used the Standard Error (SE) to decide between solutions by re-sampling until SE estimates converged. These estimates are then used to characterise noise which is then used, to determine the re-sampling rate. More broadly, re-sampling has also been applied to multi-objective optimisation problems [25, 119]. For more works on handling noise in optimisation, we refer interested readers to [134, 146].

The main shortcoming of the static and dynamic re-sampling techniques described above is the assumption that the noise is normally distributed which does not hold in smart-phones environments. In addition, the above works used fast artificial problem in their experiments, which allowed for high evaluation budget unlike real environments where the budget is restricted.

5.3 A Motivating Example

Like all measurements of real systems, measurements of energy consumption using the internal meter are subject to noise. As an example of this noise Figure 5.1 shows repeated measures of energy consumption, as measured by the internal meter, for the same variant of the Rebound library described in previous chapters running on a Nexus 6 using Android 7. In particular, the only difference between the two repetitions (one orange and one blue) was that the battery was recharged between the two repetitions. As can be seen, the measurements exhibit significant random and systematic noise. Such noise can be created by a variety of factors including battery state, system state, and temperature, among others[1]. This noise has the potential to impact on the accuracy of fitness evaluations performed during comparisons of program variants which, in-turn, can affect any optimisation process informed by these comparisons.

[1]Chapter 6 shows how system states can cause a drastic increase in the measurements similar to the initial chunk of the presented datasets in Figure 5.1.

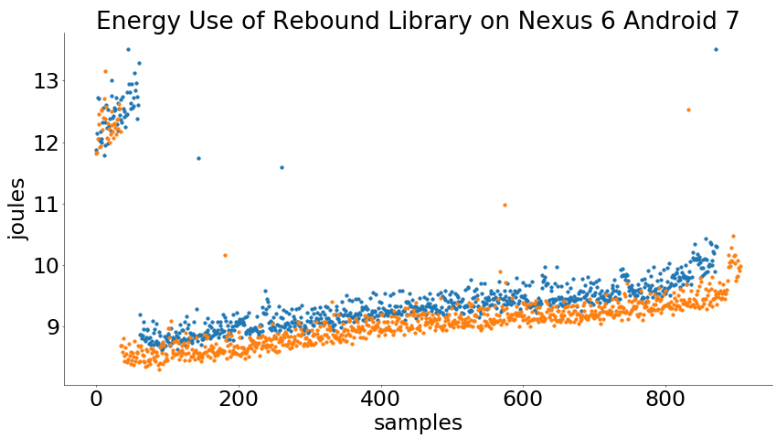

FIGURE 5.1. Energy use of repeated runs of the original Rebound library on Nexus 6 running Android 7. Setup: Rebound was run about 900 times (blue), then the device was recharged, then Rebound was run again about 900 times. Samples are in chronological order.

To formally characterise the distribution of the samples (here: to test for normality) we used the Shapiro Wilk normality test [143] described in Chapter 4. Table 5.1 presents the results (p-values) of the test on five segments (every 20%) of the collected data shown in Figure 5.1. As can be seen, none of these sub-populations conforms to a normal distribution, regardless of the number of the used segment. The number of samples in each segment are 174 and 181 for run 1 and run 2, respectively. Although we alleviated the impact of the unexpected increase in the energy use over time by partitioning the samples into sub-populations, the resulted distributions are not normal.[2]

[2]It is worth mentioning that we used different statistical tests such as Kolmogorov-Smirnov, Jarque-Bera and Liliforse tests and all reported similar results to Shapiro Wilk (i.e. *p-value* < 0.05) but with different magnitudes.

Battery Level	run 1	run 2
100% to 80%	2.0e-17	6.5e-21
80% to 60%	1.2e-20	9.0e-13
60% to 40%	1.0e-4	5.0e-4
40% to 20%	3.7e-5	1.3e-19
20% to 10%	5.6e-20	2.3e-16
All samples	1.4e-87	1.0e-3

TABLE 5.1. Shapiro Wilk normality test *p-value* on run 1 and run 2 from Figure 5.1

We further conducted the statistical test on every 30-samples subset of the collected data. Figure 5.2 shows on the y-axis the *p-values* of the test on log scale, the subsets of size 30 on x-axis, and the red horizontal line for the *p-value* of 0.05 which indicates a typical confidence threshold. Interestingly, even after dividing the data into small subsets that are less affected by the unexpected increase over time, over two thirds of the subsets have non-normal distributions for both runs.

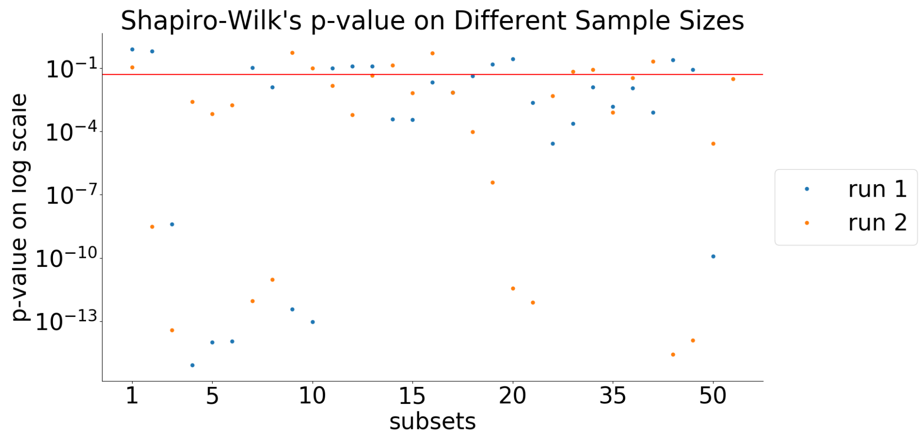

FIGURE 5.2. Shapiro Wilk normality test *p-value* on run 1 and run 2 from Figure 5.1 on every 30-samples subset of the collected data shown in Figure 5.1. The horizontal red line shows the boundary of *p-value* = 0.05.

As can be observed, the problem of energy consumption is different to those problems used in the literature in terms of probability distribution. This is due to complex interactions and different states of the system components, systematic noise and the upward drift of power consumption for the same variants as optimisation progresses. Moreover, even when variants are freshly compared in a tournament, at the same point of optimisation process, we *still* find that the distributions of sampled energy use are not normally distributed when using the Shapiro Walik and other normality tests. This lack of normality in noise distributions introduces the risk that existing static and dynamic resampling strategies will not perform as expected in this setting.

5.4 Phone Farm for Energy Measurements

One objective of our overall research project was to build a framework that can help researchers to measure mobile phone's energy consumption easily and accurately. Figure 5.3 shows the hardware setup of our framework in action. In the following, we present the high-level design of the framework. We begin with an outline of how the framework can be applied from a user's perspective. After that, we briefly outline how we control the hardware and which hardware we use in our study.

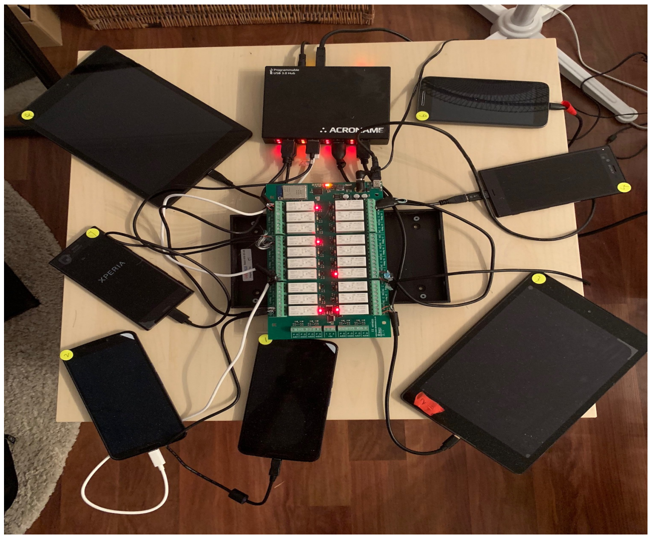

FIGURE 5.3. Phone farm in action, multiple experiments running in parallel. The relay in the centre is responsible for physically disconnecting the phones from the charger. Screens are off to minimise noise from the screen's consumption, and wake-locks are used to prevent the device from entering various standby modes.

5.4.1 Operation

Our framework supports a number of modes of operation. In the most complex mode, the user can interact with the system as follows:

1. A user issues multiple commands to measure multiple applications in multiple devices.
2. The framework queues and runs these applications in the devices and records the energy consumption using the internal meters.
3. The framework retrieve all results from these devices.

A core function of the framework that can be identified here is that it needs to be able to run an application, record the energy consumption and pull the result from a particular device.

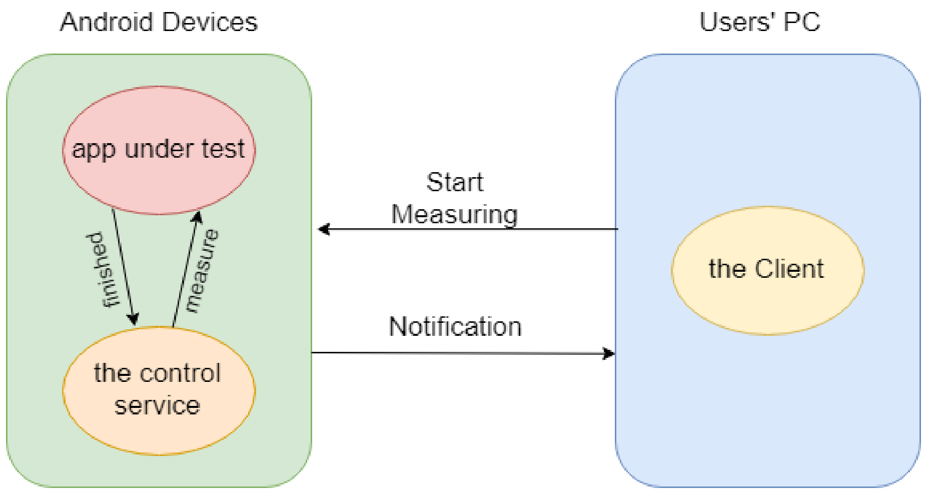

FIGURE 5.4. Main components in the framework and their interaction

In our framework, this is implemented as follows. In order for successful communication to happen, there are two services running in the PC and devices respectively, and a communication protocol for the two services. The service running in the PC is called “the client” and the service running in devices is called “the control service”. These two services, along with the “app under test”, consist of three main components of the framework. In general, the client is responsible for interacting with users and communicating with the control service; the control service is responsible for measuring energy consumption of the app under test and communicating with the client.

5.4.2 Controlling Android Devices

In order to reduce the noise from the system during the measurement of energy consumption, it is required that some aspects of Android devices be controlled. The aspects include charging, CPU frequency, screen brightness, network connection, air-plane mode etc. Control over these aspects is an important part of the framework, especially control over charging. To achieve as much control as possible, we follow the recommendations developed by Bokhari et al. [14, 16].

Control over charging is also essential for restoring battery capacity automatically. Therefore, whether control over charging can be achieved is a key factor that determines if the whole measurement process can be automated. In our implementation, we use the Wifi relay WIFI8020 from Devantech Limited [99] to physically disconnect the phones from the charging source. This has the advantage that both the client and the device can control this by sending simple `HTTP GET` requests.

5.4.3 Smartphones used

We use four different models of Android smartphones in our experiments, ranging from entry-level to top-of-the-class. The operating systems that we consider are Android 6, 7, and 8. In combination, these platforms cover 76.1% of Android's worldwide market share in December 2018 [3]. We list all devices in Table 5.2.

Device	Android	SoC	CPU	Memory
Xperia ZX	7 & 8	Snapdragon 820	Q-core (2x2.15 GHz Kryo & 2x1.6 GHz Kryo)	3 GB
Nexus 6	6 & 7	Snapdragon 805	Q-core 2.7 GHz Krait 450	3 GB
G4 Play	6 & 7	Snapdragon 410	Q-core 1.2 GHz Cortex-A53	2 GB
Nexus 9	6 & 7	Nvidia Tegra K1	D-core 2.3 GHz Denver	2 GB

TABLE 5.2. The list of the used devices and their specifications. The stock ROM is used in each device.

The devices vary significantly in their specifications. As we shall see later, this contributes – together with some "quirks" specific to certain OS versions – to an unexpectedly wide range of behaviours, even of the very same test application.

[3]Statcounter: Mobile & Tablet Android Version Market Share Worldwide, http://gs.statcounter.com/os-version-market-share/android/mobile-tablet/worldwide, visited on 25 January 2019.

To facilitate readability, we will from now on refer to phone-OS combinations just as "phones" or "devices", unless we want to explicitly highlight a particular combination.

5.5 Case Study

In our case study, we first characterise our devices for the use in tournaments for which we seek statistical significance. Then, we extract recommendations from our measurements which are then used in a proof-of-concept of on-device optimisation and in comparison with other methods for noisy environments.

5.5.1 Characterisation of Phones

The goal of this first part of the case study is to characterise an understandable base case in order to establish which phone-OS combinations we might be able to use later for optimisation purposes.

The experimental design is influenced by the internal meters in the Moto G, Nexus 6 and Nexus 9, which have a temporal resolution of 4Hz. Our goal is to investigate first whether it is possible to distinguish between processes that are identical in nature but slightly different in duration.

For this, we use a simple dummy load that we call BusyLoop app, as it just runs a busy loop for a pre-defined duration. The durations chosen are 10 seconds as the base case, 10.25 seconds and 10.50 seconds (being equivalent to the base case plus 1 or 2 samples of the 4Hz-energy meter), and 12 seconds. For each data collection, we start with a full battery (denoted as 100%) and repeatedly run BusyLoop until the battery has only 10% charge left.

We have to highlight one important detail of the implementation before we can continue. As shown in the motivating example in Section 1, the system can exhibit different energy consumption behaviours even when the only change in system state is a battery recharge. Our approach to mitigate this is as follows. To average out the effect of different states between different recharges and of varying rates of drift, we interleave the execution of different BusyLoop variants. This means, instead of running (10s, 10s, 10s, 10s, ...) in the first run (from 100% battery down to 10%), then running (10.25, 10.25s, 10.25s, ...) and so on, we rotate through the four configurations: first run (10s, 10.25s, 10.5s, 12s, 10s, 10.25s, ...), second run (10.25s, 10.5s, 12s, 10s, 10.25s, ...), third run (10.5s, 12s, 10s, 10.25s, ...) and so on. Then, by pulling apart and reassembling the actual measurements, we have exposed the measurement of each of the four configurations to the effects of different states as well as to varying drift speeds.

Based on these measurements, we perform two analyses. First, we simulate tournaments of varying length in which we are attempting to discriminate between variants, to a given level of statistical confidence, using the Wilcoxon ranksum test. Through this process we are able to estimate the minimum number of samples needed on a device to discriminate between two variants to a given confidence level. The use of the Wilcoxon ranksum test is important – this test does not assume normality in the underlying distribution.

As an example, we demonstrate the above process here using the Moto G4 Play. The raw results of running the four configurations are shown in the left column of Figures 5.5 for Android 6 and 5.6 for Android 7. It is interesting that the upgrade to Android 7 removed the initial phase of high energy consumption. However, additional higher and lower levels of energy consumption were introduced.

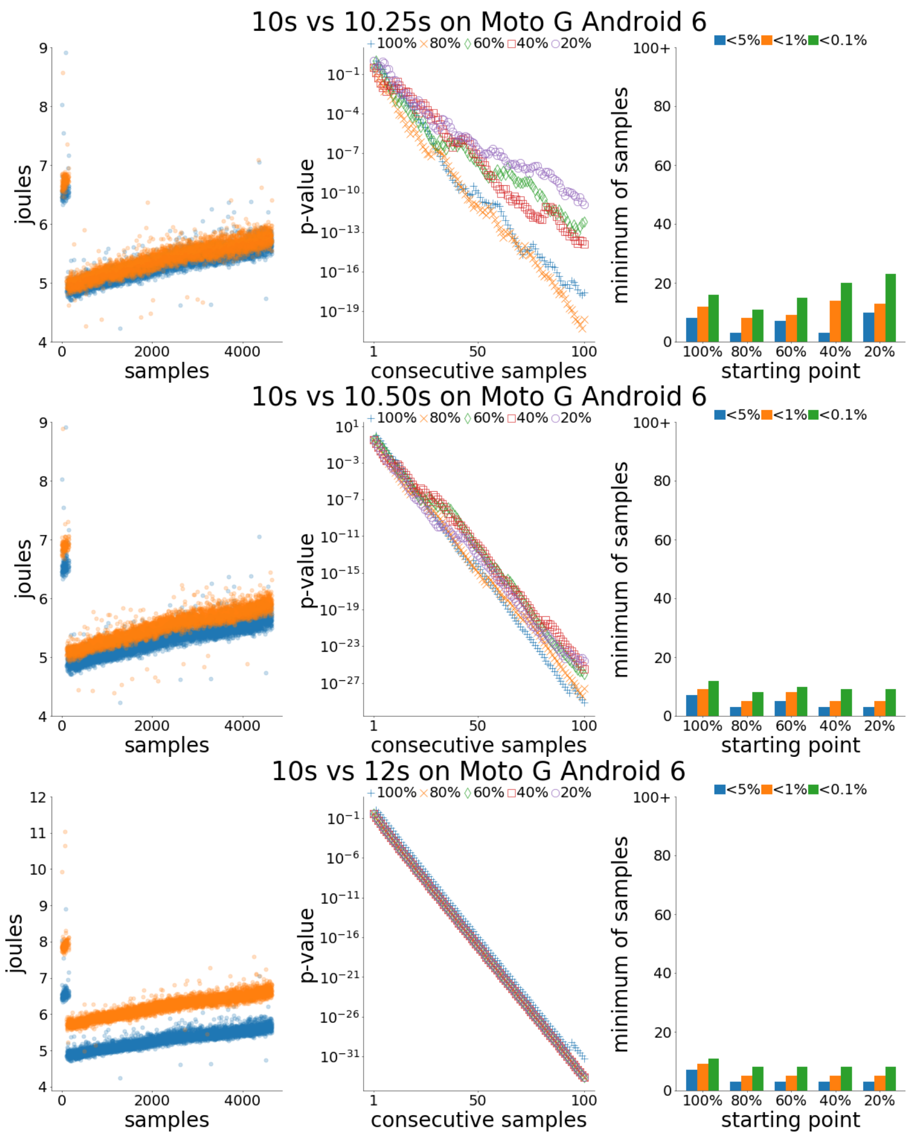

FIGURE 5.5. Energy Use of repeated runs of BusyLoop app on Moto G4 Play running Android 6. Blue: 10s, orange: as listed.

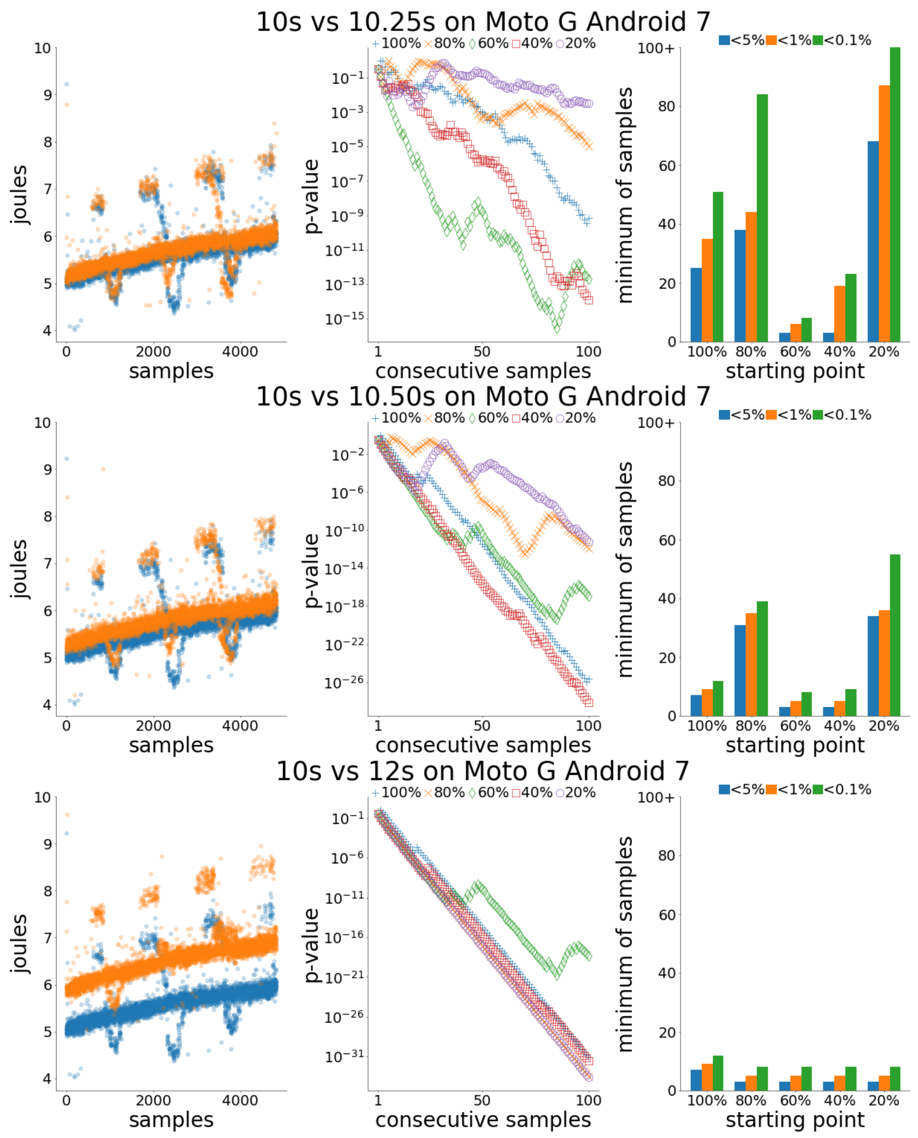

FIGURE 5.6. Energy Use of repeated runs of BusyLoop app on Moto G4 Play running Android 7. Blue: 10s, orange: as listed.

The middle column of Figures 5.5 and 5.6 show, as scatter-plots, the results of the Wilcoxon ranksum tests, simulated at different points in time and for various numbers of consecutive samples (from 1 to 100). These points in time are expressed by the percentage of battery charge used: 100% (full battery), 80%, 60%, 40% and 20% (nearly depleted battery). As expected, the outcomes become statistically more and more significant (p-values decreases) as the number of samples increases. Similarly, we can see that we need more samples to determine a significant difference if the difference in duration is small (i.e., in the case of 10s vs. 10.25s).

Lastly, the rightmost column of 5.5 and 5.6 attempt to extract an initial recommendation from the results of the statistical tests. These bar charts show, for each level of battery charge, and for each significance level the least number of samples needed from which on no test has returned a higher p-value than the required thresholds. This conservative estimate can be used to inform experimental settings, for example, in regression testing of software and also when running tournaments in a evolutionary process.

As can be observed, when the difference between the two app variants is small (i.e. 0.25 second), the high and low energy consumption introduced in Moto-Android7 have dramatically impacted the result of the test and consequently impacted the minimum sample size needed to detect statistical significance. For example, on Android 6 a sample size less than 20 is adequate to reach high significance level (i.e. 0.1%) at 100% and 80%, whereas it has to be more than 20 samples on Android 7 to get a significance level at 5% threshold at the sample level of battery.

For the other three devices, the observations are as follows. The Nexus 6 (Figures 5.7 and 5.8) and the Nexus 9 (Figures 5.9 and 5.10) largely follow the trends observed for the Moto G4 Play, except they are missing the additional states that were introduced with Android 7. For the Nexus 6, we can see short burn-in phases with the energy consumption being roughly twice as high, and we observed this independent of the experiment and of the operating system version. For the Nexus 9, these burn-in phases are just a few samples long, but result in energy consumption observations that are a factor of 4 larger than subsequent ones.

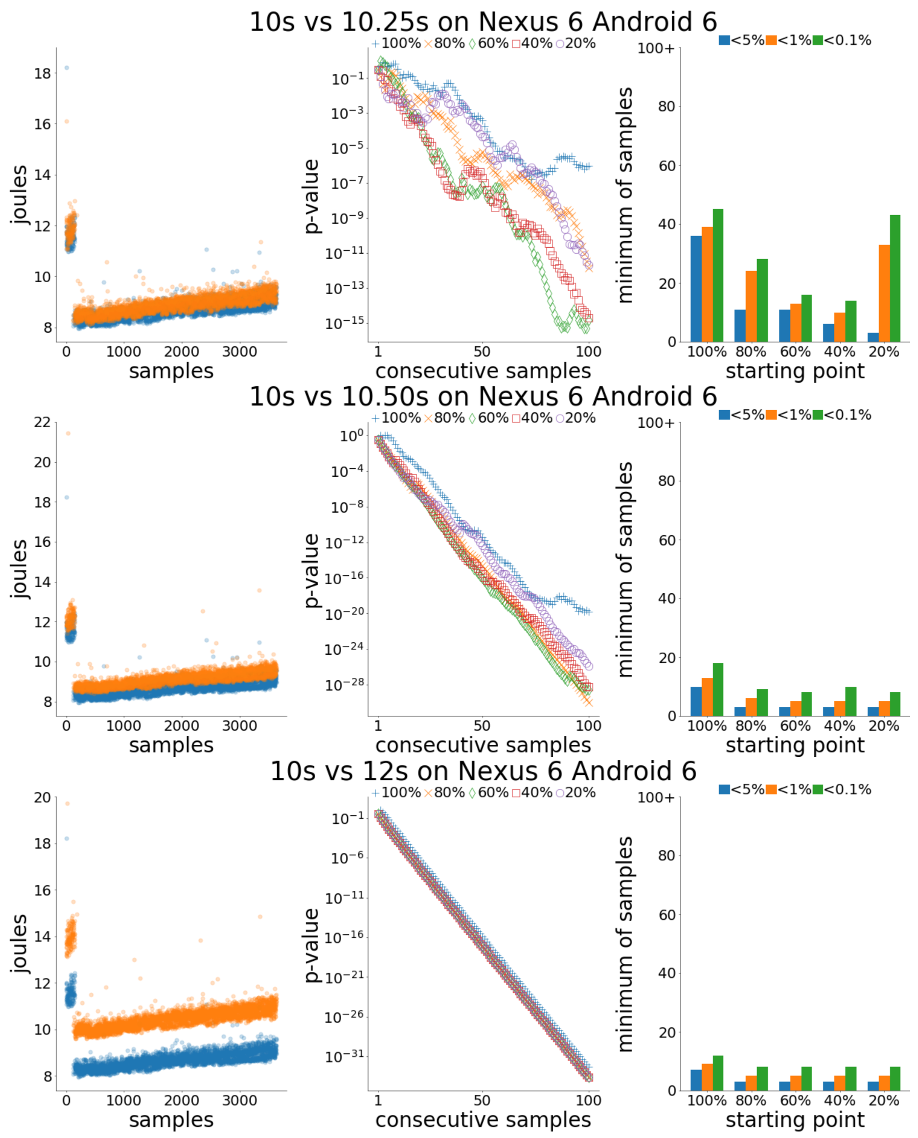

FIGURE 5.7. Energy Use of repeated runs of BusyLoop app on Nexus 6 running Android 6. Blue: 10s, orange: as listed.

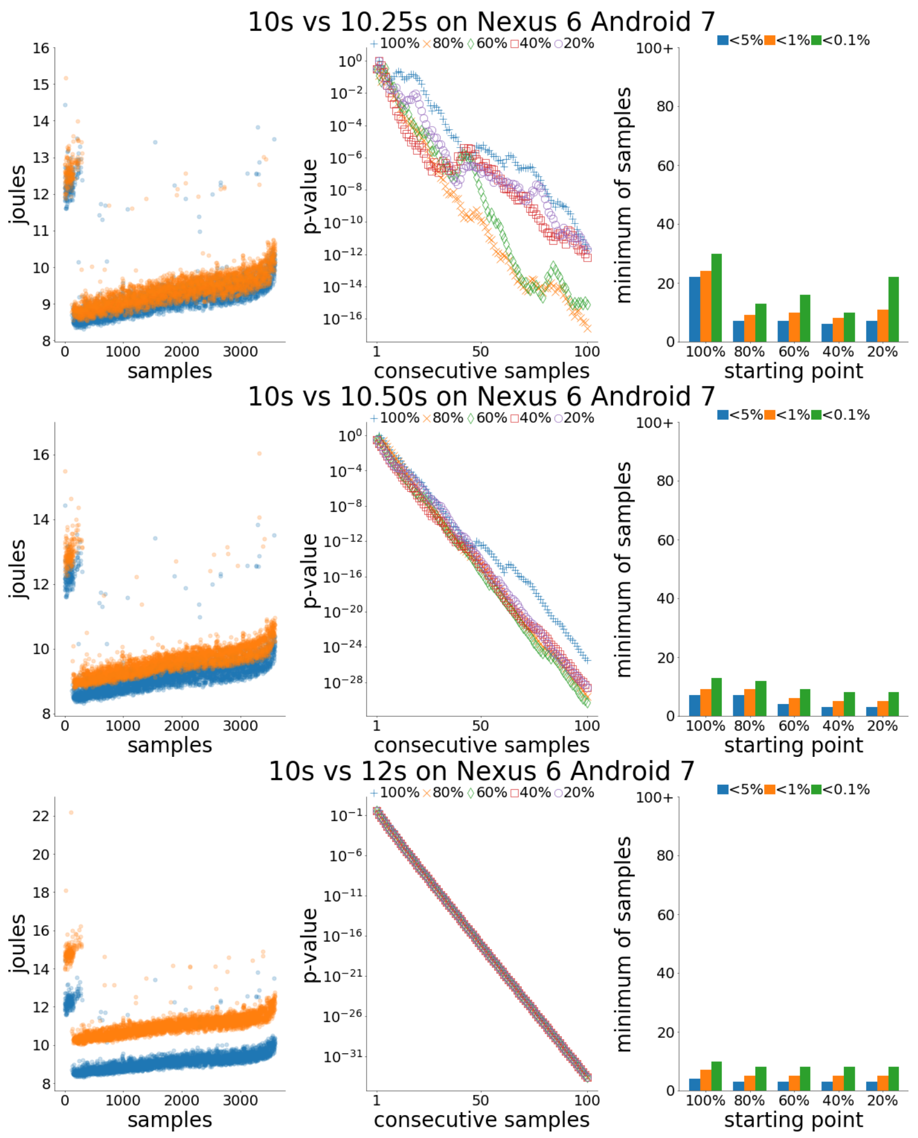

FIGURE 5.8. Energy Use of repeated runs of BusyLoop app on Nexus 6 running Android 7. Blue: 10s, orange: as listed.

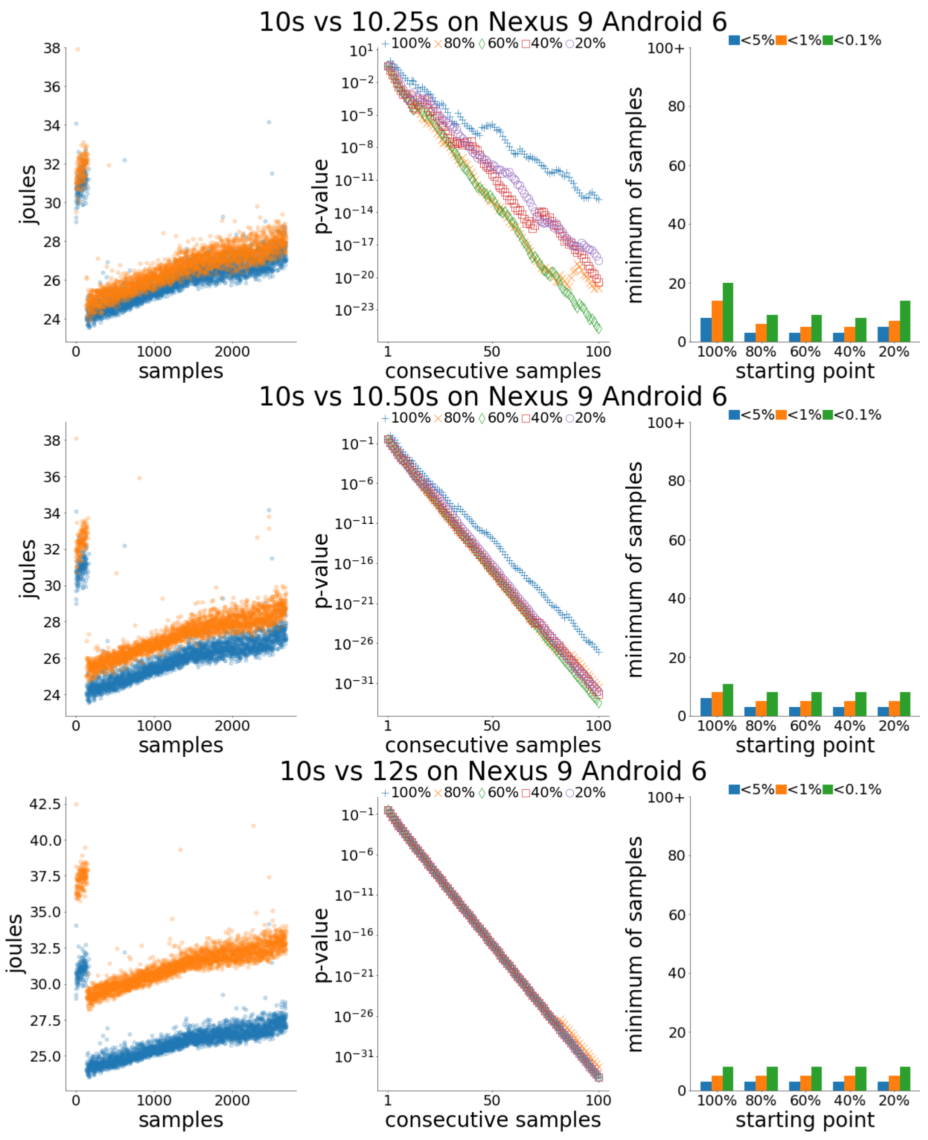

FIGURE 5.9. Energy Use of repeated runs of BusyLoop app for 10 vs. 12 seconds on Nexus 9 running Android 6. Blue: 10s, orange: as listed.

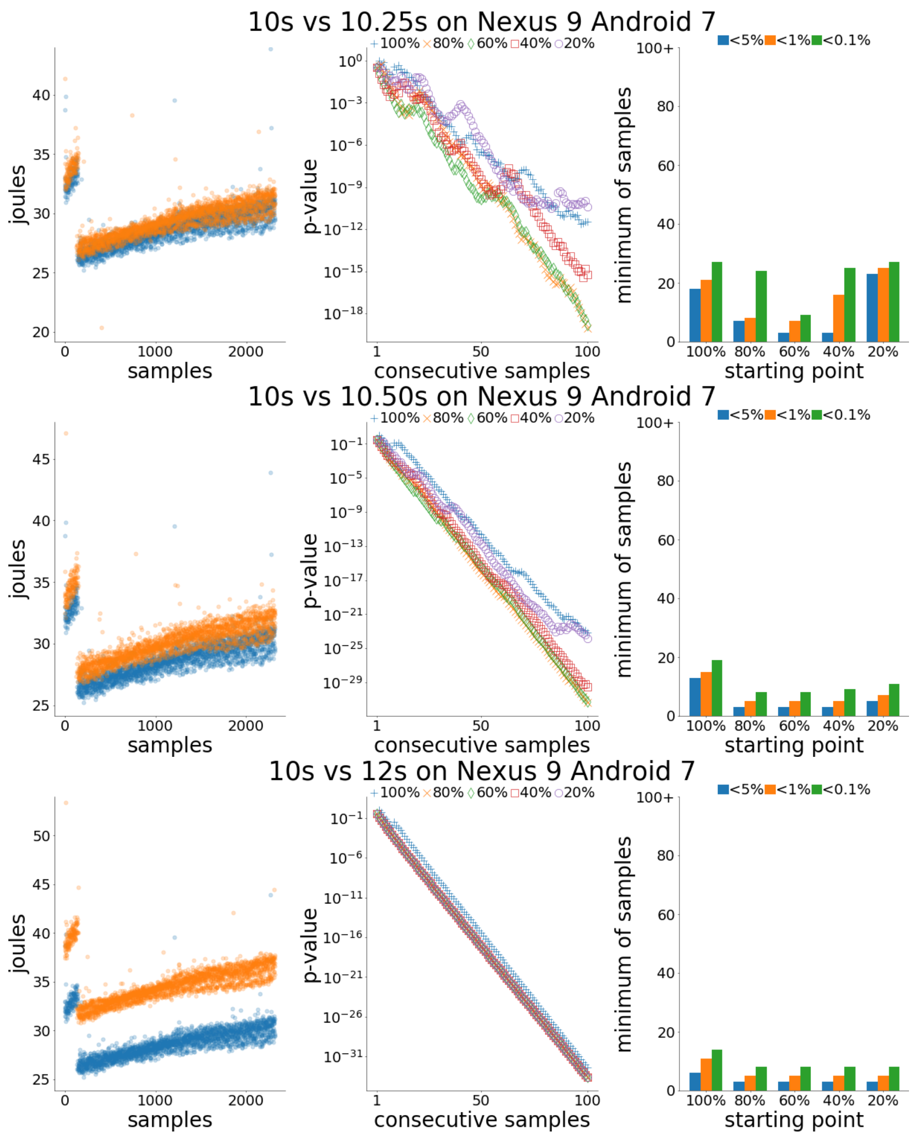

FIGURE 5.10. Energy Use of repeated runs of BusyLoop app on Nexus 9 running Android 7. Blue: 10s, orange: as listed.

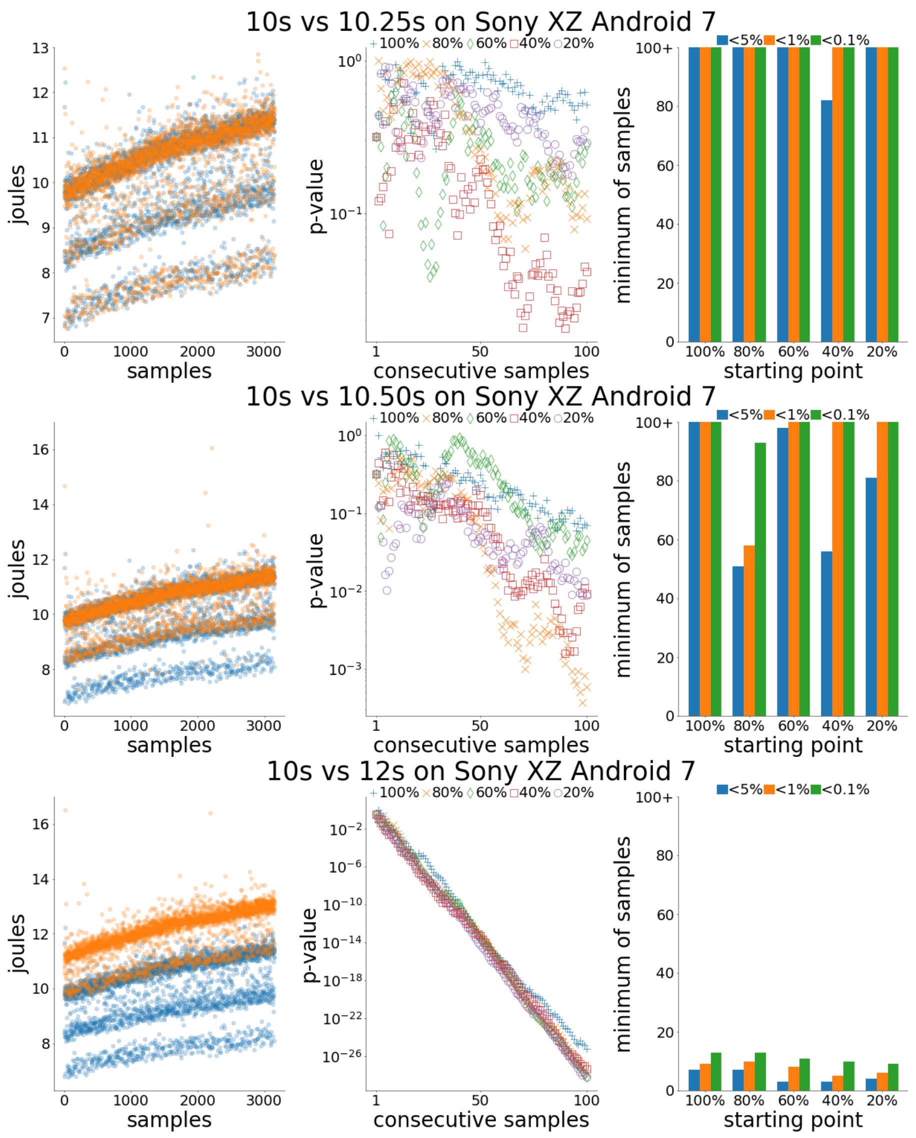

FIGURE 5.11. Energy Use of repeated runs of BusyLoop app on Sony XZ running Andoid 7. Blue: 10s, orange: as listed.

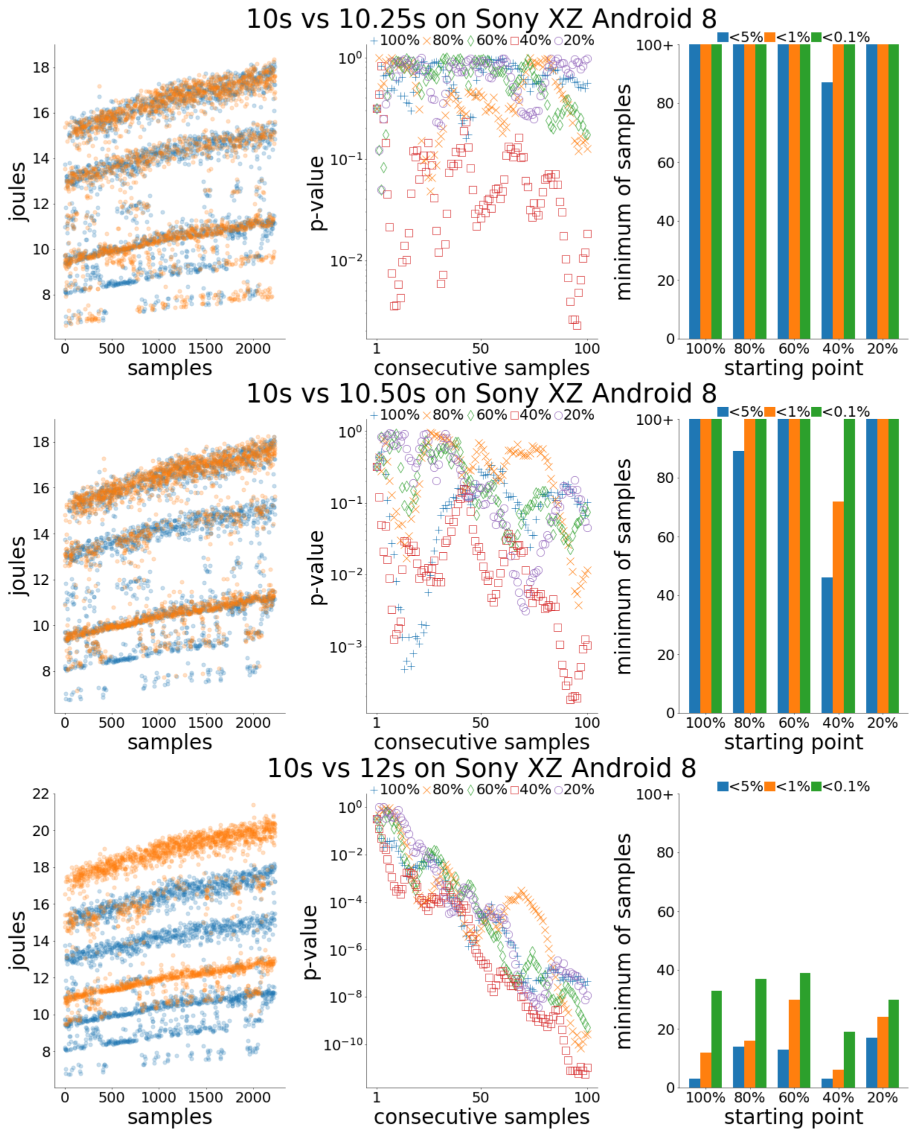

FIGURE 5.12. Energy Use of repeated runs of the BusyLoop app on Sony XZ running Android 8. Blue: 10s, orange: as listed.

The Sony XZ behaves quite differently as it shown in Figures 5.11 and 5.12. Although we employed the very same steps to attempt to control the system, we can observe at least six different levels of energy consumption of the system – despite our test application being just a simple busy loop. Essentially, we cannot make statements about statistical significance when the runtime difference is less than 5%, even when repeating a 10 second experiment 100 times.

5.5.2 Extracting Recommendations

Table 5.3 shows the results of running the BusyLoop variants on a Moto G4 Play running Android 7 with a full battery (i.e., starting at 100%) and significance level 5%. For each combination of the four variants, the table lists the minimum number of samples needed to distinguish two variants at the desired confidence level. This shows that the smaller the difference between variants the more samples are required. In this study, for each significance level of 5%, 1% and 0.1% and each battery level range, a 4-by-4 table is generated for each device-OS combination.

	BusyLoop duration			
	10s	10.25s	10.50s	12s
10s		25	7	7
10.25s	25		8	7
10.50s	7	8		3
12s	7	7	3	

TABLE 5.3. Comparing the four BusyLoop variants in pairs with starting point 100% (fully charged) and significance level 5.0%.

Having large numbers of re-samples such as 25 is expensive in real-world applications and can slow down the search process. Therefore, we minimise the number of samples further by taking the *median* of required samples taken from the variants at each battery level range for each significance level. The choice of the median is based on the assumption that the variants sampled at a given battery level are in some sense representative of the population that will be seen during optimisation. If the sampled variants are representative then the re-sampling estimate will be high enough approximately 50% of the time. Ultimately, the relative rank of the variant chosen to determine the re-sampling will need to be tested and calibrated empirically in optimisation experiments.

Table 5.4 shows the median of the minimum number of samples needed for the desired significance level at every battery range. Each element in the table represents the median of samples found in the 4-by-4 corresponding table. For example, when running a tournament between two variants in battery range 100% to 80% and confidence level of 5% is required, it is recommended to sample each solution 7 times. As can be seen, 7 is the median of the element in Table 5.3.

Android 7

		starting point				
		100%	80%	60%	40%	20%
significance level	5%	7.0	15.0	3.5	3.0	18.5
	1%	9.0	20.0	5.5	5.0	20.5
	0.1%	13.0	24.5	18.0	8.5	31.5

TABLE 5.4. Recommended number of samples based on the BusyLoop case study. Set up is the Moto G4 running Android 7. Entries are the recommended minimum number of samples for different battery charge levels points and the desired significance level.

5.6 Optimisation Experiment

In this section we report on an optimisation experiment that serves as a proof-of-concept. We use an adaptive 1+1 Evolutionary Strategy (ES) algorithm, the mutation operator is uncorrelated mutation with one step size described in [44]. ES is a very simple evolutionary algorithm that mutates a copy of a single individual and replaces the individual with the copy if the mutated version is better. For comparison, we run optimisation with our recommendation-table based dynamic sampling (*table-based DS*) and compare to the re-sampling techniques described in [156] (*confidence-based DS*) and [69] (*2-sigma DS*) described in Section 5.2). For our *table-based DS*, we accept the newly created solution if the statistical test either favours it or if no significant difference to the parent can be determined, based on the number of samples given in the lookup table shown in table 5.5.

		starting point				
		0%	20%	40%	60%	80%
significance level	5%	4.0	3.0	3.5	3.0	3.0
	1%	9.0	5.0	5.5	5.0	5.0
	0.1%	12.5	8.5	8.5	8.0	8.0

TABLE 5.5. Recommendation table for Nexus 6 running Android 7.

We used our framework described in Chapter 3 to run the optimisation experiments on a Nexus 6 running Android 7. We generated the table prescribing re-sampling rates using the process described in Section 5.5.2. The experimental runs took approximately 4 hours and were started with a full battery and we stopped when reaching a battery level of 10%. The target for optimisation was the busy-loop app – a very simple benchmark where the current best loop iteration count is mutated according to a normal distribution with $\sigma = 0.25$ seconds. The duration of the starting variant was 45 seconds. It is important to note that this experiment is designed to explore the feasibility of different re-sampling approaches in this optimisation environment, rather than to definitively compare re-sampling techniques (which would require a larger study).

Figures 5.13 and 5.14 illustrate the results of applying each re-sampling technique to optimisation of the busy-loop benchmark for equivalent four-hour runs. As can be seen, the *table-based DS* technique generated considerably more variants than *2-sigma DS* and *confidence-based DS* techniques. The *2-sigma DS* re-sampled solutions 18 times on average which considerably slowed its progress. In fact, since the distribution of the generated solutions were not normal and strongly overlapped, *2-sigma DS* unrealistically requested up-to 4639 samples per solution in one of the tournaments. However, the upper limit is set to 30 samples per solution [69]. This shows that demanding the difference between two solutions to be at least 2σ to tell them apart is costly in this environment. For *confidence-based DS*, solutions were sampled 5 times on average which is the upper limit of the algorithm [156]. Again, *confidence-based DS* assumes that samples are taken from a normal distribution, a condition not met in this setting.

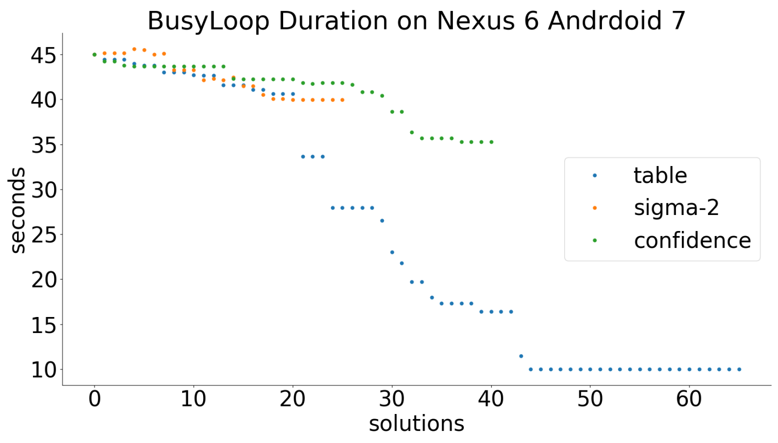

FIGURE 5.13. Solutions' duration of busy loop optimisation experiment, shown are tournament winners. The duration equals the value of the decision variable.

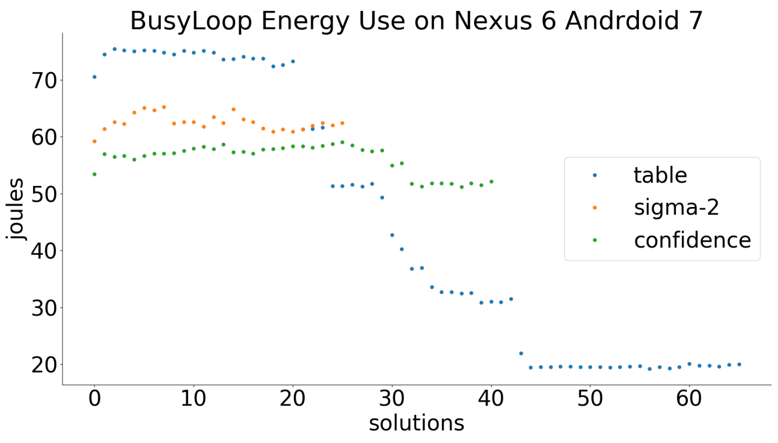

FIGURE 5.14. Solution fitness values of busy loop optimisation experiment. The measurements are the ones returned by the internal meter for the solutions shown in Figure 5.13 and thus exhibit a slight upwards trend due to sensor drift.

In contrast, the number of samples were relatively small in *table-based DS* technique at most 4 samples per solution across the entire optimisation, which, in turn, gave the search process enough time to evaluate more solutions. In addition, the use of more conservative statistical test (i.e. Wilcoxon ranksum) helped distinguishing between solutions. It should be noted that, especially, toward the end of the optimisation process, *table-based DS* produced some large outliers in both execution time and energy consumption. The presence of these outliers, being over-estimates, did not impact on optimisation. However, more experiments in different settings would be needed to verify if other optimisations are similarly robust to such outliers.

5.7 Conclusions

In this chapter we have examined re-sampling in tournaments in the context of *in-vivo* optimisation of energy use on mobile platforms. We have derived distributions of the number of samples required in two-way tournaments on a variety of mobile platforms. We have developed a process for generating preliminary recommendations for re-sampling strategies for each platform and we have applied these recommendations in a proof-of-concept experiment with promising results.

Currently these recommendations are derived from only one benchmark application. However, applications with similar usage profiles of CPU and memory generate similar recommendations. Further study is required to determine the extent to which these recommendation tables can be reused across different optimisation targets.

Future work will focus on carefully measuring the relationship between re-sampling recommendations and optimisation settings across a variety of devices. We also propose to examine how changes in platform state impact on the effectiveness of re-sampling recommendations.

As it was discussed in this chapter, the energy use of software are drastically impacted by the system version, states and the hardware platform. This introduces a threat to validity of experimental outcomes when optimising the energy use of software and therefore can mislead the researchers. In the next chapter we propose and implement a rigorous method to validate such results.

Chapter 6

Towards Rigorous Validation of Energy Optimisation Experiments

"A man's mind may be likened to a garden, which may be intelligently cultivated or allowed to run wild; but whether cultivated or neglected, it must, and will, bring forth. If no useful seeds are put into it, then an abundance of useless weed seeds will fall therein, and will continue to produce their kind."

James Allen

6.1 Introduction

In previous chapters we explored different methodologies for sampling energy use of variants for the purposes of optimisation. In this chapter, we implement a new validation approach for energy optimisation outcomes. In addition, we compare our approach against the traditional validation methods used in the literature. This chapter has been published in the Genetic and Evolutionary Computation Conference in 2020 [19].

In many applications of evolutionary search the fitness functions used during search are noisy and approximate. A key way to address these limitations is to add a validation stage to the end of the search process to more thoroughly evaluate and rank the best individuals produced by the initial search process [22]. One field in which validation is of increasing importance is the application of evolutionary search to improve the non-functional properties of programs. Examples of such properties include program execution time and energy use. Observed measurements of such properties arise from an interaction between the target program and the time-variant system state of its host platform. Unfortunately, as platforms become more complex, heterogeneous, and adaptive, it becomes much more difficult to model and compensate for changing system states. In turn, these complex interactions make it all too easy for validation runs of program variants to give misleading results.

Contribution. In this chapter we explore the problem of validating solutions from evolutionary processes aimed at optimising energy consumption on mobile platforms. This application domain is especially challenging for software optimisation, due to factors including: diversity of platforms, system configurations, and highly complex and adaptive system states. We show how these system features can combine to thwart current published approaches to validation runs of programs evolved for improved energy and execution speed. We do this by first characterising the time-variant effect of system state on a program run on multiple platforms. We show how, in spite of reasonable efforts to achieve uniformity of system state, successive experiments exhibit large and unpredictable variations in key system features. Finally, we propose a simple approach to the scheduling of the running of program variants, called rotated-round-robin (R3-VALIDATION) that increases the chance of variants being run in the same *set* of system states. We measure how effective it is in allowing variants to sample similar system states and, thus, enabling more specificity and sensitivity in comparisons of software variants. We compare it with current validation approaches on two different Android versions running on three different hardware platforms, and show that this measurement protocol aligns more precisely with actual platform behaviour.

In the remainder of this chapter, we first review literature related to system states and to validation approaches in Section 6.2. Then, we provide a horrifying motivating example in Section 6.3. To deal with these, we define a number of validation approaches in Section 6.4, including our R3-VALIDATION. Using a case study in Section 6.5, we observe and characterise system states, and we demonstrate how R3-VALIDATION mitigates the variations. In Section 6.6 and 6.7, we compare the validation approaches in terms of specificity and sensitivity. Finally, we discuss our approach in Section 6.8, and wrap up in Section 6.9.

6.2 Literature Review

Impact of System State on Non-Functional Properties The impact of highly-variable system state on measured program performance has been described in many works in the literature. Program execution time can be greatly affected by the state of cache memory [3], cache miss-rates, context switches [131], JIT compilation settings [48, 67, 137], memory layout [35], garbage collection policies [11, 67]; and OS environment variables [113], memory layout [35], garbage collection policies [11, 67] and OS environment variables [113]. Even when a platform is rebooted between runs there can be significant variation in benchmark runtimes [78]. Furthermore, the works in [77, 79] show that measurements of the benchmark operation are subject to inherent and external random changes to the benchmark operation, as well as the system state prior to execution. In this work we show that changes in system states between and during validation runs on mobile platforms can also impact energy measurements.

This variation in system state can affect the sensitivity and specificity of conclusions drawn from validation runs.

Current Validation Approaches Much existing work on using search to optimise non-functional properties of programs is described in [125]. We briefly summarise the approaches to validation described in this work and in more recent related publications. It should be noted that not all works included a validation stage to confirm the performance of solutions. Where validation was done, similarly to our work, it involved re-sampling individual program variants.

In work optimising energy use on mobile platforms, Schulte et al. [138] and Li et al. [97] used external meters with constant voltage. While external meters are precise they fail to capture different voltage states that applications encounter (and, potentially, trigger) when running on battery. This is significant because changes in voltage have an impact on energy use [158, 173]. Bokhari et al. [16] avoided this issue by using devices' internal meters, at the cost of noisier energy measurements [15].

An alternative approach to meters is to use energy models based on hardware (HW) counters [22, 26]. Although HW models produce cleaner signals, they are affected by the current system state as well as non-determinism in the virtual machine behaviour caused by factors such as garbage collection, thread scheduling and Just-In-Time optimisation [12]. In addition, such models abstract away at least some of the relevant environmental conditions and only capture the assumptions on which they were built.

Other work that relied on energy models has used simulation [168] or hybrid linear models based on small-scale energy sampling [101]. As with HW counter based models these models abstract away from at least some important details of system state.

There are also works that use the unmodified target platform for validation tests [16, 22, 26, 97, 138]. These works conducted multiple performance measurements and carried out non-parametric statistical tests and effect size measurements to achieve some confidence in the observations. One of these works, [22], revealed individuals that performed well only during optimisation – thus showing the potential value of validation. Other publications had less complete data with [97] reporting only the average of 10 samples. Likewise, the work in [138] reported the *p-values* of comparisons without naming the statistical test used and did not report the effect sizes from this test. Like most prior work we equip our R3-VALIDATION approach with non-parametric statistical analysis and effect size measure to compare between obtained solutions' samples.

Although most prior work applied rigorous statistical analyses, most work has not explicitly acknowledged or mitigated changing system states (see, e.g., [16, 22, 97, 101, 138, 168]). In an attempt to consider these, Nathan et al. [26] avoided some aspects of system noise by turning off the JIT and garbage collection Java systems whilst optimising for execution speed and reactivated these during validation runs. However, in both cases the execution times were modelled by vm-instruction counts which are somewhat decoupled from wall-clock time measurements.

A different approach to controlling the system was taken by Bruce et al. [24], who suggested running all validation experiments immediately after a system reboot to avoid the random initial state of the system. Unfortunately, on mobile platforms this protocol doesn't guarantee similar states, partly because, not all system services start immediately after reboot but also because the battery state of the system can trigger changes in system behaviour such as suspending background processes.

In summary, we propose that more progress needs to be made in rigorously validating optimisation results that come from running software on modern computing platforms. Our proposed R3-VALIDATION approach mitigates these issues by scheduling solution executions in similar system conditions, that allowing a fair comparison to be conducted.

6.3 A horrifying motivating example

We now illustrate how temporal changes in system state on Android platforms are reflected in measurements of energy consumption of even only a single program variant. We show that this changing system state can make a variant appear significantly different from itself even if the measurements take place between system reboots. But first, we introduce the software and hardware used throughout this chapter.

6.3.1 Target software and hardware

We use the optimisation outcomes produced in Chapter 4. In particular, we explored the use of the direct energy measurements (named "raw") and also the use of models based on jiffies (a time unit on Android phones) and executed lines-of-code (LOC). The experiment that used the raw energy measurements resulted in a Pareto front containing 10 solution variants named *raw1* to *raw10*. In the following, we show how system state affects the energy consumption of these variants and how this can impact on validation.

The target devices for our experiments are the HTC Nexus 9 and the Motorola Nexus 6. Both are equipped with the Maxim MAX17050 fuel gauge chip that compensates measurements for temperature, battery age and load [72]. We use this internal chip for all validation experiments, as we have shown a good correlation of the measurements with those performed with an external meter in [15]. We also use the Motorola G4 Play with Android 6 and 7, and the Sony XZ with Android 8.0 and 8.1. As these are not equipped with comparable battery fuel gauge chips, the OS provides estimates based on voltage readings and course-grained internal power profiles.

6.3.2 Observations from a simple experiment

To motivate the need for rigorous approaches to validation, we start by analysing the results of a simple experiment: we repeatedly run the original Rebound library, then reboot and recharge the device, and then repeatedly run the original Rebound library again. We do this on a Nexus 6 using Android 7.

Figure 6.1(a) shows the distribution of the energy consumption, which is clearly multimodal. Moreover, the energy consumption from the second run is skewed significantly lower – this means that in this experimental setting the variant is demonstrated to be significantly different from itself! This indicates that this – admittedly very simple – validation process lacks statistical specificity. Figure 6.1(b) provides a deeper look into the individual runs of the two experiments. In general, the measurements exhibit significant random and systematic noise. At the beginning of the experiments, the consumption is considerably higher than during the rest of the experiments. Moreover, the first experiment is clearly consuming more energy than the second experiment in most of the runs.

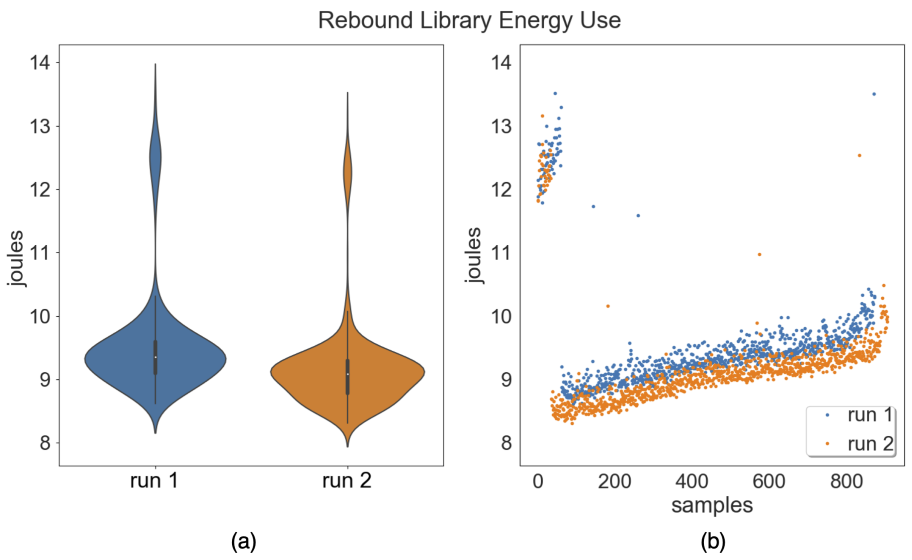

FIGURE 6.1. Energy use of the Rebound library in two experiments. Violin plots (a) and sample distributions (b). The device was rebooted and recharged between the two experiments.

Although the approach of rebooting and recharging seems to be prudent at first glance, rebooting a device can impact the energy use of the subsequent runs. This introduces clear challenges for researchers attempting to validate the outcomes of energy optimisation results, as software variants need to run under the quite similar conditions to be compared fairly.

Long-frequency changes in system state (evident in Figure 6.1(b)) are large enough to impact specificity even when validation runs are made within a single charge of a device. Such long-wave changes can be observed on multiple platforms. Figure 6.2 shows a spectral analysis of the measured energy consumption of the original rebound library on eight different hardware/OS combinations. These curves show that measurement variability induced by long-period changes in system state dominate on all platforms except Nexus 6 - Android 6 (which exhibits globally low system-induced variation). These large signals at low-frequency show that is problematic to make direct comparisons of software variants run a long time apart. Conversely, the magnitude of variation is moderate for variants run within 50 minutes of each other. That is, we are more likely to capture similar system states if we run variants close together in time. This observation can help inform the design of our validation approach.

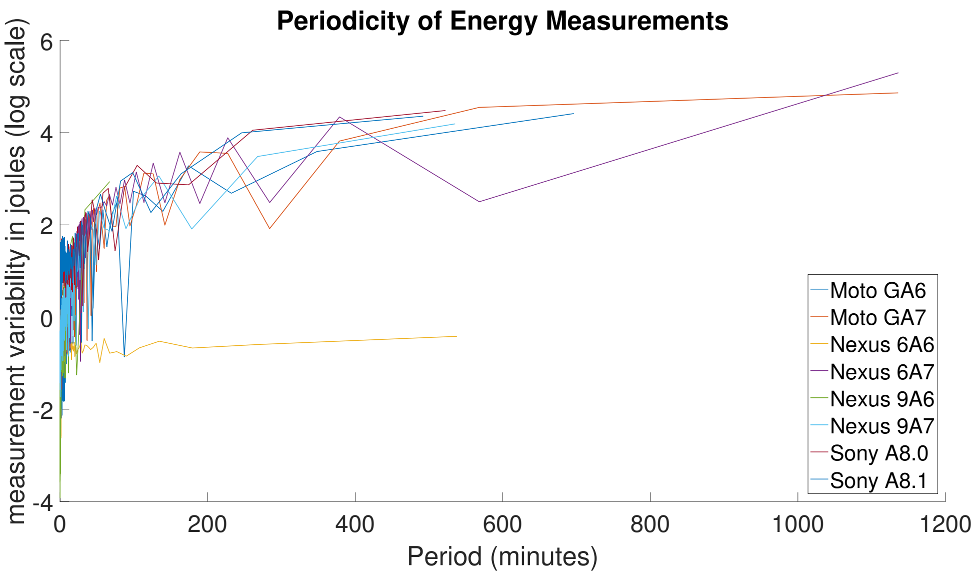

FIGURE 6.2. Log-scale spectral analysis of energy measurements of the original Rebound library run repeatedly on a full charge on eight different platform variants. On all but one platform long period changes in measurement environment dominate.

6.4 Validation Approaches

In this section we describe a number of alternative approaches to scheduling runs of program variants for validation. In all approaches we assume that there are n required samples of each program variant, where we label variants $A, B, C, D, \ldots$. We assume that we are resampling runs for long enough so that the non-functional property of energy is detectable above the noise on the internal meter of the device. In all of our experiments involving the aforementioned Rebound library and its configurations, measuring the total energy consumption of a configuration once takes between 20 seconds and a minute depending on the platform.

The first approach is called APPROACH 1 and is the modal approach used for validating the outcomes of the optimisation experiments in the literature [22, 26, 97, 138]. This approach starts with a *setup* phase where the device is rebooted and initialised by disabling unused hardware components after a recharge. *Setup* is followed by the experimental phase where each variant is run n times in turn. So for example, if we had variants A, B, C, D and $n = 4$ then Approach 1 would produce the schedule:

$$\textit{setup}, AAAA, BBBB, CCCC, DDDD$$

The second approach, Approach 2, re-samples one solution the required number of times starting from a full battery. The device is then *setup* again and then the second solution is run and so on. Under this approach with $n = 4$ the variants above would run with the order:

$$\textit{setup}, AAAA, \textit{setup}, BBBB, \textit{setup}, CCCC, \textit{setup}, DDDD$$

A third approach, Approach 3, interleaves the samples. So, for $n = 4$ and the variants above we would have the schedule:

$$\textit{setup}, ABCD, \textit{setup}, ABCD, \textit{setup}, ABCD, \textit{setup}, ABCD$$

A-priori, as long as each round $ABCD$ happens sufficiently quickly, then the variants in each round have a better chance of running in a similar system state than the previous approaches[1].

A fourth approach, Approach 4 is similar to Approach 3 but runs all trials on a single charge. This approach was used in [24]. So for, $n = 4$ Approach 4 would produce the schedule:

$$\textit{setup}, ABCD, ABCD, ABCD, ABCD$$

Our final approach R3-validation is similar to Approach 3 above except that the variants are left-shifted between setup phases and up to π sets of trials are run on a single discharge. Thus for $n = 4$ and $\pi = 2$ we would have:

$$\textit{setup}, ABCD, ABCD, \textit{setup}, BCDA, BCDA,$$
$$\textit{setup}, CDAB, CDAB, \textit{setup}, DABC, DABC$$

[1]The spectral analysis in Figure 6.2 indicates similarity could be achieved with rounds taking less than 20 minutes.

The value of π is chosen so as to not deplete the battery below 20% in a discharge cycle.[2] This approach is described in Algorithm 1.

Algorithm 1: Round-Robin & Rotate (R3-VALIDATION) validation protocol

```
input: Given a set $S$ of $N$ configurations, and $\pi$ the number of times a permutation
       is to be repeated within a discharge cycle.
1 Create a permutation $\Pi$ based on $S$;
2 repeat $N$ times
3     Reboot and recharge the device;
4     repeat $\pi$ times
5         Execute permutation $\Pi$ on the device;
6     end
7     Rotate $\Pi$ left by one position;
8 end
```

Implementation note. Technically, we are potentially left with state changes introduced by running the solutions, as we do not change the order of the solutions within the permutations. In the limit, this would require us to run all permutations, which is prohibitively expensive in practise. Also, it is not immediately clear which effect-length should be considered when optimising the overall order of the permutations, i.e., whether the number of pairs, triplets, quadruplets, ... are to be minimised. This also renders the application of exact algorithms for sequencing experiments (see, e.g., [159]) infeasible. A naive work-around would be to take a set of randomly generated permutations; however, this comes at the cost of potentially large random effects within the small number of time-consuming reboots. Therefore, our decision is to rotate through the initial permutation, as this results in a deterministic approach that does not require the user to solve any further combinatorial optimisation problems.

Claims We claim that in the systems and software variants we review, R3-VALIDATION is able to sample similar sets of system states across program variants. We also claim that, in our experiments, R3-VALIDATION is better for validation than the four alternative approaches listed above in terms of both test specificity, test sensitivity. We address these claims in turn, starting with system states in the next section and addressing specificity in Section 6.6, and sensitivity in Section 6.7.

[2]In our experiments system noise increases greatly if experiments proceed below 20% charge.

6.5 Case Study - Characterising system states and mitigating them

In this case study, we analyse data captured from a validation run using R3-VALIDATION. We then look back through the encountered states to show how R3-VALIDATION have consistently sampled for each variant.

As mentioned before, we use the variants raw1 to raw10 from Chapter 4 and the same platform configuration (Nexus 9 Android 6). Combining R3-VALIDATION with a targeted number of samples of at least 30 means that we need to restart the device 11 times and between each restart we run each configuration (in a rotating fashion) 3 times; this results in a total of 33 samples for each configuration.

Figure 6.3 shows the distributions of energy measured by sampling using the R3-VALIDATION sampling process. The variants on the x-axis are ordered by their median energy consumption (y-axis). The white dot represents the median and the coloured area shows the distribution of the sampled data. It can be seen that most distributions are compact and have a similar shape (slightly skewed to the right). The exception is the variant raw9 which has some outliers. The raw3 variant also has an extreme outlier (97 joules) which is not included in this plot. Note that the original ordering of the variants during optimisation was raw1, raw2, ... raw10, original so the validation approach has reordered some of the evolved variants.

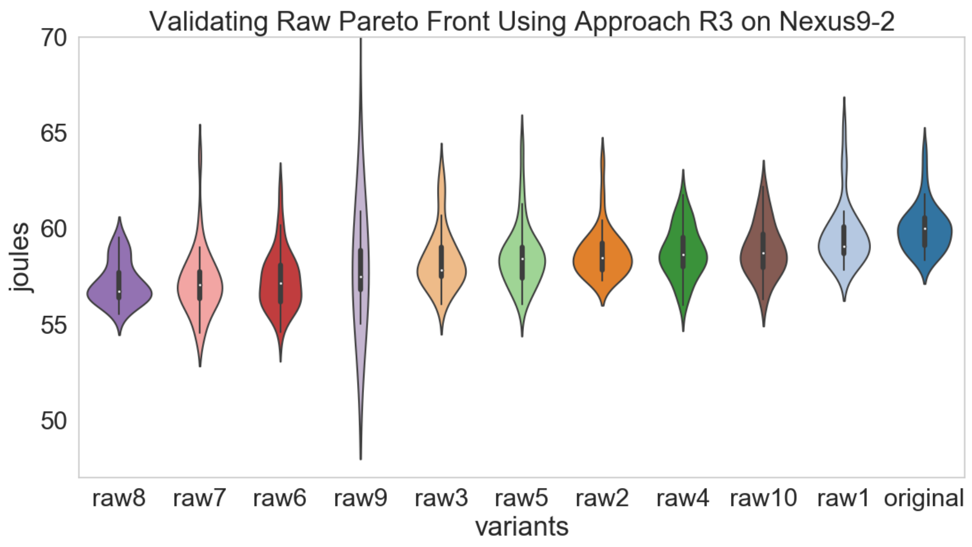

FIGURE 6.3. Results of validating 11 configurations using R3-VALIDATION. The configurations are ordered based on their median energy consumption.

6.5.1 System State Analysis

R3-VALIDATION is designed to help mitigate the effect of changing system states when comparing program variants. In designing these protocols it is interesting to observe how system states change. Some first insights were provided in Figures 6.1 and 6.2. However to see how deep and systematic changes in system state are, it is worth investigating these changing states in more detail.

In order to see the impact of changing system state in detail we ran an identical workload of the program variants original, raw1 ... raw10 executed, in round-robin fashion, three times. We then restarted the device and ran the experiment again and so on, 11 consecutive times on the Nexus 9 Android 6 platform. We name these 11 experiments exp1,...,exp11. For each experiment we sampled, energy and other information relating to system state.

Although Rebound is a CPU-bound library that does not utilise other hardware components, system states impacted its energy consumption during the eleven round-robin experiments. The plot in Figure 6.4 shows the distribution of Rebound variants' energy use. In general, each experiment, even though it was running an identical workload on a freshly booted machine produced a different distribution. The sixth experiment in particular used significantly more energy than the others and there seems to be a gentle upward trend in energy consumption overall.

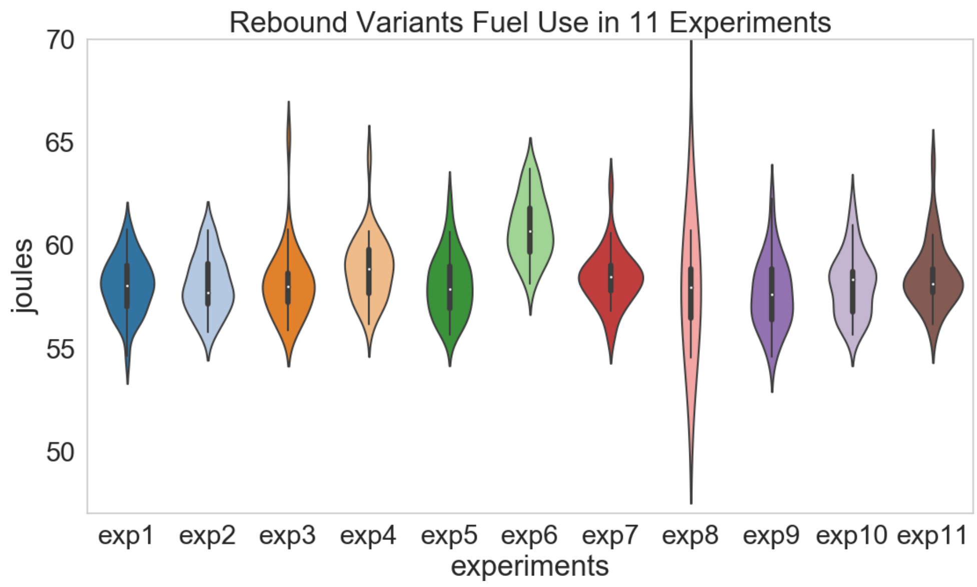

FIGURE 6.4. Each distribution here is run of the variants original, raw1 ... raw10 executed three times. That is, the same workload is run for each of exp1 through to exp11. As can be seen there is considerable variation in distributions, which is attributable to changing system state.

In the following, we shall dive a little deeper. Figure 6.5 shows individual samples of active background processes collected during the eleven experiments – each experiment's dataset is shown in a different colour. In general, the overall trend fluctuates from experiment to experiment. The number of active processes ranges approximately from 140 to 250 processes during variants execution with one extreme outlier of 500 processes.

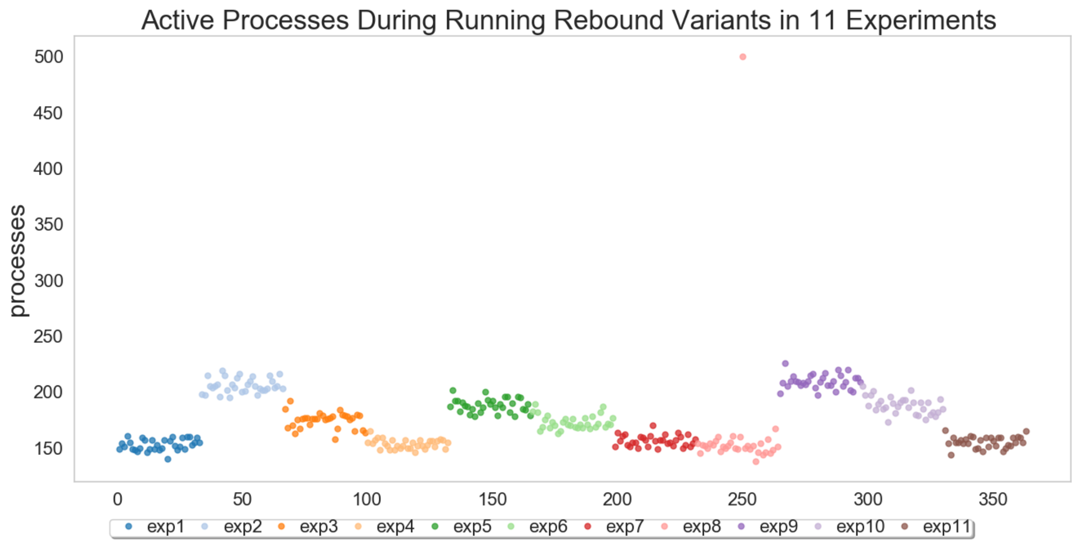

FIGURE 6.5. Active processes during the 11 experiments. Each colour is an experiment running all 11 variants in round robin order.

This outlier occurred during the execution of one variant of raw3 in experiment 8 (which happened to register extreme energy use). Further investigation of logs revealed that at that time Google Services' processes such as Google Play, Google Mobile Services and Google Cloud Messaging were, unsuccessfully, trying to perform sync operations and pull updates[3]. For example, the Google Cloud Messaging (GCM) process "gcm.HeartbeatAlarm.ConnectionInfoPersistService" which is responsible for pulling notifications from apps' servers to user devices, attempted to establish a Wi-Fi connection with a server during the eighth experiment. These services and APIs provide support functionalities such as location, wearable, advertisements, file recovery and other capabilities that developers need.

Another observation from Figure 6.5 is that samples collected in the same experiment are closer to each other in terms of active processes than other experiments. This indicates that, in terms of active processes, system state varies between system setups (reboot/recharging).

[3]Note that all communication interfaces were turned off.

Voltage level can affect the amount of energy used by a program run and, ideally, it is good to run variants in a similar voltage state. Figure 6.6 shows the voltage supply levels used during the round-robin experiments. Note that the initial voltage supply differs slightly in each experiment even though we attempted to start each at the same battery level by recharging between each experiment. Furthermore, the trace of voltage levels varies drastically between experiments. For example, the seventh experiment (red dots) has a steeper drop in voltage levels compared to the rest. The second, third, ninth and tenth experiment had roughly the same drop in voltage (by about 24 mV). Counter-intuitively three experiments (1,2 and 11) showed an increase in voltage. This contrasts starkly with models in the literature that assume a linear and an inverse correlation between voltage and state of discharge [52, 177].

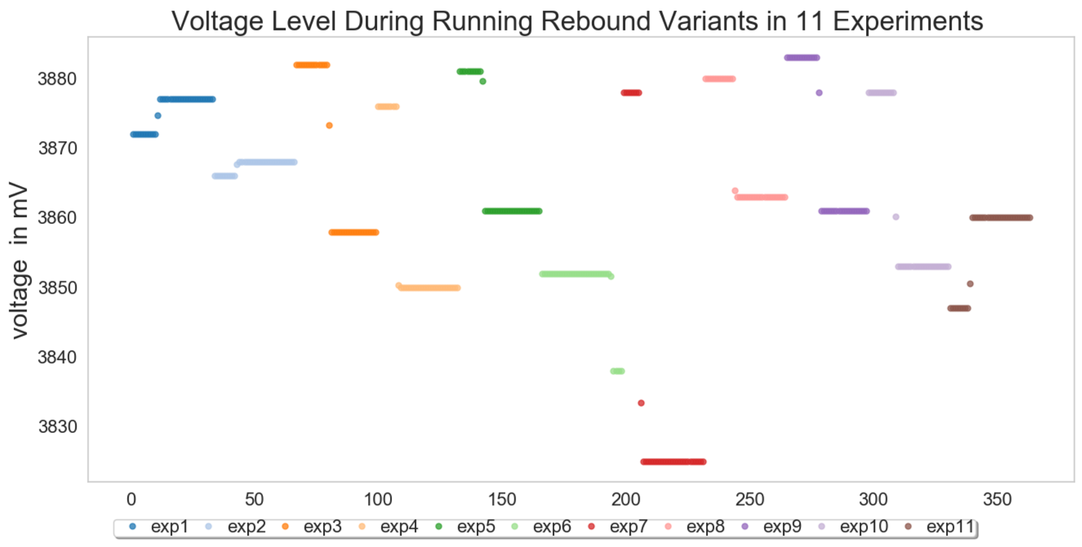

FIGURE 6.6. The voltage level during the 11 experiments. Each colour is an experiment running all 11 variants in round robin order. The voltage changes considerably between and during experiments.

It is worth mentioning that batteries are pre-configured and therefore experimenters cannot change or control the voltage supply. Such conditions show that no reasonable effort could guarantee the battery state to be the same for each experiment due to the chemical reactions inside the battery. It should be noted, however, that the battery state can stay steady for periods of time which means that two variants run close together in time have a chance of capturing a similar battery state.

Another aspect that cannot be fully controlled is the overall system memory consumption. This is due to the running background processes and the complex Android memory management system [149]. Figure 6.7 depicts interesting system memory usage behaviours. The memory consumption during the eleven experiments are spread over different distributions. In addition, after the first four experiments, the usage suddenly increased by 5% for the rest of the runs. As one might expect, the system memory fills up steadily as each experiment is executed – except in the fourth experiment where it was stable and (again) in the eighth experiment that includes a large drop in memory consumption. No explanation was found for the stable memory consumption in exp4 but the drop in exp8 is due to a system update putting a demand on memory resources and thus invoking a garbage collection[4]. This event also coincided with a peak in active processes.

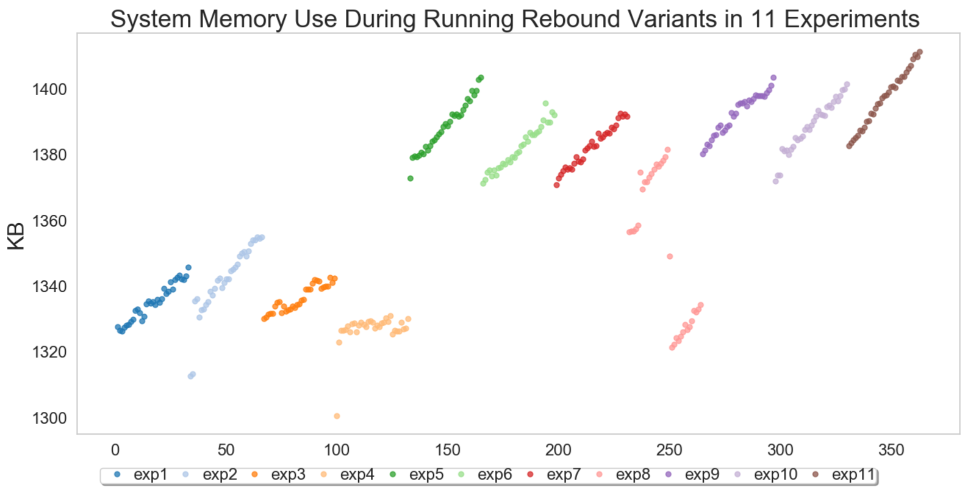

FIGURE 6.7. System memory use during the 11 experiments. Each colour is an experiment running all 11 variants in round robin order.

As a consequence of the drastic increase in number of active processes demonstrated in Figure 6.5, the CPU utilisation increased considerably to accommodate the extra workload. This is clearly depicted in Figure 6.8 which shows the CPU utilisation for the background

[4]To maximise responsiveness, Android reclaims memory from live and even dead processes only when it has to.

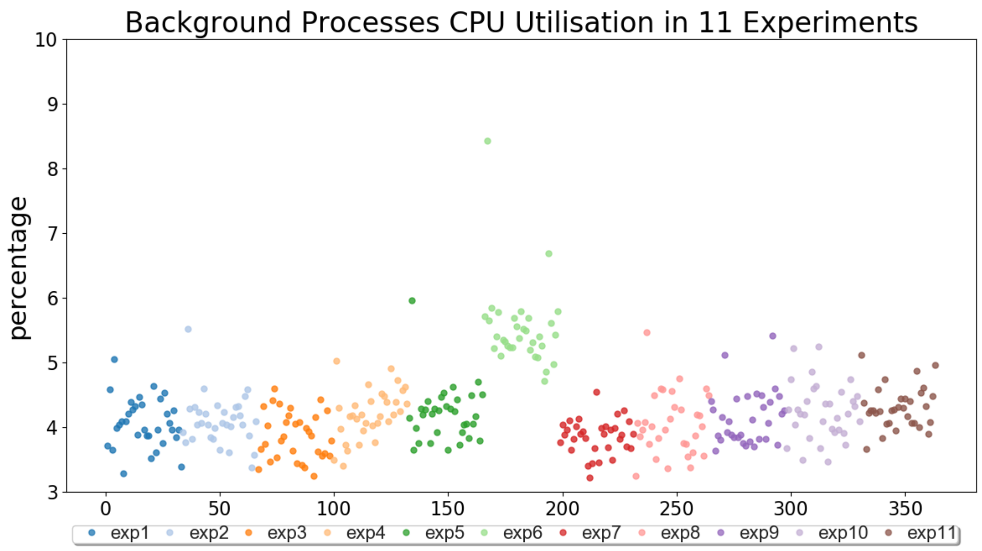

Figure 6.8. The CPU utilisation of background processes during the 11 experiments. Each colour is an experiment running all 11 variants in round robin order. The voltage changes considerably between and during experiments.

processes. As can be seen, overall, the background processes occupied roughly 3% to 8% of the CPU in most of the time. However, in the eighth experiment where the device state changed to pulling updates and syncing states, their utilisation increased to about 40.5%; the outliers were dropped from the plot to focus on the other runs.

Focusing just on Rebound's CPU use there was still variation between experiments. Figures 6.9 and 6.10 show the amount of CPU utilised and run-time needed by Rebound variants in the eleven experiments. Overall, the utilisation fluctuates between 48% to 54%, and the run-time varies between 16 seconds and 18.5 seconds. Interestingly, these two appear to be positively correlated, not negatively as one might expect for fixed loads.

In summary, we have observed that the system states affected the energy consumption. In addition, our fine-grained investigations of the experiments show that no reasonable effort to control conditions could guarantee that each solution runs on the same system state. Thus, we need to cope with the reality that the system state is will vary for each experiment run. Using our proposed method of validating solutions in a rotated-round-robin style, we attempt to ensure that such changes in state affect all solutions fairly.

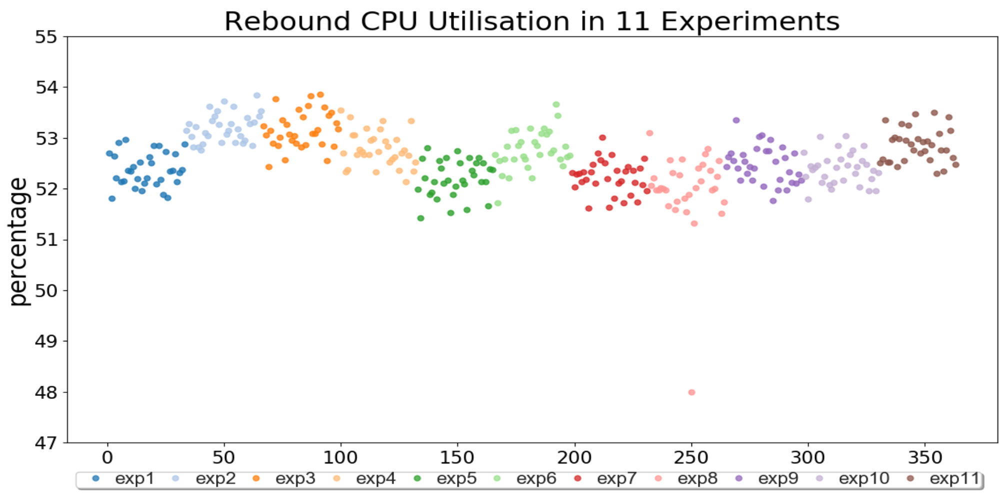

Figure 6.9. Rebound CPU utilisation during the 11 experiments. Each colour is an experiment running all 11 variants in round robin order.

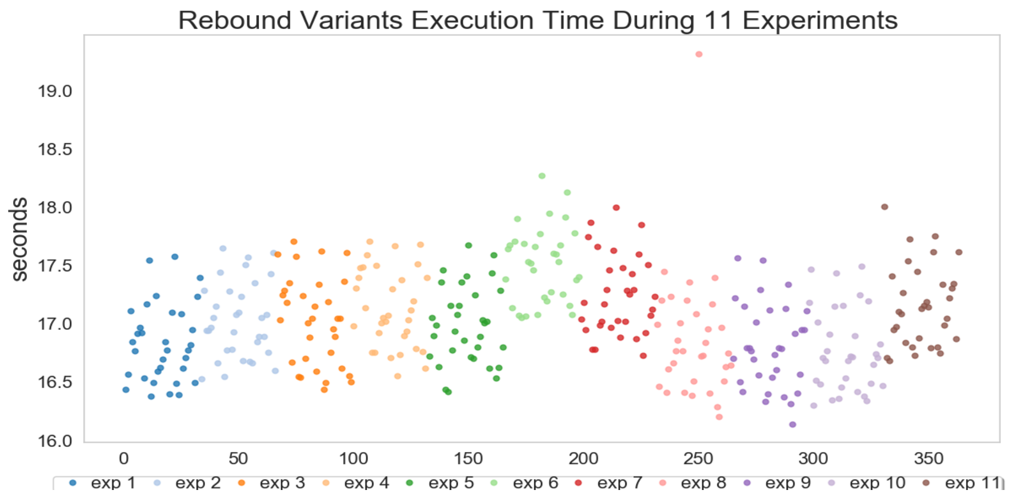

Figure 6.10. Rebound execution time during the 11 experiments. Each colour is an experiment running all 11 variants in round robin order.

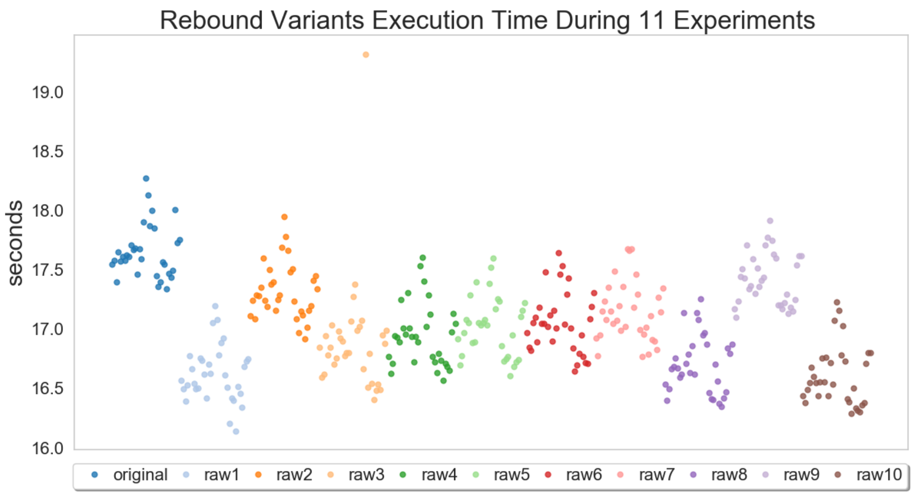

FIGURE 6.11. The run-time of each solution in the Round-robin validation experiments after grouping them. Each solution's data is in different colour and the original configuration is in blue.

6.5.2 Dealing With Different System States

Now we show how R3-VALIDATION helps mitigate the problem of changing system states by attempting to sample variants across similar sets of system states. R3-VALIDATION works by sampling variants in a rotated round-robin style and then aggregating the measurements for each variant together in preparation for statistical comparisons. The beneficial effect of this on energy measurements can be seen in Figure 6.3 where distributions of energy measurements are compact and similar. Other system features show similar levels of consistency between variants.

For example, Figure 6.11 shows the collected individual sample run-times for each variant. The dark blue samples represent the execution time of the original configuration. As can be seen, except for some outliers, there is remarkable consistency in the relative shape of the sequence of measurements of execution time with measurements for each variant generally trending up to a small peak and then dropping suddenly and then rising again. This consistency makes it relatively easy to distinguish between the underlying CPU utilisation of each variant.

This consistency of the set (and even sequence) of system states between variants under R3-VALIDATION can be seen in the plots of active processes in Figure 6.12. With the

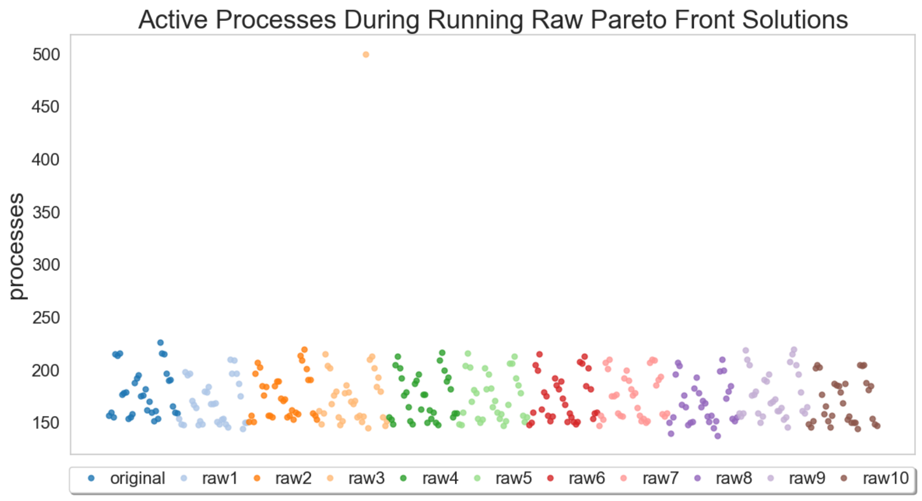

FIGURE 6.12. The plot shows the active processes during running each solution in the Round-robin validation experiments after grouping them. Each solution's data is in different colour and the original configuration is in blue.

exception of one outlier point, these sequences of active process readings have very similar shape and level.

Moreover, our approach ensured solutions to run in similar battery voltage levels which is beyond the device control. Figure 6.13 shows the distribution of the supplied voltage level during executing each solution. As can be seen, the overall voltage distributions for each solution is quite similar. Additionally, the bi-modal effect of the voltage is found in every solution. This was possible because the approach runs solution in short tournaments and therefore they experience similar battery conditions.

Lastly, although the state of the overall system memory usage cannot be fully controlled, our proposed approach ensured that each solution runs in similar overall main memory conditions. Figure 6.14 shows the overall system memory consumption during running each solution. Again, we can see that all configurations have been exposed fairly to the changing conditions.

We have seen so far how much system state can change in hard-to-predict ways and how the R3-VALIDATION scheduling approach can mitigate this problem in order to sample across similar system states. Next we compare R3-VALIDATION in terms of precision and then in terms of sensitivity.

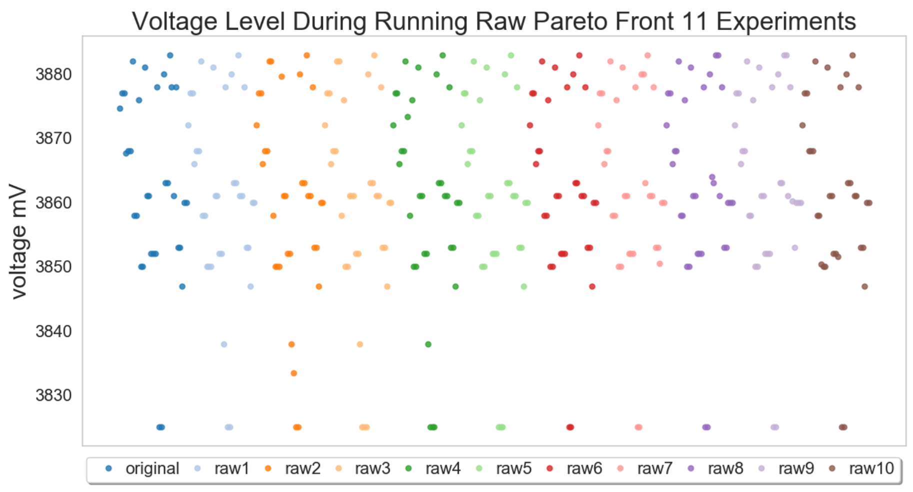

Figure 6.13. The plot shows the voltage level in mV of each solution in the Round-robin validation experiments after grouping them. Each solution's data is in different colour and the original configuration is in blue.

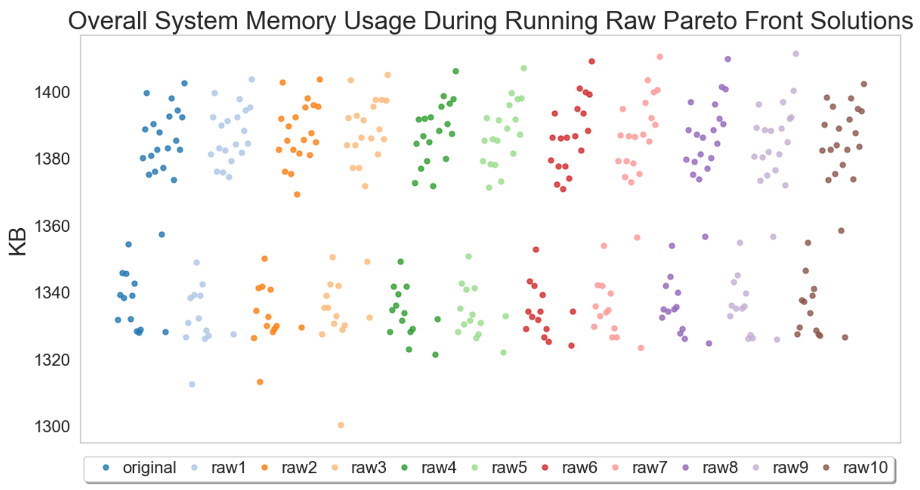

Figure 6.14. The plot shows the overall system memory usage during running solution in the Round-robin validation experiments after grouping them. Each solution's data is in different colour and the original configuration is in blue.

6.6 Test Specificity

As mentioned previously R3-validation is designed to maximise the similarity of the set of system states in which the energy used by program variants are sampled. The intent of achieving this similarity is to produce datasets that improve the specificity and sensitivity of statistical tests run over that data. We demonstrate that R3-validation scheduling produces energy readings that cater for more statistical specificity in our current experiments than the other approaches described above. We do this by showing that the other approaches are more prone to producing false positives.

We start by sampling a corpus of repeated energy measurements from running the original Rebound variant (Rebound run 1000 times followed by a sample of energy use via the internal meter). These measurements span seven reboots on each platform, except Nexus 9A6 which exhibited caching problems in energy measurements. Because this corpus contains measurements from running the *exact same variant* a good validation schedule should sample these variants in an order such that statistical tests detect *no difference* between groups of samples.

The seven reboots for each platforms supports seven samples each for $n = 7$ simulated variants. Each scheduling approach will group these samples into different sets of seven samples for each of seven variants according to the schedule determined by each approach. We compare each of the variant groups, for each platform, for each approach using a two-tailed Wilcoxon rank-sum test with a threshold of $p \leq 0.05$.

We aggregated the counts of false-positives ($p \leq 0.05$) across all seven platforms and produced the matrices in Figure 6.15. It can be seen that the approach that produces the most false positives is Approach 2 this shows that rebooting has a significant impact on system state as reflected in energy measurements. Approach 1 shows the effect of running variants far apart in time during a single charge cycle. The matrices for Approach 3 and Approach 4 indicate that interleaving has some success in capturing similar sets of system states. For Approach 4 in particular, the only false positives are on the Sony Android 8 and 8.1 platforms. R3-validation produces no false positives but produces a p-value close to the 0.05 threshold on the Sony Android 8.1 platform.

6.7 Test Sensitivity of Validation Approaches

We now compare the test sensitivity of the validation approaches described in Section 6.4 and our R3-validation. As before we consider the Rebound variants, original, raw1,...,raw11 as the set of configurations to be validated.

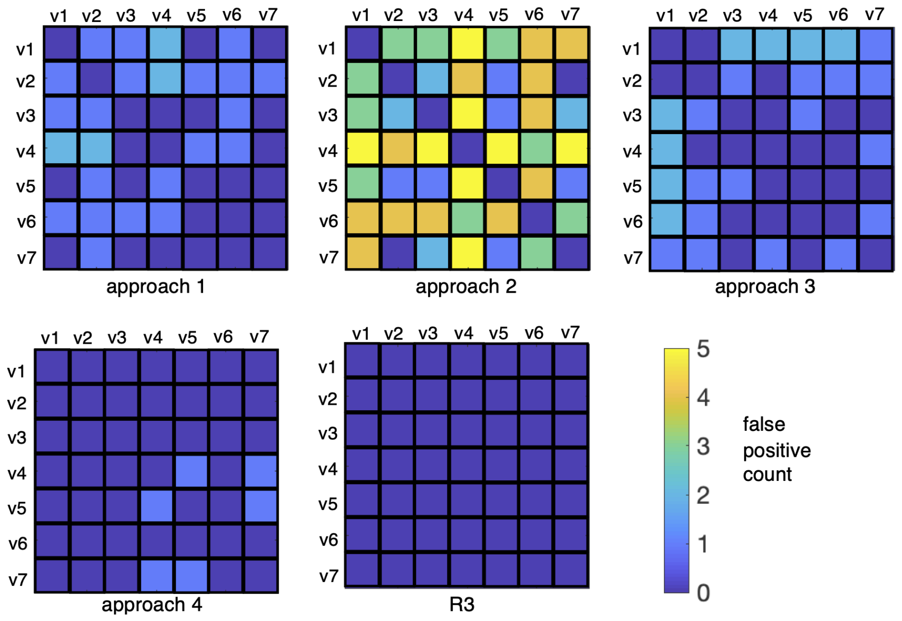

FIGURE 6.15. False positive count for grouped variants across all platforms when data from the original Rebound variant is grouped into seven simulated program variants (v1,..,v7) by the five different sampling approaches. The only approach not to produce false positives is R3-VALIDATION.

To check how the validation methods affect the statistical sensitivity of comparisons between each solution and the original configuration, we use an unpaired right-tailed Wilcoxon statistical rank-sum test. This non-parametric test does not make any assumptions about the distribution of the data-sets. Our null hypothesis is the original configuration's energy use samples are greater than the optimised solutions. The confidence level *alpha* used is 0.05. To augment this comparison we also compute the Vargha and Delaney $\hat{A}_{12}$ effect size to measure the approximate differences between the original configuration energy use and the optimised solutions. This measure is non-parametric and calculates the proportional difference between two data-sets [162]. It quantifies the difference in four ranges/thresholds for interpreting the effect size: 0.5 means no difference; up to 0.56 indicates a small difference; up to 0.64 indicates medium effect; over 0.71 is a large difference. This approach calculates the expected probability that solution 1 consumes less energy than solution 2. For example if $\hat{A}_{12} = 0.8$, then solution 1 is expected to consume less than solution 2 in 80% of the time.

We use these statistical tests and measures for each approach on each platform in three ways. First, to report on the median effect size *median*($\hat{A}_{12}$). Second, to measure how often we observe at least a "medium" effect in the direction of dominance, we count how often $es \geq 0.64$. Third, we count the number of times the Wilcoxon rank-sum tests that indicate a significant difference at $p \leq 0.05$.

	Approach 1			Approach 2			Approach 3			Approach 4			R3-validation		
	median($\hat{A}_{12}$)	$es \geq 0.64$	$p \leq 0.05$	median(es)	$es \geq 0.64$	$p \leq 0.05$	median(es)	$es \geq 0.64$	$p \leq 0.05$	median(es)	$es \geq 0.64$	$p \leq 0.05$	median(es)	$es \geq 0.64$	$p \leq 0.05$
Nexus 9A6	0.57	5	5	0.38	1	2	0.73	7	7	0.65	6	7	**0.82**	**10**	**10**
Nexus 9A7	0.42	**3**	**3**	0.27	1	1	**0.50**	1	1						
Nexus 6A6	0.50	5	5	0.17	1	1	0.83	9	9	0.75	8	9	**0.87**	**10**	**10**
Nexus 6A7	0.12	0	0	0.57	4	3	0.60	5	5	0.50	0	0	**0.82**	**10**	**10**
Moto GA6	0.38	1	2	0.72	7	8	0.72	**9**	**9**	0.70	8	8	**0.9**	**9**	**9**
Moto GA7	0.50	4	4	0.68	6	6	0.50	5	5	5	0	0	**0.82**	**10**	**10**

Table 6.1. Performance comparison of different validation approaches on different platforms. $es \geq 0.64$ denotes the number of at least "medium" effects. $p \leq 0.05$ denotes the number of Wilcoxon rank-sum tests that indicate a significant difference. The best values per device-OS configuration are highlighted in bold.

Table 6.1 reports the results of validating 11 software configurations using five validation approaches on six combinations of hardware and operating systems. Two blocks of data are unavailable for Approach 4 and R3-validation for the Nexus 9 (Android 7) platform due to the device burning out under experimental load. It can be seen that, where data is available, R3-validation is consistently better than other approaches on these measures. This indicates that, at least on these benchmarks and platforms R3-validation exhibits higher sensitivity than other methods. We excluded the Sony XZ from our validation experiment since it was shown to have inaccurate energy readings in [18].

In addition, Approach 1 and Approach 2 have the lowest performance among the approaches. The former does not take into consideration the variation in battery/voltage levels and their consequences on the system state. The latter, on the other hand, suffered from the random initial system states caused by rebooting.

Running solutions in round-robin fashion benefited Approach 3 and Approach 4 in most cases. In case of the former, reboots did not affect the collected samples, however, the fixed order did not expose all solutions to the same condition. For example, *raw1* was always executed when the battery level was at 100%. This issue has also impacted Approach 4. Both platforms however, performed well on Android 6 because solution run-times were twice as fast as on Android 7. This aligns with our findings presented in Section 6.3.2.

6.8 Discussion

The above results lead to several useful observations. First, energy use for a given program is highly time variant to the extent that a validation process can easily mis-rank program variants. Second, across multiple platforms we have observed that variations in energy use induced by system state vary most over longer time periods and between reboots. The proposed round-robin validation scheduling algorithm R3-validation is designed to rotate through runs of program variants so as to maximise the chances of catching the same sequence of system states for each variant. Our application of R3-validation to a set of program instances leads to a third observation that R3-validation succeeds in capturing very similar sequences of system states according to multiple metrics (Figures 6.11-6.14). A fourth observation is that when used to compare program variants R3-validation seems to exhibit both good precision and specificity when compared to other approaches.

These observations lead to some prescriptions for how validation might be done on a mobile platform. First it is important to note that R3-validation assumes that long period variations energy states dominates short term variations. Before validating on a new system it is important to repeatedly run a test payload to and perform a frequency analysis in measured energy use over time to ensure that this assumption holds. Second when validation is run on different program variants it is important to instrument the runs so that dimensions of system states for every run are captured. More confidence can be assigned to rankings of variants if one can observe, as we did here, very similar trajectories for system states across all runs. Finally, it is always possible that on new platforms new dimensions of system behaviour could start to impact on performance - it is important that we always look for this possibility into the future so it can inform the design of new validation schedule designs.

6.9 Conclusion

Conventional means of validating software variants resulting from evolutionary processes are risky, because modern compute platforms exhibit a large number of system states beyond the control of a researcher. In this chapter, we have shown how a range of conditions – such as system software states, memory consumption, and battery voltage – can affect experimental results, which in turn may mislead the researchers. To mitigate the effects, we have proposed R3-VALIDATION: it exercises the tests in a rotated-round-robin fashion, while accommodating the necessary regular restarting and recharging of the test platform. We have observed in a comparison with other validation approaches that state changes have not misled our procedure, and that the measurable differences between configurations were the strongest for ours.

Chapter 7

Conclusion

"Like the legend of the Phoenix all ends with beginnings, what keeps the planets spinning, the force from the beginning."

Pharrell Williams and Nile Rodgers

In this thesis we have presented research into the genetic improvement (GI) of software for energy efficiency. In this final chapter we shall discuss the broad contributions of this dissertation and possible future directions we believe could be most advantageous to even the greater field of search-based software engineering.

7.1 Contributions

Prior to this research journey, little was known on how GI of software for energy efficiency can be carried out on mobile platforms, i.e. *in-vivo*. Unfortunately, smart devices are diverse and very complex, and therefore modelling energy behaviours is difficult. This complexity presents challenges to the effectiveness of off-line optimisation of mobile applications. Smart-phones will continue to evolve both in hardware (HW) and software in order to improve the user experience. However, this is likely to result in further challenges for researchers interested in creating relatively noise-free test-beds that are to support the automated optimisation of non-functional properties. Although the internal battery meters by which the fitness function is computed are less accurate than external power meters, they are precise and can be used for ranking candidate solutions.

At the beginning of this journey, in Chapter 3, our investigations for carrying out the *in-vivo* optimisation on smart-phones have demonstrated and overcome expected and unexpected obstacles. Those obstacles are due to the fact that isolating the test-bed environment from the HW and the OS is impractical. Such coupling produces noise that impacts the energy signals which is crucial for fitness evaluation. Therefore, energy optimisation experiments need to run in environments where the noise is minimised as much as possible. We have documented challenges ranging from HW hurdles such as sampling induced error, ambient temperature and CPU throttling, to problematic OS behaviours such as Doze-mode and issues with the Android Debugging Bridge. We view highlighting and solving the encountered technical challenges as valuable contributions for practitioners in the search-based software engineering field. This is because researchers and developers can use our findings as guidelines in conducting energy optimisation experiments.

In the next stop of our journey, in Chapter 4, we have demonstrated the technical feasibility of *in-vivo* optimisation on smart-phones. Although the optimisation was guided by executing the target application's test suite, which adds more noise, our proposed approach has helped the search process by simple code rewrites to increase energy signals. Instrumenting the target code with *dummy-loops* has minimised the induced noise and amplified the small energy signals of the case study, where the target code exclusively uses CPU and RAM. Our main assumption is that adding uniform per-statement overhead does not adversely affect search. Our validation process of the optimised configuration using the non-amplified evolved solution code showed the feasibility of our approach.

In addition, we have supplemented our *in-vivo* optimisation experiment with model-based optimisation experiments using CPU utilisation and line of code energy models. The models where customised to the optimisation platform exploiting the internal meter capability. Interestingly, the generated optimal solutions by the three experiments are quite similar in the decision space. This, in this instance, verifies that the developed models are reasonably consistent with the real world, and the results are reproducible, even when using a different model or the real device. Our results shows that the *in-vivo* optimisation of software on smart devices is possible by the means of GI. Our work helps extend the application of GI to *in-vivo* optimisation of energy consumption in the presence of noise and small signals in energy readings.

In terms of approximate computing where incomplete outputs are acceptable in order to save computation resources, our results have shown that degradation in the animation performance of rebound can save energy. The improved solutions generated using *in-vivo* and model-based optimisation have breached the code semantics in our case study. Consequently, the produced animations deviated from the original behaviour intended by the developers of the target app. Although this seems unacceptable, it has decreased the energy consumption up to 22% with slight degradation in the animation that did not affect the main purpose of the required animation.

More stumbling blocks were littered on our rough journey – as reported in Chapter 5, we have observed that the noise in energy readings is more difficult to model than expected. Broadly speaking HW noise conforms to a normal distribution [69]. Nevertheless, induced noise in the fuel gauge is not normally distributed. This is owing to the unexpected increase in the energy use caused by dissipation, voltage and discharge currents relationship and sensor drifts. It is worth mentioning that we have used smaller sampling windows to alleviate the problem of the energy increase overtime, however, even then, the resulting distributions are not normal. This observation is critical to take into consideration when choosing a statistical test to determine whether there is significant difference in energy use between software variants.

Our conceptual framework can determine the minimum number of samples required to show significant differences between software solutions competing in a tournament. Our results show that the number of samples can vary substantially from one point of time to another within each platform. For instance, when sampling starts on Moto G4 Play with a full battery, at least seven samples are required to show a statistical difference at 95% confidence level, whereas, at least fifteen samples are needed when sampling starts at 80% of remaining battery.

Moreover, we have discovered that the required number of samples differs notably between platforms (combinations of HW and OS). This is because different platforms have different severity levels of noise even in identical HW specifications. For example, the Moto G4 Play device running Android 6 has exhibited an initial burn-in phase with the energy consumption being roughly twice as high as the rest of the samples, whereas, on Android 7 the initial phase of high energy consumption has disappeared. However, additional higher and lower levels of energy consumption have emerged. In addition, our results show that as the platforms evolve the noise in reading rises. Such findings are crucial when using optimisation in the wild (uncontrolled environment) or even in the lab (controlled environment).

The misleading signs detected during our journey have shown that current validation approaches – post optimisation – are inadequate and have led us to propose and implemented a new validation approach in Chapter 6. Unfortunately, as the complexity of platforms increases, the number of hidden factors impacting the outcomes of energy optimisation measurements increases. Thus, ignoring these factors' effects can mislead researchers and practitioners to drawing wrong conclusions. We have demonstrated and documented a broad range of factors and their impacts on the final outcomes. For example, variations in the battery voltage, system states (the random initial state and the current state), and periodic tasks cannot be controlled when re-sampling software variants, will cause variants to run in different conditions. This gives unjustified advantages to some of the variants which leads to measurement bias. Our validation approach mitigates the problem of these hidden and uncontrolled factors. It executes optimised variants in round robin order to guarantee – to a greater extent than the other approaches – that each variant runs under similar conditions even across reboots. More generally, our in-depth analysis and the new validation approach are of importance for the software performance benchmarking field, and in particular for the community of search-based of software engineering where noisy fitness functions are used to guide the search for fitter solutions in terms of non-functional properties such as execution time, memory and energy consumption.

7.2 Future Work

After concluding this journey, we plan to commence a new journey to advance the current state of *in-vivo* optimisation using GI. In this section we outline our plan. Our proposals have been published as position papers in the seventh and ninth editions of the International Genetic Improvement Workshop at GECCO in 2019 and 2020 [17, 20].

The quest of a distributed and scalable GI approach. In-vivo optimisation provides the ground-truth for the behaviour of an application on a given platform, unlike model-based optimisation which only provides what the model was trained on. However, in-vivo optimisation faces several obstacles. The noisy fitness function resulting from sampling non-functional properties, such as speed, memory or energy use, is a key challenge. In general, precise and robust measurement of non-functional software properties is difficult to obtain. Complete isolation of the effects of software optimisation is simply infeasible, due to random and systematic noise from the platform. Furthermore, noise reduction requires re-sampling the new software variants, which makes the fitness function very expensive. The search space is gigantic and can be non-monotonic (Chapter 4), requiring a large number of fitness evaluations in order to discover more interesting regions of the solution space. These issues raise the need to incorporate scalable and high performance techniques to help the search process. We aim to incorporate genetic improvement of software methods with distributed evolutionary computation models such as islands model to improve the energy use of software across several platforms. The need for a distributed technique is also due to our observations in Chapter 5 and 6, where we have found that different platforms have different characteristics. This dictates a future study on the robustness of the evolved solutions across the other platforms.

Machine learning for GI. Model-based fitness functions can speed up the evaluation process and reduce the effects of noisy environments. However, the behaviour of complex systems such as PCs and smart-phones cannot effectively be reduced into a simple model that describes the environment in terms of only one dependent variable. The problem with the current models is their simplicity, which defies the main purpose of evaluating the performance (run-time and energy use) of real software in realistic scenarios. In addition, non-determinism in the host system, in which the software is evaluated, drastically affects the obtained fitness values. This means a bad solution may sometimes outperform good ones because the current state of the system, a situation which neither of *in-vivo* and simple model-based fitness evaluation can handle. The first proposed solution is to integrate machine learning approaches with GI. This helps learning complex models that take into consideration the complexity of the host environment and the non-linearity in energy usage. For example, the background processes have repeated patterns that ML can help GI to model and compensate for their noise. Voltage drops and spikes can divert GI from interesting search spots, nevertheless, ML learns the impact of such variation and compensates for GI. Although memory usage can be argued to have a second-order correlation with non-functional properties such as energy and run-time, its side effect reflected by garbage collection (GC) can be enormous. GC can stop the world in some virtual machines used in Android [149]. Our second proposal is a synergy between learned models and *in-vivo* evaluations. The result of such a synergy is adaptive models that get re-calibrated as the optimisation proceeds. During the optimisation, the *in-vivo* evaluation takes in place to evaluate selected solutions and their samples are used to fine-tune the model.

Special benchmarks for GI. Optimising the non-functional properties of software such as energy use requires benchmarks that can be used to profile the energy behaviour of the host platform's HW and software components. In this thesis, we introduced the BusyLoop app to profile the energy behaviour of the CPU and the main memory on different platforms. It would be interesting to see future works that create benchmarks to investigate the other HW components such as GPU, GPS, NIC and built-in sensors. Additionally, these benchmarks need to be optimisable (similar to our BusyLoop app). For example, to profile and optimise the energy use of NIC, one can create a benchmark app that downloads and uploads dummy data. During the data transfer operation, the energy use is recorded for profiling purposes. The optimisation can be achieved by tuning special parameters such as the size and number of packets. It is worth mentioning that the problem of the benchmark availability is a universal problem in GI [89].

7.3 Final Remarks

“Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.”– Frederick Phillips Brooks.

Generally speaking, smart devices’ HW, software and their complexities will continue to evolve to improve and transform our lives. Software needs continuous improvements to satisfy its users expectations, otherwise its quality degrades. Since there is no accounting for tastes, device vendors will manufacture diverse devices to cover the most possible market segments. Software practitioners need more tools in their arsenals to overcome the battle of improving software efficiency on diverse platforms. Creating supporting tools requires sound and systematic methodologies. *In-vivo* optimisation using GI can be used to automatically create customised software variants that respond differently to a variety of conditions depending on the host platform. It is worth mentioning that there is no silver bullet and we do not consider *in-vivo* optimisation using GI as one. Nevertheless, it can be considered a factory for creating different bullets made from different materials.

In this research journey, we have outlined our vision in Chapter 1 which is “a future where software developers are only concerned with what their applications should do and leave the burden of how they should behave on different smart devices to *in-vivo* optimisation using GI”. Although making such a dream comes true could take more than several theses in this field, our work has established the foundations for the upcoming works by presenting and solving the technical issues to set a test-bed for the required experiments, mitigating the problem of noisy readings, illustrating the differences between platforms, and providing a novel rigorous validation approach of the final outcomes.

Bibliography

[1] Codenames, tags, and build numbers, 2020. URL https://source.android.com/setup/start/build-numbers. Accessed on 30 June 2020.

[2] I. E. Agency. Key world energy statistics 2015, 2015. URL http://www.iea.org/publications/freepublications/publication/key-world-energy-statistics-2015.html.

[3] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *the 9th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2003.

[4] *Android Power Profiles*. Android. URL https://source.android.com/devices/tech/power.html. retrieved 03/2016.

[5] Android Developers. Optimizing for doze and app standby. URL https://developer.android.com/training/monitoring-device-state/doze-standby.html. Accessed 23 March 2017.

[6] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014.

[7] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *the International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[8] Y. Bee Wah and N. Mohd Razali. Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2011.

[9] S. Bhadra, A. Conrad, C. Hurkes, B. Kirklin, and G. M. Kapfhammer. An experimental study of methods for executing test suites in memory constrained environments. In *Workshop on Automation of Software Test*, 2009.

[10] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. Orbs: Language-independent program slicing. In *the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[11] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Notices*, volume 41. ACM, 2006.

[12] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister, et al. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38, 2016.

[13] M. Bokhari and M. Wagner. Optimising energy consumption heuristically on android mobile phones. In *Genetic and Evolutionary Computation Conference Companion (GECCO)*. ACM, 2016.

[14] M. A. Bokhari, B. R. Bruce, B. Alexander, and M. Wagner. Deep Parameter Optimisation on Android Smartphones for Energy Minimisation: A Tale of Woe and a Proof-of-concept. In *Genetic and Evolutionary Computation Conference Companion (GECCO)*. ACM, 2017.

[15] M. A. Bokhari, Y. Xia, B. Zhou, B. Alexander, and M. Wagner. Validation of internal meters of mobile android devices. *CoRR*, abs/1701.07095, 2017.

[16] M. A. Bokhari, B. Alexander, and M. Wagner. In-vivo and offline optimisation of energy use in the presence of small energy signals: A case study on a popular android library. In *15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous)*. ACM, 2018.

[17] M. A. Bokhari, M. Wagner, and B. Alexander. The quest for non-functional property optimisation in heterogeneous and fragmented ecosystems: A distributed approach. In *the Genetic and Evolutionary Computation Conference Companion (GECCO)*. Association for Computing Machinery, 2019.

[18] M. A. Bokhari, L. Weng, M. Wagner, and B. Alexander. Mind the gap–a distributed framework for enabling energy optimisation on modern smart-phones in the presence of noise, drift, and statistical insignificance. In *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2019.

[19] M. A. Bokhari, B. Alexander, and M. Wagner. Towards rigorous validation of energy optimisation experiments. In *the 2020 Genetic and Evolutionary Computation Conference (GECCO)*, 2020.

[20] M. A. Bokhari, M. Wagner, and B. Alexander. Genetic improvement of software efficiency: The curse of fitness estimation. In *the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO)*, 2020.

[21] A. E. I. Brownlee, N. Burles, and J. Swan. Search-Based energy optimization of some ubiquitous algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1, 2017.

[22] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2015.

[23] B. R. Bruce, J. M. Aitken, and J. Petke. Deep parameter optimisation for face detection using the viola-jones algorithm in OpenCV. In *Symposium on Search-Based Software Engineering (SSBSE)*. Springer, 2016.

[24] B. R. Bruce, J. Petke, M. Harman, and E. T. Barr. Approximate oracles and synergy in software energy search spaces. *IEEE Transactions on Software Engineering*, 45, 2019.

[25] D. Buche, P. Stoll, R. Dornberger, and P. Koumoutsakos. Multiobjective evolutionary algorithm for the optimization of noisy combustion processes. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 32, 2002.

[26] N. Burles, E. Bowles, A. E. I. Brownlee, Z. A. Kocsis, J. Swan, N. Veerapen, and Y. Labiche. Object-oriented genetic improvement for improved energy consumption in Google Guava. In *Symposium on Search-Based Software Engineering (SSBSE)*. Springer, 2015.

[27] Canalys. Smart phones overtake client PCs in 2011, 2012. URL https://www.canalys.com/static/press_release/2012/canalys-press-release-030212-smart-phones-overtake-client-pcs-2011_0.pdf. Accessed on 24 March 2017.

[28] A. Cañete, J.-M. Horcas, I. Ayala, and L. Fuentes. Energy efficient adaptation engines for android applications. *Information and Software Technology*, 118, 2020.

[29] E. Cantú-Paz. Adaptive sampling for noisy problems. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2004.

[30] C. K. Chan, H. Peng, G. Liu, K. McIlwrath, X. F. Zhang, R. A. Huggins, and Y. Cui. High-performance lithium battery anodes using silicon nanowires. *Nature Nanotechnology*, 3, 2008.

[31] S. Chand and M. Wagner. Evolutionary many-objective optimization: A quick-start guide. *Surveys in Operations Research and Management Science*, 20, 2015.

[32] N. Chang, K. Kim, and H. G. Lee. Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI. In *the Symposium on Low Power Electronics and Design (ISLPED)*. ACM, 2000.

[33] Cisco. Cisco annual internet repor, 2018. Accessed 23 June 2020, https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf.

[34] B. Cody-Kenny and S. Barrett. The emergence of useful bias in self-focusing genetic programming for software optimisation. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2013.

[35] C. Curtsinger and E. D. Berger. Stabilizer: statistically sound performance evaluation. In *ACM SIGPLAN Notices*, volume 48. ACM, 2013.

[36] N. V. DATABASE. HTC 802.1x password exploit, 2012. URL https://nvd.nist.gov/vuln/detail/CVE-2011-4872#vulnCurrentDescriptionTitle. Accessed on 30 June 2020.

[37] J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The human competitiveness of search based software engineering. In *the 2Nd International Symposium on Search Based Software Engineering (SSBSE)*, Washington,

DC, USA, 2010. IEEE Computer Society. doi: 10.1109/SSBSE.2010.25. URL `http://dx.doi.org/10.1109/SSBSE.2010.25`.

[38] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, Part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18, 2014.

[39] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 2014.

[40] A. Di Pietro. *Optimising evolutionary strategies for problems with varying noise strength*. 2007.

[41] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15, 1972.

[42] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2011.

[43] R. C. Eberhart, Y. Shi, and J. Kennedy. *Swarm intelligence*. Elsevier, 2001.

[44] A. Eiben and J. Smith. Introduction to evolutionary computing (natural computing series). 2008.

[45] F. Siegmund, and A. H. C. Ng, and K. Deb. A ranking and selection strategy for preference-based evolutionary multi-objective optimization of variable-noise problems. In *IEEE Congress on Evolutionary Computation (CEC)*, 2016.

[46] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *2nd Workshop on Mobile Computer Systems and Applications*. IEEE, 1999.

[47] M. Gabel and Z. Su. A study of the uniqueness of source code. In *the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, 2010.

[48] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *ACM SIGPLAN Notices*, volume 43. ACM, 2008.

[49] F. Glover and M. Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

[50] Google. Batterymanager. https://developer.android.com/reference/android/os/BatteryManager. Accessed on 8 May 2020.

[51] Google. Platform architecture. https://developer.android.com/reference/android/os/BatteryManager, 2019. Accessed on 23 June 2020.

[52] L. Z. M. Gordon and B. Tiwana. A power monitor for android-based mobile platforms. *Application can be downloaded from: http://ziyang. eecs. umich. edu/projects/powertutor*, 2012.

[53] B. C. Group. The Mobile Revolution: How Mobile Technologies Drive a Trillion-Dollar Impact, 2015. Accessed 23 June 2020, https://www.bcg.com/publications/2015/telecommunications-technology-industries-the-mobile-revolution.

[54] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2012.

[55] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *European Test Symposium (ETS)*. IEEE, 2013.

[56] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating android applications' cpu energy usage via bytecode profiling. In *1st International Workshop on Green and Sustainable Software*. IEEE Press, 2012.

[57] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *International Conference on Software Engineering (ICSE)*. IEEE Press, 2013.

[58] S. O. Haraldsson and J. R. Woodward. Genetic improvement of energy usage is only as reliable as the measurements are accurate. In *Genetic and Evolutionary Computation Companion*. ACM, 2015.

[59] S. O. Haraldsson, J. R. Woodward, A. E. Brownlee, and K. Siggeirsdottir. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *the Genetic and Evolutionary Computation Conference Companion (GECCO)*, 2017.

[60] M. Harman and B. F. Jones. Search-based software engineering. *Information and software Technology*, 43, 2001.

[61] M. Harman, Y. Jia, and W. B. Langdon. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2014.

[62] A. Hindle. Green software engineering: The curse of methodology. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, 2016.

[63] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *11th Working Conf. on Mining Software Repositories (MSR)*. ACM, 2014.

[64] J. Hoffman and S. Frankel. *Numerical Methods for Engineers and Scientists*. CRC Press, 2018.

[65] J. H. Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[66] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma. Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Computing Surveys (CSUR)*, 48, 2016.

[67] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. *ACM SIGPLAN Notices*, 39, 2004.

[68] A. Hussein, M. Payer, A. Hosking, and C. A. Vick. Impact of gc design on power and performance for android. In *8th ACM International Systems and Storage Conference (SYSTOR)*. ACM, 2015.

[69] G. Iacca, F. Neri, and E. Mininno. Noise analysis compact differential evolution. *Systems Science*, 43, 2012.

[70] A. Inc. Background locationOLD limits, 2017. URL `http://tiny.cc/locold`. Accessed on 28 March 2018.

[71] Q. I. C. Inc. Trepn profiler, 2014. URL https://developer.qualcomm.com/software/trepn-power-profiler/tools. Accessed 10 May 2018.

[72] M. Integrated. MAX17047/MAX17050 ModelGauge m3 Fuel Gauge, 2016. Accessed 2 February 2020, https://datasheets.maximintegrated.com/en/ds/MAX17047-MAX17050.pdf.

[73] ISO/IEC. Iso/iec 25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models, 2017.

[74] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. Ecodroid: An approach for energy-based ranking of android apps. In *4th International Workshop on Green and Sustainable Software*. IEEE Press, 2015.

[75] Jordan. 802.1x password exploit on many htc android devices, 2012. URL http://blog.mywarwithentropy.com/2012/02/8021x-password-exploit-on-many-htc.html. Accessed on 30 June 2020.

[76] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. Devscope: A nonintrusive and online power analysis tool for smartphone hardware components. In *Eighth International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, 2012.

[77] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. *ACM SIGPLAN Notices*, 48, 2013.

[78] T. Kalibera, L. Bulej, and P. Tuma. Automated detection of performance regressions: The mono experience. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2005.

[79] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *the International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS)*. SCS, 2005.

[80] S. Keskin. Comparison of several univariate normality tests regarding type i error rate and power of the test in simulation based small samples. *Journal of Applied Science Research*, 2, 2006.

[81] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32, 2015.

[82] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 1983.

[84] W. B. Langdon and M. Harman. Evolving a cuda kernel from an nvidia template. In *IEEE Congress on Evolutionary Computation*. IEEE, 2010.

[85] W. B. Langdon and M. Harman. Genetically improved cuda c++ software. In *European Conference on Genetic Programming (EuroGP)*. Springer, 2014.

[86] W. B. Langdon and M. Harman. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19, 2014.

[87] W. B. Langdon and M. Harman. Grow and graft a better cuda pknotsrg for rna pseudoknot free energy calculation. In *the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO)*, 2015.

[88] W. B. Langdon, J. Petke, B. R. Bruce, E. Hart, P. R. Lewis, M. López-Ibáñez, G. Ochoa, and B. Paechter. Optimising quantisation noise in energy measurement. In *Parallel Problem Solving from Nature (PPSN)*. Springer, 2016.

[89] W. B. Langdon, W. Weimer, J. Petke, E. Fredericks, S. Lee, E. Winter, M. Basios, M. B. Cohen, A. Blot, M. Wagner, et al. Genetic improvement@ icse 2020. *ACM SIGSOFT Software Engineering Notes*, 2020.

[90] A. M. Law, W. D. Kelton, and W. D. Kelton. *Simulation modeling and analysis*, volume 2. McGraw-Hill New York, 1991.

[91] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38, 2011.

[92] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012.

[93] B. Li, J. Li, K. Tang, and X. Yao. Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys*, 48, 2015.

[94] D. Li and W. Halfond. Optimizing energy of http requests in android applications. In *3rd International Workshop on Software Development Lifecycle for Mobile*, 2015.

[95] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2013.

[96] D. Li, A. H. Tran, and W. G. Halfond. Making web applications more energy efficient for OLED smartphones. In *36th International Conference on Software Engineering (ICSE)*, 2014.

[97] D. Li, A. H. Tran, and W. G. Halfond. Making web applications more energy efficient for oled smartphones. In *the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014.

[98] M. Li, J. Lu, Z. Chen, and K. Amine. 30 years of lithium-ion batteries. *Advanced Materials*, 30, 2018.

[99] D. Limited. Wifi8020 - 20 x 16a wifi relay. `http://www.robot-electronics.co.uk/wifi8020-20-x-16a-relay-module.html`, 2018. Accessed on 8 May 2020.

[100] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *the 2013 9th joint meeting on Foundations of Software Engineering (FSE)*, 2013.

[101] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2015.

[102] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *the 36th international conference on software engineering (ICSE)*, 2014.

[103] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2010.

[104] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *36th International Conference on Software Engineering (ICSE)*. ACM, 2014.

[105] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause. An empirical study of practitioners' perspectives on green software engineering. In *38th International Conference on Software Engineering (ICSE)*, 2016.

[106] S. McConnell. *Code complete*. Pearson Education, 2004.

[107] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, 2011.

[108] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *IEEE International Conference on Software Maintenance (ICSM)*. IEEE Press, 2013.

[109] M. Mendes and A. Pala. Type i error rate and power of three normality tests. *Pakistan Journal of Information and Technology*, 2, 2003.

[110] E. Mininno and F. Neri. A memetic differential evolution approach in noisy optimization. *Memetic Computing*, 2, 2010.

[111] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, 2018.

[112] R. Murmuria, J. Medsger, A. Stavrou, and J. M. Voas. Mobile application and device power usage measurements. In *Software Security and Reliability*. IEEE, 2012.

[113] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *ACM SIGARCH Computer Architecture News*, 37, 2009.

[114] J. Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[115] A. Noureddine and A. Rajan. Optimising energy consumption of design patterns. In *37th International Conference on Software Engineering (ICSE)*. IEEE Press, 2015.

[116] S. Overflow. Android - Proximity Sensor Accuracy , 2013. Accessed 22 June 2020, http://stackoverflow.com/questions/16876516/proximity-sensor-accuracy/29954988.

[117] B. R. Overholser and K. M. Sowinski. Biostatistics primer: part 2. *Nutrition in clinical practice*, 23, 2008.

[118] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33, 2016.

[119] T. Park and K. R. Ryu. Accumulative sampling for noisy evolutionary multi-objective optimization. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2011.

[120] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[121] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *European Conference on Computer Systems (EuroSys)*. ACM, 2011.

[122] F. Pellacini. User-configurable automatic shader simplification. *ACM Transactions on Graphics*, 24, 2005.

[123] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. P. Fernandes. The influence of the java collection framework on overall energy consumption. In *5th International Workshop on Green and Sustainable Software (GREENS)*. ACM, 2016.

[124] J. Petke, W. B. Langdon, and M. Harman. Applying genetic improvement to minisat. In *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2013.

[125] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22, 2018.

[126] A. D. Pietro, L. While, and L. Barone. Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions. In *IEEE Congress on Evolutionary Computation*, volume 2, 2004.

[127] G. Pinto and F. Castor. Energy efficiency: a new concern for application software developers. *Communications of the ACM*, 60, 2017.

[128] G. Pinto, F. Soares-Neto, and F. Castor. Refactoring for energy efficiency: A reflection on the state of the art. In *the IEEE/ACM 4th International Workshop on Green and Sustainable Software*. IEEE, 2015.

[129] W. H. Press and S. A. Teukolsky. Adaptive stepsize runge-kutta integration. *Computers in Physics*, 6, 1992.

[130] A. O. S. Project. Fix systemui wallpaper crash, 2020. URL `https://android-review.googlesource.com/c/platform/frameworks/base/+/1321016`. Accessed on 1 July 2020.

[131] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread tranquilizer: Dynamically reducing performance variation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8, 2012.

[132] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *the 36th International Conference on Software Engineering (ICSE)*, 2014.

[133] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *the International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[134] P. Rakshit and A. Konar. *Principles in Noisy Optimization: Applied to Multi-agent Coordination*. Springer, 2018.

[135] C. Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46, 1895.

[136] R. Saborido, V. V. Arnaoudova, G. Beltrame, F. Khomh, and G. Antoniol. On the impact of sampling frequency on software energy measurements. Technical report, PeerJ PrePrints, 2015.

[137] N. Sachindran and J. E. B. Moss. Mark-copy: Fast copying gc with less space overhead. In *ACM SIGPLAN Notices*, volume 38. ACM, 2003.

[138] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. Post-compiler software optimization for reducing energy. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014.

[139] H.-P. Schwefel. *Numerische optimierung von computer-modellen mittels der evolutionsstrategie.(Teil 1, Kap. 1-5)*. Birkhäuser, 1977.

[140] ScientiaMobile. Mobile overview report (movr) 2019 q2, 2019. URL https://www.scientiamobile.com/movr-mobile-overview-report/. Accessed on 30 June 2020.

[141] D. P. Searson, D. E. Leahy, and M. J. Willis. Gptips: an open source genetic programming toolbox for multigene symbolic regression. In *International multiconference of engineers and computer scientists*, volume 1. Citeseer, 2010.

[142] Shanhong Liu. Android os platform version market share 2013-2019, 2020. URL https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/. Accessed on 30 June 2020.

[143] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52, 1965.

[144] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Symposium on Microarchitecture*. ACM, 2009.

[145] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (FSE)*, 2011.

[146] F. Siegmund, A. H. C. Ng, and K. Deb. A comparative study of dynamic resampling strategies for guided evolutionary multi-objective optimization. In *IEEE Congress on Evolutionary Computation*, 2013.

[147] F. Siegmund, A. H. C. Ng, K. Deb, C. Henggeler Antunes, and C. C. Coello. Hybrid dynamic resampling for guided evolutionary multi-objective optimization. In *Evolutionary Multi-Criterion Optimization*. Springer, 2015.

[148] O. Signal. Android fragmentation visualised, 2015. URL `www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf`. Accessed on 30 June 2020.

[149] D. Sillars. *High Performance Android Apps: Improve ratings with speed, optimizations, and testing*. O'Reilly Media, Inc., 2015.

[150] A. Sinha and A. P. Chandrakasan. Jouletrack-a web based tool for software energy profiling. In *38th Design Automation Conference (DAC)*, 2001.

[151] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence. Genetic Programming for shader simplification. *ACM Transactions on Graphics*, 30, 2011.

[152] C. Spearman. " general intelligence" objectively determined and measured. 1961.

[153] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *Computer*, 27, 1994.

[154] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2001.

[155] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 1997.

[156] A. Syberfeldt, A. Ng, R. I. John, and P. Moore. Evolutionary optimisation of noisy multi-objective problems using confidence-based dynamic resampling. *European Journal of Operational Research*, 204, 2010.

[157] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. Repairing crashes in android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

[158] S. Tarkoma, M. Siekkinen, E. Lagerspetz, and Y. Xiao. *Smartphone energy consumption: modeling and optimization*. Cambridge University Press, 2014.

[159] H. Thimbleby and D. Williams. A tool for publishing reproducible algorithms & a reproducible, elegant algorithm for sequential experiments. *Science of Computer Programming*, 156, 2018. doi: https://doi.org/10.1016/j.scico.2017.12.010. URL `http://www.sciencedirect.com/science/article/pii/S0167642317302952`.

[160] J. Tiongson. Mobile app: Marketing insights, 2015. URL `http://tiny.cc/twg`.

[161] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice. Exhausting battery statistics: understanding the energy demands on mobile handsets. In *Workshop on Networking, systems, and applications on mobile handhelds*. ACM, 2010.

[162] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25, 2000.

[163] K. D. Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. *CoRR*, abs/1401.4655, 2014.

[164] P. Walsh and C. Ryan. Automatic conversion of programs from serial to parallel using genetic programming - the paragen system. In *PARCO*, 1995.

[165] L. Wei, Y. Liu, and S.-C. Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[166] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *the IEEE 31st International Conference on Software Engineering (ICSE)*. IEEE, 2009.

[167] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013.

[168] D. R. White. *Genetic programming for low-resource systems*. PhD thesis, University of York, 2009.

[169] F. Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 1992.

[170] R. S. Williams. Computing in the 21st century: nanocircuitry, defect tolerance and quantum logic. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 356, 1998.

[171] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke. Deep parameter optimisation. In *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2015.

[172] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[173] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[174] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *the FSE/SDP workshop on Future of software engineering research*, 2010.

[175] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22, 2012.

[176] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In *USENIX Conference on Annual Technical Conference (ATC)*. USENIX Association, 2012.

[177] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis*. ACM, 2010.

[178] Z. Zhang and T. Xin. Immune algorithm with adaptive sampling in noisy environments and its application to stochastic optimization problems. *IEEE Computational Intelligence Magazine*, 2, 2007.

[179] L. Zhong and N. K. Jha. Graphical user interface energy characterization for handheld computers. In *the international conference on Compilers, architecture and synthesis for embedded systems (CASES)*, 2003.