Improving the Non-Functional Properties of Android Applications with Genetic Improvement

James Callan

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

March 19, 2024

I, James Callan, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work. The following chapters are based on the publications listed below. With regards to **Chapter 8**, the contribution of this thesis is the completion of a proof-of-concept implementation described in the chapter.

Chapter 3 is based on:

James Callan and Justyna Petke. "Improving Android App Responsiveness Through Automated Frame Rate Reduction." *Proceedings of the 13th International Symposium on Search-Based Software Engineering. Springer International Publishing. 2021.*

Chapter 4 is based on:

James Callan, Oliver Krauss, Justyna Petke, and Federica Sarro "How do Android developers improve non-functional properties of software?." *Empirical Software Engineering, volume 27, no.5. Springer. 2022.*

Chapter 5 is based on:

James Callan, and Justyna Petke. "Improving responsiveness of Android activity navigation via genetic improvement." *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. 2022.*

Chapter 6 is based on:

Janes Callan and Justyna Petke. "Multi-Objective Improvement of Android Applications." *arXiv preprint arXiv:2308.11387 (2023). Under Review at Springer's Automated Software Engineering journal.*

Chapter 7 is based on:

James Callan and Justyna Petke. "Reducing the Network Usage of Android Applications with Genetic Improvement." *To be submitted to the 13th Genetic Improvement Workshop at ICSE 2024.*

Additional works

1. Giovani Guizzo, Aymeric Blot, James Callan, Justyna Petke, and Federica Sarro. "Refining Fitness Functions for Search-Based Automated Program Repair: A Case Study with ARJA and ARJA-e." *Proceedings of the 13th*

International Symposium on Search-Based Software Engineering. Springer International Publishing. 2021.

2. James Callan, and Justyna Petke. "Optimising SQL queries using genetic improvement." *IEEE/ACM International Workshop on Genetic Improvement (GI). IEEE, 2021.*
3. James Callan, and Justyna Petke. "Multi-objective Genetic Improvement: A Case Study with EvoSuite." *Proceedings of the International Symposium on Search Based Software Engineering. Springer International Publishing. 2022*
4. Dominik Sobania, Alina Geiger, James Callan, Alexander Brownlee, Carol Hanna, Rebecca Moussa, Mar Zamorano, Justyna Petke, and Federica Sarro. "Evaluating Explanations for Software Patches Generated by Large Language Models." *International Symposium on Search-Based Software Engineering, Challenge Track, Accepted. 2023.*
5. Alexander E.I. Brownlee, James Callan, Karine Even-Mendoza, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania "Enhancing Genetic Improvement Mutations Using Large Language Models" *International Symposium on Search-Based Software Engineering, Challenge Track, Accepted. 2023.*
6. David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, Federica Sarro: "User-Centric Deployment of Automated Program Repair at Bloomberg." *International Conference on Software Engineering, Software Engineering In Practice Track, Accepted. 2024.*

Abstract

There are 3.5 billion Android applications on the Google Play Store. However, surprisingly little work exists on automatically improving the source code of Android apps, especially when compared to traditional software. Genetic Improvement is a technique for automatically improving software which has proven successful in the past. However, its applicability in the Android domain is yet to be explored. In this thesis, we explore how GI can be used to improve Android apps.

The first contribution of this thesis is an investigation into applying GI to Android with minimal changes from the standard technique, however, we achieved limited success. Next, we mined git repositories to try to find the changes that real developers make to improve the non-functional properties of applications. With what we learned in this study, we modified and successfully applied GI to improve the responsiveness of Android applications. We then moved on to multi-objective improvement, improving execution time and memory usage, but failing to improve network usage. We also provide a benchmark of tested applications that can be used to evaluate future automated improvement tools. We developed a profiler to find the most network-intensive methods to target and a novel mutation operator. Our final contribution is an adapted version of GI framework for network usage optimisation.

We found that Genetic Improvement is an effective tool for improving multiple non-functional properties of Android apps. We found that by using simulation-based testing, rather than testing variants on devices, we could make GI faster and more practical. We found that there were many opportunities for GI to more closely mimic the types of changes made by developers and that caching in particular is an effective change type. We recommend future work further explores this direction.

Impact Statement

This thesis can impact a number of groups of people, including Android Developers who can use the software developed to automatically improve the applications that they develop, and also researchers who are working on both genetic improvement (GI) and the automatic optimisation of Android applications.

In our first piece of work, we attempt to improve the frame rate of apps. However, found that, in its current state, GI is not suitable for this particular task. In the work, we gained many insights into how GI can be applied to Android and highlighted the most pressing current limitations to future researchers.

Next, we mined real developer commits which improved the non-functional properties of Android apps. This work will give developers knowledge on the kinds of changes they can make to their applications to improve them in the future. It will also inform future research into improving the non-functional properties of apps about the types of changes that are most effective.

With the results of this study, we rethought the framework to improve it, based on the results of our study and experiences applying GI to Android. We applied this framework to successfully optimise navigation responsiveness, providing a fully open-source tool for the automatic improvement of the responsiveness of Android apps to both developers and researchers.

Then, we extended this tool with three Multi-objective algorithms, allowing more than one property to be improved simultaneously, or for trade-offs between properties to be found. Again we made this tool public so that Android developers can use it for their apps. However, when applying this approach to improving the network usage of apps, we failed to find improvements, showing researchers that

genetic improvement needed modifications to be applied to this particular problem.

Finally, we tried to adapt genetic improvement to improve the network usage of applications. We began by developing a profiler to detect the most network-intensive methods in applications so that we could target them. We made this tool freely and openly available to developers to find the parts of their apps that use the most network and researchers can use it in future work in this area. We also provide a version of our framework with new mutation operators to target network usage specifically. Unfortunately, these operators were not successful in finding improvements but may be useful to researchers improving other properties, again we make this tool openly available.

Acknowledgements

Firstly, I'd like to thank my supervisor, Professor Justyna Petke, for her motivation, patience, and guidance throughout my degree and for teaching me countless valuable lessons. Working with Justyna has been a great pleasure and I couldn't have asked for a better supervisor.

Thank you to everyone in CREST and SOLAR for both the invaluable help you have given me in my research and for the great times we have had socialising. A special thanks to Dr. Max Hort and Dr. Giovani Guizzo for our many great trips to the pub.

Finally, I would like to thank all of my friends and family, especially my parents Chris and Colette Callan, who have been there for me throughout my entire education. And Paige, for supporting me for all for of those years. Without her, I never would have even started this degree, let alone made it to the end.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android applications (or apps for short) are one of the most widely used types of software [1]. They are designed for direct user interaction, with the main entry point for the software being its UI components. Due to the small size of Android devices (phones and tablets) compared to traditional desktop devices, their hardware capabilities are naturally limited. These two factors result in non-functional properties being especially important to both users and developers. In fact, non-functional properties are so important to Android users that 1/3 of instances of users abandoning applications [2] and 59% of bad reviews were due to poor performance [3]. Banerjee and Roychoudhury [4] analysed 170,000 user reviews of mobile applications, and classified reasons for user downvotes. Three out of five identified categories related to non-functional properties. Khalid et al.'s [5]'s study of iOS applications also showed that unresponsive, resource-heavy applications and those with network-related issues were among the top most frequent sources of complaints. Liu et al. [6] found that out of 60,000 randomly sampled applications, over 10,000 contained reported or repaired performance issues. Gao et al. [7] reported that users would leave negative reviews for applications that had advertisements that negatively impacted performance. Gao et al. [8] found a strong correlation between performance cost and negative user reviews.

However, despite the impact of non-functional properties on user experience and the prevalence of bugs in source code which affect them [9], there is limited work into improving Android performance.

In the past, work improving the non-functional properties of Android apps has utilised approaches including prefetching, offloading, OLED screen management, and device configuration. However, there has been limited work into automatically refactoring Android applications. Genetic Improvement (GI) has proven successful in improving non-functional properties through patch generation. GI uses meta-heuristics to generate patches that improve some aspect of a program. In this thesis, we will explore how we can use GI to improve the non-functional properties of Android applications.

The contributions of the thesis are as follows:

1. An open-source tool, GIDroid, for running Genetic Improvement in Android.
2. Three novel mutation operators for Genetic Improvement specifically developed to provide improvements to memory, execution time, and network use.
3. A profiler for detecting network-intensive methods in Android.
4. An empirical study detailing the ways in which developers improve the non-functional properties of Android apps.
5. Experiments showing that improvements of up to 50% to frame rate, 30% to responsiveness, 35% to execution time, and 69% to memory consumption are possible with GI in the Android domain.

We make all tools and results available at our website `solar-group.github.io/os/android.html`

This thesis is structured as follows. We begin by discussing the related work and background of this work. Next, we detail our work in improving the frame rate of Android applications with GI. In the fourth Chapter, we show how we improved the responsiveness of Android applications using GI with only local testing. Then, we discuss a study in which we mined real developer commits to see how they actually improve the non-functional properties of Android Applications. In Chapter 6, we discuss how we modified our framework with the information gathered from commits to improve GI for multi-objective optimization of Android apps. Next, we detail our attempts to reduce the network usage of apps with GI. Finally, we discuss the threats to the validity of our work and the general conclusions made.

Chapter 2

Background and Related Work

This chapter is concerned with the background knowledge needed to understand the work presented in the thesis, along with the related work from the literature, needed to contextualize the work. We begin by discussing the Android environment. In particular, how Android app development differs from traditional software development. Then we discuss the previous approaches used to improve non-functional properties of Android applications.

Next, we discuss Genetic Improvement. Discussing how GI works, and how it can be applied to software. Finally, we present the previous applications of GI found in the literature.

2.1 Android Applications

Android applications are programs that are able to be installed and run on devices running the Android Operating System. The majority of apps are developed for mobile phones, with 2.87 million apps on the Google Play Store alone [10]. Android applications are what does generally written mean generally written in languages that run in the Java Virtual Machine (JVM), although other languages can be used. These languages are primarily either Java or Kotlin, but components of apps can also be compiled native code written in C/C++.

Despite this, Android applications differ from traditional JVM-based software in a number of ways.

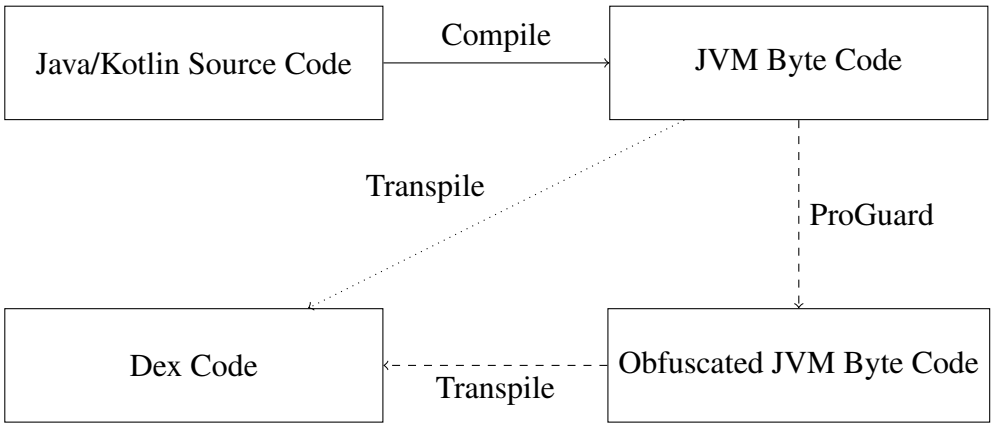


Figure 2.1: Android Application Compilation Process.

2.1.1 Compilation

The first way that Android apps differ from traditional software is in compilation. In a standard JVM-based language like Java or Kotlin, the source code is simply compiled into JVM byte code, and can then be optionally packaged into a jar file. In Android, we must perform extra steps. Android Applications are not run with the JVM. Instead, they are run with either the Dalvik Virtual Machine (DVM), in Android versions up to 4.4, or the Android RunTime (ART)[1] in more recent versions. The DVM works in a similar way to the JVM, translating byte code into machine code at run time. However, the ART performs compilation at installation time, making installation slower, but offering performance benefits at run time. However, rather than running the same byte code as the JVM, Android has its own byte code called dex code. To get dex code from Java or Kotlin source code, we must first compile it to JVM byte code, then transpile it to dex code. Optionally between these two steps we can obfuscate the byte code using a tool called ProGuard. This process is illustrated in Figure 2.1.

Next, we must package the code into an Android Application Package (APK) or, less commonly, an Android App Bundle (AAB). APKs are the standard file format for applications, whereas AABs contain everything needed for the Google Play Store to generate an APK whilst allowing for smaller file sizes. In this process, we combine the application's byte code and resources into an executable file that

[1] `https://source.android.com/docs/core/runtime`

can be installed and run on an Android device. An application's resources include XML files defining the layouts of UI elements, images, audio files, and the strings displayed in the app.

Finally, the application can be installed and run on a device or emulator.

We generally perform this whole process using the Gradle build system, using the Android-specific Gradle plugin. Gradle can manage most of the tasks needed to successfully deploy an application, including compilation, testing, dependency management, and linting.

2.1.2 Testing

The other major difference between traditional software and Android applications is the way that they are tested. There are two types of tests that can be used to validate the behavior of Android applications: connected tests, local tests [11].

Firstly there are connected tests. These tests run on either devices or emulators. These tests can exercise any Android code and are suggested to be used for integration testing. We can utilise testing libraries such as Espresso [2] to interact with the actual application under test. These tests are, however, very slow. They require the entire application to undergo compilation, packaging, installation, and launch.

Secondly, Android apps can be tested using local tests. These tests require much less time to compile and run than connected tests, as they only require the source code of the application to be compiled into JVM byte code. However, many of the APIs that apps use are not available in these tests and are replaced with stubs. The missing APIs include UI rendering, File IO, and sensor reading. Crucially, there is no way to access the state of the application as no application is actually launched. This severely restricts the amount of code that can be exercised with unit tests and they are mostly only useful for small, static methods.

Finally, random input generation techniques can be used to perform system testing on apps. The first of these techniques is Monkey [12], which is developed and maintained by the Android Development team. Monkey simply simu-

[2] `https://developer.android.com/training/testing/espresso`

Tool	Property
systrace	Memory Usage
dumpsys-gfxinfo	UI rendering
dumpsys-netstats	Network Usage
dumpsys-batterystats	Energy Usage
dumpsys-meminfo	Memory Usage
perfetto	Memory Usage, CPU Time

Table 2.1: Tools available in the Android SDK for measuring the performance of apps.

lates random touches on the device or emulator on which the application is running. These simulated inputs aim to find crashes in the app. There is a large body of work aiming to improve upon Monkey. This began with Amalfitano et al. [13] using a model-based approach to generate simulated inputs which could more effectively explore code and uncover more crashes. It continued with more model-based approaches, including Azim et al.'s [14] A^3E, Choi et al.'s [15] SwiftHand, and Yang et al.'s [16] ORBIT. Other works, including Mahmood et al.'s EvoDroid [17] and Mao et al.'s Sapienz [18] opted for search-based approaches, where input sequences are stochastically generated and improved with search algorithms, aiming to optimise the amount of code which is exercised and the number of crashes which are induced. However, no assertions can be made when using these tools so they cannot find bugs in the same way that traditional unit/integration tests can, making them useful for regression testing.

2.1.3 Measuring Non-Functional Properties of Android Apps

When improving non-functional properties we must be able to measure them. Whilst some non-functional properties, such as code quality, can be measured statically, many require the application to be exercised. For example, to measure the speed of a method in an application, we must actually run the method.

To exercise apps we can use the previously described testing techniques, however, the choice of technique may limit the properties that can be measured. If testing is performed on an actual device, many of the tools that would be used to measure the non-functional properties are not available.

Only a small selection of tools is made available through the Android operating

system. The tools made available through the Android SDK are shown in Table 2.1

One of the most useful tools available is `dumpsys`. `dumpsys` provides a wide range of information about different apps and services on a device, this includes information about memory usage, battery usage, and UI performance.

Whilst the other tools are useful, they do not provide their information in a concise machine-readable format like `dumpsys`, being more useful to human developers than automated tools.

When measuring non-functional properties using unit tests many other tools can be used as the tests just become a normal process. For example, the Linux time tool [19] can be used to measure the execution time of a method.

2.2 Non-functional Improvements to Android Apps

Hort et al [20] conducted a survey into the work on improving the NFPs of Android apps. Below we detail the works listed in the survey, along with more recent publications.

To find more recent works, we searched a number of online repositories for related works. The repositories were IEEE Access, ACM Digital Library, DBLP, and Google Scholar. We used a keyword search of the titles and abstracts of papers for keywords relating to Android (Android, Mobile) and non-functional properties (speed, performance, memory, bandwidth, network, efficiency) and Genetic Improvement to identify relevant papers. We also searched the proceedings of relevant venues (ICSE, FSE, ASE, GECCO, SSBSE, and MOBILESoft).

We have identified the main approaches for improving the NFPs of Android applications as:

1. Offloading
2. Preloading
3. OLED Screen Management
4. Device Configuration
5. Refactoring

In this section we will discuss the previous work that concerns these ap-

proaches.

2.2.1 Offloading

Offloading involves not only running the application on the actual Android device on which it is installed but also running the application on an external server. Most of the research in this area attempts to find the parts of the application that should be run remotely and when the application on the device and the application on the server should communicate. The main aim of offloading is to reduce the energy used by the Android application on the device, thus extending its battery life. However, it can also improve the responsiveness of the application if the remote code can be run quicker than it would on the device.

Das et al. [21] introduced the APPS system which supports both class-level offloading and thread migration to an external server, decreasing the execution time of applications by 60% and decreasing the energy usage by 70%. Montella et al. [22] used offloading to successfully adapt an application to run on low-resource devices by running its most expensive portions remotely. Chen et al. [23] derived the best dynamic offloading in ultra-dense networks where the high number of nodes in the network made regular offloading prohibitively complex. They did this by finding optimal times to offload computation, accounting for both the state of the mobile device and the state of the network. Kwon et al. [24] perform offloading on areas of code that are marked by developers as suspect. This achieves a 25-50% decrease in energy usage. Gordon et al. [25] run copies of apps both locally and on a remote server, in order to avoid deciding which will be faster. This results in an up to 2-3x speedup. Gordon et al. [26] introduced the COMET system to allow unmodified, multi-threaded applications to automatically run across multiple machines on the local network. COMET achieves this by implementing a distributed shared memory accross different machines and only sharing the fields which have been modified to minimise the interactions between different machines. Kemp et al. [27] introduced the Cuckoo framework for simplifying the development of applications that can exploit offloading and then chose what to offload at runtime. Chun et al. [28] developed the CloneCloud which partitioned applications into components that could

be run on a clone version of the program in the cloud. Kosta et al. [29] focused on increasing the scalability of mobile offloading by introducing multiple virtual machines in the cloud, allowing multiple components to be offloaded at once. Saarinen et al. [30] showed that offloading communications components of applications could lead to energy savings. Ding et al. [31] achieved energy savings of 80% by finding and offloading using the most energy-efficient WiFi access points available. Khairy et al. [32] used supervised learning to better predict when to offload, reducing both energy usage and execution time. Berg et al. [33] created safe points in execution from which offloading can happen. This allowed the applications to function correctly in cases where offloading was interrupted by executing the offloaded process locally. Ki et al. [34] combined offloading with two other energy saving techniques, dynamic voltage/frequency scaling and hybrid memory allocation, to make more energy efficient systems.

That being said, offloading requires external server architecture to be set up and engineering effort to modify applications or operating systems to allow them to take advantage of offloading. This setup would then only be available to users with internet access and may not be viable for users on limited data plans.

2.2.2 Prefetching

Prefetching involves downloading online assets at launch time and storing them until they are needed. This saves waiting for resources to be downloaded whilst the application is in use. Prefetching must be carefully considered so that large unnecessary files are not downloaded, increasing the network and memory usage of the application. Resources may also become outdated between download and usage, the ability to avoid this is known as freshness. If prefetching is performed on a cellular network rather than WiFi, it may result in higher energy usage [35].

Higgins et al. [36] provide a self-adjusting cost-benefit algorithm for determining when prefetching should be performed. They achieve this by only making network requests when the network is not already busy. This algorithm is run as a part of the operating system so no changes of source code are needed. Baumann et al. [35] propose the Every Byte Counts algorithm for choosing the best

time for prefetching based on probabilistic models of application usage and network traffic. This algorithm reduces the network usage of pre-fetching by 10% and improves the freshness, or the time between the acquisition of a network asset and its usage, by 36%. Mohan et al. [37] explore the prefetching of mobile ads in bulk to reduce energy usage. They find that prefetching ads can reduce the energy usage of mobile apps by 50% without impacting revenue. However, Chen et al. [38] found that 57 out of the top 100 free apps on the Google Play store could only reduce the energy consumption of ads with prefetching by an average of 3.2%. Zhao et al. [39] reduce the latency experienced by users by prefetching http requests before users trigger the requests.

Whilst pre-fetching has proven successful, it has also been show to quickly take up large amounts of the limited storage available on Android devices [40]. Moreover, prefetching can only be applied to areas of code which access the network and cannot take advantage of local caching opportunities.

2.2.3 OLED Screen Management

Both prefetching and offloading can improve the energy usage of apps. However, much of the work on improving energy consumption concerns OLED screen management.

Lin et al. [41] developed an app that allowed users to select an area of interest in the screen and dim the rest of the screen, whilst keeping it readable. Chen et al. [42] dimmed the parts of the touch screen covered by users' fingers, achieving energy savings of 13%. Lin et al. [43] reduced energy consumption by combining dynamically disabling pixels and scaling resolution. Lin et al. [44] proposed an algorithm for scaling images without impacting their visual quality by segementing images into different regions and dimming those which the user is less likely to be looking at. Linares-V'asquez et al. [45] propose an approach that uses multi-objective optimization to find energy-efficient colour schemes whilst also maintaining the contrast accross the UI, such that it is still readable, and the schemes are faithful to the original application. Li et al. [46] wrote a tool that rewrote web pages to use darker and more energy-efficient colour schemes, finding energy savings of 40%.

Anand et al. [47] used tone mapping techniques to increase the brightness of images, allowing the LED backlight to be dimmed, saving up 68% of display power. Chen et al. [48] dimmed areas of the screen, whilst highlighting the components of images being displayed which were important to the structure of the image using a highly parallelised region detection algorithm which could be run entirely on a GPU in real-time.

Approaches involving OLED screen management can however only affect the UI power usage and not any of the power used by other expensive components such as GPU sensors. Also, if the area of the screen that the user is interested in is dimmed, it may adversely affect their experience.

2.2.4 Device Configuration

Android devices have a number of configuration options that can improve the energy consumption and execution time of applications. Rao et al. [49] found the most energy-efficient options for CPU frequency and memory bandwidth options for a number of applications. To achieve this, they simply measure energy usage and performance for 18 different configurations for these parameters. When the application is running, they can the modify the CPU frequency and memory bandwidth to reduce energy consumption based on the previous measurements. Pyle et al. [50] improved energy usage by switching WiFi to a lower power usage state when its audio streams are silent. However, these techniques can come at the cost of performance. Kim et al. [51] achieved speed-ups by configuring devices to perform sequential IO calls with the pack command rather than seperately. Kim et al [52] achieved a 13% speed-up by tuning 5 parameters of the Ext-4 file system.

2.2.5 Refactoring

In refactoring, source code is modified to improve some properties of the software, whilst maintaining its functionality.

Lin et al. [53] proposed ASYNCDROID, a refactoring tool that helps a developer to convert incorrectly used asynchronous constructs into correct ones. However, this approach requires developers to identify each line of code that they wish

to execute asynchronously and there is no indication of the actual impact on performance of these changes. Lyu et al. [54] automatically refactored inefficient database writes, found in loops, into more optimal code.

Saborido et al. [55] investigated the cost of various versions of the map data structure for memory usage, execution time, and energy usage. They then suggest the most optimal version to use in different situations.

Cito et al. [56] automatically detected and throttled recurrent requests for advertisement and analytic requests. Li et al. [57] found http requests which could be bundled into single larger, but more energy-efficient requests. Banerjee et al. [4] found parts of code that violated energy usage guidelines and automatically refactored them, decreasing the energy usage of apps. Banerjee et al. [58] developed EnergyPatch to detect and fix resource leaks, improving energy efficiency. Cruz et al. [59] refactored apps to follow known energy-efficient design patterns Morales et al. [60] automatically improved the design quality of Android apps, whilst controlling for energy efficiency. Bokhari et al. [61] exposed and optimized 'deep parameters' in Android applications. Deep parameters are variables added into source code to modify the arguments of function calls [62]. These parameters are then tuned to reduce the energy usage of the application.

2.2.6 Summary

The approaches that have proven successful so far are either limited in their applicability or require patterns to be defined (prefetching, device configuration, and refactoring), require external hardware (offloading, OLED screen management), or are not fully automatic (offloading, refactoring). Bokhari et al.'s work [61] is the only approach that could be generically applied to source code, with no need for patterns, external hardware, or user input. Deep Parameter optimisation is a form of Genetic Improvement, finding optimal variants of source code using meta-heuristic search. Genetic Improvement has shown promise in improving the non-functional properties of traditional software and we believe that it could be used to improve many more non-functional properties in Android than just energy usage.

2.3 Genetic Improvement

According to our review (see Section 2.2), Genetic Improvement has yet to be fully explored for improving the non-functional properties of Android Applications. [63]

Genetic Improvement [63] uses meta-heuristics to generate patches for programs that improve them with regard to some property. Genetic Improvement has been used to repair programs, improve their non-functional properties, and transplant functionality between programs.

2.3.1 Representation

Genetic improvement attempts to find patches for source code. The source code variants used in genetic improvement are represented as a set of edits. This greatly reduces the size of the individuals which, thus reducing the memory footprint of the GI process compared to representing individuals with the whole source code of the program, which is standard in Genetic Programming – the most popular search strategy in GI. To find improvements, we must generate patches. These patches consist of a series of edits or mutations to source code, which could include line [64], statement [65], and bytecode [66, 67] changes.

The edits used rely on the plastic surgery hypothesis, which states:

> "Changes to a codebase contain snippets that already exist in the codebase at the time of the change, and these snippets can be efficiently found and exploited." [68]

2.3.2 Mutations

When using GI we must decide upon what kinds of changes we will make and define the search space of patches which we will explore.

GI aims to recreate the patches developers often make, consisting of the movement and deletion of code in the existing code base. The standard statement-level mutation operators used in Genetic Improvement [63] are:

delete(s_1) : Removes statement s_1

copy(s_1, s_2) : Copy statement s_1 in front of s_2

swap(s_1, s_2) : Swap statements s_1 and s_2

replace(s_1, s_2) : Replace statement s_1 with statement s_2

Similar mutation operators have also been used for lines of source code [64] and lines of byte code [66, 67].

Other mutation types have been used which do not simply remove and move code. Brownlee et al. [69] used mutations that inserted return and break statements, in some cases wrapped in if statements to speed up programs. Deep parameter optimisation exposes so-called deep parameters and modifies their values [62]. The mutation operators depend on the type of the parameter being mutated. In the case of integers, mutations will increase or decrease the value of the parameter. Mutations that replace data structures with others that have the same interfaces but different implementations (e.g. ArrayList vs LinkedList in Java) have been used to improve NFPs of software [70, 71].

2.3.3 Fitness

After patches are generated they are evaluated and the fitness of a patch is measured.

To evaluate a patch, the test suite of the software is run. When repairing a faulty program, we take the number of passing tests as the fitness. The fitness measurement is an indicator of how good a patch is in achieving a given objective When improving non-functional properties, all tests must pass for us to consider a patch valid. During this test suite's execution, the non-functional property that is being improved is measured to determine the fitness of the patch. For example, when improving the execution time of software we could use the time taken for the test suite to execute as a fitness measurement, which we would then aim to minimise.

2.3.4 Search

To generate and search for patches that improve programs, heuristics or meta-heuristics are used. Meta-heuristics are a class of algorithms that can be applied to a wide array of problems in order to find optimal, or near-optimal, solutions. According to Blum and Roli:

> "Metaheuristics are high level concepts for exploring search spaces by

> using different strategies. These strategies should be chosen in such a way that a dynamic balance is given between the exploitation of the accumulated search experience (which is commonly called intensification) and the exploration of the search space (which is commonly called diversification). This balance is necessary on one side to quickly identify regions in the search space with high quality solutions and on the other side not to waste too much time in regions of the search space which are either already explored or don't provide high quality solutions." [72]

In the context of GI, these algorithms are used to generate initial patches randomly, then iteratively improve the patches based on the selection of the fittest individuals, and the insertion and removal of edits in selected patches.

Blot and Petke [73] performed an empirical comparison of the most commonly used GI search algorithms, and their variants. In particular, they compared 8 variants of GP and 6 variants of LS. Thus, we explain the main components of each below.

In GP a population of patches is stochastically generated. Next, the fitness of these patches is measured. A new population is then created by repeatedly selecting individuals from the existing population. An Illustration of GI using GP to evolve increasingly optimal versions of software is shown in Figure 2.2.

Selection can be completed in a number of ways. In roulette wheel selection, each individual x in a population of size N has an associated probability of being selected $Pr(x)$ which is proportional to its fitness $f(x)$. We set $Pr(x)$ to the fitness of x, i.e., $f(x)$, divided by the sum of the fitness for each individual y in the population:

$$Pr(x) = \frac{f(x)}{\sum_{y=1}^{N} f(y)} \tag{2.1}$$

In tournament selection [74], groups of N individuals are selected, often 2, and either the fittest individual in the group is selected or a roulette wheel selection is run on the group.

Selections are then repeatedly made based on this distribution until N individ-

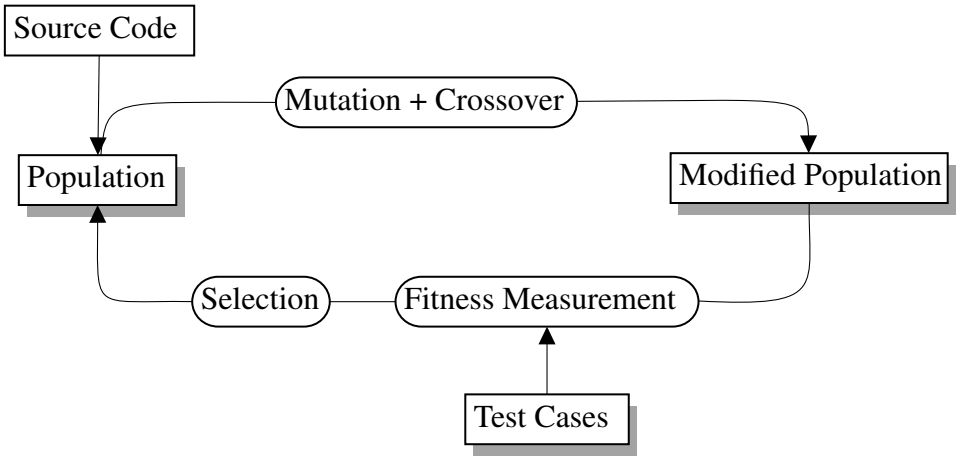


Figure 2.2: The Genetic Improvement Process Using a Genetic Programming Style Meta-Heuristic.

uals have been selected. This results in a population with a higher average fitness as there are likely to be multiple instances of high-fitness individuals and no instances of low-fitness individuals.

Mutation operators (removing/adding new edits) are then applied randomly. Crossover can also be used to combine patches, splitting the lists of edits for a patch in two and recombining with a segment of another selected patch's edit list, or appending a section of one patch to another. This process repeats for a set number of iterations or until a certain fitness is reached.

Local Search (LS) is another commonly used search algorithm. In local search, an empty patch is initialised and its fitness tested. Single random edits are then added or removed and the fitness is measured, if the fitness is improved the edited patch replaces the empty patch as the current best. This process repeats for a set number of evaluations, continually updating the current best solution. In the end, the current best is given as the best solution.

In general, different search algorithms offer different balances between exploration of the search landscape and exploitation of local optima in the landscape. In the case of GP, a more diverse set of individuals can be evaluated. But in LS improvements that have been found can be built upon, offering more exploitation.

When improving one property of a program, others may be affected. For example, if a variable is cached for use later it may improve execution time at the

detriment of memory usage. Multi-Objective Search Algorithms (MOSAs) can be used to consider these trade-offs during search and find the best trade-offs. A naive MOSA may just take weighted averages of different objectives. However, this requires a good understanding of the relationships between the objectives. The concept of Pareto dominance can instead be used to determine whether one solution is better than another. One individual x with objective values of $\{x_0, x_1, .., x_n\}$ Pareto Dominates another individual y, if all of x's objectives are as good as y's and at least one objective is better, that is:

$$\forall i \in 1..n : x_i \geq y_i$$
and
$$\exists i \in 1..n : x_i > y_i$$

The set of solutions that are not Pareto dominated by any others is the Pareto Front. Solutions in the Pareto Front are considered to be optimal trade-offs and the whole Pareto front is given instead of the single best patch. This allows developers to see which trade-offs are possible and select the most appropriate for the deployment environment of their software.

Once edit types, selection operators, mutation/crossover operators, and an appropriate meta-heuristic have been selected, we can use GI to improve any number of properties of a particular piece of software – as much of the previous work using GI has shown.

2.4 Applications of Genetic Improvement

Whilst there have been a number of works that try to find transformations to the code of Android applications, they have either required patterns, only been applicable to small areas of code, or only targeted energy efficiency

GI has shown promise in improving many properties of traditional software and is generally applicable to all areas of code.

Genetic Improvement has been used a number of times in the field of Automatic Program Repair (APR). Le Goues et al. [75] searched for variants of programs that pass all tests in a test suite, the similarity of the variant to the original code was also improved using delta debugging and structural difference algorithms to make

patches more similar to those that developers would write. Arcuri et al. [76] co-evolved source code and test cases based on a formal specification of the program, repairing bugs.

Genetic Improvement has also been used to improve other properties of, extend, or modify the functionality of programs on several occasions. Petke et al. [77] used GI to transplant functionality into a program to specialise it to a particular task. Barr et al. [78] used GI to transplant functionality between various programs, including video encoding functionality from the x264 system, to the VLC media player. Al Najar et al. [79] evolved programs that could more accurately forecast shoreline evolution. Federiks et al. [80] used GI to evolve grammars for better procedural generation of stories in games. O'Brien et al. [81] retargeted quantum programs for different hardware using GI.

Furthermore, one of the most common applications of Genetic Improvement has been the optimisation of various different non-functional properties of software.

Execution time has been one of the most common properties that previous work using genetic improvement has targeted. Langdon et al. [82] improved the execution time of the SEEDS image segmentation algorithm by 13%. Sitthi-amorn et al. [83] improved the execution time of a shader by over 80%, whilst sacrificing some of the fidelity of the images it produced. Langdon et al. [84] improved the speed of the bowtie2 tool by 45% for genome comparison using GI. Petke et al. [77] used GI to transplant code and improve execution time by 17%. Langdon et al. [85] improved the execution time of CUDA code for 3D medical imaging by 35%. Walsh et al. [86] used GI to automatically parallelise code, thus improving the execution time. Bruce et al., [87] decreased the execution time of a face detection algorithm by up to 45% using deep parameter optimisation. Langdon et al. [88], evolved LLVM representations of geographical open standards and achieved up to 2% better performance than code optimised by compilers. Zhong et al. [89] used genetic improvement to speed up Python programs by up to 94% automatically converting Python into statically typed Cython code, which is generally more efficient.

Genetic Improvement has also proven useful in improving the energy con-

sumption of software. Burles et al. [71] found energy-efficient replacements for data structures using GI. Bokhari et al. [61] used 'deep parameter optimisation' for energy optimisation on Android devices. White et al. [90] used genetic programming in order in to improve energy consumption of software on embedded systems. White et al. [91] optimised the energy consumption of random number generation software using genetic improvement. Schulte et al. [92] applied GI to compiled machine code in order to find energy optimisation.

Other work has targetted memory consumption and bandwidth usage. Wu et al. [62] exposed and modified 'deep parameters' to improve and find trade-offs between both memory and execution time. Basios et al. [70] swapped between 'Darwinian data structures' to improve both memory usage and execution time. Carbognin et al. [93] used GI to evolve novel congestion policies for network components, increasing network throughput by 5%.

Despite the resource-constrained environment of Android applications making them prime targets for improvement, GI had only been used once before the work presented in this thesis for Android applications. Bhokari et al. [61] improved the energy usage of applications using deep parameter optimisation. Therefore, in this thesis, we investigate how GI can be applied, and how effective it can be at the improvement of non-functional properties of software in the mobile domain, with a focus on Android. We start with trying to improve a property that is especially important to mobile phone users – Android app responsiveness.

Chapter 3

Improving Frame Rate

In this Chapter, we detail our initial experiments in applying Genetic Improvement to improve non-functional properties of Android applications. We start by applying single-objective GI to the responsiveness of Android apps.

Responsiveness is one of the most important qualities of Android applications to their users. Inukollu et al. found that 59% of users would give a bad review to an unresponsive app [3]. Khalid et al. [5] found that unresponsiveness was one of the most frequent reasons that users left bad reviews on mobile applications. Lim et al. [2] found that unresponsiveness was to blame 1/3 of the times when users abandoned applications.

In order to apply GI to responsiveness, we must first define a fitness function which we can measure. Responsiveness, however, can be difficult to quantify. Moreover, measurements such as execution time are inherently noisy and do not necessarily reflect user experience if long-running background operations are measured. Inspired by previous work [94], we propose using the frame rate of an application as a metric for responsiveness.

To explore this idea, we define a new framework for applying Genetic Improvement to improve the non-functional properties of Android. This framework allows Genetic Improvement to run on a desktop PC, whilst fitness measurements are delegated to devices or emulators. We use the gfxinfo tool which is built into the Android operating system to measure frame rate.

We identified the 20 most popular Android applications. However, GI relies

on testing to evaluate the fitness of individual program variants. In particular, we require tests that exercise the UI of applications, generating frames, to measure the frame rate of the application. Out of the 20 apps, only 4 had tests exercising UIs, thus we focused our experiment on those apps.

We find improvement of up to 50% in frame rate, although closer inspection reveals many of the patches to be invalid. This is due to weak test suites — used as proxies for correct program behavior, as is common in GI work. Nevertheless, we found a valid mutation that reduced the frame rate by 43%. Subsequently, we apply GI on code with high test-suite coverage. However, in these cases, no consistent improvements were found.

Our results show that although GI could be successful in the improvement of Android apps' responsiveness, any such test-based technique is currently hindered by the availability of test suites covering UI elements.

The rest of the chapter is structured as follows:

1. Section 3.1 details how we have modified GI to be suitable for Android. Including changes changes to the testing and compilation components of an existing framework.
2. Section 3.2 lists and explains the research questions that we wish to answer about how suitable GI is to the improvement of the frame rate of Android apps.
3. Section 3.3 describes the experiments we ran to answer our research questions.
4. Section 3.4 details the results of the experiments described in our methodology, showing how effectively GI can improve the frame rate of Android apps.
5. Section 3.6 discusses the conclusions of this research.

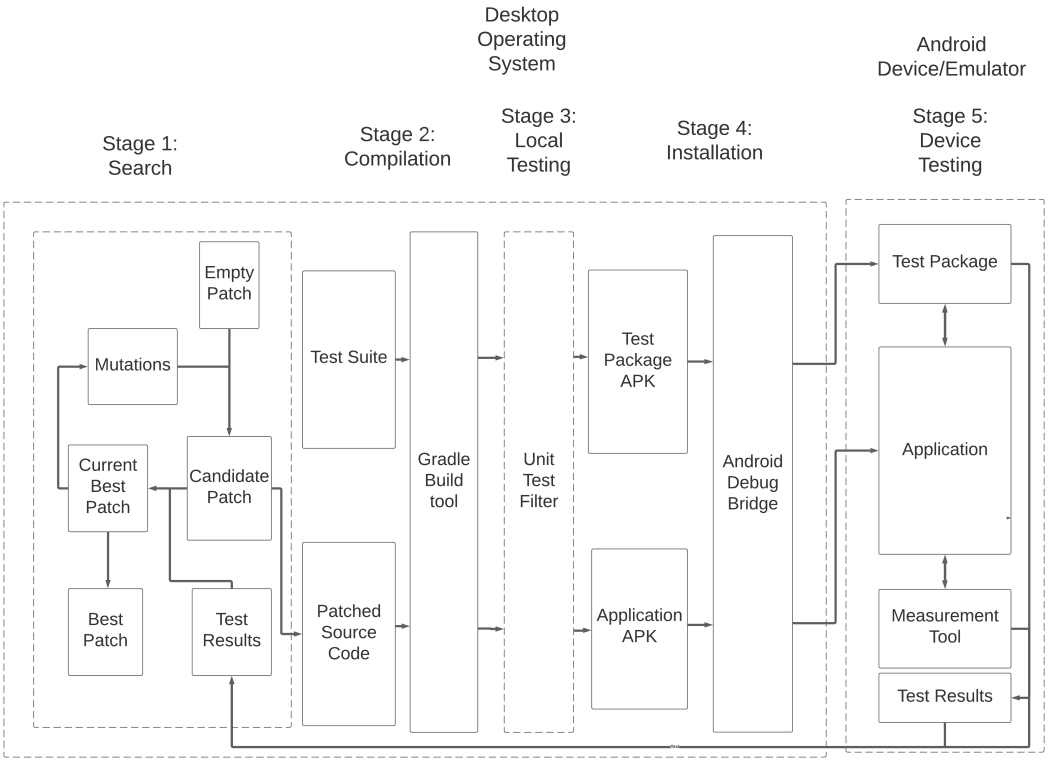


Figure 3.1: Genetic improvement framework for Android applications.

3.1 Improvement of Android App Responsiveness Using GI

The main challenge of applying GI in the Android domain to improve responsiveness lies in defining and evaluating the fitness function. In the past, responsiveness has been measured using the execution time [26, 29, 95] of test cases. Whilst this may capture responsiveness, it will be negatively impacted by long-running background processes which do not impact the actual responsiveness of the application. Gordon et al. [25] measured the "user-perceived latency" of interactions with applications, which is the time between a user input and the completion of the action it triggers. This metric requires user scenarios to be manually defined, including start and end points and does not allow us to utilise developer-defined UI tests. However, we chose to use frame rate as a proxy for responsiveness, as it is both easily measured and directly captures delays in updates to the UI. An application whose frames are not rendered in a timely manner will be unresponsive. Therefore, fixing

these delays will result in a more responsive application. We believe that frame rate, and thus responsiveness can be improved through source code transformations.

To measure an application's frame rate, we must exercise the application's UI on a device or emulator, so we cannot rely solely on local unit tests. This means that applications must be packaged and installed on a device or emulator, which is a costly process. It also removes our ability to use optimisation techniques such as in-memory compilation.

Therefore, we propose the general framework shown in Figure 3.1. The improvement process takes place across two devices: a desktop and an emulator or mobile device. All communication between the desktop device is performed by the Android debug bridge, running on the desktop device.

In the desktop environment, new patches are generated, through mutation and selection (Stage 1), patches are applied, and applications are built and packaged (Stage 2). Finally, local unit tests are run to determine whether or not a patch should be installed on the actual device (Stage 3). This step is important to vastly increase GI efficiency, as it reduces the number of program variants that need to be packaged and installed on a device or emulator, in order to measure their fitness. Patched applications that pass unit testing are then installed (Stage 4). On the Android device, modified versions of the application are exercised by the test package, and fitness measurements can be taken (Stage 5).

This framework could be easily used to improve any non-functional property, simply by specifying different measurement tools. It could also be extended to automated program repair by removing the measurement of a non-functional property and using the number of passing tests as the fitness function. Different search algorithms and mutation operators could also be tried in Stage 1 of the process. This framework also allows for parallelisation of the fitness evaluation process, by connecting multiple devices or emulators, though careful measures need to be taken to achieve reliable measurements (depending on the fitness function of interest).

3.2 Research Questions

In order to investigate the effectiveness of genetic improvement for the purpose of improving Android app responsiveness, we set out to answer the following research questions:

RQ1 *How effectively can genetic improvement optimise the responsiveness of Android applications?*

This question will explore how well simple line-level modifications to Android applications can improve their responsiveness and how easily we can automatically find effective transformations.

RQ2 *What type of source code changes are effective at decreasing frame rate in Android applications?*

The changes that we find to have the largest impact on frame rate could be used to inform developers of ways in which they can improve the responsiveness of their apps. They could also be useful in inspiring future automated techniques for improving the responsiveness of Android applications.

RQ3 *How expensive is it to improve the frame rate of Android applications using genetic improvement?*

This question will allow us to quantify whether it is worth it to run GI in this manner. We will be able to present the balance of cost running vs the improvement to allow developers to make an informed decision about applying GI. We will also explore how the cost varies between applications and what impacts the cost of running GI.

3.3 Methodology

In order to answer our research questions we implemented the framework presented in Figure 3.1, and run it on a selection of Android applications.

3.3.1 Framework

We realise the abstract framework presented in Figure 3.1 in the Gin GI tool [96]. We chose it as among non-functional property improvement GI tooling, Gin is scalable to large real-world software and is optimised for Java — a popular choice for

Android software. We utilise the pre-existing functionality from Gin which allows the generation and modification of source code files with line-level changes. We also use the existing local search algorithm from Gin. By default, local search is run for 100 steps, at each step either copying, deleting, or replacing a randomly selected line of code. We elected to run it for 400 steps to try to increase the chances of finding effective changes.

In order to run on Android and gather data for fitness evaluation, we have modified the components that compile the projects being improved and run their tests. We also added the functionality to install applications on Android devices and measure their frame rendering statistics.

3.3.1.1 Fitness

There are a number of different metrics which can be used to measure frame rate. They include the frames per second (FPS), the average time taken to render a frame, and the number of delayed frames. In order to measure the frame rate of an application, we first need to run it, exercising its UI. We use UI tests for this purpose and use the built-in *dumpsys gfxinfo* tool to gather various measures. The tool gives detailed statistics about the render times of frames of a particular process. These statistics include the number of janky frames (those that take longer than 1/60th of a second to render), the median and, various percentiles (50th, 75th, 90th, 95th, 99th) of frame render time are given. We ran the whole test suite of our selected applications 100 times, measuring all these metrics, and found that the 95th percentile of frame render time to be least noisy, thus we use it as our frame rate measurement. Improving this metric will mean that the largest delays in responsiveness have been fixed.

3.3.1.2 Testing

Patch evaluation consisted of running all test cases that covered the area of code being modified to ensure that the functionality of the project had been preserved. UI tests also had to be run to measure the frame rate of the application. To improve the efficiency of the GI process, we identify test cases that cover the given class for improvement, using *jacoco* [97]. Next, we use *espresso* [98] to identify UI tests.

Finally, we split the UI tests into two, based on 60% delayed frame rate measure. The reason for this split is two-fold: first, running tests on the emulator or device is expensive, so we want to avoid unnecessary runs; second, we want to have a held-out test suite to check the generalisability of improvements found. Therefore, for Stage 3 and Stage 5 of our GI process presented in Figure 3.1 we use UI tests causing the largest frame delays (over 60% frame delays), as well as all non-UI tests covering a given class. If all tests pass at Stage 3, we keep this program variant and evaluate its improvement in Stage 5, where each test is run 10 times, and the median 95th percentile frame render time is recorded. Due to the measurement of frame rate sometimes missing the test execution and not capturing the full execution of the test, small 3-second delays were added to the end of each UI test. This allowed the frame rate measurement to be consistently captured. Each performance test suite was then run until 200 frame measurements had been recorded. Before this the measurement could experience noise, leading to false positive improvements. Once 200 frames have been recorded we can see if the patch is in fact an improvement by comparing the median proportion of delayed frames in each test to that of the current best solution.

3.3.1.3 Search

Before we used the default local search implemented in Gin, we conducted a pre-study, to see if genetic programming (also implemented in Gin) might have been a better choice. Local search showed more promising results than GP as it was able to find optimised solutions faster. This is in line with the findings of Blot et al. [73]. We performed 20 runs on each of the selected classes in each of the projects. This allows us to collect a large amount of data and be confident about the efficacy of our setup, despite the non-deterministic nature of GI. We perform statistical tests on our results in order to quantify the effectiveness of GI at finding improvements.

3.3.2 Validation

For each final patch from each GI run, we use all tests covering a given class for validation purposes. We ran each 10 times and recorded median frame rate improve-

ment. This allowed us to run statistical tests on the results and see which patches offered significant improvements. The number of delayed frames was measured in the same way as during the GI runs. We performed this evaluation on a real device rather than an emulator, to ensure that improvements were valid in a real-world environment and test for device overfitting. We also conducted a manual analysis of the patches to confirm their validity.

3.3.3 Benchmarks: Mobile application Selection

We aim to improve real-world software and, therefore, choose to use real open-source applications. Since we are using the Gin improvement tool, our modifications are limited to Java source code. Android applications may consist of mixtures of Kotlin and Java source code, but only the part of the application being modified needs to be written in Java. In our GI framework, each patch is validated using the test suite of the application. This limits us to improving open-source applications with areas of code that are well-tested. Moreover, we need UI tests to measure frame rate. Therefore, a number of criteria had to be met by applications used in this study:

- The application must be open source and at least partly written in Java.
- The application must be able to be compiled and deployed on an Android Emulator.
- The application must have sufficient areas of code covered by a test suite (at least one class with 40% line coverage).
- The application must contain at least one test that exercises it's UI.
- The application must contain at least one non-trivial UI class. [1]

Checking these criteria for a given app is costly (particularly test coverage). The application must be downloaded, compiled, installed and tested. The coverage of the unit tests and the instrumented tests must be measured separately. Fortunately, Pecorelli et al. [99] performed an analysis of all applications from FDroid, documenting both the number of tests and the coverage of those tests.

[1] Based on manual judgment we decided to select applications with at least one UI class with at least 100 lines of code.

In order to curate a set of applications to evaluate our approach on, we checked applications analysed in [99] in descending order of line coverage. We then discarded applications that were not written in Java, those that could not be compiled, those whose tests could not be run successfully, and those that were too small for meaningful improvement to be found. If an application was not discarded, the areas of the application covered by its test suite had to be checked.

The first step in this process was to remove unreliable tests which would occasionally fail without any changes to the application - for two reasons. Firstly, the *jacoco* test coverage plugin [97] requires all tests to pass so unreliable tests could disrupt the coverage measurement. Secondly, unreliable tests may produce false negatives in patch validation. If a test fails due to unreliability, rather than due to the applied patch, it will make a valid patch appear invalid. Thus, they must be excluded from the experiments and, therefore, should be excluded from coverage measurements. In some cases, build files had to be modified to remove conflicting dependencies or enable test coverage measurement. No source code was modified in this process.

When running GI on a desktop application, automated test generation tools such as EvoSuite [100], can be used to supplement test suites and increase code coverage. Sadly there are limited tools available for automated test generation for Android applications and none that can automatically generate regression tests were found. We found 3 tools that could generate automatic UI test input, however, none worked on the recent versions of Android on which we ran our experiments. Even if they did work they generated no assertions, so could not be used to confirm patch validity. Therefore, the existing test suite of the application had to be relied on to validate patches.

Due to the large cost of validating a suitable application and the rarity of these applications, this process was repeated until 4 applications were found. Beyond this point line coverage was less than 15% so it was unlikely that more suitable applications would be found. Overall, we examined 192 applications, and 188 were discarded.

3.3.3.1 Profiling

Next, we profile each application we want to improve to identify code where changes influencing frame rate are most likely to be found. We thus focused on the UI implementing classes, the activity, view, and fragment classes. For each application, we select the class which is most covered by the jankiest UI tests, that has at least 100 lines of code. We added the second condition, as classes with few lines of code are unlikely to hold improvements.

However, UI tests often contain very few assertions, relative to the amount of code that they exercise, and unit tests for UI classes are very uncommon. Our proposed GI approach uses testing as a proxy for correctness. Because of this, while targetting UI-related classes may find the strongest improvements, it may also find invalid improvements due to the weaknesses of the test oracle.

Therefore, for each application, we select a class for improvement that is best covered by the whole test suite, and covered by at least one UI test, so we could measure frame rate.

In order to identify covered classes we used the *jacoco* Android coverage tool on each of the selected test cases Firstly, as *jacoco* only runs on whole test suites, we added JUnit's `@Ignore` decorators to all tests but the test case being investigated. We then ran *jacoco* on the modified test suite and extracted the coverage information, this process was repeated for each test. The classes which were most commonly exercised were then manually analyzed to check for suitability, as described above.

Table 3.1 shows the final set of applications we found using our selection procedure, including the classes we identified using our profiling procedure and their test coverage.

3.3.4 Physical setup

Our experiments were run on a research cluster, with 16GB of RAM and an Intel Xeon e5 CPU, with an emulator using Android version 7. The evaluation of improvements was performed on a NOKIA 9 running Android version 10.

Table 3.1: The number of test cases and % line coverage for each of the selected classes.

App Name	Class Name	Line Cov.(%)
AntennaPod	PreferenceActivity (Exp1)	43
	MainPreferencesFragment (Exp2)	68
Gnu Cash	AccountsListFragment (Exp1)	64
	GnuCashApplication (Exp2)	76
MicroPinner	MainDialog (Exp1)	44
	MainPresenterImpl (Exp2)	75
WikimediaCommons	AboutActivity (Exp1)	45
	RecentSearchesContentProvider (Exp2)	63

Table 3.2: Improvements found by our GI framework in poorly tested UI classes.

Project	No. imps. found	Max. % dec in 95th per. render time
AntennaPod	0	0.0
Gnu Cache	1	11.11
MicroPinner	1	5.56
Wikimedia Commons	8	50.00

3.4 Results

Below we present the results of our experiments. In our first set of experiments (Exp1), we ran GI 20 times on the class in each of the four projects which was most covered by janky UI tests. In our second experiment, for each project, (Exp2) we ran GI on the class with the highest line coverage, which was also covered by at least one UI test.

3.4.1 RQ1: Improvements to responsiveness

In order to answer RQ1, we present the improvement of frame rate before and after our patches are applied. Improvement is presented as the percentage decrease in the 95th percentile of frame render time. We also performed the Mann-Whitney U statistical test with the null hypothesis: *"There is no difference between the frame*

Table 3.3: Improvements found by our GI framework in well-tested classes.

Project	No. imps. found	Max. % dec in 95th per. render time
AntennaPod	0	0.00
Gnu Cache	0	0.00
MicroPinner	1	5.26
Wikimedia Commons	0	0.00

rate of the unpatched application and the patched application." for each patch discovered. This is to determine whether or not the improvements were statistically significant at the 95% confidence level. We treat those improvements as which are not statistically significant as 0% improvements. Tables 3.2 and 3.3 show our results.

We find that only 11 out of 160 of the GI runs performed found statistically significant improvements and 8 of those were in one application. In the vast majority of cases, no improvements were found and the GI execution simply returned an empty patch. In 7 cases in the first experiment, patches were found that suggested improvements during search, however, validation resulted in them being found not to offer statistically significant improvements.

We also measure the execution time and memory usage of the patches where statistically significant improvements to frame rate were found, in order to quantify the way frame rate improvements affect other metrics for responsiveness. However, we find that where improvements are found, there is very little effect on either memory consumption or execution time. These measurements are noisy and may not be sensitive to the types of improvements that we found.

There is also the chance that the applications simply are not unresponsive enough to find significant improvements. Visual observation of UI tests does show noticeable improvements, though not significant. This shows that indeed frame rate measurements we take are more sensitive to UI changes, and have a real, albeit small, impact. If tests were deliberately made to expose the unresponsive areas of applications, we may have an even better chance of finding improvements.

Answer to RQ1:. In most cases in our experiments, GI did not improve the frame rate of Android Applications. However, in cases where application test suites are effective, real improvements of up to 50% can be found.

3.4.2 RQ2: Types of Improvements

To understand the types of improvement that can improve the frame rate of an application, we undertook a manual investigation of patches. We investigated the edits

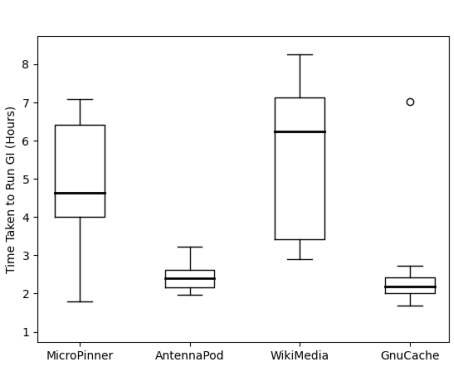
of the patch which was found to offer the most improvement in each project in order to find the most effective changes.

One patch in particular offered significantly better improvements than any other. A patch to the WikiMedia Commons application offered improvements of 50% to frame render time. This patch contained 3 edits, 1 more than any other patch found. 2 of these edits remove text from the screen, making the whole patch invalid. However, one of the changes removes a line setting the gravity of a drop-down menu's animation. Running this single change alone still produces a 43% improvement to frame render time, showing that it is the most important change. When deploying the modified version of the app we can see that opening and closing the drop-down menus is significantly smoother and there is no obvious visual impairment to the animation. This improvement will not have large effects on the execution time or memory consumption of the test suite, however, it does make the application run more smoothly from a user's perspective, fixing a stuttering animation.

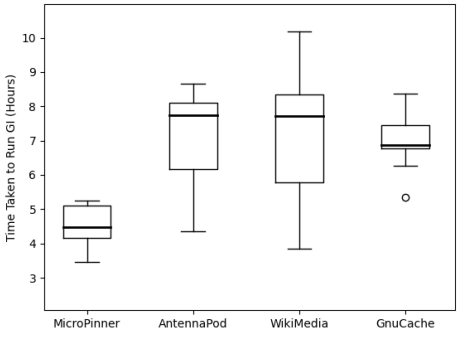
It is possible that there are other opportunities for this kind of change available. However, the majority of open-source applications have no tests, and those that do have very poor coverage [99].

In the cases where improvements found turned out to be invalid, again the classes being improved did not have adequate coverage and the tests which did cover were not very robust. Some patches removed lines of text that were meant to be displayed or prevented a dialog box from being displayed. In some cases, the lines which were removed were covered but there were no assertions to check that the text was being displayed correctly. Much stronger regression testing would be needed to remove the risk of invalid patches being produced. This issue was not found for the single improving patch produced for well-covered classes, only for the UI classes with lower coverage.

Answer to RQ2: The most effective change across our experiments was found to remove a line. This change modified the animation of a drop-down menu to run more smoothly.



(a) Times taken for experiments on UI classes.



(b) Times taken for experiments on well-covered classes.

Figure 3.2: Boxplots of the Time Taken for Each Run on Each Project in Hours

3.4.3 RQ3: Cost of Improving Responsiveness

In order to answer RQ3, to evaluate the cost of improvement, we timed the execution of each GI run. The results of this evaluation can be found in Figure 3.2. The runs took between 2 and 16 hours to complete. All of the experiments took a total of 883 hours of compute time.

The execution time varied greatly between projects and the runs on particular projects. This variance comes from differing lengths of test suites and the number of patches that could be built, and therefore tested, that were found. Trying to target classes that are covered by small, fast test suites would help to reduce the cost of GI.

Running tests on the emulator is very expensive, and almost certainly responsible for the long runtimes. When analysing the Wikimedia commons setup used for the About Activity class we find that running the unit test filter only requires a median of 5s over 10 runs. Whereas compiling, installing, and running the UI tests once takes a median of 2 minutes and 12 seconds over 10 runs. When running GI on Android in the future, it may be significantly faster to target properties that can be measured exclusively using local tests, removing the need for an emulator or real device.

Answer to RQ3: GI takes between 2 and 16 hours to run in our setup. The mean time taken by a run in this setup is 6.3 hours. The main factor determining the time taken is the time taken by the test suite of the application.

3.5 Threats to Validity

In this section, we discuss the threats to the validity of this work. Firstly, we only attempt to improve a small number of applications, this is primarily down to the lack of availability of apps tested whose UI is tested. However, the apps that we do test are from a wide range of domains and developers. We also attempt to make improvements to multiple classes in each project, finding similar results throughout.

Another threat is using tests to validate our improving patches. As we observed in our results, if test suites are weak, some patches generated by GI may not preserve the functionality of the application. But, we also demonstrated that standard code review can find disruptive changes, and we can even still extract effective changes from these patches.

Finally, noise in measurements is a threat to the validity of this work. If measurements are noisy, there is a chance that the improvements we observe are just random fluctuations in our measurement. We mitigate this threat in two ways, firstly, we measure the 95th percentile of the frame render time which filters out minor changes in how most frames are rendered and allows us to only observe when large changes occur. Secondly, when discussing our results, we use statistical tests upon repeated observations to ensure the improvements we observe exist.

3.6 Conclusions

In this chapter, we developed a framework for the application of Genetic Improvement to Android Applications. In particular, for the improvement of frame rate, we evaluated our framework by applying it to 4 open-source Android applications. However, we found that Genetic Improvement could only find a single patch that could improve frame rate without disrupting the functionality of the application. We do however show, that it is feasible to use meta-heuristic search to generate patches

for Android Apps. Whilst genetic improvement is capable of finding improvements to the frame rate of Android applications it is greatly limited by the number and distribution of available tests. In order for genetic improvement to be applied successfully, applications need more UI tests to allow janky areas of code to be exposed and more unit testing of UI elements to increase the code coverage.

Another possibility is that the current mutation operators used by GI are simply not well suited to improving the NFPs of Android apps. Therefore, in the next chapter, we undertake a mining study to see the types of changes that real developers make to improve the NFPs of Android apps. This study will allow us to understand how well-suited the current operators are, and which other operators could be developed in the future.

Chapter 4

How Do Developers Improve the Non-Functional Properties of Android Apps?

Given the limited success of our initial approach, we decided to look into the ways in which real developers make changes to improve the NFPs of Android apps. With this knowledge, we will be able to adjust our approach to more accurately mimic the changes which realm developers make and thus find better patches.

We pose that software repositories offer researchers a wealth of information about the behaviour and techniques used by actual developers. These can be used to find patterns that can be mimicked by search-based software engineering approaches for optimisation of non-functional software properties ([101]). Although several previous studies focus on performance bugs in traditional software, such as the study by Jin [102], only a few studies on mining performance improving commits in the Android domain exist (e.g., [21, 103]). Moreover, those do not provide fine-grained enough information to guide developers of search-based software development tooling. Previous studies were also concerned with finding general patterns across as many projects as possible, thus employed a sampling strategy that would alleviate the expensive manual analysis cost. This leads to an under-approximation of the true number of non-functional-property-improving changes.

To fill this gap we mined the most popular Android repositories, using single-

keyword search, and analyse all returned results, to find patterns that could be utilised in search-based automated software improvement tooling. We focus on four non-functional properties in particular: execution time, memory consumption, bandwidth usage, and frame rate. We chose these as they are most related to mobile app performance, the key issue for users, as previous studies show [4, 20].

First, we mined the repositories of the 20 most popularly downloaded mobile applications, according to Fossdroid, and manually examine the resultant 3,132 commits, finding 229 were actually NFP-improving ones. [1] Although this process should give us a good overview of non-functional property improving strategies for performance, it only allows for analysis of a relatively small number of repositories. However, the detailed analysis provides us with a corpus of data on which we can train a classifier that could help gather and analyse more data. Therefore, we devised such a classifier and analysed a further randomly selected set of 80 repositories, manually analysing 495 commits found, which added 331 non-functional property improving commits to our dataset. We categorised all the commits found, to help us identify emerging patterns. We also report on whether current automated improvement tools already allow for such transformations to be found, and if not, if such tools could be extended to provide new, useful software transformations. Finally, we examined features of the repositories we analysed. This is to provide recommendations for software developers, for what types of mobile applications non-functional improvements are likely to be found.

Our results show that non-functional property improvements to app performance are rare: from 74,408 commits mined across 100 repositories, only 560 were deemed to improve execution time, memory consumption, frame rate or bandwidth (229 identified by manual search and 331 by using a classifier). However, we can still draw interesting conclusions about their nature. In particular:

- In 10.7% of cases, developers were willing to sacrifice one non-functional property over another, while in 6.5% of cases, developers were able to im-

[1] In comparison, [103] found 371 energy-aware commits from a sample of 2,189 curated commits. It should be noted these span different numbers of repositories, and different keywords, corresponding to relevant non-functional software properties.

prove upon multiple properties at once. This shows the need for tooling that can handle multi-objective optimisation.

- The strongest indicators for the number of non-functional-property-improving commits in a repository was the total number of commits, number of contributors, and number of stars.
- Current search-based improvement tooling mimics 5 out of 23 non-functional improvement strategies found.
- Future automated techniques for improvement of non-functional properties could be enhanced by incorporating automated caching, SQL query, and image transformations. We propose detailed transformation patterns to aid researchers and developers in the design and adoption of such strategies.

Overall our results provide recommendations for software engineers, aiming to provide better tooling for automated software improvement; and for researchers, providing patterns of how developers improve mobile applications' non-functional properties related to mobile app performance, as well as a classifier that can help with future mining studies in this domain.

All our data and scripts are freely available to allow for reproduction (`https://github.com/SOLAR-group/NonFunctionalAndroidCommits`), replication and extension of our work.

The rest of this chapter is organised as follows:

1. Section 4.1 describes our methodology for mining commits related to non-functional properties.
2. Section 4.2 presents results of our mining study;
3. In Section 4.3 we discuss the implications of our study in software engineering research and practice.
4. Section 4.4 presents threats to validity.
5. Section 4.5 presents related work.
6. Section 4.6 concludes this chapter.
7. Appendix A contains additional material.

4.1 Methodology

In order to answer *how Android developers improve the performance-related non-functional properties of software (performance NFPs)* , and how we can use this knowledge to potentially devise new software transformations for tools for automated software improvement, we mine open-source Android projects for commits that improve four non-functional software properties (NFPs): *execution time*, *memory consumption*, *bandwidth usage*, and *frame rate*. Along with energy efficiency, previous research shows these are often found in user reviews ([58, 5]), yet have not been extensively tackled in the literature ([20]). [2]

We aim to answer the following research questions:

RQ1 *With what prevalence do developers improve performance NFPs of Android apps?*

NFPs of mobile applications impact user satisfaction, however it is not clear to what extent Android developers change their code to improve performance NFPs. The aim of this question is twofold: understanding if there exist NFP commits in Android open-source repositories to extract general patterns from, and understanding their characteristics.

RQ2 *How and when do Android developers improve app's performance NFPs?*

We want to know at which stage in software development do performance NFP-improving commits occur, whether these are considered as standalone improvements, and whether these improve multiple NFPs or prioritise one whilst possibly sacrificing another. These should give us an overview of the current Android development practice with respect to performance NFP improvement.

RQ3 *What type of code changes do Android developers make to improve app's performance NFPs?*

We want to also investigate what sort of changes developers make to source code to improve its performance NFPs. Examining these changes will allow

[2] We omit energy commits, as very similar studies targeting these have already been conducted (e.g., [103]), with [4] already implementing a refactoring tool for energy bugs.

us to compare current search-based improvement techniques to real-world commits and make suggestions for how these techniques can be improved.

To answer these research questions we have manually curated a corpus of 560 non-functional property improving commits, which were collected by analysing a total of 74,408 commits mined from 100 open-source Android repositories. In the following section we explain in detail our collection procedure. We have made this corpus publicly available to allow for replication and extension of our work (`https://github.com/SOLAR-group/NonFunctionalAndroidCommits`).

4.1.1 Overview of Methodology

Below we present the methodology used to create our corpus. It consists of three steps:

Keyword mining: In this step we collect a set of performance NFP-improving commits by filtering them first based on keywords and then manual analysis.

Classifier mining: In this step we expand this set by using a classifier trained on the commit messages gathered in the previous step.

Categorisation: In this step we attempt to manually group the commits into categories. These categories allow us to find common patterns used to improve the four non-functional properties of interest: runtime, memory consumption, bandwidth and frame rate.

4.1.2 Corpus

In the first step, we mined the twenty most popularly downloaded Android applications according to Fossdroid[3], and extracted a total of of 28,028 commits. As it would have been infeasible to manually inspect such a large set to identify NFP-improving commits, we have adopted a semi-automatic approach that examines every commit message based on keyword search (as detailed in Section 4.1.3). This lead us to a total of 3,132 commits, which were then manually analysed in order

[3] `https://fossdroid.com/`

to label them as performance NFP-improving commits or not. A final set of 229 NFP-improving commits was deemed to improve one of the four non-functional properties of interest. We note that in previous work [103] opted for two-word key-phrases rather than keywords to massively narrow down the number of commits to manually analyse. [21] only mined commits from the main modules of applications, missing any changes to back-end modules. We opted not to take these actions, and avoid missing possible useful software transformations by mining all commits with generic keywords.

In the second step, we leverage this curated set of NFP-improving commits, to train a classifier to be able to automatically identify such commits. This allowed us to automatically analyse a much larger set of commits (46,378), mined from 80 randomly selected F-droid repositories, and filter out irrelevant (i.e., not NFP-improving) commits with a precision of 95%, as detailed in Section 4.1.4. Specifically, we used the classifier to automatically identify 331 additional NFP-improving commits by randomly sampling F-droid. We initially found a total of 495 commits, which were then manually validated by two of the authors to make sure they improve any of the four non-functional properties of interest. This manual check led to the identification of 331 performance NFP-improving commits.

The final size of our manually curated corpus thus consists of 560 NFP-improving commits (229 from the first and 331 from the second step) . We then manually categorised these commits by the type of change which was made to improve the NFP, by analysing their commit messages and diffs. This resulted in 23 categories of improvement types being found.

Next, we detail how we mine NFP-improving commits by using keyword search (Section 4.1.3) and the classifier (Section 4.1.4), as well as how we manually validate the NFP-improving commits and categorise them (Section 4.1.5).

4.1.3 Step 1: Identifying NFP-improving Commits Based on Keyword Search

We mined 28,028 commits from the twenty most popularly downloaded applications according to Fossdroid (as of 18/03/2020), a website which offers an alterna-

Table 4.1: Properties of Repositories Mined Based on Keyword Search.

Repository	Type of App	Comm.	Stars	Age (Days)	Contrib.	Forks	KLoC
Aeons End	Game	26	5	963	1	5	3.0
AFH Downloader	Network.	69	18	1407	2	7	2.4
Android CUPS Print	Printing	274	142	1802	14	45	4.9
ANNO 1404	Game	13	1	1127	2	2	5.0
Apple Flinger	Game	463	22	972	37	-	11.8
Calculator	Calculator	1142	190	4220	18	286	32.8
Call Recorder	Audio	1590	97	1523	10	-	6.1
DNS66	Network.	341	1400	1304	15	153	7.8
Editor	Text Editor	405	110	1038	14	38	5.3
F-Droid	App Store	6157	1382	3492	99	-	85.1
Firefox	Browser	2592	1500	1249	82	585	309.2
FOSS Browser	Browser	927	427	1292	22	165	15.9
Frozen Bubble	Game	157	71	3829	4	64	38.4
G-Droid	App Store	625	78	557	60	-	17.2
Gadgetbridge	Network.	5163	74	1587	298	40	103.9
Gloomy Dungeons	Game 2	46	73	2006	4	38	91.2
MaterialOS	Themes	139	117	1834	7	37	14.5
Mi Mangu Nu	Books	1827	230	1839	23	60	33.3
Mighty Knight	Game	18	11	1250	2	12	1.0
NewPipe	Video Stream.	6054	8000	1711	439	1200	82.9

tive user interface to the standard F-Droid web page. These applications are diverse in nature (e.g., gaming applications, streaming applications, browsers) and size, having between 13 and 6,157 commits. Details of each application repository can be found in Table 4.1 and Table 4.5. Whilst the repositories of these applications are hosted on a variety of platforms (GitHub, GitLab, etc.), all repositories use the git version control system. The `git log` command was used to generate a list of commit messages, which was then parsed and searched for sets of relevant commits, that suggest improvements to the following four non-functional properties:

1. Execution Time: Decreasing the amount of time needed for computation.
2. Memory Consumption: Decreasing the amount of RAM used.
3. Bandwidth Usage: Reduction of the load on the network.
4. Frame Rate: Decreasing frame rendering and display rate.

In order to identify relevant performance NFP-improving commits, each repository was mined by searching every commit message for a series of keywords (or parts of words in some cases, e.g. “effic” to capture all words similar to “efficient”, “inefficient”, etc.) associated with the particular property, by following a

three-step process, as described below, and then manually validated.

Initial Selection. An initial set of keywords was generated by a combination of our knowledge of relevant terminology (which we have gained by writing NFP-improving commits ourselves) and the examination of the language used in commit messages written by others. We then augmented this set with 15 keywords[4] used in previous work conducting similar analysis ([102, 9, 21, 104, 105].) Any commit containing any of these keywords was selected for manual evaluation. Every selected commit message was manually evaluated to see if it actually suggests that an NFP has been improved or not. This approach aims to highlight as many commits as possible that could improve non-functional properties and therefore result in many false positives being manually evaluated. This helps to reduce the number of false negatives and allows us to detect as many relevant commits as possible.

Keyword Expansion. Synonyms for all keywords were searched for using the SEThesaurus ([106]), a natural language processing (NLP) tool for finding synonyms in an SE context. Terminology found during manual evaluation of commits which suggests improvement but was not present in the initial keyword set was added to a new keyword set. Another search took place with the new keywords in the same way as the original search. The keywords used can be found in Table 4.2.

Keyword Validation. To validate the keywords we conducted a text analysis by tokenising and lemmatising all words over all commit messages. The resulting 12,230 tokens were grouped according to the commits *relevant (229)*, *irrelevant (3132 - 229)*, and *filtered out (27028 - 3132)*, based on the keywords used. These tokens were then ranked by how often they occur in each group. From these rankings we attempted to identify possible keywords that we may have missed. First we removed all tokens that occur less than 10 times in the commits identified as improving performance NFPs: This resulted in the identification of 76 tokens, which could potentially be used as keywords. Then we further filtered out tokens by focusing only on those that occur in the *relevant* group more or as often than those occur-

[4]The keywords taken from previous work were: ‘wait’, ‘tim’, ‘stuck’, ‘react’, ‘latenc’, ‘throughput’, ‘suboptimal’, ‘bloat’, ‘utilization’, ‘ANR’, ‘OOM’, ‘bottleneck’, ‘hot-spot’, ‘length’, ‘consumption’

ring in the *irrelevant* group. This step allowed us to filter out words such as 'and', which are common in all commits. Of the 6 remaining tokens, three were already included as keywords (i.e., *memory*, *faster*, and *leak*). The remaining three were *save, reduce* and *low*. These three terms may be considered as additional keywords to identify additional NFP commits, yet their use could increase the already high manual effort needed to inspect the selected commits. In fact, in our study, these three keywords (save, reduce, low) relate to 111 *filtered out* commits. After manual inspection, we found that of these 111 commits only a single one could be identified as relevant; this commit also contains the word 'mem' instead of 'memory', suggesting that using keyword search may miss those commits that use abbreviations like this or contain misspellings of keywords. However, as most commits contain more than one keyword, the keyword set used herein can capture the majority of those commits too. As only three more relevant keywords were identified out of the 12,230 unique tokens present in the *relevant* commits, and they led to the identification of only one additional relevant commit out of 111, we are confident that the set of keywords used to conduct our study is comprehensive and effective.

Furthermore, the first author of this paper manually analysed the resultant commit set. Some commit messages were found ambiguous as to whether or not they offer any improvement. Developers sometimes write commit messages about what they have done but not why they have done it. Such commits were also independently analysed by another author. If the second author also found the commit to be ambiguous and not explicitly labeled as and improvement, it was discarded. We also discarded those commits which were merged with a single child commit as they were considered duplicates. We refer to the final set of manually curated commits gathered in this step as the "manual set".

4.1.4 Step 2: Identifying NFP-improving Commits Based on Automated Classification

While in the previous step, we use keyword search to narrow down the number of commits for manual investigation, in this step we explore the use of an automated classifier, which leverages on the manual set obtained from Step 1.

Table 4.2: Keywords Used to Search for Commit Types, from Initial Selection and **Keyword Expansion** stages. Note that extensions of keywords are also captured during search, e.g., speeding, performance, and other.

Property	Keywords
Execution Time	speed, time, perform, slow, fast, optimi, wait, tim, stuck, react, suboptimal, utilization, ANR, bottleneck, hot-spot, length, **effic**
Memory	memory, leak, size, cache, buffer, bloat, consumption, OOM **space**, **storage**
Bandwidth	network, bandwidth, size, download, upload, socket latenc, throughput,
Frame Rate	frame, lag, respons, latenc , **hang**

The classifier we propose has been trained with the classified data from Step 1, i.e., all commits manually excluded after the keyword search are labeled as *irrelevant*, while all commits included are labeled as *relevant*.[5] In addition, we have included 368 commits manually identified as relevant towards execution time in previous work ([9]) to the *relevant* commit set. We train the classifier using only the commit messages of the commit.[6]

In order to search for an accurate prediction model, we have investigated a total of 20 classification algorithms exploiting 6 different settings for feature selection. The settings were derived from the featurization of text tokens via TF/IDF, Bag of Words ([107]), and an adapted version of Bag of Words where only words occurring with a discriminative significance in either the irrelevant or relevant groups were used in the feature vector. Next, we present only the best result of these attempts, while more information about the training of the classifier can be found in Appendix A. The best classifier was achieved via stemming as a pre-processing step, TF/IDF for featurization using a Decision Tree classifier. We assessed its effectiveness via cross-validation by using 10 hold-out repetitions (80%/20% train/test split),

[5]We decided to group all relevant commits into one single group, as preliminary analysis showed that attempting to classify the commits into multiple classes (i.e., execution time, memory, bandwidth and frame rate) produces classes that are too small for building an accurate classification model (the Recall in all groups was less than 0.1).

[6]We considered also using issue messages to identity commits. However, the analysis of our KM data set showed that only 13% of commits had associated issues. Most (52%) of those issues were associated with 10 or more commits, meaning that only a small fraction of their messages and comments would be related to the commit that we are interested in.

Table 4.3: Decision tree classification of NFP-improving commits allows an accurate classification (0.80 recall) with a tolerable level of irrelevant commits mixed in (0.73 precision).

	Precision	Recall	F1-score
Relevant	0.73	0.80	0.76
Irrelevant	0.95	0.92	0.93

Table 4.4: Comparing our keyword search to our classification-based approach on two datasets. The *368* number of relevant commits for the Mazuera-Rozo et al. dataset was taken from their work `https://github.com/amazuerar/perf-bugs-mobile/blob/master/bug-fixing-commits-performance.csv`. We note that authors report 380 in their paper, but 11 commits don't exist anymore.

		Our Dataset	[9]
	Total Commits	28,028	420,352
Our Keyword Search	Identified	3,132	32,308
	Relevant	229	*368*
Classifier	Identified	669	3477
	Relevant	219	355
	Missed	10	13
	Additional	440	3109

each time using a different seed. The results show a good level of classification with a precision of 73% and recall of 80% in the *relevant* class (see Table 4.3) .

In order to show the reduction in manual effort required when using our classifier we run it on two datasets. Table 4.4 shows a comparison of commits identified via keyword search or via the classifier. For the dataset from Mazuera-Rozo et. al. [9] we applied the keywords from Table 4.2, after compiling the git logs from the repositories used in the dataset by Mazuera-Rozo et. al. [9]. The table shows that keyword search requires a much higher manual effort as the search returns several thousand keywords (3,132 in our dataset and 32,308 from Mazuera-Rozo et al.) containing only a few relevant commits (229 and 368). The classifier returns only 669 commits, with 219 from the manual identified ones contained (only 10 missed), and an additional 440 commits that may be relevant, but were filtered by the keyword search.

As the cross-validation confirms the effectiveness of the classifier, we re-train it on the entire available dataset in order to classify performance NFP-improving commits on unseen data, thus further validating our classifier in a real usage scenario, and extending our corpus of NFP-improving commits with the commits correctly classified as such.

To this end, we randomly selected 80 repositories from F-Droid and used the classifier to automatically classify all 46,378 commits extracted from these repositories .[7] Details of the repositories are provided in Table 4.5. The classifier identified 475 commits *relevant* commits. Two of the authors manually analysed these commits, as they did in Step 1, to check whether the commits classified as relevant are actually NFP-improving commits, i.e., true positives. They found that only 164 commits were false positives, giving a manually evaluated precision of 66.87 % for this classifier, and 331 commits added to our corpus. [8]

To further verify our classifier, we evaluated its performance on 5 randomly selected repositories from the set that was mined with the classifier. We perform keyword mining on these repositories in order to identify the false negatives of the classifier. Of the 5 repositories selected, 3 were found in both CM and KM to contain no performance NFP-improving commits. In the repositories where commits were found, one was found to have 5 performance NFP-improving commits compared to 3 found by the classifier, and in the other, the same 3 commits were found by both approaches. These repositories are all small yet representative of many of the repositories which were mined. In order to evaluate the classifier on a larger repository with many commits, we also ran it on the Koreader repository where the most CM commits were found (147 overall, see Table 4.7). We manually analyse all commits found using keyword search. In this project we found 2 additional relevant execution time commits, 2 additional memory commits, 1 additional frame-rate-improving commit and the same set of bandwidth-improving commits with the

[7] We had to set a limit on the number of repositories due the manual effort required to analyse the precision of the classifier.

[8] We note that a lot of these were small repositories, as the 40 repositories in which NFP-improving commits were found had altogether 39,420 commits, while the other 39 had altogether 6,958 commits.

keyword search. This means that the classifier only missed 5 relevant commits.

Our final manually curated corpus of performance NFP commits thus contains a total of 560 commits, which we use to answer our RQs.

Table 4.5: Properties of Classifier Mined Repositories.

Name	Commits	Stars	Age (Days)	Contrib.	Forks	KLoC
Alwayson	362	75	882	5	13	10.0
Android-inventory-agent	982	41	3584	14	25	12.2
Android-usb-serial-monitor-lite	50	140	3237	2	77	2.2
Anewjkuapp	1332	12	2423	14	6	27.4
Ankieditor	47	21	1226	2	6	41.6
Atmospherelogger	65	14	3354	1	4	3.0
Audioanchor	243	109	662	12	21	8.3
Audiometer	61	20	1493	3	7	1.3
Ausweisapp2	52	275	1311	9	42	137.6
Autoairplanemode	20	15	1398	3	8	1.9
Avare	1790	114	2940	28	116	55.2
Blexplorer	92	46	2080	4	23	1.7
Boogdroid	189	9	1819	5	3	4.0
Botbrew-gui	135	49	3141	1	15	5.10
Changedetection	219	584	938	11	72	14.9
Cmus-android-remote	45	11	2512	1	5	4.4
Controlloid-client	95	62	683	2	8	15.1
Covid19stats	81	139	273	4	39	2.0
Dailypill	251	5	401	2	2	1.10
Dandelion	651	102	1753	16	36	21.2
Droid48	96	59	3747	2	22	19.2
Easytoken	102	44	2364	1	13	4.2
Easywatermark	125	596	153	5	60	7.10
Gears2	63	17	3521	1	11	3.6
Gigaget	128	205	2219	2	52	5.5
Glesquake	14	15	2312	1	6	150.3

Glt-companion	482	9	2106	5	3	11276.9
Gpodroid	69	25	3523	2	3	3.2
Http-shortcuts	1144	354	2067	8	66	59.10
Headingcalculator	38	1.2	2329	1	1	1.2
Holokenmod	90	10	2083	3	2	5.2
Kerneladiutor	1347	21	1335	60	6	55.6
Koreader	7908	8104	2828	164	857	119.3
Languagepack	609	112	3147	31	189	1833.10
Lifecounter	2.2	17	2647	1	5	2.2
Lightning-browser	2125	1726	2879	72	744	23.10
Listmyaps	112	60	2670	2	21	2.3
Logmein-android	270	13	2394	7	11	1.10
Mlauncher	6	7	1968	1	2	0.2
Media-button-router	93	25	1949	1	6	1.3
Memento	82	142	1474	8	55	9.4
Memopad	61	7	3503	2	1	1.8
Openbikesharing	388	61	2327	28	52	4.10
Open-money-tracker	559	13	457	8	3	11.10
Openfoodfacts-androidapp	7218	576	2055	113	401	417.2
Openmw-android	862	192	1071	6	25	12.4
Permissionsmanager	54	4	1063	2	2	1.4
Pi-hole-droid	53	111	1400	4	15	147.6
Pixivformuzei3	870	90	603	11	11	5.0
Portauthority	1004	181	2200	14	53	4.10
Privacy-friendly-netmonitor	393	117	1510	14	30	8.1
Privacy-friendly-passwordgenerator	290	23	1497	4	12	8.3
Privacy-friendly-reckoning-skills	72	10	1326	5	2	4.9
Proexpense	276	40	188	3	12	16.2
Qbittorrent Client	930	211	2483	5	21481	29.0
Qrscan	96	25	1076	2	8	0.4
Rbb	2347	34	3150	1	4	27.6
search based launcher	257	40	2905	3	18	2.0

Smssync	1816	926	3608	21	468	42.7
Siteswap-generator	193	10	1174	2	4	8.2
Synctool	142	22	941	2	10	8.8
Tvhguide	364	44	3518	2	25	5.8
Taxiandroidopen	127	87	2573	1	115	8.0
Towercollector	561	97	1756	3	17	28.4
Trickytripper	301	43	3264	6	13	25.8
Ushahidi-android	949	205	4240	10	156	0.5
Vitosha-blackjack	16	7	1968	1	3	3.0
Voipms-sms-client	451	148	2283	4	49	12.9
Votar	65	14	2538	3	6	3.10
Weather	142	40	1753	7	13	22
Wulkanowy	1119	114	1351	23	18	100.2
Yashlang	161	23	473	1	1	26.10
Zeus	920	187	673	12	34	40.0

4.1.5 Step 3: Categorisation of Mined Performance NFP-improving Commits

To gain a greater understanding of the set of commits, we manually classified them by the type of change that was made. For each performance NFP (i.e., execution time, memory consumption, bandwidth, and frame rate) the set of extracted commits was examined and categories were generated, based on commit type. Commits were inserted into relevant categories or into new categories if they could not be classified inside current ones. Commits that could be classified into more than one category due to multiple changes were added to both categories. If two categories had a large shared membership or it became difficult to place a commit into either category, the categories were combined into a single category encompassing the traits of both.

Some commits were unclassifiable, e.g., due to improvements being buried in a large list of changes, or changes requiring domain-specific knowledge that is not explained in the commit message. If many commits were left without a category,

Table 4.6: Comparison between categories identified by keyword search vs. classifier. Percentages from total cumulate to >100% as some commits address multiple NFP.

	Keyword	Classifier	Discrepancy
Total	229 (100%)	331 (100%)	-
Execution Time	125 (54.5%)	211 (64.2%)	9.7%
Memory Usage	73 (31.9%)	115 (34.3%)	2.3%
Bandwidth	26 (11.4%)	5 (1.5%)	9.9%
Framerate	15 (6.6%)	15 (4.6%)	2%

the uncategorised commits were re-examined to determine if any categories had not been uncovered in the first class. Next another author examined the categorised commits to analyse whether or not they belonged to a given category. In case of disagreement the commit was placed in the unknown category, this occurred in 15 instances. The two authors also independently examined the issues associated with these commits in order to gather any extra information that could aid us in categorising them. In the case of non-classifiable commits we had to rely on the description of the optimisation written by the developer in the commit message as evidence that there was an improvement.[9]

Table 4.6 summarizes the categories of the two separate datasets of found commits via keyword and classifier search respectively. The results show only around two percent discrepancy in the memory usage and framerate categories. The Execution time is represented 9.7% stronger via the classifier, and the bandwidth is around 9.9% less via the classifier. The reason for this may be a slight bias towards run time performance, but may also be simply because bandwidth is not as relevant for many projects. 19 out of 26 bandwidth commits we identified via keyword search are only from two repositories (see Table 4.7), and thus may simply be overrepresented in the keyword dataset. The analysis implies that the classifier can produce datasets that reflect the real-world considerations of Android developers towards NFP.

[9] An example of a commit which could not be classified can be found at `https://github.com/ar-/apple-flinger/commit/bbc70cdc0a8190153195f46fe8c873def6ca3e98`

4.2 Results

In this section, we present the results of our mining for commits improving execution time, memory consumption, bandwidth usage, and frame rate. We report on these non-functional property improving (NFP) commits returned from our keyword search (KM) and those returned using our classifier (CM) separately. We investigate how developers improve the four NFPs, and categorise them to see whether source-code level changes could be implemented in automated software improvement tooling, targeting the mobile domain.

4.2.1 RQ1: Numbers of NFP-Improving Commits Found

Table 4.7 shows the number of commits which were found to improve each particular NFP in each of the repository mined, split between KM and CM commits. We also report on what percentages of total commits improve each of the four NFPs considered.

KM commits: We found NFP-improving commits in 12 out of 20 most popular Android repositories. 229 out of a total of 28,028 were deemed to improve 1 of our four NFPs considered. Execution time is the most commonly improved non-functional property in our KM set of commits (with 125 commits identified), appearing to be the most important non-functional property to Android developers. The next most common improvement was memory usage, with 73 commits. In three repositories, the number of commits improving memory consumption was actually greater than those improving execution time, showing its importance varies across projects. Bandwidth is improved less often than the previous two properties. It is to be expected, as some applications use little to no network data. A lot of network traffic also consists of large files, such as videos, pictures, or application APKs. Decreasing the network data used by these files is often not possible with source code changes. Frame rate is not improved very often. This could be due to developers being willing to tolerate frame rate in their applications, or that changes which improve frame rate are less well-known amongst developers. Also we found that the changes are larger than those for memory and execution time, so may require more effort to implement (see Table 4.10).

Table 4.7: RQ1: Number of NFP-improving Commits in Each Repository (% of Total Commits in Repository). Repositories with zero NFP-improving commits are not listed. The "Total NFP Commits" column does not count duplicates (as some commits could have improved multiple properties at once).

App Name	Execution Time Commits	Memory Commits	Bandwidth Commits	Frame Rate Commits	Total NFP Commits
Android CUPS Print	1 (0.36%)	1 (0.36%)	1 (0.36%)	0	3 (1.09%)
Apple Flinger	1 (0.22%)	0	0	0	1 (0.22%)
Calculator	11 (0.96%)	1 (0.09%)	0	1 (0.09%)	13 (1.14%)
Call Recorder	0	2 (0.13%)	0	1 (0.06%)	3 (0.19%)
DNS66	4 (1.17%)	8 (2.34%)	1 (0.29%)	0	10 (2.93%)
F-Droid	56 (0.89%)	15 (0.24%)	9 (0.15%)	6 (0.10%)	85 (1.38%)
Firefox	10 (0.35%)	4 (0.15%)	2 (0.08%)	2 (0.08%)	18(0.66%)
Frozen Bubble	1 (0.64%)	3 (1.91%)	1 (0.64%)	0	3(1.91%)
G-Droid	2 (0.32%)	0	2 (0.32%)	0	3(0.48%)
Gadgetbridge	9 (0.17%)	17 (0.33%)	0	2 (0.04%)	28 (0.62%)
Mi Mangu Nu	6 (0.33%)	6 (0.33%)	0	0	12 (0.66%)
NewPipe	24 (0.41%)	16 (0.26%)	10 (0.17%)	3 (0.05%)	50 (0.82%)
KM Total	125	73	26	15	229
Alwayson	9 (2.48%)	1 (0.27%)	0	1 (0.27%)	11 (3.03%)
Android-inventory-agent	1 (0.10%)	1 (0.10%)	0	0	2 (0.20%)
Android-usb-serial-monitor-lite	1 (2.0%)	0	0	0	1 (2.0%)
Anewjkuapp	5 (0.37%)	0	0	0	5 (0.37%)
Atmospherelogger	2 (3.07%)	0	0	0	2 (3.07%)
Audioanchor	1 (0.41%)	0	0	0	1 (0.41%)
Avare	15 (0.83%)	8 (0.55%)	0	2 (0.11%)	22 (1.34%)
Changedetection	2 (0.91%)	0	0	1 (0.45%)	3 (1.36%)
Cmus-android-remote	1 (2.22%)	0	0	1 (2.22%)	1 (2.22%)
Controlloid-client	3 (3.15%)	0	2 (2.10%)	0	3 (3.15%)
Easytoken	0	1 (0.98%)	0	0	1 (0.98%)
Easywatermark	1 (0.8%)	0	0	0	1 (0.8%)
Gigaget	2 (1.56%)	1 (0.78%)	0	0	3 (2.34%)
Glt-companion	6 (1.24%)	4 (0.82%)	0	0	10 (2.07%)
Http-shortcuts	2 (0.17%)	0	0	0	2 (0.17%)
Kerneladiutor	1 (0.07%)	5 (0.29%)	0	0	6 (0.37%)
Koreader	54 (0.67%)	23 (0.25%)	0	3 (0.03%)	71 (0.89%)
Lightning-browser	26 (1.22%)	21 (0.98%)	0	2 (0.09%)	45 (2.11%)
Listmyaps	1 (0.89%)	1 (0.89%)	0	1 (0.89%)	2 (1.78%)
Media-button-router	1 (1.07%)	0	0	0	1 (1.07%)
Open_money_tracker	1 (0.17%)	0	0	0	1 (0.17%)
Openbikesharing	0	1 (0.25%)	0	0	1 (0.25%)
Openfoodfacts-androidapp	6 (0.08%)	2 (0.02%)	1 (0.01%)	0	8 (0.11%)
Openmw-android	2 (0.23%)	1 (0.11%)	0	0	3 (0.34%)
Pixivformuzei3	10 (1.14%)	8 (0.91%)	0	1 (0.11%)	17 (1.95%)
Portauthority	27 (2.68%)	25 (2.39%)	2 (0.19%)	1 (0.09%)	50 (4.88%)
Privacy-friendly-reckoning-skills	1 (1.38%)	0	0	0	1 (1.38%)
Proexpense	0	2 (0.72%)	0	1 (0.36%)	3 (1.08%)
Qbittorrent-client	2 (0.21%)	0	0	0	2 (0.21%)
Rbb	4 (0.17%)	2 (0.08%)	0	0	6 (0.25%)
Search-based-launcher-v2	1 (0.38%)	0	0	0	1 (0.38%)
Siteswap_generator	1 (0.51%)	0	0	0	1 (0.51%)
Smssync	2 (0.11%)	2 (0.11%)	0	1 (0.05%)	3 (0.16%)
Towercollector	6 (1.06%)	2 (0.35%)	0	0	8 (1.42%)
Trickytripper	1 (0.33%)	0	0	0	1 (0.33%)
Tvhguide	3 (0.82%)	2 (0.54%)	0	0	5 (1.37%)
Ushahidi_android	4 (0.31%)	1 (0.10%)	0	0	5 (0.42%)
Voipms-sms-client	4 (0.88%)	0	0	0	4 (0.88%)
Weather	1 (0.70%)	0	0	0	1 (0.70%)
Wulkanowy	2 (0.17%)	2 (0.08%)	0	0	4 (0.26%)
CM Total	211	113	5	15	331
Total	346	188	31	30	560

CM commits: We found NFPs in 40 out of 80 randomly selected repositories. We find that our automatic classifier mostly selected performance improving commits (211) . The next most commonly captured improved property was memory consumption with 115 out of 331 CM commits. Bandwidth and frame rate improving commits were, as with KM, much less common. With 15 frame rate commits and only 5 bandwidth-improving commits found.

Answer to RQ1: Our study reveals that non-functional property improving commits are rare, with our single-keyword search not returning any NFP-improving commits for 8 out of 20 most popular Android repositories considered. In the remaining 12 only 229 NFP-improving commits were found, for the four properties of interest. Our CM repositories tell a similar story as we found that only 40 out of 80 repositories contain NFP-improving commits related to performance .

4.2.2 RQ2: How Android developers improve NFPs

To answer RQ2 we first analyse general characteristics of the commits found, i.e., age of commits, functionality changes made at the same time as NFP-improving changes, commit size, multiple NFPs improved at once, and optimisation trade-offs between NFPs . This should give us an overview of the current software development practice with respect to NFP improvement.

Age of Repository When Commits Are Made. To determine when developers make commits that improve performance NFPs , we look at the age of the repository when a commit was made in days, i.e., how long after the first commit was it made (see Table 4.8). In this table we show the median number of days between the first commit and the commits of each category, we also show the upper and lower quartiles to show the spread of the ages. We find that the age of the repository when these commits are made varies greatly between repositories. Not only that, but the spread of this figure also varies. For example, in the F-Droid repository (highlighted in the table), NFP-improving commits were spread out in age, even more so than other commits. Perhaps making NFP commits throughout the life cycle of the application rather than in small time windows is one of the reasons F-Droid was found to have the most NFP-improving commits.

Table 4.8: RQ2: Age in Days of Repositories When Commits Were Made (CM repositories are highlighted in italic).

Repository	Execution Time Commits			Memory Commits			Band. Commits			Frame. Commits			Other Commits		
	LQ.	Med.	UQ.	LQ.	Med.	UQ.	LQ.	Med.	UQ.	LQ.	Med.	UQ.	LQ.	Med.	UQ.
Android CUPS Printer	1210	1210	1210	1206	1206	1206	1264	1264	1264	-	-	-	410	518	708
Apple Flinger	18	18	18	-	-	-	-	-	-	-	-	-	26	379	411
Calculator	1789	2000	2396	2444	2444	2444	-	-	-	1445	1445	1445	1092	1568	2003
Call Recorder	-	-	-	594	844	1095	-	-	-	-	-	-	212	478	698
DNS66	99	132	142	131	134	166	3	3	3	-	-	-	10	166	250
F-Droid	**1179**	**2047**	**2472**	**1150**	**1875**	**2724**	**1637**	**1778**	**1992**	**1744**	**2158**	**2380**	**1430**	**1949**	**2431**
Firefox	172	276	350	59	103	212	394	465	536	117	117	117	187	363	606
Frozen Bubble	1642	1642	1642	1505	1642	1654	1642	1642	1642	-	-	-	1282	1638	2270
G-Droid	87	98	110	-	-	-	122	122	122	-	-	-	45	76	124
Gadgetbridge	688	1222	1448	427	688	723	-	-	-	376	525	674	479	868	1406
Mi Mangu Nu	444	620	732	480	616	824	-	-	-	-	-	-	371	674	1154
NewPipe	819	1277	1440	879	903	1271	1204	1374	1482	760	785	900	614	947	1360
Alwayson	209	719	739	763	763	763	-	-	-	716	716	716	410	668	727
Android-inventory-agent	92	92	92	2847	2847	2847	-	-	-	-	-	-	2388	2459	2788
Android-usb-serial-monitor-lite	28	28	28	-	-	-	-	-	-	-	-	-	16	26	72
Anewjkuapp	403	1177	1200	-	-	-	-	-	-	-	-	-	221	543	1604

Atmospherelogger	132	134	135	-	-	-	-	-	-	-	-	-	131	439	2669
Audioanchor	533	533	533	-	-	-	-	-	-	-	-	-	130	336	533
Avare	584	1003	1493	287	1135	1274	-	-	-	393	517	641	387	763	1319
Changedetection	35	47	59	-	-	-	-	-	-	130	130	130	23	40	126
Cmus-android-remote	8	8	8	-	-	-	-	-	-	8	8	8	3	5	9
Controlloid-client	54	54	82	-	-	-	54	54	54	-	-	-	13	50	115
Easytoken	-	-	-	34	34	34	-	-	-	-	-	-	7	11	13
Easywatermark	57	57	57	-	-	-	-	-	-	-	-	-	23	37	44
Gigaget	11	12	14	27	27	27	-	-	-	-	-	-	8	15	83
Glt-companion	462	928	1026	894	974	1060	-	-	-	-	-	-	357	979	1519
Http-shortcuts	251	272	294	-	-	-	-	-	-	-	-	-	653	1121	1816
Kerneladiutor	594	594	594	459	590	613	-	-	-	-	-	-	114	301	597
Koreader	727	1402	2314	933	1727	2289	-	-	-	1381	1580	2420	617	1115	2303
Lightning-browser	554	954	1704	1059	1104	1202	-	-	-	318	584	849	800	1310	1725
Listmyaps	248	248	248	248	248	248	-	-	-	2	2	2	24	31	186
Media-button-router	332	332	332	-	-	-	-	-	-	-	-	-	6	16	262
Open-money-tracker	601	601	601	-	-	-	-	-	-	-	-	-	530	602	1193
Openbikesharing	-	-	-	193	193	193	-	-	-	-	-	-	83	200	541
Openfoodfacts-androidapp	1218	1648	1813	919	1230	1540	1859	1859	1859	-	-	-	717	1003	1427

Openmw-android	544	793	1042	1827	1827	1827	-	-	-	-	-	-	321	1292	1717
Pixivformuzei3	162	274	364	85	89	92	-	-	-	393	393	393	119	273	383
Portauthority	226	601	794	647	739	791	1088	1146	1205	615	615	615	273	676	831
Privacy-friendly-reckoning-skills	277	277	277	-	-	-	-	-	-	-	-	-	78	110	277
Proexpense	-	-	-	174	175	176	-	-	-	167	167	167	20	43	150
Qbittorrent-client	227	246	264	-	-	-	-	-	-	-	-	-	394	615	960
Rbb	122	267	433	127	222	317	-	-	-	-	-	-	216	350	543
Search-based-launcher-v2	939	939	939	-	-	-	-	-	-	-	-	-	952	988	1014
Siteswap-generator	10	10	10	-	-	-	-	-	-	-	-	-	46	197	397
Smssync	1380	1578	1775	1293	1520	1746	-	-	-	1182	1182	1182	1135	1551	1837
Towercollector	14	34	167	528	886	1244	-	-	-	-	-	-	674	972	1447
Trickytripper	611	611	611	-	-	-	-	-	-	-	-	-	445	1088	1499
Tvhguide	37	52	173	227	386	545	-	-	-	-	-	-	23	69	210
Ushahidi-android	663	683	915	667	667	667	-	-	-	-	-	-	710	974	1417
Voipms-sms-client	724	726	960	-	-	-	-	-	-	-	-	-	694	1002	1667
Weather	209	209	209	-	-	-	-	-	-	-	-	-	107	166	1047
Wulkanowy	598	796	993	275	275	275	-	-	-	-	-	-	702	946	1174

Table 4.9: RQ2: Number of commits changing both functional and non-functional properties.

Type of Commit	No. of Commits	
	KM	CM
Exec. Time Commits	17 (14%)	24 (11%)
Memory Commits	10 (14%)	10 (9%)
Bandwidth Commits	8 (30%)	1 (20%)
Frame Rate Commits	2 (13%)	5 (33%)

Functionality Changes in Commits. Table 4.9 shows the number of commits of each type in which functional changes were also made. For the KM commits three of the types contained very similar numbers (14%), however bandwidth improving commits contained more (30%). This is mostly due to many of the bandwidth commits coming from repositories in which commits tended to be larger and contain many changes. The CM commits are similar, with the exception of frame rate, where 1/3 of commits also modified functionality.

Size of Commits. To determine the size of non-functional property improving commits we consider four measures: the number of files changed, the number of chunks changed, the number of classes modified, and the number of lines of code inserted/removed. Git splits the diff of each commit into chunks, where a chunk is the set of lines containing a change and the surrounding lines. If changes are close together they will be contained within the same chunks, so chunks can be used to measure the distribution of changes. These measures will provide a holistic picture of both the size of commits and the distribution of the changes that are made. Some of the commits contained multiple changes which will inflate their size, however so do many standard commits. We also take median values to attempt to mitigate this distortion and allow for a valid comparison. We compare the size of our identified performance NFP-improving commits to the size of every other commit found in the repositories. As commits from both categories will contain multiple changes, we do not attempt to distinguish between functional changes and non-functional changes in individual commits.

As shown in Table 4.10 we can see that KM commits improving performance

Table 4.10: RQ2: Median Commit Sizes. 'Other' category represents all commits that were not deemed to improve any of the four NFPs of interest.

Type of commit	Files changed		Chunks changed		Classes changed		Lines inserted		Lines removed	
	KM	CM	KM	CM	KM	CM	KM	CM	KM	CM
Exec. Time	2	2	5	5	4	2	16	9	12	6
Memory	1	2	4	4	3	2	16	7	6	5
Bandwidth	3	1	13	1	10	1	51	10	13	1
Frame Rate	1	3.0	9	16	7	4.5	47	77	4	4
Other	1	1	3	2	1	1	6	4	2	1

Table 4.11: RQ2: Commits Improving Multiple Properties.

	Execution Time		Memory		Bandwidth		Frame Rate	
	KM	CM	KM	CM	KM	CM	KM	CM
Exec. Time	**125**	**211**	6	15	5	5	0	5
Memory	6	15	**73**	**115**	2	0	0	3
Bandwidth	5	5	2	0	**26**	**5**	0	0
Frame Rate	0	5	0	3	0	0	**15**	**15**

NFPs tend to be larger than a generic commit, in every measure. They often span multiple files, multiple chunks and change multiple class definitions. They also add in more lines than they remove. For the CM commits, execution time and memory improving commits are of similar size to those in the KM set, spanning multiple files and make larger changes than other commits. The CM bandwidth commits are smaller than those in KM set, however still make larger changes than non NFP-improving commits. Finally, the CM frame rate improving commits are very large (median lines inserted of 47 for KM and 77 for CM). This could be due to the large rate of frame improving commits which also alter the functionality of their projects, as shown in Table 4.9.

Table 4.12: RQ2: Commits with Trade-Offs Between Properties.

Optimised \ Impaired	Exec. Time		Memory		Bandwidth		Frame Rate	
	KM	CM	KM	CM	KM	CM	KM	CM
Execution Time	**0**	**0**	20	25	0	0	0	0
Memory	6	2	**0**	**0**	0	0	0	0
Bandwidth	0	0	9	0	**0**	**0**	0	0
Frame Rate	0	0	0	0	0	0	**0**	**0**

Table 4.13: RQ3: Categories of Commits by Non-functional Property (% of commits improving a particular NFP).

Category	Subcategory	Execution Time		Memory		Bandwidth		Frame Rate	
		KM	CM	KM	CM	KM	CM	KM	CM
Add Condition	-	2 (1.6%)	13 (6.2%)	4 (5.5%)	2 (1.8%)	1 (3.8%)	2 (40.0%)	1 (6.7%)	2 (13.3%)
Add Delay	-	0	2 (<1%)	0	0	0	0	2 (13.3%)	0
Animation Length Reduction	-	1 (<1%)	0	0	0	0	0	1 (6.7%)	0
Caching	-	20 (16.0%)	25 (11.8%)	0	1 (<1%)	8 (30.8%)	0	0	2 (13.3%)
Change In Operation Order	-	2 (1.6%)	6 (2.8%)	0	2 (1.8%)	0	0	0	1 (6.7%)
Data Structure Replacement	-	8 (6.4%)	9 (4.3%)	0	2 (1.8%)	0	1 (20.0%)	0	1 (6.7%)
Data Structure Size Reduction	-	0	0	6 (8.2%)	4 (3.5%)	0	0	0	0
Decrease Asset Size	-	1 (<1%)	1 (<1%)	3 (4.1%)	5 (4.4%)	0	0	0	0
Different Algorithm	Use String Builder	0	4 (1.9%)	0	0	0	0	0	0
	Use Char instead of String	0	3 (1.4%)	0	0	0	0	0	0
	Improve Regex Performance	2 (1.6%)	4 (1.9%)	0	0	0	0	0	0
	Other	5 (4.0%)	12 (5.7%)	0	1 (<1%)	0	0	0	1 (6.7%)
Early Return	-	2 (1.6%)	2 (<1%)	0	0	0	0	0	0
Freeing Up Memory	-	0	0	10 (13.7%)	0	0	0	0	0
Increase in Concurrency	Move code to background	7 (5.6%)	9 (4.3%)	1 (1.4%)	1 (<1%)	0	0	5 (33.3%)	0
	Alter timing	3 (2.4%)	0	0	0	0	0	0	0
	Use a thread pool	0	2 (<1%)	0	0	0	0	0	0
Layout Optimisation	-	0	6 (2.8%)	0	0	0	0	0	0
Leak Fix	-	0	0	35 (47.9%)	74 (65.5%)	0	0	0	0
Make Final	-	2 (1.6%)	0	0	0	0	0	0	0
Make Static	-	0	0	0	1 (<1%)	0	0	0	0
Network Throttling	-	0	0	0	0	3 (11.5%)	0	0	0
Parameter Change	-	0	12 (5.7%)	0	0	0	1 (20.0%)	0	0
Remove Caching	-	1 (<1%)	0	1 (<1%)	0	0	0	0	0
Remove Redundancy	-	34 (27.2%)	56 (26.5%)	0	15 (13.3%)	10 (38.5%)	3 (60.0%)	2 (13.3%)	2 (13.3%)
SQL Query	Change Primary Key	0	2 (<1%)	0	0	0	0	0	0
	Specify column	0	2 (<1%)	0	0	0	0	0	0
	Combine Queries	0	3 (1.4%)	0	0	0	0	0	0
	Move file into Database	1 (<1%)	2 (<1%)	0	0	0	0	0	0
	Remove unneeded JOIN	1 (<1%)	0	0	0	0	0	0	0
	Flatten queries	3 (2.4%)	0	0	0	0	0	0	0
	Add table indices	3 (2.4%)	0	0	0	0	0	0	0
	Use transactions	1 (<1%)	1 (<1%)	0	0	0	0	0	0
	Parameter Binding	0	1 (<1%)	0	0	0	0	0	0
Time Out Reduction	-	1 (<1%)	7 (3.3%)	0	0	0	0	0	0
Use Different Library	-	1 (<1%)	8 (3.8%)	2 (2.7%)	3 (2.7%)	0	0	0	1 (6.7%)
Unknown	-	20 (16.0%)	35 (16.6%)	14 (19.2%)	4 (3.5%)	5 (19.2%)	0	5 (33.3%)	6 (40.0%)

Multiple Improvements. Table 4.11 shows how often commits improve multiple properties at the same time. The vast majority of NFP-improving commits improve one property at a time. This is true across both CM and KM commits. However, in 6.5% of cases (5.7% of KM and 7.0% of CM) developers are able to improve multiple properties at once. It is possible that some commits do improve multiple properties in ways that developers are not aware of or do not report, as this is not the primary purpose of the commit.

Tradeoffs. Some commit messages report that changes that improve one non-functional property negatively affect others. Table 4.12 shows that memory and execution time improvements are often traded off against each other. The most often impaired property is memory. This is due to the use of caching. Caching can be used to avoid having to repeatedly call the same code by storing the result. If the code not being called accesses the network, caching can reduce bandwidth usage. Sometimes caches can be too big so their size must be reduced. This can have a negative impact on execution time.

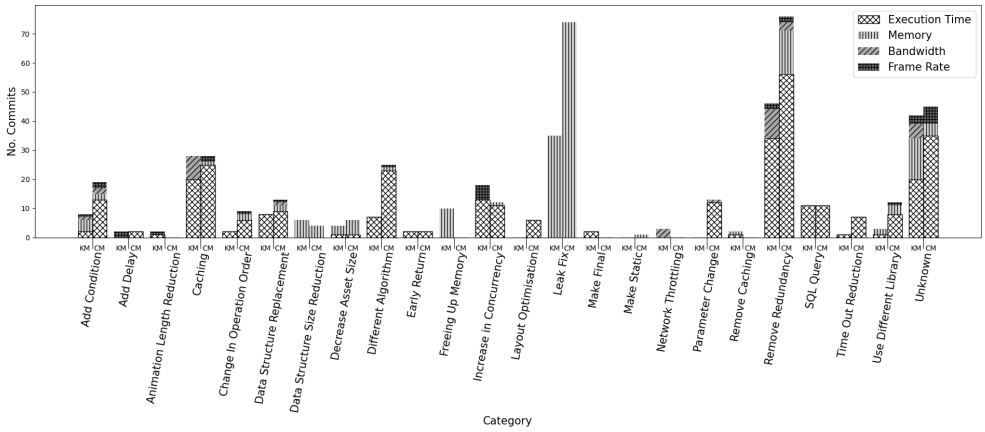


Figure 4.1: Histogram showing the distribution of commits amongst different categories.

Answer to RQ2: Our study reveals that developers commit changes aiming at improving the performance non-functional properties of mobile apps throughout the lifecycle of the development process. Roughly one in three commits contain also functionality changes. Performance NFP-improving commits tend to be larger than those that do not focus on improvement of NFPs . Most interestingly, these commits tend to spread across multiple files, meaning that automated approaches to improving NFPs should also run on code spread over multiple files. Execution time and memory can often be improved at once, although memory is much more frequently sacrificed to improve other NFPs. Approaches such as caching, which sacrifice memory for other properties, should be considered when improving performance NFPs.

4.2.3 RQ3: Types of NFP commits

Subsequently, we discuss the results of the manual categorisation of NFP-improving commits, as explained in Section 4.1.5. This should help us establish whether changes made by developers are already automated by existing tooling, and if not, whether new refactorings could be suggested for future work. In order to answer RQ3, our categorisation is presented in Table 4.13, we also show this data in Figure 4.1 as a histogram for easy comparison.

Add Condition: Conditions were added in 8 KM and 19 CM commits, this allowed

applications to avoid additional computation unless it was actually necessary.

Execution Time: 2 KM and 13 CM commits improved execution time by using caching. This is the most obvious application of caching as one can use it to avoid repeatedly performing the same computation unnecessarily.

Memory: 4 KM commits and 2 CM commits optimised memory usage by adding new conditions.

Bandwidth: 1 KM and 2 CM commits utilised caching to avoid making unnecessary network requests.

Frame Rate: 1 KM and 2 CM commits utilised new conditions to avoid unnecessary work on the UI thread, thus improving the frame rate.

Pattern: Wrap blocks of code in if statements, or add new conditions to existing if statements.

Add Delay: Delays were introduced in 2 KM and 2 CM commits, these delays allowed background execution to finish before proceeding, thus improving performance NFPs.

Execution Time: 2 CM commits improved execution time using increased delays.

Frame Rate: 2 KM commits utilised delays to improve frame rate.

Pattern: Insert calls to the Time.sleep() method into code.

Animation Length Reduction. Visual changes, such as changes to animations or UI elements, are used in 2 KM commits to reduce the frame rate of an application.

Frame Rate: 1 KM commit animation length reduction improved the application execution time.

Frame Rate: 1 KM commit used programmatic animation changes were used to improve the application frame rate.

Pattern: When animations are done programmatically we recommend using profilers to identify hotspots. Frames could also be removed from animations to speed them up and a trade-off between speed and smoothness could be found.

```
for (int i = 0; i < 100; i++){
    ...
    int x = foo();
    ...
}
```

↓

```
int y = foo()
for (int i = 0; i < 100; i++){
    ...
    int x = y;
    ...
}
```

Figure 4.2: An Example of the Caching Pattern.

Caching. Caching data to avoid rerunning code was one of the largest categories of change, with 28 KM and 28 CM changes being made.

Execution Time: 20 KM and 25 CM commits improved execution time using caching. This is the most obvious application of caching as we one use it to avoid repeatedly performing the same computation unnecessarily.

Memory: Whilst caching can often increase memory usage if used to avoid memory-intensive computation it can actually save memory. However, this is uncommon, with only 1 CM commit showing this use of caching.

Bandwidth: 8 KM commits utilised caching to avoid making unnecessary network requests.

Frame Rate: 1 CM commit utilised caching to avoid making unnecessary work on the UI thread, improving frame rate.

Pattern: Caching is a common pattern for decreasing the execution time of an application. It can often be easily implemented by assigning the results of a method call to a variable and replacing future calls to the method with that variable. An example of a caching pattern being applied can be found in Figure 4.2.

Change in Operation Order. 2 KM and 9 CM commits altered the order of operations. This involved swapping the order in which lines of code are executed,

```
if (foo()){
    ...
}
else if (bar()){
    ...
}
```

↓

```
if (bar()){
    ...
}
else if (foo()){
    ...
}
```

Figure 4.3: An Example of the Change in Operation Order Pattern.

such as conditions in if statements.

Execution Time: 2 KM and 6 CM commits improved the execution time of code by changing the order of operations.

Memory: 2 CM commits were found to reduce the memory consumed.

Frame Rate: Only 1 CM commit changed the order of operations in order to improve the application frame rate.

Pattern: Change in operation order can be achieved by swapping lines or blocks of code, or nodes in the abstract syntax tree. The swap operator has already been utilised in program repair ([75]), but is yet to be widely adopted in automated search-based techniques for improvement of NFPs of software ([63]). An example of this pattern can be seen in Figure 4.3.

Data Structure Replacement. Data structure replacements were used in 8 KM and 12 CM commits. A data structure replacement could consist of swapping an `ArrayList` for a `LinkedList` where the `LinkedList` is more efficient.

Execution Time: 8 KM and 9 CM commits replaced data structure to improve execution time.

Memory: 2 CM commits implemented the usage of more memory-efficient

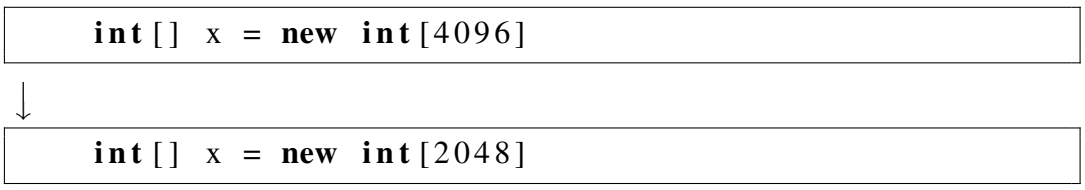


Figure 4.4: Data Structure Size Reduction Pattern

data structures.

Bandwidth: 1 CM commit replaced a data structure transmitted over the network with a smaller equivalent to improve bandwidth usage.

Frame Rate: 1 CM commit used a more efficient data structure to improve frame rate.

Pattern: Various NFPs can be improved by finding data structures and automatically replacing them with compatible ones. Similar approach has been implemented by Basios et al. [70], although their tool is not publicly available.

Data Structure Size Reduction. Reduction in the size of data structures, such as arrays, were used in 6 KM and 4 CM commits.

Memory: All KM and CM commits improved the memory consumption of applications by reducing the sizes of data structures.

Pattern: Reduce the size of data structures. This can be achieved, for instance, by changing the size of declared arrays. Program analysis could be required to prevent overflows. An example of this pattern can be seen in Figure 4.4.

Decrease Asset Size. Changes to assets such as images and fonts, mostly to improve the efficiency of loading said assets, account for 3 KM changes and 5 CM changes.

Execution Time: 1 KM and 1 CM commit improved the execution by reducing asset size and speeding up their loading.

Memory: 3 KM commits and 5 CM commits improved memory by reducing the amount of memory that large assets consume.

Pattern: Use compression algorithms, such as gzip, to reduce the size of assets.

Different Algorithm. The implementation of more efficient algorithms constituted 7 KM commits and 24 CM commits.

Execution Time: 7 KM and 23 CM commits implemented more time efficient versions of algorithms.

Memory: 1 CM commit replaced one algorithm with another more memory efficient one.

Frame Rate: 1 CM commit implemented a more efficient algorithm to improve frame rate.

Subcategories:

Use String Builder: In 4 CM commits, String Builders were used in place of naive string construction to improve execution time.

Use char instead of String: In 3 CM commits, method calls with Strings for arguments, such as indexOf, were replaced with equivalent methods with char arguments to improve execution time.

Improve Regex Performance: In 2 KM and 4 CM commits, regular expressions were modified to execute more quickly.

Other: 5 KM and 12 CM commits consisted of changes to algorithms that could not be grouped with others. We detail each of these changes in Appendix A.1.

Early Return. Earlier return statements were introduced in 2 KM and 2 CM commits, preventing whole methods from executing when unnecessary.

Execution Time: 2 KM and 2 CM commits used earlier returns to speed up application execution time.

Pattern: Insertion of a return statement. An example of this pattern can be found in Figure 4.5.

Freeing Up Memory. Adjustments to the amount of memory in use when devices were low on memory was another common change made to 10 KM commits.

Memory: All commits which used this strategy were used to reduce the memory consumption of applications.

Pattern: Program analysis would be required to identify which resources

```
...
if(foo()){
    ...
    int x = bar();
}
...
return x;
```

↓

```
...
if(foo()){
    ...
    int x = bar();
    return x;
}
...
return x;
```

Figure 4.5: An Example of the Early Return Pattern

could be freed from memory.

Increase in Concurrency Multi-threading changes were used in 15 of the KM and 12 of the CM changes.

Execution Time: 10 KM and 11 CM commits used more multi-threading to speed up applications.

Memory: 1 CM commit used more concurrency to become more memory efficient.

Frame Rate: 5 KM commits improved frame rate by using multi-threading to reduce the load on the UI thread.

Subcategories:

Move code to background: 6 KM and 4 CM improved execution time, 1 CM commit improved memory consumption, and 3 KM commits improved frame rate by executing code on separate threads.

Alter timing: 3 KM commits improved execution time by altering the timing of threads.

Use a thread pool: 2 CM commits found improvements to execution time by

introducing thread pools to manage thread execution more efficiently.

Layout Optimisation. 6 CM commits modified the layouts of applications. This consists of flattening layouts to reduce nesting.

Execution Time: All 6 CM commits were used to speed up applications.

Pattern: Automatic layout flattening is obtained by removing layout components and replacing them with their child components. This can be achieved with the lint tool[10].

Leak Fix Fixing memory leaks was the most common approach to decrease memory consumption, with 35 KM and 74 CM commits.

Memory: Leak fixes were made in 35 KM commits, which is almost half of KM memory improving changes. 74 CM commits fixed leaks.

Pattern: Automatically detecting objects that are being unnecessarily instantiated or not properly disposed of, using a tool like infer[11], and removing them with program repair techniques like GenProg ([108]).

Make Final: 2 KM commits introduced the final keyword to local variables, allowing more efficient code to be compiled.

Execution Time: both commits improved execution time.

Pattern: Add the static keyword to local variables.

Make Static: 1 CM commit added the static keyword to a method in order to reduce memory usage.

Memory: The only commit in this category improved memory usage.

Pattern: Add the static keyword to methods.

Network Throttling. Network throttling (which is an intentional slowing down of internet speed) was used in 3 KM commits to improve bandwidth usage. In fact, while this may not reduce the total amount of bandwidth used, it can still be useful to reduce the load on the network by speeding it up for other users.

Bandwidth: All 3 KM commits used Network throttling to improve bandwidth usage.

Pattern: Create a network monitor that reduces the networking of the appli-

[10] `https://developer.android.com/studio/write/lint`
[11] `https://fbinfer.com/`

cation when traffic is high, a tool like android-varanus[12] could be used to this end.

Parameter Change. 13 CM commits simply changed parameters in various function calls, speeding up the application.

Execution Time: 12 CM commits were used to decrease execution time.

Bandwidth: 1 CM commit was used to improve frame rate.

Pattern: Techniques such as deep parameter optimisation [62], have proven useful for finding optimal parameters in source code.

Remove Caching: 2 KM commit removed a cache to improve NFPs.

Execution Time: 1 commit in this category improved execution time by removing a costly caching operation.

Memory: 1 commit in this category improved memory consumption.

Pattern: Replace cached variables with method calls.

Remove Redundancy. The removal of redundant function calls or iterations is the largest category we identified, with 46 KM and 65 CM commits found. Removing unnecessary code can be an easy and simple way to optimise software.

Execution Time: 34 KM and 56 CM changes improved execution time by removing unnecessary execution of code.

Memory: 15 CM commits removed code instantiating objects, thus reducing memory consumption.

Bandwidth: 10 KM and 3 CM changes removed unnecessary network access, reducing bandwidth usage.

Frame Rate: 2 KM and 2 CM commits removed redundant code that was causing frame rate to be low.

Pattern: Remove lines or blocks of code, or nodes in the abstract syntax tree. This operation is standard in Genetic Improvement [63] tooling used for automated improvement of non-functional software's properties, such as execution time, energy consumption, but also for automated program repair.

[12] `https://github.com/Yelp/android-varanus`

SQL Query. A large number of changes (11 KM and 11 CM) to SQL requests appear to improve performance NFPs. These changes were only present in 7 projects. Some of the changes removed unnecessary JOIN statements, and others changed the order of JOIN statements.

Execution Time: All SQL query commits improved execution time.

Subcategories

Change Primary Key: 2 CM commits improved execution times by changing the primary keys used in SQL tables.

Specify Column: 2 CM commits moved from selecting all columns to selecting individual columns, speeding up query execution.

Combine Queries: 3 CM commits combined multiple queries together to save executing them each individually.

Move File to DB: 1 KM commit and 2 CM replaced file I/O with an in memory database to improve the execution time.

Remove Unneeded JOIN: 1 KM commit improved execution time by removing an unnecessary JOIN statement.

Flatten Queries: 3 KM commits flattened queries containing sub-queries into a single select query, improving execution time.

Add table indices: 3 KM commits added indices to tables to speed up SQL queries.

Use transactions: 1 KM commit and 1 CM commit wrapped a series of queries up into a single transaction, allowing them to be executed more quickly.

Parameter Binding: 1 CM commit introduced parameter binding, which allows similar queries to be made repeatedly with only their parameters changed.

Time Out Reduction. In total, 1 KM and 7 CM commits made changes to the length of timeouts, waiting for other computations to complete.

Execution Time: 1 KM commit and 7 CM commits reduced unnecessarily long timeouts to improve the execution time.

Pattern: Change timeout values in source code.

Use Different Library. 3 KM 11 CM commits replaced the libraries they used with more efficient alternatives.

Execution Time: 1 KM and 8 CM commits used different faster libraries.

Memory: 2 KM and 3 CM commit used a more memory efficient library.

Frame Rate: 1 CM commit used a more efficient library to improve frame rate.

Pattern: Use a set of similar libraries and automatically replace their existing usages in code.

Unknown. Some of the commits (41 KM and 46 CM) were not classifiable as the type of optimising change was not obvious from the commit message or the diff. In fact, some changes were bundled within large commits making optimisations hard to pinpoint, yet from the developer message it was clear that execution time, memory consumption, bandwidth, or frame rate had been improved.

Answer to RQ3: Our study provides 18 categories of non-functional property-improving code changes. For execution time simply removing redundancies leads to most improvements (27 % for KM and 27 % for CM commits). For memory optimisation leak fix is the predominant strategy (48% for KM and 66 % for CM commits). Redundancy removal is also the favoured strategy to improve bandwidth, followed by caching, while an increase in concurrency leads to a better frame rate.

4.3 Discussion

In this section, we discuss potential avenues for future research that stem from our study.

4.3.1 Recommendations for NFP Mining

Our work shows that a more fine-grained mining of NFP commits is feasible. The classifier that we presented has a good chance of finding commits with a recall of 0.8 (4 in 5 commits are found in our test set). The precision of 0.72 suggests

Table 4.14: Correlation between properties of repositories and the number of NFP-improving commits found in them.

Property	Correlation Coefficient (ρ)	$p-value$
Total commits	0.754	$< 2.2e-16$
No. Stars	0.678	$1.10e-13$
No. Contribs.	0.601	$2.34e-10$
Age of Repo.	0.169	0.107
No. Forks.	0.043	0.697
KLoc	0.033	0.756

that around 3 in 4 commits will be relevant. This means that the manual effort to identify relevant commits is much less than searching via keywords. This is confirmed by our mining via the classifier where we find 331 relevant commits by analyzing 495 commits. In comparison via keyword search we mined 229 relevant commits by manually checking 3,132 commits. I.e. analyzing 1.50 commits via classifier returns one relevant commit, vs. 13.68 commits for one relevant commit via keyword search.

This enables the mining of repositories for NFP commits on a larger scale than previously possible as the manual effort is reduced significantly. With larger datasets in the future, it may even be possible to expand the classification to identify categories and subcategories of NFP that a commit falls under.

4.3.1.1 Characteristics of Repositories Containing NFP-improving Commits

As we observed that the number of performance NFP-improving commits greatly varies among the mined repositories, we further analyse these repositories in an attempt to find the characteristics of those repositories that contain many commits. This can allow future NFP mining studies to target the most commit rich repositories.

Table 4.1 and Table 4.5 show the properties (i.e., the total number of commits made in the repository, the number of stars a repository has, the number of developers who have contributed, the age of the repository, the number of times the repository has been forked, and the number of lines of code in the repository) of

the repositories together with the number of performance NFP-improving commits found in each of them.

In order to quantify the relationship between each property and the number of NFP-improving commits found, we calculate the Pearson Correlation Coefficient ([109]). Pearson's correlation (ρ) measures the linear relationship between two pairs of observations. It ranges from +1 (indicating perfect correlation) to - 1 (indicating perfect inverse correlation), no correlation is indicated by 0. The results of this analysis are shown in Table 4.14.

We find that the total number of commits in a repository shows the strongest, statistically significant, positive correlation with the number of performance NFP-improving commits ($\rho = 0.754$, p-value $< 2.20e-16$) . We also find strong, statistically significant, positive correlations between the number of NFP-improving commits and both the number of stars ($\rho = 0.678$, p-value $= 1.10e-13$) and the number of contributors ($\rho = 0.601$, p-value $= 234e-10$) . We find no statistically significant correlation with the age of a repository ($\rho = 0.169$, p-value $= 0.107$) , the number of lines of code in a repository ($\rho = 0.043$, p-value $= 0.756$) , or the number of forks of the repository ($\rho = 0.033$, p-value $= 0.697$) .

We also attempt to see if there is a relationship between the categories of applications and the number of performance NFP-improving commits we found. We present in Figure 4.6 a box plot of the categories reported on F-Droid for each application against the number of performance NFP-improving commits we found in it. We can observe that there are no categories that tend to have significantly more commits than others, however the applications in the connectivity category have the highest median number of NFP-improving commits.

The above results suggest that it would be preferable to target large repositories that receive many stars by GitHub users and have a large number of contributors, when mining for performance NFPs commits.

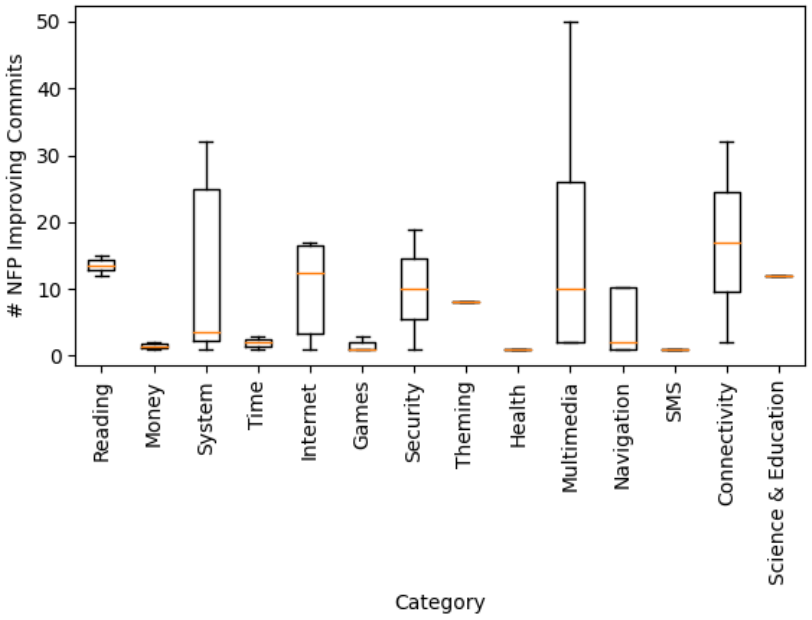


Figure 4.6: Box plot showing the relationship between repository category and number of performance NFP-improving commits.

4.3.2 Recommendations for Performance NFP-Improving Tooling

Our analysis of commits improving execution time, memory consumption, bandwidth usage, and frame rate of mobile applications provides several findings. In particular, our main goal has been to establish how Android developers improve these four NFPs, and whether we could utilise this knowledge to enhance automated software improvement tooling.

After careful analysis of the literature, we found that 5 out of 23 improvement strategies (see Section 4.2) are already mimicked by software mutation operators in automated software improvement tooling, as described below.

Deletion of redundancy has been used in many fields, ranging from slicing to newer strategies such as genetic improvement ([63]). The *delete* operator, which can be applied at various granularity levels, e.g., line or statement level has proved effective in automated program repair ([75]). Similarly, 75% of the non-functional properties studied were improved through the removal of a redundancy (used to im-

prove execution time, memory consumption, and frame rate). Android applications contain many lines of redundant code and its removal is one of the main techniques developers use to improve the four NFPs considered in this work.

Burles et. al. [71] and Basios et. al. [70] swap the data structures to improve non-functional properties. This is also reflected in our mined data. Both the *execution time* and *memory consumption* commit sets contained changes which consisted of simply swapping one data structure for a more efficient one. This suggests that patches produced with their techniques will be similar to those made by humans.

The *early return* category is small but offers a simple software transformation. Rather than assigning a value to a variable, just return the value. This will only be valid if the return value will not change in the rest of the method. This could be used as an opportunity to insert early returns. Recently, Brownlee et. al. [69] incorporated this type of operator in their automated software improvement tooling for Java software.

Switching the order of operations in which these are executed is also mimicked by the *swap* operator in genetic improvement research. Such swaps have been done at the statement ([96]) and expression level ([92, 110]).

The *parameter change* category is reminiscent of deep parameter optimisation ([62]), where parameters are exposed and automatically tuned to improve some property. Bokhari et. al. [61] successfully applied deep parameter optimisation to energy consumption in Android. Our results suggest that deep parameter optimisation could be effective for improving other properties in Android.

Although the changes mentioned above have been incorporated into automated software improvement tooling, these have generally not been applied in the Android ecosystem, but just to traditional software.

Our study has also revealed several improvement strategies that are not yet employed in current search-based software improvement tools.

One such strategy is caching. Caching has been repeatedly used to improve execution time, memory consumption, and bandwidth usage. Caching is useful as it reduces calls to areas of code by storing their output. The stored value is then

reused instead of calling the code. However, it does increase memory usage. The trade-off between the NFPs that caching improves must be balanced with the cost to memory, pointing to multi-objective optimisation. A caching pattern could be as simple as replacing multiple calls to a function, with the same arguments, with a single call to the function. The result of this call should be stored in a variable, and the original calls to the function replaced by the variable. Another option would be to cache a function called in a loop, replacing the function call in the loop with a variable storing the result of the function which is called before the loop, as in Figure 4.2.

Changes to assets, such as pictures, are found among the commits that improve execution time and memory consumption. Assets can be resized to improve memory consumption (and potentially bandwidth usage) or handled differently to improve execution time. When modifying assets, the changes made must result in the asset still being acceptable to the users. Measuring this acceptability and finding reasonable modifications poses an interesting challenge. However, a few genetic improvement approaches have tried to allow for decrease of output quality, when improving for other properties, such as energy consumption ([111]), or shader simplification ([83]). This, however, has not been regarded as a code changing operator itself, but as a side-effect of the other software transformations.

SQL queries were modified a substantial number of times in order to improve execution time. Using search-based techniques to transform SQL queries could allow developers to automatically achieve large improvements to execution time in applications that have many database interactions. Das et al. [21] found also that not only databases but also file systems led to *speed bugs*, suggesting that I/O could be a good target for improvement.

Many search-based techniques only focus on making changes within a single file. Table 4.10 suggests that a multi-file approach may be more useful for generating patches to improve non-functional properties. The number of chunks also suggests that patches are distributed widely across source code and changes are found in many locations. Thus, for Android applications, changes could be deliberately

spread out across multiple files.

4.4 Threats to Validity

Although we cannot claim that our results can apply to any type of mobile application, we mitigate the conclusion validity threat by mining commits from applications diverse in many ways, including their type, size, number of commits made, and number of contributors.

Moreover, developers of open-source applications may view non-functional properties at a different priority than their closed-source counterparts. However, the user base is the same (Android mobile users), and open-source repositories provide a rich amount of data that can be analysed to improve current software engineering practices. The number of NFP-improving commits we analysed may give a partial picture of the commits which are actually made by developers, and it was limited by the manual effort needed to analyse the 3,132 commits. However, this is in par with the number of commits examined in previous studies (e.g., [103, 21]). Some commits which improve NFPs may have been missed by the keyword search. This may have been due the sets of keywords used being incomplete or the commits not having messages which reflect the non-functional properties which they improve. To mitigate the former, the keywords were repeatedly expanded to try to catch as many commits as possible and had a very large false positive rate (73%), so it is unlikely that many were missed. It is also likely that developers did not report trade-offs that were made. The classifier may also be influenced by the keywords used, as it was trained on the gold set based on keywords search and manual analysis. We do provide the source code of both the keyword search and classifier (`https://github.com/SOLAR-group/NonFunctionalAndroidCommits`) so that they can be expanded in the future.

Finally, the NFP-improving commits could be categorised differently. The aim of the current categorisation was to provide meaningful classes from which insights could be drawn towards possible new mutation operators for non-functional property improvement using search-based approaches, such as genetic improve-

ment ([63]). To alleviate such a threat and facilitate this aim, we provide our detailed commit categorisation online (`https://github.com/SOLAR-group/NonFunctionalAndroidCommits`).

4.5 Related Work

There have been a few studies that mined for non-functional property improving commits in Android applications. Moura et al. [103] performed a study on software repositories mining "energy aware commits". They mined commits which attempted to improve the energy efficiency of an application and categorised the commits that did. They used a two-word-key-phrases approach, to narrow down the returned results to 2,189 commits, rather than 112,900 commits that were initially returned from single keyword search. The majority of the commits found in this study ($\sim$ 70%), belong to categories which concern device configuration, e.g. Voltage Scaling. Das et al. [21] mined Android repositories for "performance related commits". They found such messages in only 7% Android application repositories considered. Moreover, their categorisation was more top-level than what we propose, classifying commits into, e.g., "Memory", "File system", and other. They have also used one set of keywords for all performance-related issues. Their study restricted their searches to the main application modules. The patterns that they extracted mostly concerned what was being fixed, rather than how it was fixed, making direct comparison difficult. Although there are a few similarities, we both find changes to regular expressions to be useful for improving performance, we find that developers move computation to the background, and we find that caching is used to improve performance. In their work, Das et al. [21] find that developers improve multiple properties 10.4% of the time, whereas we only found that in 6.5% of cases We have also attempted to use as many keywords as possible, this includes all of the relevant keywords from the related work, and keywords which we determined from analysing commit messages, this is detailed in Section 4.1 .

Mazuera-Rozo et al. [9] performed a similar study, producing a topology of Android performance bugs, and pointed out several other studies that focused on

non-functional bugs. We are concerned with non-functional property *optimisation* rather than *repair*, and thus we describe the types of changes which can improve NFPs, whilst they describe the bugs which can be detrimental to them. In their study they uncovered a number of bug types, for which we have found related fixes. These include the leak fix category, the data structure replacement category, and the layout optimisation category. Interestingly, we find around 3-4x the number of commits dealing with caching for execution time improvement than they do. They also do not capture many of the categories for improving memory usage and bandwidth usage that we do, in particular the Remove Redundancy category. This may be due to our usage of a larger keyword set. As we are concerned with a detailed analysis of the improvement strategies found with respect to four specific performance non-functional property improvement criteria, we use a more comprehensive keyword search than the three aforementioned studies. Consequently, we manually analyse more commits. Moreover, we train a classifier to gather additional data, which we make open source. We also categorise and analyse commits in ways that allow the comparison to existing, and derivation of new, mutation operators for automated software engineering approaches for non-functional property improvement, such as genetic improvement [20, 63].

Linares-Vasquez et al. [45] attempted to uncover ways in which developers address performance bottlenecks in Android, they did this by surveying developers. They found that developers use similar techniques to the caching and SQL query categories uncovered in this study. Our approach aims to find a more complete picture of the changes developers make by directly exploring the content of the changes they made, as developers may exclude techniques used less frequently when answering a survey.

Chen et al. [105] perform a similar study to Linares-Vasquez et al. [9], but on 100 projects with C/C++ code. Changes to C/C++ are likely to be different to Java as there is more direct access to lower-level operations like memory allocation. However, their arguments pattern is similar to the change parameter pattern that we found and the memorization is similar to our caching pattern. Jin et al. [102]

conducted a study across a number of different programs in different languages, but did not include Android applications. However, they did find fix patterns similar to change parameter, change order of operations, and add condition, showing that some changes made in Android may applicable in other types of software. We also find that changes in Android are larger than those in C/C++. Only a median of lines of code are changed in C/C++ commits, but for all categories, we find median changes of 10+ lines and in some cases up to 70+ lines.

Our work uncovers a number of new patterns with respect to those discussed in related work. We find changes to animations, layouts, and assets that have not been discussed in previous work, these categories all seem to be particularly relevant UI-focused Android applications. We also find that searching for source code changes results in a number of patterns that have not been discussed in previous work, like remove redundancy, early return, different library, data structure changes and time out reduction.

The idea of mining repositories for software-improving commits has shown promise in the automated program repair field. Long et al. [112] mined Java repositories to automatically find fix patterns to apply to buggy software. Kim et al. [113] manually evaluated human-written patches from GitHub to generate patches which could then be automatically applied to buggy software. Bader et al. [114] leverage the version history of a piece of software to extract fix patterns to suggest to developers. Martinez et al. [115] showed that repair templates generated by mining existing code could be used to generate a large number of bug fixes. Several of the automated program repair techniques mentioned the use of search-based approaches, which have been utilised for improvement of non-functional properties, in the field of genetic improvement, in particular. Therefore, we believe that the results of our study could be similarly used to provide recommendations for future tooling [116].

4.6 Conclusions

In this chapter, we have explored the ways in which Android application developers improve non-functional properties, in order to guide the development of new automatic transformations to improve the NFPs of Android applications. To this end we analysed 74,408 commits from 100 repositories, finding 560 commits that improve four performance-related non-functional properties (NFPs): *execution time*, *memory consumption*, *bandwidth usage*, and *frame rate*. We employed a combination of keyword search, and a classifier to filter irrelevant commits, and manually analysed over 3,000 commits to obtain our corpus. We have found that developers more commonly improve the execution time of their applications than other NFPs. However, manual changes to improve non-functional properties are uncommon, suggesting that automated tools could aid developers in this challenging task. Moreover, we find that developers occasionally improve multiple non-functional properties at once (5.2% of cases), or sacrifice one property for another (10.7% of cases). This suggests developers are willing to take multi-objective approaches. However, the rarity of these changes suggests a need for automatic tools that could propose a Pareto of solutions for the developer to choose from ([117]). We have also found similarity between 5 current mutation operators in automated software improvement and the changes that app developers make. Code removal was a very common technique used by app developers for improving multiple non-functional properties, as was data structure replacement. However, we also identify a number of changes, which GI is currently not capable of making. It would be interesting to explore whether the corresponding mutation operators, which have already been successfully used to optimise NFPs of traditional software, are effective in automatically improving NFPs of mobile applications too. We have also found novel ways in which real commits could be more closely mimicked by tools for automated NFP improvement, for example, by making changes across multiple files simultaneously.

Our results highlight a need for automated tools which improve the non-functional properties of Android applications and provide initial guidance on what types of changes such tools could implement. These changes include automatic

caching to improve speed and bandwidth, SQL query transformation to improve speed, and image modification to reduce memory consumption and execution time. In later Chapters of this thesis, we explore how GI can integrate these types of changes.

Chapter 5

Improving Responsiveness with Local Genetic Improvement

Despite the negative results we found when attempting to improve the frame rate of Android apps, we believed that with the insights gathered in the previous chapter we could improve our framework and find better improvements to responsiveness. Firstly, we observe that there are a very limited number of ways in which framerate can be improved with patches to source code and thus it may not be the most effective target. However, we observe that there are many ways in which execution time can be improved, including transformations that are possible when using Genetic improvement. Thus, if we can find a type of execution time that can be used as a metric for responsiveness, we may be more likely to find improvements.

We observe that one simple source of poor responsiveness is slow navigation between activities. Most non-trivial applications consist of multiple activities, often including a main activity and a settings activity. When a user navigates between activities, specific methods are called. If these methods are sufficiently fast, the user will not notice a delay when moving between activities. But if they are slow, the user will notice a delay or unresponsive transition, as the application will hang whilst these methods execute. By simply measuring the execution times of these transition methods, we can easily quantify the responsiveness of activity transitions for any application. In the cases where these transitions are slow, reducing their execution time will improve the responsiveness of the application.

In order to make automated improvement of navigation responsiveness issues more practical, we explore GI for this purpose. We introduce an additional caching mutation operator, proven effective for execution time improvements, which we found in the work described in the previous chapter.

We use simulation-based testing in order to speed up the GI process as compared to Chapter 3. We evaluate our technique on 7 applications and analyse how well the improvements found with simulation-based testing translate to real applications.

The conclusions of this chapter are as follows:

1. We can achieve improvements of up to 30% to the navigation repsonsiveness of Android applications.
2. These improvements do not result in changes to application functionality.
3. Improvements found using local Robolectric tests translate to improvements on real devices.
4. Finally, the main limitation to our approach is the lack of testing in Android apps.

The rest of this chapter is divided as follows:

1. Section 5.1 describes our approach, using only local unit testing, to improve the navigation responsiveness of Android apps.
2. Section 5.2 lists the research questions we wish to answer about our approach.
3. Section 5.3 describes the methodology used to answer our questions.
4. Section 5.4 discusses and analyses our results.
5. Section 5.5 describes the threats to the validity of our work.
6. Section 5.6 details the conclusions of our experimentation and analysis.

5.1 Improving Android Navigation Response Time Using GI

In order to apply GI to Android lifecycle transitions, we propose the system presented in Figure 5.1.

The input to the framework is the application's source code (including tests)

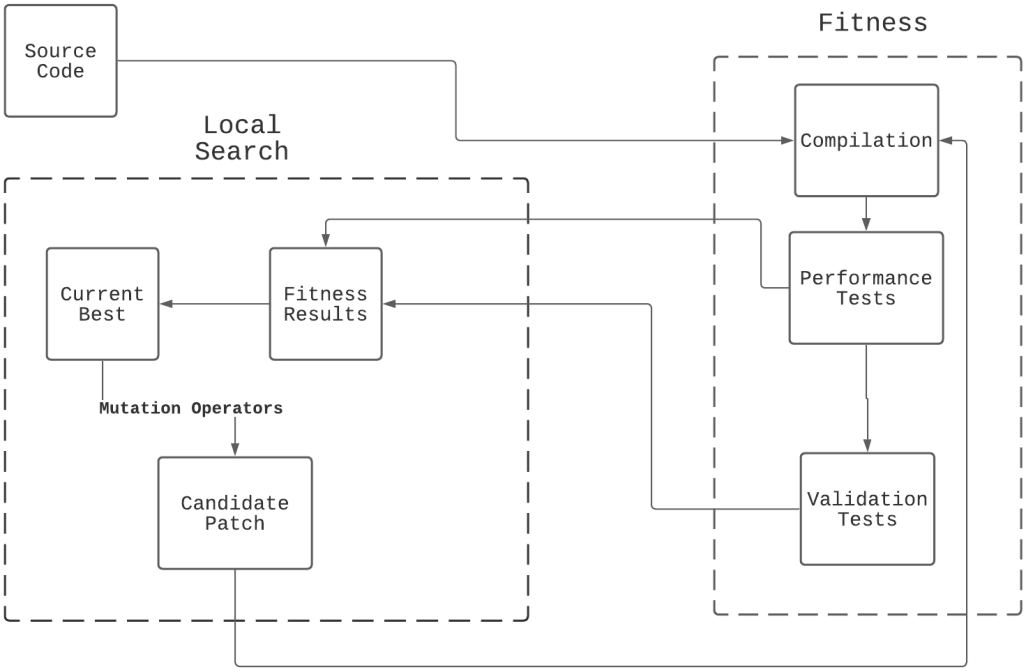


Figure 5.1: Local Android GI Framework, using the Local Search Meta-Heuristic

with a list of activities' callback methods to be optimised. The tests are split into 2 categories: performance and validation. After each software mutation the two sets of tests are run to check if the given mutant is more time-efficient, yet still maintains the same behaviour as original software. Although in this work we apply local search to navigate the space of mutations to callback methods, other search strategies can be used in the future. As shown in Figure 5.1, meta-heuristics can easily be substituted in place of local search. Our framework then outputs the best found software variant.

In the following subsections we describe our fitness evaluation procedure and mutation operators in more detail.

5.1.1 Mutation Operators

To generate mutants, we use 5 mutation operators on the abstract syntax tree (AST). 4 of these operators have been used previously in genetic improvement:

1. **Delete Statement** Remove a statement from the AST.
2. **Copy Statement** Add a copy of an existing statement in AST to a new loca-

tion

3. **Replace Statement** Insert one statement in the AST in place of another.
4. **Swap Statement** Swap two AST statement nodes.

We also introduce a new operator, the **Cache Operator**. This mutation finds *target methods* which are called at least twice within a *parent method* with the same parameters. A random subset of these *target methods* is then selected for caching. A single call to the *target method* is inserted above the first caching location and its result stored in a new variable. All of the previously selected instances of the *target method* are then replaced with the newly created variable.

5.1.2 Android Testing

The main challenge of applying GI to the Android domain comes from tests needing to exercise APIs which are exclusive to Android devices and emulators, such as GPS sensor information or UI components.

When testing in Android, tests can be split into two categories:

1. `Unit Tests`: In the context of Android, these are tests which can be run on desktop operating systems. They do not utilise any device specific functionality, such as GPS sensors.
2. `Connected Tests`: These tests can only be run on actual Android devices or emulator. They require a whole application to be compiled, packaged, and installed.

Normally, in order to test variants of Activity classes, we would need to use connected tests. This is because the APIs which are needed to render the UI components that we wish to test are only available on Android devices. When applying Genetic Improvement, each fitness evaluation (in which code compiles) would require the compilation, packaging, installation, and on device testing. This would be prohibitively expensive.

To avoid this cost, the Robolectric testing library [118] can be utilised. This library implements the Android specific APIs, whose use is normally restricted to connected tests. This library allows the testing of UI elements of applications on desktop devices using JUnit, removing the expensive steps usually needed for UI

testing. Crucially, Robolectric allows us to test the activity transitions of applications locally and significantly more quickly than we would otherwise be able to.

This allows us to avoid the significant overhead of running instrumented tests on actual devices or emulators.

To evaluate each software variant, we split relevant tests into two groups:

1. **Validation Tests** All tests which cover the lifecycle transitions which we wish to optimise. These are used to determine validity of the mutated software variant.
2. **Performance Tests** Tests which only exercise the lifecycle transitions which we wish to optimise. The execution time of these tests is used to determine the fitness of a mutant. The performance test set is a subset of the validation set.

We use the execution time of the performance tests on unmodified source code as a baseline to compare the improvements found.

5.2 Research Questions

In order to evaluate the utility of using Genetic Improvement to improve the navigation responsiveness of Android applications, we propose the following research questions:

RQ1 *How effectively can genetic improvement optimise the responsiveness of Android applications?*

This question will explore how well we can decrease the delays experienced by users when navigating through Android applications, using a fully automated search-based approach. We will also explore how well the approach generalises across different applications.

RQ2 *Which types of source code changes are most effective when improving navigation responsiveness in Android applications?*

The changes that we find to have the largest impact on navigation speed could be used in the future by developers who are trying to improve the responsiveness of their apps. They could also be useful in future automated techniques,

whether that be avoiding changes which are unlikely to be successful, or applying any fix-patterns we uncover to applications.

RQ3 *How expensive is it to improve the responsiveness of Android applications using genetic improvement?*

This question will allow us to quantify how feasible it is to run our setup and balance that with the improvements found. We can also find out what affects the cost of setup and how it varies between applications. With this we may also be able to propose ways to speed up GI, and discover the type of applications that GI will be most suitable for.

5.3 Methodology

To answer our RQs we modified an existing genetic improvement tool, Gin [96], to work with Android applications written in Java.

Next, we evaluate it on a systematically selected set of real-world applications, to check if indeed navigation responsiveness improving edits can be found.

5.3.1 Implementation

We chose Gin, as it targets method-level modifications. This is unlike other GI tooling, which largely focuses on class-level changes. Furthermore, Gin has been optimised for Java software, a common choice of programming language for Android developers. We use a simple local search strategy, default in Gin. We apply a single mutation to the current best patch (this is an empty patch at the start of search). If the execution time is shorter for the mutated code, than the current best patch, it is set as the current best. This continues for N individuals, in our case 200 as this was found to be a good trade-off between time spent searching and improvements found in our initial test experiments. Once all 200 patches have been evaluated, the current best is returned as the best patch found.

We have extended Gin to compile and test Android applications, as the existing code is only compatible with standard Java code.

We measure the execution time of performance tests using the linux `time` tool, this allows us to capture the CPU time of test execution, which is significantly less

noisy than just measuring the execution time. This is due to the other processes on the system running at the same time. Any time that is spent waiting whilst these other processes are running on the CPU would be added into an execution time measurement. Whereas, CPU time only captures the time when the process is actually running

In order to find the areas of code to improve, we first profiled the time taken by the lifecycle transitions using either pre-existing tests which started activities, or created our own test which did this. These tests were also used during search as our **Performance Tests**.

We then selected the activities with the greatest start-up time for improvement. We then found the areas code which were exercised by the activities start-up, within the activity class, to transform to find improvements.

In order to check the validity of the mutants produced during search, we run all of the tests which cover the areas of code selected for mutation. In many cases, there are numerous Robolectric tests first create an activity and then test some of an activities behaviour, giving us confidence that invalid mutants will be detected.

5.3.2 Benchmarks

In order to evaluate our approach, we run it on a number of real world applications. In order to run our setup on an application it must meet the following requirements:

1. Applications must be written, at least partly, in Java. Gin currently only supports the modification of Java code. However, our extension can run tests written in either Java or Kotlin.
2. The application must have passing tests written using the Robolectric library which exercise activity transition code.

With these requirements, we checked every application listed on the FDroid [119] online application repository. We found 97 applications which contained both activities written in Java, and utilised the Robolectric Library.

Next we manually analysed each application, the tests which used Robolectric were first checked to see if they actually exercised the code of interest or not. As Robolectric is a generic library which implements many different device spe-

cific APIs, often the test did not exercise the UI components we were interested in. Most commonly, these tests used the `Uri.parse` method as this is not available in standard Android unit testing and must otherwise be performed on an Android device/emulator.

When investing the applications which used Robolectric, we found that the transitions which entered activities was far more complex than the code which exited them. Most of the time the `onDestroy()` method was the only exiting method overridden and only consisted of a few lines of code. Because of this we chose to focus our setup on the methods used when entering or launching an activity.

Next, we checked the coverage of these tests using the Android Studio Coverage tools. In some cases, we needed to change the versions of dependencies in order to get the tests to run, however no source code was modified at this stage. We ran every Robolectric test in the application's test suite and investigated how well each activity was covered. Then for the activities that were covered we also collected the parts of code which were covered only when an activity was instantiated. This was done either using pre-existing tests or tests which we created. This allowed us to see exactly which areas of code were used when activities are launched.

We then selected the activities with over 100 lines of code in their transition callback methods. This gives a reasonable opportunity for improvements to be found. Finally, if multiple activities were found that were suitable, we chose the activity with the greatest launch time.

Our search yielded 7 suitable applications, these applications and the activities which we targeted for improvement are shown in Table 5.1. The main limitation is the lack of suitable tests, this is unsurprising as open-source Android applications tend to be very poorly tested [99]. If Robolectric tests could be automatically generated, our approach could be applied to almost any application. However, unlike with traditional software, where tools like EvoSuite [100] can be used, there exist no tools for automatically generating Android unit tests. Whilst random test input generation tools exist, such as Monkey [12] and Droidbot [120], they do not generate assertions. Therefore, they are not useful for checking the validity of a patch,

Table 5.1: Applications and targeted activities in our benchmark

Application	Target Activity
Amaze File Manager	MainActivity
Anki Android	CardTemplateEditor
Budget watch	TransactionViewActivity
Catima	MainActivity
Gift Card	GiftCardViewActivity
Loyalty Card	LoyaltyCardViewActivity
Rental Calculator	PropertyWorksheetActivity

other than for potentially finding crashes that are not present in the original app. These tools were also developed for, and sometimes only run on, Android 4.4, the latest release of Android is version 11, further limiting their utility. Therefore, we chose not to use random test input validation to validate our patches.

5.3.3 Validation

In order to validate the patches produced by GI, we undertake a number of checks. First the performance tests are rerun 10 times each for each of the best found patch in each run and the empty patch. This allows us to check whether the improvements found are genuine and not just noise. We also use statistical tests to check whether the improvements are significant or not.

Next we undertake manual analysis. This consists of both looking at the source code and running the app on an actual device. We check the patches to see if they do in fact appear to be improvements and navigate between the affected activities to look for a noticeable impact on the navigation responsiveness.

5.3.4 Experimental Setup

We run our set up 20 times on each of our 7 benchmarks, this is due to the stochastic nature of local search. With 20 runs we can see how our system performs in the average case, with different randomly selected mutations.

All computation was performed on a high performance cloud computer, with 16GB RAM.

Table 5.2: CPU times (CPUTs) of activity launch before and after GI.

Application	Orig. CPUT (s)	Med. improved CPUT (s)	Min Improved CPUT (s)
Amaze File Manager	1.15	1.07	0.98
AnkiDroid	1.55	1.17	1.09
Budget Watch	0.96	0.88	0.87
Catima	1.12	1.07	0.97
Gift card	0.88	0.84	0.83
Loyalty Card	0.90	0.82	0.78
Rental Calculator	0.90	0.87	0.85

Table 5.3: Percentage improvement to CPU time after GI.

Application	Med. improvement in CPU time	Max improvement in CPU time
Amaze File Manager	6.7%	14.5%
AnkiDroid	24.1%	29.6%
Budget Watch	8.6%	9.5%
Catima	4.4%	13.2%
Gift card	5.2%	6.4%
Loyalty Card	8.7%	13.1%
Rental Calculator	3.9%	6.0%

5.4 Results

Below we present the results of our experiments, answering our research questions

5.4.1 RQ1: Effectiveness of Genetic Improvement

In order to answer RQ1, we first measure the CPU time of the targeted activity transitions of both patched and unmodified applications. The CPU times of each variant of source code is measured 10 times and the median reading taken.

We also perform the Mann-Whitney U test [121], author = H. B. Mann on the data collected here with the null hypothesis:

> "Test running on the patched source code have the same CPU time as those running on unpatched code."

Those which did not show statistical significance at the 95% confidence level were set to 0% improvement.

The results of this can be seen in Table 5.2 for the absolute improvement in CPU time, and Table 5.3 for percentage improvements. It is worth noting that CPU represents only a small fraction of the actual execution time, as much of a processes execution time is spent waiting for it's code to be run on the CPU.

Table 5.4: Launch times (LTs) before and after the application of GI.

Application	Original LT (ms)	Med improved LT (ms)	Max. LT (ms)
Amaze File Manager	8330	7420	6368
AnkiDroid	4952	3808	3429
Budget Watch	1606	1541	1521
Catima	2223	2133	2119
Gift card	2059	2004	1977
Loyalty Card	6963	6721	6700
Rental Calculator	991	963	956

Table 5.5: Percentage Improvements to launch times after GI.

Application	Med. imp. in launch time	Max imp. in launch time
Amaze File Manager	10.9%	23.6%
AnkiDroid	23.1%	30.8%
Budget Watch	4.1%	5.3%
Catima	4.0%	4.7%
Gift card	2.6%	4.0%
Loyalty Card	3.5%	3.8%
Rental Calculator	2.8%	3.5%

These results show that GI is indeed capable of finding improvements to the CPU time taken by the code which instantiates activities. We find median improvements of between 4.4% and 24.1%, and maximum improvements of between 6.4% and 29.4%. We find the greatest improvement for the least responsive application.

To further validate our improvements, we installed them on a real Android device, a NOKIA 3.2 running Android version 10, and measured the time taken to open the activity which had been improved. In this case, we measure the launch time of an activity. An improvement in this figure will result in faster transitions between activities during usage. This means that an improvement to the activity's launch time will result in a more responsive app.

In order to perform these measurements, we specified UI tests, using the espresso testing library. Each test simply consisted of an ActivityRule, which launched the activity being improved. The application was then patched, compiled, packaged, installed and tested, with the execution time of the test execution being recorded. This does however have an overhead, the time spent launching the app, which will be included in measurements and may decrease the relative size of

improvements as this cost is unaffected by our patches.

Again we repeated 10 measurement for each patches version of the application and performed the Man-Whitney U test, at the 95% confidence level, with the null hypothesis:

> "Patched activities will have the same launch time as unpatched activities."

Again, all non significant improvements were set to zero. The results of these measurements can be seen in Table 5.4 and Table 5.5. These improvements were found to have medians of between 3.9% and 24.1% and maximums of between 6.0% and 29.6%. Thus, showing the time taken to launch activities has decreased shows an improvement in responsiveness.

We can see that the improvements found during GI do in fact translate into real improvements to execution on actual devices. For two applications, very large improvements were found of over 20%. These improvements knocked seconds off the launch times of activities, offering massive improvements to responsiveness. In every other case some improvements were found, although smaller. In these cases, the CPU time was relatively low to begin with, so finding improvements was much more difficult.

It is also worth noting that when visually inspecting the best patches, there is a clear difference observed when navigating into the improved activity.

Overall, GI is capable of improving the responsiveness of Android applications. GI is best suited in situations where applications are suffering from responsiveness issues, rather than searching for minor optimisations for already responsive apps.

Answer to RQ1: Genetic Improvement can find statistically significant improvements to the navigation responsiveness of Android apps of up to 30.8%.

5.4.2 RQ2: Most effective transformations

To find the most effective types of improvement, we analyse the patches which produced the best improvements for each project.

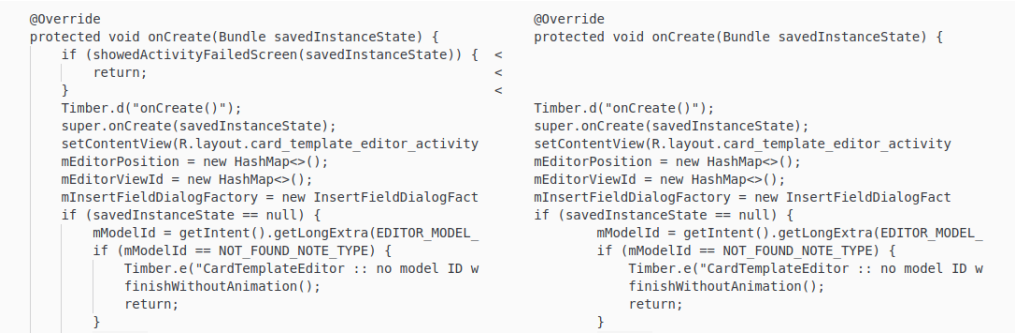


Figure 5.2: The most effective patch found. This patch which removes a mostly redundant, yet expensive check.

For the Amaze File Manager project, the most effective change consisted of the removal of redundant code. A number of components are created, but they are never used and have no effect on the application execution. Therefore removing them offers a good improvement in responsiveness. Visual inspection reveals no difference in functionality, however, the application renders significantly more quickly.

In the AnkiDroid app, and overall, our best improvement was very simple, it removed the call to a costly check in the case where an activity is created without an application, this patch is shown in Figure 5.2. This seems highly unlikely to occur, as activities should only be created by an application, therefore the high cost appears unjustified. There was no noticeable difference to this when using the patched application normally. However, the patch offers a choice between a huge optimisation, or better protection from a very unlikely edge case, and clearly the existing code is causing responsiveness issues. Interestingly, this patch was found in 3 separate GI runs.

For the Budget Watch application, the code which set the title of the page was removed, this involved querying a database multiple times in a large conditional to attempt to determine what the correct title should be. This does result in a missing title, however the title is not a key component of the app's functionality and would probably go unnoticed by most users, so the trade off may be worth it.

In the Catima project, a line that fetches an intent that is never utilized is removed in the best patch. This is a very simple patch, however produces a reasonable improvement.

For the Gift Card project, the best patch simply changed the order of operations when creating the activity. This changed the order of initialization between a button on the screen and the taskbar. When adding these two components to the activity, they are likely to interact with the already existing components, checking for conflicts. So perhaps, in this configuration, the process is quicker.

For the Loyalty Card application, the best change is relatively large, reordering a large section of code. In this patch, a BarcodeImageWriteTask is submitted before some of the other rendering jobs are competed, whereas before it was submitted last. This results in this task executing whilst the other UI components are being set up, thus speeding up the overall rendering.

The best Rental Calculator patch, again, changes the load order of the activities components, along with loading the text on the screen in a different order.

Overall, there appear to be 2 types of effective improvement. The removal of redundant code and changing the order in which activities load their components. Unfortunately, the newly introduced caching operator was not present in any of the produced patches. This is because the caching operator is only applicable in specific circumstances, i.e., when the same function is called repeatedly. In the code which we targeted, there were no caching opportunities found, so the operator could never be utilised. In the future, isolating and focusing on these 2 types of change may provide quicker search and better results. In some cases, the changes do slightly affect the way that the application looks or functions. However, this offers developers the option to have a more responsive, but slightly different application.

Answer to RQ2: We have found that the removal of redundant code and changing the order in which activities load their components are the two most effective types of changes for improving the navigation responsiveness of Android applications.

5.4.3 RQ3: Cost of Genetic Improvement

To quantify the cost of improvement, we measure the execution time of each run of our set up. We present the time taken in a box-plot in Figure 5.3, grouped by project.

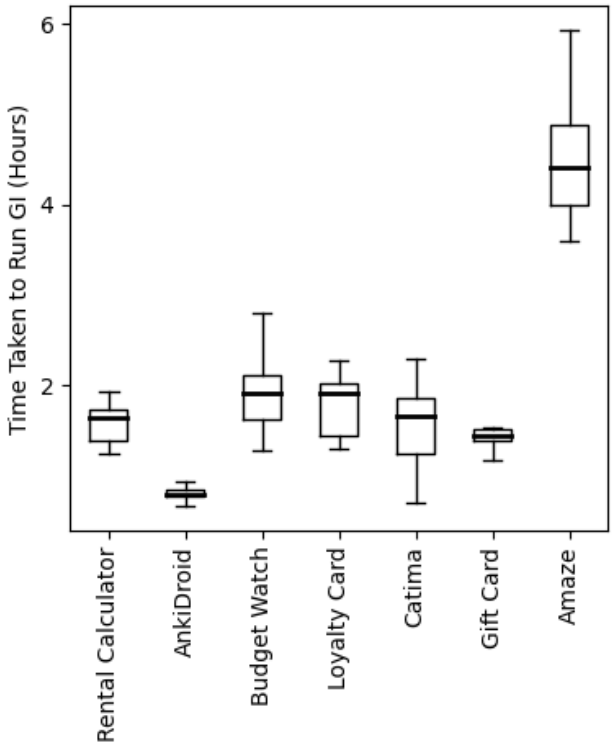


Figure 5.3: Boxplot of the times taken by each GI run for each project

Table 5.6: Comparison of test suit execution time, build time, and GI run time

Application	Test Suite Exec Time (s)	Build Time (s)	GI Exec. Time (h)
Amaze File Manage	87	32	4.04
AnkiDroid	10	3	0.802
Budget Watch	36	16	1.91
Catima	25	19	1.67
Gift Card	31	8	1.91
Loyalty Card	41	36	1.91
Rental Calculator	45	5	1.64

We see that the cost of GI varies by project and takes between 36 minutes and 6 hours, in the worst case, to complete. Whilst this can a relatively long time, the improvement is fully automated, requiring no human interaction. The median time across all runs and applications is only 1.75 hours. If there is a responsiveness problem, then running GI will be a cost-effective approach to solving this problem. If the code targeted is relatively static, GI is likely to be more suitable as it will only have to be run occasionally and the improvements will have a large impact over long periods of time.

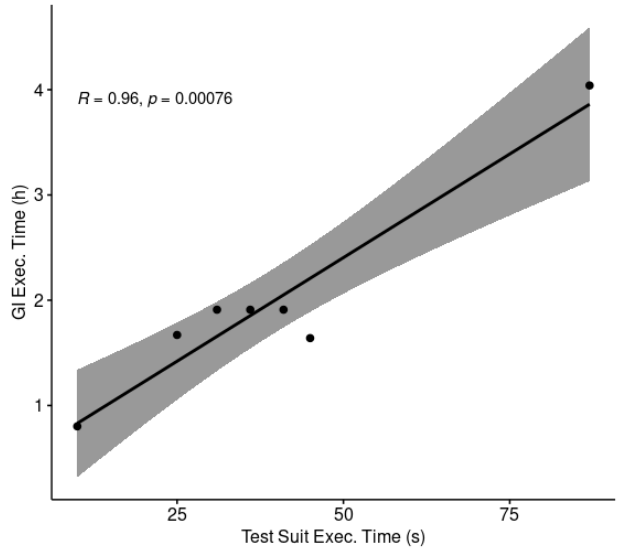


Figure 5.4: A scatter plot showing the correlation between test suite execution time and GI execution time

In order to understand what causes the execution time of GI to vary by project, we calculate the correlation between various attributes of the projects with the median execution time of GI for that project. In particular, we investigate what caused the Amaze file manager runs to be longer than the runs for other applications. We believe that the attribute with the main impact on the running time of GI will be the execution time of the test suite, as this seems to be where GI spends most of its time. We also compare the build time of each application to the overall run time. These are the main factors that seem like to impact the running time of GI. Due to the caching used by gradle during GI to speed up runs, we first clean the gradle cache and then run the same build/test task that is run during GI.

In order to calculate the correlation between these two properties, we calculate Pearson's correlation co-efficient [122] (R). R is a value between 1 and -1, representing positive and negative correlations respectively. The closer to zero R is, the less of a correlation there is. We also calculate p-values, determining the statistical significance of the correlation, and consider those with 95% confidence statistically significant.

For the test suite execution time, we calculate $R = 0.956, p - value = 0.0007$, suggesting a very strong statistically significant correlation between these two fac-

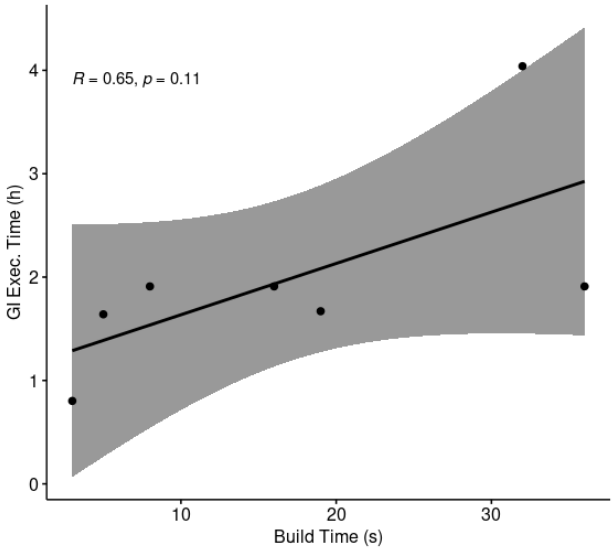


Figure 5.5: A scatter plot showing the correlation between build time and GI execution time

tors. A graph of this correlation can be seen in Figure 5.4, including Amaze File Manager with its very long test execution time. This is likely the cause of running GI on Amaze file manager being so costly. For build time, we calculate $R = 0.65, p - value = 0.112$, suggesting the correlation is not statistically significant, this relationship is shown in Figure 5.5.

When running our set up, minimising test execution time is likely to result in quicker runs. Guizzo et al. [123] showed that regression test selection can be used to speed up GI, whilst still producing valid patches. This technique would almost certainly be useful in this domain and could offer a significant speed-up.

Answer to RQ3: We find that our setup takes between 36 minutes and 6 hours to run. We find a strong correlation between the time taken by tests and the run time of GI but no statistically significant correlation between the build time of the apps and the time taken by GI.

5.5 Threats to Validity

In this section, we discuss potential threats to validity to our study.

Firstly, we use testing to validate patches. Although testing does not guarantee correctness, the GI approaches ultimately simply return a list of edits that can then undergo a standard code review process. This is standard practice in GI that operates

on source code. Moreover, during search, we exclusively utilise local unit testing, not testing patches on actual devices or emulators. We also utilise the Robolectric library to simulate Android UI rendering. Patches that are produced are validated using this library and any discrepancies between this library and the actual APIs may result in patches that are correct with respect to Robolectric, but patches that are not actually correct. In order to mitigate this threat, we take a number of additional steps to validate our patches. The first step was to manually investigate the diff of the source code before and after we patch application. We also then installed and ran the patched applications on a real device. We ran all existing tests and performed visual inspections to confirm the validity of the patches.

Another threat to validity comes from the fact that we only used 7 subjects in our empirical evaluation, thus results might not generalise. Nevertheless, we made our best effort to mitigate this threat. Since our approach relies on source code modification, we chose a popular open-source Android app repository, F-Droid. Although we found 97 apps to which our approach could potentially be applied, after manual analysis, only 7 of those had suitable test coverage. This is a general roadblock to the wider application of GI in the Android domain. Most, if not all, automated test generation tools currently only generate test input, rather than assertions, and thus more research is needed in this domain.

To mitigate this threat further, we make all our code and results freely available, allowing other researchers and developers to use and extend our tool for other software.

5.6 Conclusions and Future Work

Responsiveness is one of the most important properties of mobile software. Although several approaches exist for automated improvement of responsiveness issues, they either require network or hardware access. Moreover, arguably more lightweight approaches that target anti-patterns in source code, often do not come with available tooling and target specific performance bug types.

In this work we applied a GI-based approach to 7 diverse mobile applications,

showing improvements in time to navigate between activities to up to 30%. We carefully evaluated our results in order to verify the functionality of our tool.

Our results show that significant improvements to app responsiveness can be found with negligible changes to app functionality. This calls for the use of multi-objective approaches that would explore the space of responsiveness-improving solutions that might have minimal impact on other properties of the given app. There may also be properties which are being hindered by improvements which improve responsiveness, multi-objective approaches may be able to find trade-offs between responsiveness and other properties.

We also identify one challenge to the wider uptake of GI in the mobile domain, namely the availability of mobile applications with good test coverage of their UI elements. We also find that GI takes a relatively long time to find improvements, exploring a large search space. In the future, more intelligent mutation operators may allow us to find improvements more quickly.

We also release our framework, so that researchers could replicate our results, extend it with new mutation operators, while practitioners could already apply it to their Java-based Android applications.

Our tool is available at: `https://github.com/androidresponsiveness/AndroidResponsivenessGI/`

Given the success of our framework for this task, in the next chapter, we attempt to apply it to a more challenging problem. We adapt the framework to not only improve a single property but apply it to the improvement of multiple properties at once and explore the trade-offs that can be found between properties.

Chapter 6

Multi-Objective GI for Android

Given our success in the previous chapter, we decided to apply GI to a more complex problem. We note that when improving a single property, we can have negative impacts on others. For example, assigning more memory to n array when it is initialized may save time in the long run if many elements are inserted, saving the need for more allocations in the future. However, if only a few insertions are made and no reallocations are needed, the memory usage of the software will be unnecessarily high. Another example would be caching, if we introduce a caching mechanism into GI, we may make faster programs that use too much memory. In this case, if a user is running an optimised version of the application along with other applications, it may result in slowdowns whilst the OS attempts to free up memory. Whilst existing approaches for automated improvement of Android apps are capable of improving multiple properties simultaneously, e.g., by removing unnecessary computation and reducing runtime and energy use, in most cases such correlations have not been considered [20].

Extending GI to improve multiple properties can be accomplished by swapping out these single-objective algorithms with multi-objective ones. This allows us to consider patches that find trade-offs between various properties, rather than just those that improve one, without consideration of the impact on others. We can thus provide a choice to developers between different versions of source code, showing different trade-offs. Nevertheless, only a few works explore the potential of multi-objective GI and only in the desktop domain [124, 62].

The rest of this paper is structured as follows:

1. Section 6.1 describes how GI can be applied in a setting with multiple objectives.
2. Section 6.2 presents challenges of applying GI to the Android domain and our proposed framework that overcomes these challenges.
3. Section 6.3 presents research questions we aim to answer to evaluate our approach.
4. Section 6.4 outlines our methodology.
5. Section 6.5 presents our results showing how effective is MO-GI at improving three non-functional properties of Android apps.
6. Threats to validity are presented in Section 6.6, with Section 6.7 concluding.

6.1 Multi-Objective Optimization

Performance properties such as runtime and memory consumption often are at odds with each other, i.e., one can improve runtime by caching results, thus increasing memory use, and vice versa. In order to improve such conflicting properties, multi-objective (MO) algorithms have been proposed [125], which produce a Pareto front of non-dominated solutions. A solution x Pareto dominates another y if all of x's objectives are as good as y's and at least one objective is better than y's.

Past work utilising MO algorithms for GI is sparse, with the majority of work focusing on single-objective improvement. However, in the work where MO improvement has been successful Genetic Algorithm (GA) based algorithms have been used. Wu et. al. [62] and Callan et. al. [126] used NSGA-II [127], White et. al. [91] used SPEA2 [128], with Mesecan et. al. [124] comparing four MO algorithms, with SPEA2 and NSGA-III [129] performing best.

In each algorithm, a population of solutions (in our case program variants) is generated and their fitnesses are measured. In order to generate new patches, mutation, and crossover operators are used to generate child populations and then individuals are selected for the next generation from both child and parent populations.

The algorithms vary in their selection phases. The algorithms use Pareto dominance to compare different individuals who may find trade-offs between different properties.

Both NSGA-II and NSGA-III sort the population into Pareto fronts based on their fitnesses. The population of the next generation is then selected from the top fronts, one at a time, until a set number of individuals are chosen. If a front needs to be split, as it is too big for the population size, it is sorted by a crowding metric, and the least crowded members are selected. In NSGA-II, crowding is based on distance from other individuals in the fitness landscape. Whereas in NSGA-III, crowding is based on reference lines and the number of individuals that are closest to them, or niched to them. NSGA-III selects individuals spread across as many niches as possible in the final front to maintain diversity.

Unlike the NSGA algorithms, SPEA2 does not separate the population into Pareto fronts. Instead, the strength of each individual is calculated. This is equal to the number of other individuals that the individual Pareto dominates. The raw fitness of an individual is then calculated as the sum of the strengths of all other individuals which it dominates. Like the NSGA algorithms crowding metric is calculated. For this, all other individuals are sorted into a list based on proximity to the individual of interest. The metric is inversely proportional to the distance of the kth individual in the list. The parameter k is equal to the square root of the total population size. Finally, the raw fitness and the crowding metric are simply added together and used to select individuals.

It is yet unclear which multi-objective approach works best for the purpose of genetic improvement, thus we explore the capabilities of these three algorithms shown successfully in previous work.

6.2 Multi-Objective GI for Android

There are a number of practical changes when using genetic improvement to enhance the performance of Android apps when compared to traditional desktop environments.

Android applications make use of APIs for features like UI elements which are only present on actual devices. The Android library available when testing applications on desktop operating systems overwrites the APIs such that they throw errors when invoked. Most Android code utilises the Context class [130], in the applications we use in our experiments, the context class is explicitly imported in over 1/3 of files. This does not include the instances where it is implicitly imported as a nested dependency. This class gives the code access to the shared state of the application but is only available on devices. This means that in order to run tests that exercise any component of an application's code that accesses this state, the entire application must be compiled, packaged, transpiled, installed, and launched before it can be tested. This can take a considerable amount of time, often longer than the tests themselves [131].

Android apps are generally built using Gradle with the Android Gradle plugin. This makes them incompatible with much of the tooling surrounding automatic compilation and testing of code [11, 132].

Another challenge of applying GI to the mobile domain is the accurate measurement of the fitness function. Previous work has only applied GI to problems that take seconds/minutes to run. In the mobile domain, it was shown that apps that run in more than 150ms are considered to be 'laggy' by users [133]. Therefore, although previous work used approximate fitness measurements, these are not appropriate in the mobile domain as they may not capture such minor, yet important, differences in non-functional behaviour.

In order to overcome the aforementioned challenges, we propose the GI framework shown in Figure 6.1. The framework is split into two main components: the Search, and the Fitness sections. These components can be swapped out depending on the properties being improved.

6.2.1 Representation

We use a program representation consisting of a list of edits, which are applied sequentially to the source code. This representation has been used in GI many times in the past and has proven successful [63]. We use a list of edits, rather than

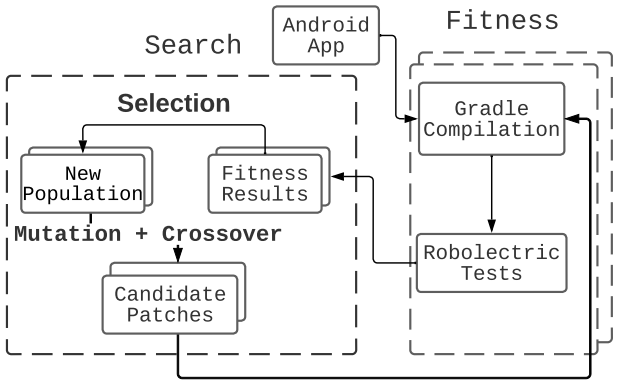


Figure 6.1: GI framework for Android app improvement, with search based on a genetic algorithm. In the case of local search, only mutation is applied.

```
| gin.edit.statement.DeleteStatement Example.java:608
| gin.edit.statement.CopyStatement Example.java:1307 ->
Example.java.java:320:365 |
```

Figure 6.2: An example of a program variant that deletes the statement with ID 608 and then copies the statement with ID 1307 to position 265 into the block with ID 365 in the file Example.java

representing the whole program in the genome, as may be done in traditional genetic programming, to reduce the memory footprint of the search process. An example of this representation, as used in GIDroid, is shown in Figure 6.2.

6.2.2 Fitness

In the Fitness section in our framework (see Figure 6.1), we measure the properties that we are improving. As in previous GI work, we patch the application, compile it and run unit tests on it. If all unit tests pass, the patch is considered valid, if not, it is discarded. Then, the property being improved is measured. For example, if we are improving execution time, the time taken by the test suite is measured. Multiple different properties are measured in the case of MO improvement.

Due to the complexity of the Android build system and significant use of UI elements, a minor change usually requires a time-costly process of installation on the actual device for testing. Our framework thus utilises only the local tests which run on the JVM. This would normally limit the components that could be tested to only those which do not use the device-only APIs. If we attempt to use these

APIs in a local test, we will simply call stubbed versions of the methods that throw exceptions. However, by using the simulation-based Robolectric testing library, we are able to test any application component with fast local tests. Robolectric has two main features that allow us to test apps. Firstly, the simulation of the application and Android environment, which creates a headless version of the application within a local JVM. Secondly, shadowing which allows the bytecode of classes to be overwritten at runtime. This is used to overwrite the API calls with simulated API calls and allows the simulated app to be exercised. Shadowing is useful for mocking hard dependencies and can be used to avoid the complex setup needed when testing certain components. Using this simulation-based approach, we can quickly compile and test application variants, and use measurement tools that aren't available in the Android operating system. Callan et. al. [134] found that improvements that could be demonstrated with unit tests written in the Robolectric library translated to improvements on Android applications run on real devices, in every case where improvements were found. Thus, with a combination of Robolectric testing and manual review of improvements, we can be confident whether we have found an actual improvement or not. We use the Gradle build system with the Android plugin to compile and test applications.

Khalid et. al [5] identified execution time, memory, bandwidth, and energy usage as the most complained about and impactful non-functional properties of Android apps. In this work, we will attempt to improve execution time, memory, and bandwidth. Previous work on automatically improving energy usage of Android apps [61, 60] found energy estimates to be too noisy, thus requiring external devices for physical energy measurements. Although GI can be used to optimize energy consumption [61], we want to provide a general, easy-to-use tool that does not require extra hardware. It is worth mentioning that thus far the primary technique for improving bandwidth has been prefetching [37]. No attempts have been made to improve it using source-code transformations, despite such changes being made by developers [135]. We are the first to try to do so.

6.2.3 Search

The Search section of our framework for Android app improvement (see Figure 6.1) determines how the search space of patches is navigated. Most GI work so far has used single-objective algorithms, such as genetic programming and local search. Only a few consider more than one objective. We pose that consideration of multiple objectives in the mobile domain is especially important, due to limited resources. To fill this gap, we propose to utilise multi-objective approaches in the search process. Multi-objective algorithms will allow us to evolve patches that will balance different trade-offs, producing Pareto fronts of solutions. The user will then be left with a choice of which patch fulfills their particular needs. The multi-objective approach will provide relevant information on how runtime reductions might for impact memory use etc.

To start our search we need to generate an initial set of patches. Our patch representation is not of fixed size and may contain any number of edits. We create an initial population containing individuals consisting of single random edits. Further creation is guided by a given search algorithm, where mutations and crossover are applied to create new patches.

6.2.3.1 Mutation and Crossover

Patches are created via mutation and crossover on the list of edits representation. In the single-objective search used in GI so far crossover typically appends the lists of edits together from patches selected using binary tournament selection. We apply this type of crossover in our MO algorithms as well. A mutation simply adds or deletes an edit. In our case we operate on the statement-level, thus each mutation can delete, remove, or replace another statement. Additionally, we investigated which other mutation operators might be beneficial in the Android domain.

Callan et. al's work [135] showed that one of the most common techniques for improving non-functional properties of Android apps is caching. Caching was found to be effective across all properties studied and improved a number of different applications in different domains. Outside of the changes already implemented by standard GI mutation operators (remove code, change order of opera-

tions), caching is the most generically applicable strategy found, and thus, the most suitable for multi-objective improvement. Based on a manual analysis of the commits from Callan et al.'s work, in which caching is used, we propose using two caching mutation operators. Caching could prove useful for the three properties that we wish to improve. Firstly, if we no longer need to execute a method as we already have the result we will save time. If the method has a larger memory footprint than the stored result, we will reduce the memory footprint of the app. Finally, if the cached method accesses the network, we will be able to avoid this operation and reduce network usage. However, caching may negatively impact memory usage if the stored result is large. This will mean that we will have to consider possible tensions between objectives when we run our search.

First, we utilise the simple **In-Method Caching Operator** from the previous Chapter. This operator simply stores the result of calling a method in a local variable and replaces future calls to this method with the local variable (see Algorithm 1). An example of this operator can be seen in Figure 6.3. The second caching operator creates new fields in the associated class for storing cached method calls. This **Class Caching Operator** allows cached variables to persist beyond the end of individual method calls and could prove particularly useful if a method is called repeatedly. An example of this operator is shown in Figure 6.4. We wrap the statement which accesses the cached variable with a null guard so that the first time it is called we actually call the method. For both of these operators, we consider method call expressions to be cachable to the same variable only if their arguments consist of the same variables. As shown in Algorithm 2, the class caching operator can be applied to any method call expression. However, as local variables do not persist after a method is executed, there must be at least two instances of the expression for it to be cached. These operators will not disrupt the source code syntax as they simply replace a method call expression with a variable name expression which is the same type as the method's return type.

Algorithm 1 Find method calls to cache in Method M

```
1: function METHODCACHEFINDER(C)
2:     seen = ∅
3:     cachable = ∅
4:     for each expression e in M do
5:         if e is a method call expression then
6:             if e ∈ seen then
7:                 cachable = cachable ∪ e
8:             else
9:                 seen = seen ∪ e
10:            end if
11:        end if
12:    end for
13:    Return cachable
14: end function
```

Original Code	Mutated Code
`int x = foo(a,b,c);`	**`int cachedVar1 = foo(a,b,c);`**
`int y = foo(a,b,c);`	`int x =` **`cachedVar1`**`;`
	`int y =` **`cachedVar1`**`;`

Figure 6.3: An example of the In-Method Cache Operator. The resultant code stores the results of a method call foo, with parameters a, b and c. This stored result can then be used later in the same method.

6.3 Research Questions

To evaluate how effective the multi-objective GI approach for improvement of Android apps' runtime, memory, and bandwidth use is, we pose the following research questions:

RQ1: Can MO-GI optimize Android apps in the same way as real developers?

In order to validate our approach, we want to see if MO-GI can reproduce real-world improvements that Android developers have manually implemented in the past.

RQ2: How effective is MO-GI at optimising Android apps without known improvements?

Answering this question will allow us to see how well our approach generalises. In particular, if it's able to find improvements in current code.

RQ3: Which MO algorithm is the most effective for MO-GI for Android?

Algorithm 2 Find method calls to cache in Class C

```
1:  function CLASSCACHEFINDER(M)
2:      cachable = ∅
3:      for each method m in C do
4:          for each expression e in m do
5:              if e is a method call expression then
6:                  cachable = cachable ∪ e
7:              end if
8:          end for
9:      end for
10:     Return cachable
11: end function
```

Original Code

```
class C1 {
public void foo(){
    int x = a();
    }
}
```

Mutated Code

```
class C1 {
int cachedVar1;
public void foo(){
   if (cachedVar1 == null){
      cachedVar1 = a();
   }
   int x = cachedVar1;
   }
}
```

Figure 6.4: An example of the Class Cache Operator. The result of a method call is stored in a field of the class for later use in any method.

There are a number of different MO algorithms available. We want to ensure that our approach utilises the most effective one, thus we investigate and compare a selection of MO algorithms successfully used in the GI domain in the past.

RQ4: How do the improvements found by MO-GI compare to those found by SO-GI for Android apps?

We wish to see if using MO algorithms limits GI's ability to improve apps, when compared to improving only a single objective. This is especially important in cases where one improvement can enhance two objectives (e.g., deletion can improve both runtime and memory use). We want to see if MO are still competitive in such cases.

RQ5: What is the runtime cost of MO-GI for Android?

Any improvements found by MO-GI must be considered against the cost of running

it. The improvements found must justify this cost.

RQ6: How does GI compare with available state-of-the-art techniques for Android performance improvement via code modification?
We want to compare GIDroid with state-of-the-art tools that are readily available to developers to see if our tool could offer an attractive alternative.

6.4 Methodology

In order to answer our research questions, we propose a series of experiments, running both multi- and single-objective GI on a benchmark of real-world Android applications.

To answer **RQ1**, **RQ3**, and **RQ5**, we run GI with three multi-objective algorithms on a set of applications, in some of which we know potential improvements are present, in order to validate our approach. To answer **RQ2**, we use the same setup to improve the latest versions of applications, to see if our framework can find yet-undiscovered optimizations

Next, to answer **RQ4**, we run GI with a single-objective hill climbing algorithm, to compare with a multi-objective approach. With this set of experiments, we can evaluate whether or not our multi-objective algorithms can find improvements that are as good as those found by single-objective search. This allows us to compare the trade-offs found by different search algorithms.

Finally, to answer **RQ6**, we use an Android linter to identify performance issues within our benchmarks. Linters are the only tools available to Android developers which can identify issues with source code that may affect performance properties we target. By manually repairing these issues we can see how our tool compares in terms of both effort and effectiveness with respect to existing tools available to developers.

6.4.1 Genetic Improvement Framework

We implement our multi-objective GI approach for Android in a tool called GIDroid, and use it to answer our RQs. Although there are many existing GI frameworks, Zuo et. al [136] found that PYGGI [137] and the Genetic Improvement In No time

tool (Gin) [96] were the only GI tools that could be readily applied to new software, with a more recent tool by Mesecan et. al [124] not yet available. However, none of the aforementioned work can be run on Android applications. Whilst Gin is compatible with most Java programs, and thus could potentially be easiest to extend, it is not compatible with the Android compilation and testing environments.

In GIDroid, we implement three MO algorithms: NSGA-II [127] as it is one of the most widely used multi-objective algorithms; NSGA-III [129], that was specifically developed for problems with 3 or more objectives in mind; and SPEA2 [128], which has recently proven successful for MO-GI in the desktop domain [124]. We use MO algorithms, as we believe that we will be able to find better improvements to some properties if we are able to sacrifice others. In particular, with our caching operators – these operators are likely to negatively impact the memory consumption of the applications, however a small increase in memory consumption may be worth it if it can sufficiently improve another property. The parameters used in these implementations can be found in Table 6.1.

To measure execution time we use Linux's time tool [19], we measure memory usage with the Java Runtime's memory allocation tracking [138] and we use Linux's built-in process-level network traffic tracking [139] to measure bandwidth.

6.4.2 Benchmarks

Genetic improvement requires a set of tests that cover the areas of code being improved, in order to validate that a non-functional property-improving patch does not negatively affect the app's functionality. Unfortunately, most open-source Android applications do not have test suites, and those that do are limited, achieving a median line coverage of 23% [99]. Furthermore, there is not a single tool that we have found in an extensive search of the literature which can automatically generate unit tests for Android applications. All automated testing tools for Android found ([140, 141, 14, 142, 17, 18, 143, 120, 144]) focus on testing UI via input generation in order to induce crashes and only run on devices/emulators, so would not be compatible with our framework. Moreover, they do not generate assertions — crucial for capturing correct app behaviour.

This meant that we had to manually construct unit tests for every single benchmark. We first had to attempt to understand each application and the component being improved and then attempt to create thorough, high-quality tests for them. In many cases, we had to account for asynchronous code, which was scheduled by the target code, and ensure that it executed completely during test execution. In other instances, we had to hunt down various parts of the state of the application to ensure they were correct. For each test we created, we ensured that it covered the methods which we wished to improve. We also added assertions about the state of the components of the application that were modified during execution. We achieved at least 75% branch coverage for methods used in our study. We do, however, note that developers would find this process simpler, as they already have an understanding of the application. They would get many other benefits from writing tests [145, 146] so the cost cannot be only placed upon the application of GI. Given the cost associated with manual testing, we set a threshold of 20 benchmarks for all our experiments.

To validate our approach, we first run GIDroid on applications with known performance issues. [135] has recently conducted a study of the changes that Android developers make to improve app performance. They propose that some of those changes are within the GI search space. For instance, moving an operation outside of a FOR loop, if it only needs to be executed once. While others are not yet achievable, e.g., requiring new code to be added that could not be achieved via mutation of the existing code base. We thus use [135]'s criteria to iteratively analyse the commits from their dataset that improve runtime, bandwidth, or memory use, until we reach our 20 benchmark target. In particular, we found 14 commits in previous work, spread over different versions of 7 applications. Since we also want to find improvements in current software, we stop our selection procedure here and use the current versions of the 7 apps, giving us a total of 21 benchmarks.

Once we had our set of versions of apps, we prepared them for GI. Firstly, we had to ensure the apps would build. Over time, a number of changes have been made to the Android build tools, making older versions of code incompatible with modern Android Studio. We require these build tools to function with Android

Table 6.1: Parameter settings for the MO algorithms used in our study.

Parameter	Value
Mutation Rate	0.5
Crossover Rate	0.2
No. Generations	10
No. Individuals	40
Selection	Binary Tournament
Crossover	Append Lists of Edits
Mutation	Add/Remove an Edit
Reference Points	Worst Observation (for each prop. and bench.)

Studio, so we can test and measure the test coverage of applications confirming that they can be safely improved. This meant that we had to update build scripts with newer versions of libraries and build tools. In some cases, there were bugs such as unescaped apostrophes in resource files, which prevented applications from building. These bugs were fixed. In a few cases, the benchmarks also used outdated non-Gradle build systems, so we wrote the necessary build scripts, and modified the project's directory structure, to be compatible with Gradle and thus with GIDroid. No source code was modified in this process.

We ran the PMD static analyser on the 7 applications and ran GIDroid on the classes which showed the most performance issues. This way we could see how our approach compares against human effort for finding performance-improving code transformations of existing code bases, for the 14 previously patched app variants. We could also see whether our approach is able to find yet unknown performance improvements in the current versions of the 7 apps.

6.4.3 Experimental Setup

For each version of code we improve, we run GIDroid 20 times with 400 evaluations. To minimise measurement noise, we use the Mann-Whitney U test at the 5% confidence level to determine whether there is an improvement of a given property (i.e., runtime, memory or bandwidth use). For the evolutionary algorithms, we divide these 400 evaluations into 10 generations with 40 individuals each, as was shown to be effective in previous work, including in the Android domain [147, 134].

We set the number of evaluations to 400 as, even when using simulation-based testing, the evaluation of an individual is slow, taking up to 2 minutes. We use the Genetic programming parameters in Table 6.1 as they have been used successfully in the past [126].

We had 2520 runs in total, taking a mean of 3 hours per run, resulting in roughly 7500 hours of computing time to test our approach.

All of our experiments were performed on a high-performance cloud computer, with 16GB RAM and 8-core Intel Xenon CPUs. We ran jobs across 10 nodes, each running separately to avoid interference between fitness measurements.

6.5 Results and Discussion

In this section, we present and analyse the results of our experiments, answering our Research Questions (Section 6.3). Throughout this section we will refer to the CPU time (s) of the test process as execution time, the size of the occupied Java heap as memory consumption (MB), and the number of bytes sent and received by the test process as network usage (B). Each of these objectives is a fitness function that we aim to minimize.

6.5.1 RQ1: Known Improvements

Figure 6.5 and 6.6 show the improvements found in the benchmarks in which we knew improvements were possible. We find improvements to both execution time and memory, but not bandwidth. We believe this is due to the nature of the benchmarks. Although feasible, only one application had bandwidth improvements in its history that would be achievable by GI. This improvement[1] required 2 insertions and 2 deletions at once to be achieved and thus was more difficult to evolve over time.

We find improvements to execution time of up to 26% and memory of up to 69%. We manually analysed the patches found in order to determine whether GI was capable of finding the same patches that developers made to improve their

[1] `https://github.com/erikusaj/fdroidTvClient/commit/bf8aa30a576144524e83731a1bad20a1dab3f1bc`

Table 6.2: No. of times GIDroid finds patches that contain edits semantically equivalent to developer patches, providing at least the same % performance improvement (Rep.) and no. runs where an improvement was found (Imp.). Each MO run was repeated 20 times.

Application Version	NSGAII		NSGAIII		SPEA2	
	Rep.	Imp.	Rep.	Imp.	Rep.	Imp.
Port Authority 1	4	16	8	18	3	15
Port Authority 2	0	17	0	15	0	14
Port Authority 3	0	13	0	14	0	18
Port Authority 4	4	15	8	17	10	13
Port Authority 5	5	19	3	19	0	12
Port Authority 6	4	13	7	18	2	11
Tower Collector 1	10	14	6	13	8	20
Tower Collector 2	0	15	0	18	0	19
Gadgetbridge 1	0	15	0	12	0	13
Fosdem Companion 1	3	13	4	12	7	14
Fdroid 1	0	19	0	17	0	13
Fdroid 2	8	14	4	12	6	16
Lightning Browser 1	2	12	3	18	4	17
Frozen Bubble 1	13	15	12	16	12	18

applications. The result of this analysis can be found in Table 6.2. In 64% of benchmarks GIDroid is able to find patches containing edits semantically equivalent to developer patches, providing at least the same % performance improvement. In other words, aside from reproducing improvements, in some cases, we find additional edits, further improving app performance.

Answer to RQ1: We find that MO search can find improvements of up to 26% for execution time and up to 69% for memory consumption on code where there are known improvements. In 64% of benchmarks, we are able to automatically produce edits that are semantically equivalent to developer patches.

6.5.2 RQ2: Improvements of Current Apps

Next, we analyse the results of the experiments on the benchmarks of current versions of applications, to see how well our approach generalizes to code in which

Table 6.3: Normalised Hypervolumes of the Pareto fronts found by GIDroid across our experiments, by algorithm.

Application Version	NSGAII	NSGAIII	SPEA2
PortAuthority 1 (PA1)	0.145	0.186	**0.458**
PortAuthority 2 (PA2)	0.223	0.267	**0.327**
PortAuthority 3 (PA3)	0.259	**0.285**	0.249
PortAuthority 4 (PA4)	**0.429**	0.225	0.112
PortAuthority 5 (PA5)	**0.247**	0.073	0.196
PortAuthority 6 (PA6)	0.053	0.053	**0.143**
PortAuthority Current (PAN)	0.051	0.133	**0.59**
Tower Collector 1 (TC1)	0.03	0.019	**0.127**
Tower Collector 2 (TC2)	0.027	0.052	**0.088**
Tower Collector Current (TCN)	0.254	0.017	**0.309**
Gadgetbridge 1 (GB1)	**0.611**	0.568	0.158
Gadgetbridge Current (GBN)	0.008	**0.384**	0.007
Fosdem Companion 1 (FS1)	0.318	**0.383**	0.359
Fosdem Companion Curr. (FSN)	0.105	**0.138**	0.021
Fdroid 1 (FD1)	0.016	**0.206**	0.012
Fdroid 2 (FD2)	0.022	0.042	**0.525**
Fdroid Current (FDN)	0.206	0.065	**0.233**
Lightning Browser 1 (LB1)	**0.322**	0.159	0.028
Lightning Browser Curr. (LBN)	0.038	0.037	**0.039**
Frozen Bubble 1 (FB1)	**0.097**	0.094	0.076
Frozen Bubble Current (FBN)	0.024	0.024	**0.026**

there are no known improvements.

The performance of each algorithm on versions of software is shown in Figure 6.5 and Figure 6.6. We find improvements to the execution time of up to 35% and to memory consumption of up to 32%. Again no improvements were found to bandwidth. We believe this is due to the nature of our benchmarks, where only FDroid 2 uses bandwidth extensively.

We have compiled the best changes found by GIDroid in these experiments to demonstrate the capabilities of GIDroid. We detail each of these patches below:

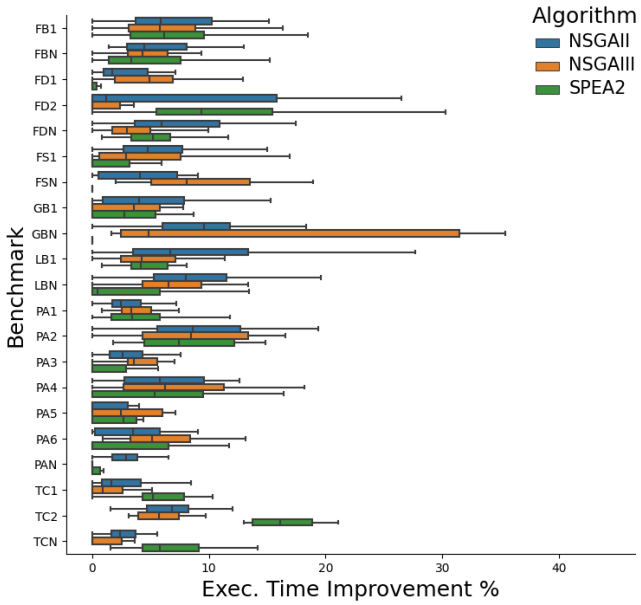


Figure 6.5: Execution time improvements (%) achieved by GIDroid using three MO algorithms on 21 versions of 7 Android apps.

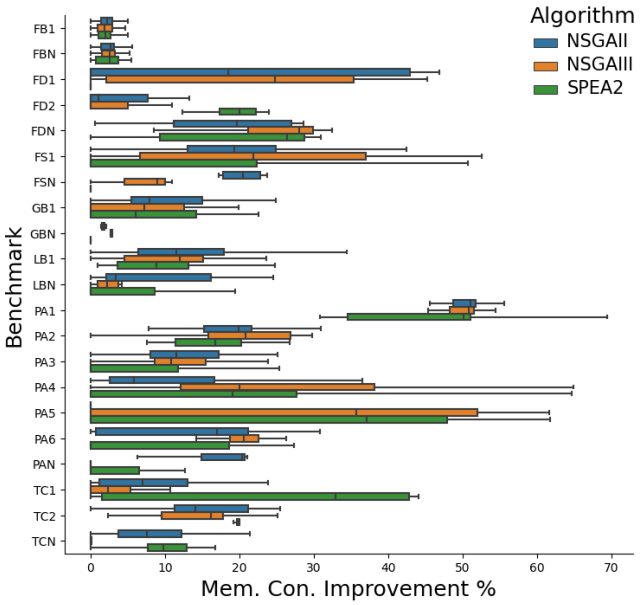


Figure 6.6: Memory consumption improvements (%) achieved by GIDroid using three MO algorithms on 21 versions of 7 Android apps.

6.5.2.1 Port-Authority (PAN)

In the Port Authority application, our best change found consisted of removing an unnecessary try-catch statement, which resolved the IP address of a URL. It would not only attempt to resolve URLs, but also, redundantly, IP addresses. Furthermore,

the resolved IP address is then passed to the constructor of the InetSocketAdress class, which already performs IP resolution, rendering the statement completely redundant. The error handling is also performed in the same way when the IP address is passed to the InetSocket.

6.5.2.2 F-Droid (FDN)

The improvement for F-droid refactored an if/else statement. Before, the statement checked if an object was null or not, instantiating it if it were null, and canceling its animation if not. However, after this statement, the object was instantly re-initialised. Meaning that in the case where the object was null, it would be instantiated once and then instantiated immediately after. We refactor the statement to remove the null clause and only cancel the animation if the object is not null.

6.5.2.3 Tower Collector (TCN)

In the TowerCollector, the best-evolved change consisted of changes to the way in which a database is handled. It ensured that the connection to the database is closed when no longer needed and that the database is only instantiated when it is actually needed. This change reduces memory usage but slightly increases execution time due to an extra function call.

6.5.2.4 Frozen Bubble (FBN)

In the Frozen Bubble application, the best improving change consisted of modifying how new rows of bubbles were instantiated in a row. We found that checking for -1 in the newly generated row was redundant as the row cannot contain a -1, it can only contain positive integers. We also found that the game pushed the sprite to the back of the board, but inspecting the application with and without this change shows no noticeable difference.

6.5.2.5 Fosdem-Companion (FSN)

In the Fosdem application the most improving change consists of moving the instantiation of two objects outside of a loop. This means the same object can be reused in the loop, with the need for a new object to be assigned, thus saving both memory and execution time.

6.5.2.6 Gadget-Bridge (GBN)

In the best change for the GadgetBridge Application we cache the method call which resolves the name of a file that is repeatedly used and removes the redundant rendering of a view that is already visible.

6.5.2.7 Lightning Browser (LBN)

In Lightning Browser, the best-evolved mutation consists of removing a check for whether or not a list of bookmarks is null. The list is an argument decorated with @NonNull so should never be null, and in the case that it is there will be no errors.

> **Answer to RQ2:** We find that MO search can find improvements of up to 35% for execution time and up to 32% for memory consumption on code where there are no known improvements. Many of these changes consist of caching method calls and removing unnecessary code.

6.5.3 RQ3: Multi-Objective Search

In order to compare the different algorithms used in search, we consider the procedure proposed by Li et. al [148], for comparing different multi-objective algorithms in a search-based software engineering context. We choose to measure the hypervolume (HV) of the data, as it is considered to be a good indication of the general quality of the Pareto fronts produced and is considered appropriate when there is no preference between the different properties being improved. In order to measure the hypervolume we specify the reference points as the worst observation for all fitness measurements, for each objective, as done in previous work [149, 150]. Due to different fitness scales, we normalise the values. Normalised hypervolume values are presented in Table 6.3. We find that across our experiment we find patches spread across the Pareto front (see Figure 6.7), showing that trade-offs between properties must be considered in the search process, due to the natural tension between them.

We find that NSGA-II performs similarly to NSGA-III, with the biggest hypervolume in 5 cases for both algorithms. We find that SPEA2 performs best, finding the best fronts in 11 cases. In general, the different algorithms seem to perform similarly in terms of the best improvements found, as shown in Figure 6.5 and Figure 6.6. We find that the caching operators we introduced turned out to be highly

Table 6.4: A effect size for each algorithm on each benchmark. Effect sizes larger than 0.5 show positive improvement. differences: N=negligible, S=small, M=medium, L=large

Benchmark	Exec. Time			Mem. Con.		
	NSGA-II	NSGA-III	SPEA2	NSGA-II	NSGA-III	SPEA2
PortAuthority 1	1.0 (L)	1.0 (L)	0.93 (L)	1.0 (L)	1.0 (L)	1.0 (L)
PortAuthority 2	0.98 (L)	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)
PortAuthority 3	0.97 (L)	0.97 (L)	0.97 (L)	1.0 (L)	1.0 (L)	0.93 (L)
PortAuthority 4	0.99 (L)	0.99 (L)	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)
PortAuthority 5	0.67 (M)	0.81 (L)	0.18 (L)	0.82 (L)	1.0 (L)	0.79 (M)
PortAuthority 6	0.88 (L)	0.99 (L)	0.71 (M)	0.91 (L)	1.0 (L)	0.9 (L)
PortAuthority Current	1.0 (L)	1.0 (L)	0.67 (M)	1.0 (L)	1.0 (L)	1.0 (L)
Tower Collector 1	1.0 (L)	1.0 (L)	0.89 (L)	0.98 (L)	1.0 (L)	0.92 (L)
Tower Collector 2	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)
Tower Collector Current	0.92 (L)	1.0 (L)	0.85 (L)	1.0 (L)	0.67 (M)	0.98 (L)
Gadgetbridge 1	0.87 (L)	0.96 (L)	0.53 (N)	1.0 (L)	1.0 (L)	0.54 (N)
Gadgetbridge Current 1	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)	1.0 (L)
FosdemComp. 1	1.0 (L)	0.95 (L)	0.67(M)	1.0 (L)	1.0 (L)	0.83 (L)
FosdemComp. Current	1.0 (L)	0.95 (L)	0.67(M)	1.0 (L)	0.83 (L)	1.0 (L)
Fdroid 1	0.77 (L)	0.92 (L)	0.73 (L)	0.82 (L)	1.0 (L)	0.76 (L)
Fdroid 2	0.99 (L)	0.93 (L)	0.92 (L)	1.0 (L)	1.0 (L)	1.0 (L)
Fdroid Current	0.74 (L)	1.0 (L)	0.99 (L)	0.98 (L)	1.0 (L)	0.99 (L)
LightningBro.	0.79 (L)	1.0 (L)	1.0 (L)	1.0 (L)	0.95 (L)	1.0 (L)
LightningBro. Current	0.9 (L)	0.83 (L)	0.59 (S)	1.0 (L)	0.9 (L)	0.92 (L)
FrozenBubble 1	0.98 (L)	1.0 (L)	0.97 (L)	0.98 (L)	1.0 (L)	0.97 (L)
FrozenBubble Current	1.0 (L)	0.93 (L)	0.88 (L)	1.0 (L)	1.0 (L)	1.0 (L)

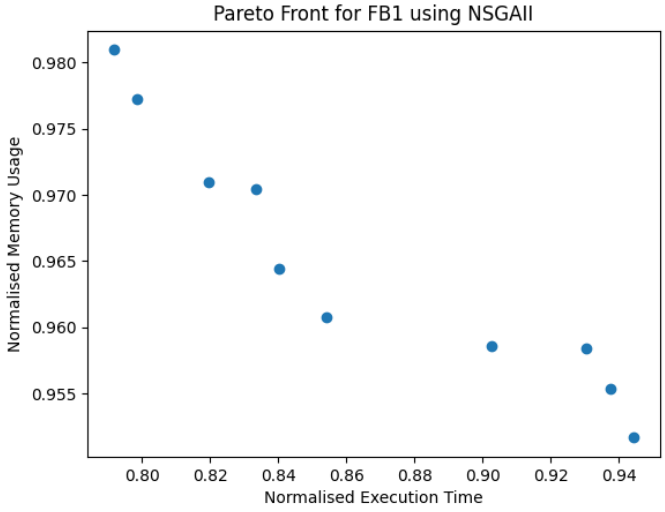


Figure 6.7: Pareto Front from NSGA-II experiments on the FB1 Benchmark.

Table 6.5: Maximum improvements to execution time and memory use found by GIDroid using SO-GI (no bandwidth improvements were found).

Application Version	Exec. Time (%)	Mem. Con. (%)
PortAuthority 1	23.39	71.69
PortAuthority 2	21.2	53.05
PortAuthority 3	23.13	33.76
PortAuthority 4	26.32	60.59
PortAuthority 5	28.03	59.13
PortAuthority6	24.44	24.43
PortAuthority Current	29.9	9.32
Tower Collector 1	16.01	30.82
Tower Collector 2	26.92	34.61
Tower Collector Current	20.9	32.43
Gadgetbridge 1	29.52	31.29
Gadgetbridge Current	26.73	5.89
FosdemComp. 1	32.8	36.81
FosdemComp. Current	10.31	13.62
Fdroid 1	21.82	17.06
Fdroid 2	27.94	33.01
Fdroid Current	14.14	32.18
LightningBrow. 1	28.45	8.96
LightningBro. Current	23.71	32.43
FrozenBubble 1	16.67	36.11
FrozenBubble Current	19.88	4.09

effective, appearing in 26% of improving patches.

We also evaluate the effect size of the improvements found by each of the MO algorithms, as show in Table 6.4. We use the Vargha and Delaney A measure ([151]) to calculate the magnitude of the differences between the observations of the NFPs of original applications and the improved versions. This measure is non-parametric so does assume data is normally distributed. We find that in all but 8 cases we find large effect sizes, and only find negligible differences in 2 cases.

Answer to RQ3: We find that the SPEA2 achieves the highest hypervolume of the 3 algorithms that we compared. We also find that the caching operator appears in 26% of patches.

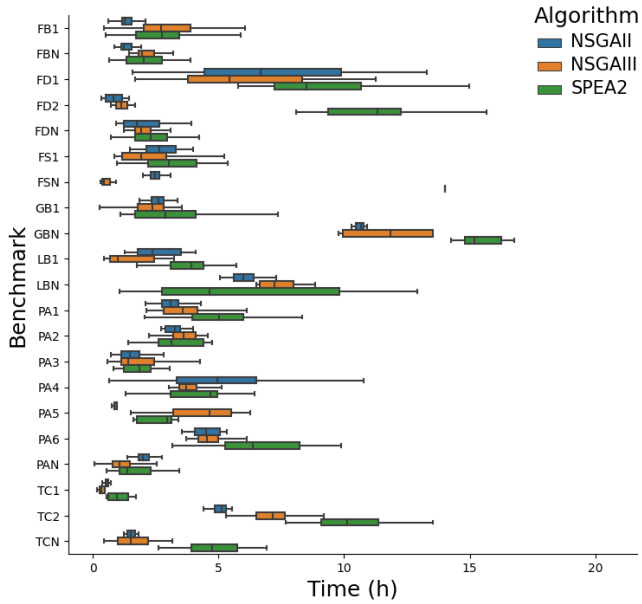


Figure 6.8: Time taken by GIDroid using different MO algorithms to evolve 10 generations, each with 40 individuals.

6.5.4 RQ4: Comparison to SO-GI

Next, we run single-objective genetic improvement on each of our benchmarks. We measure the effects of the changes found by SO-GI on our other properties. The results of this evaluation can be found in Table 6.5. We found improvements to execution time of up to 33% and memory consumption of up to 72%.

We find that SO search generally performs better when improving individual properties than multi-objective search. However, a multi-objective search was capable of finding improvements to both execution time and memory in a similar time as a single-objective search could find improvements to individual properties. Single-objective search produces results that improve one property in 753 of 1260 cases (21 benchmarks $*$ 20 runs $*$ 3 properties) but in 47% of these cases, patches are detrimental to another property.

Answer to RQ4: We find that SO search performs better than multi-objective search when improving individual properties. However, in 47% of cases, these patches are detrimental to other properties.

6.5.5 RQ5: Cost of GI

In order to evaluate the applicability of our approach, we analyze its cost. Figure 6.8 shows a boxplot of the time taken in hours for our experiments. We find that the time taken varies a lot between different benchmarks and in some cases even across different runs on the same benchmark. We find that MO-GI takes between 0.1 hours and 20.6 hours, with a median time across the benchmarks of 2.6 hours. The main source of variation across the benchmarks is the difference in time taken by the test suites. In the slowest benchmark, the test suite takes 8 seconds to execute, whereas the quickest one takes 2 seconds. In the slowest experiments, there were more patches that compiled, rather than instantly failing, further slowing down the experiments.

We find that SO-GI takes longer than MO-GI, with a minimum of 0.4 hours, a maximum of 19.0 hours, and a median of 3.5 hours. SO-GI can only find improvements to one property at a time, showing the much-improved efficiency of using MO-GI. Despite hour-long runtimes, we note that this is a one-off cost. Given that users consider apps running for 150ms laggy, which might lead to them abandoning an app, we deem the cost of running MO-GI worth it.

> **Answer to RQ5:** We find that MO-GI takes between 0.1 hours and 20.6 hours, with a median time across the benchmarks of 2.6 hours. We find that MO-GI is quicker than SO-GI which takes a median of 3.5 hours.

6.5.6 RQ6: Comparison to Linter

In order to compare our approach to the currently available tooling for improving performance for Android, we run a well-known Linter (PMD) on all of the benchmarks which we improved. We configured it to provide warnings when any of its performance rules were violated. We then manually analyzed each of the warnings that it provided, and in the cases where they could be repaired without disrupting the functionality of the application, we repaired them.

We then measured the performance differences between the repaired and unrepaired versions of the applications. We found that in our 21 benchmarks, 5 had

either no warnings or warnings that could not be repaired without introducing buggy behavior. For example, a warning about instantiating an object in a loop could be "unfixable" as a reference to each instantiated object is held in an array. So, moving the instantiation outside of the loop would result in an array with the same reference repeated multiple times.

In all cases where possible, the fixes were easily created and very similar to the examples given in the PMD documentation.

Of the 16 where fixes were possible, only 9 actually offered any improvement. The maximum improvement to execution time was 4.5%, while to memory it was 10.42%, when compared with 35% and 69%, respectively, achieved by GIDroid. No improvements to bandwidth usage were found. Only a single one of these patches improved multiple properties, and 6 were detrimental to other properties. Of those improvements, none had any impact on the bandwidth of the applications. The linters were, however, significantly quicker than GI, taking a maximum time of 20 minutes to repair the warnings. However, unlike GI this process is not automatic and requires a developer to be engaged at all times and the improvements found were much smaller than those found by GI.

Answer to RQ6: We find that our setup is more effective than linters for improving the non-functional properties of Android apps. We find improvements to execution time that are 6.6x larger than those found by the linter 7.8x larger for memory consumption.

6.6 Threats to Validity

There are a number of threats to the validity of our study. We discuss these next, including steps we took to mitigate them.

The measurements we use for our fitnesses are noisy. To mitigate this threat, we repeat each measurement 20 times during search and after the search is complete. We use the Mann-Whitney U test at the 5% confidence level to determine whether there is an improvement. We tested our measurements on known improvements and found that they are consistently detected.

Furthermore, we use tests to determine whether or not a patch is valid. This

Table 6.6: Improvements (%) from repairing linter warnings, for benchmarks where viable improvements were found.

Application Version	Exec. Time	Mem. Con.	Time (min.)
PortAuthority 1	-2.5	2.8	2
PortAuthority 5	2.4	10.4	9
PortAuthority Current	0.9	-2.8	1
TowerCollector 2	0 1	0	5
TowerCollector Current	0.0	1.9	7
Fdroid 1	4.5	0	13
Fdroid Current	2.3	-0.2	9
LightningBrow. 1	-2.2	0.4	1
LightningBrow. Current	0.9	-1.6	5
FrozenBubble 1	3.5	-0.1	20
FrozenBubble Current	-1.6	0.4	15

does not guarantee correctness. However, the patches produced can undergo the standard code review procedure as any other code being integrated into a project would. We conducted a manual analysis of all the patches on the Pareto fronts (1753 total), to ensure the improvements reported here do not disturb app functionality. Through manual analysis, we found that 1352 out of the 1753 best patches found did not disrupt the functionality of the apps, demonstrating the strength of our test suites. Disruptive patches included the removal of some error handling and the deletion of some components rendered on screen that could not be detected with unit tests. They would be easily discarded by code review.

Using stochastic search may result in us finding improvements out of sheer luck. In order to avoid this issue, reliably compare different algorithms, and demonstrate generalisability of our approach, we run each of the algorithms tested 20 times on each of our 21 benchmarks.

The search algorithms we use rely on parameters such as mutation and crossover rate. The values of these parameters can have an effect on the effectiveness of the algorithms. To mitigate this threat, we use the same parameters across all experiments for fair comparison. We use settings used in previous work that found improvements in software.

We tested our approach on 21 versions of 7 Android apps, which poses a threat to generalisability to other software. However, these apps are diverse in size and type. Moreover, we found improvements in current app versions, which were previously undiscovered. Unfortunately, currently, the big obstacle to wider adoption is test availability. For each benchmark, these took us hours to produce. However, the benefits of testing go beyond the applicability of our approach. We envision with the development of more fine-grained automated test generation tooling for Android and better testing practices, further benefits of GI can be unlocked.

To mitigate such threats further, we will make all our code and results freely available upon publication, allowing other researchers and developers to use and extend our tool and validate our work.

6.7 Conclusion

In this chapter, we use multi-objective genetic improvement (MO-GI) to automatically improve Android apps. We are the first to apply MO-GI with three objectives to improve software performance and evaluate the feasibility of MO-GI for bandwidth and memory use in the Android domain. To evaluate the effectiveness of the proposed approach we developed GIDroid, which contains 3 MO algorithms and 2 novel cache-based mutation operators. We have tested GIDroid on 21 benchmarks, targeting runtime, memory, and bandwidth use. We find improvements to the execution time of up to 35% and memory consumption of up to 65%.

However, we find that for the benchmarks we used, our approach cannot find improvements to bandwidth, even though they are within GIDroid's search space. We believe that only certain parts of applications actually affect the network usage of the app. In our experiments, we did not specifically target code that accessed the network and thus may not present any opportunities for improvement. There is also the chance that the changes that could improve network usage are either not possible or very difficult to achieve with current mutation operators. Network usage specific operators may offer a better chance at improvement and should be explored in the future.

In the next chapter, we explore the problem of improving network usage. We attempt to profile apps to find their most network-intensive methods and modify the mutation operators used to specifically improve bandwidth usage.

Chapter 7

Reducing Network Usage with Genetic Improvement

Following the negative results attained in the previous chapter, we decided to see if the GI technique could be adapted further to specifically target network usage.

Work improving network usage for Android apps, has mostly utilised prefetching. Prefetching allows online assets which are fetched to be cached locally. If the asset is fetched multiple times, the amount of network usage will be reduced. However, the primary aim of pre-fetching is to avoid having to wait for assets to be downloaded when they are needed and improve the responsiveness of applications. If a pre-fetching scheme is too aggressive it may fetch assets that are never actually needed and thus increase network usage. Thus far, only one work has found improvements to network usage through modifications to the source code of apps. This is despite the fact that developers do make many kinds of changes to source code which improve network usage, as shown in Chapter 4.

In the previous Chapter, we attempted to improve the network of Android applications using genetic improvement, however, were unsuccessful. We believe that a combination of the benchmarks used not being network intensive and the limited power of the current GI mutation operators were to blame.

Thus, we propose using a profiler to identify the most network-intensive areas of code and applying genetic improvement with novel, network-specific mutation operators to the identified code.

We apply our approach to 7 applications that we have identified as being network-intensive. However, we find no improvements, suggesting that network usage may not be an appropriate target for GI.

Overall, with this work, we provide the following contributions:

1. A new profiler to identify network-intensive methods in Android applications.
2. A set of Android applications that use network extensively, for future research in network usage optimization.
3. A set of mutation operators that mimic modifications made by developers that improve network usage, including a novel operator that avoids unnecessary HTTP requests.
4. An empirical study showing the effectiveness of our approach at finding improvements to network usage.

Our framework and profiler, as well as all the results obtained, are available on the following website, so others can verify and extend our work: `https://github.com/SOLAR-group/NetworkGI`

The rest of the chapter is structured as follows:

1. Section 7.1 describes how we refine the approach in the previous chapter to improve network usage.
2. Section 7.2 lists the research questions we have about our approach for improving network usage.
3. Section 7.3 describes the methodology we use to answer our RQs.
4. Section 7.4 details the results of our experiments.
5. Section 7.5 explains the threats to the validity of our work and how we mitigate them.
6. Section 7.6 lists the conclusions of our work.

7.1 Approach for Improvement of Android App Network Usage

We use the framework used in Chapter 6, also presented in Figure 7.1.

GIDroid is a framework for applying genetic improvement to Android appli-

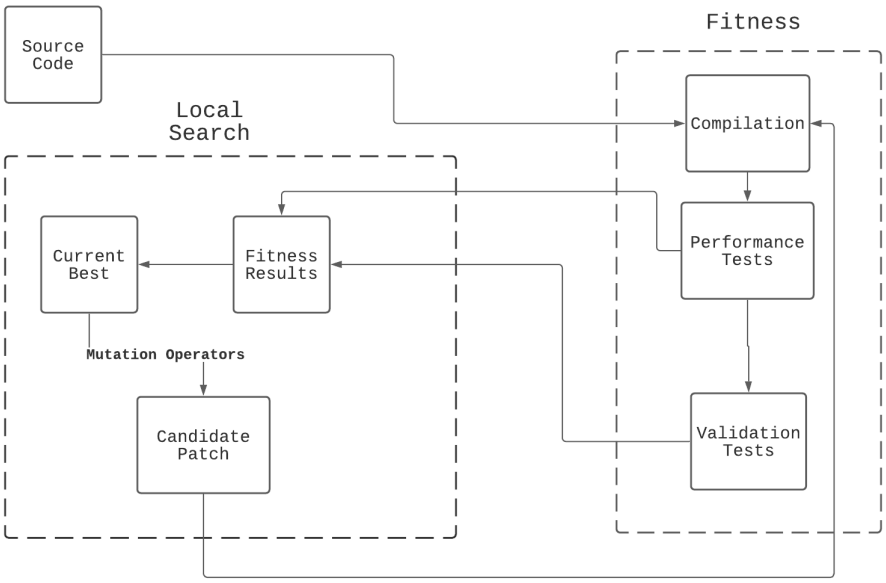


Figure 7.1: Overview of the GIDroid framework for optimization of non-functional properties of Android applications using genetic improvement.

cations. GIDroid generates variants of Android applications in the form of lists of edits to their source code. These edits include the "traditional" GI operators, which can copy, replace, delete and swap statements in the AST, along with two caching operators to avoid unnecessary method calls. These variants are then improved with the use of search algorithms (such as genetic programming). GIDroid randomly generates variants and validates them with unit tests, particularly tests written in the simulation-based library Robolectric [118]. This saves on the time needed to transpile, package, and install the applications on actual devices or emulators, speeding up the GI process. If all tests pass, the variant is considered valid, if it is invalid its fitness will be set to the worst possible value. During test execution, the non-functional property/ies being improved can be measured and then set as the fitness for valid patches. The fitness is then given to the search algorithm to generate further variants.

We modify the framework in two key ways. Firstly, rather than using the PMD static analyzer to detect target methods for modification, we develop our own profiler, specifically for identifying methods that make large HTTP requests. The PMD static analyzers performance patterns only concern memory usage and execution

```
URL url = new URL("http://www.android.com/");
HttpURLConnection urlConnection =
(HttpURLConnection) url.openConnection();
BufferedReader br =
new BufferedReader(urlConnection.getInputStream());
String strCurrentLine;
while ((strCurrentLine = br.readLine()) != null) {
    Log.d("AndroidHttpProfiler", "ThisClass.getAndroid");
    Log.d("AndroidHttpProfiler", strCurrentLine.size());
    doSomething(strCurrentLine);
}
```

Figure 7.2: An example of an instrumented `HttpUrlConnection` request. First, a HttpURLConnection object is instantiated, and then its input stream is read with a buffered input stream. We instrument the code to log the method name (ThisClass.getAndroid) and the data received over the network.

```
String url = "http://www.android.com/";
StringRequest stringRequest =
new StringRequest(Request.Method.GET, url,
    new Response.Listener<String>() {
        @Override
        public void onResponse(String response){
            textView.setText("Response is: "
            + response.substring(0,500));
             Log.d("AndroidHttpProfiler", "ThisClass.getAndroid");
             Log.d("AndroidHttpProfiler", response.size());
        }
    }, new Response.ErrorListener(){
        @Override
        public void onErrorResponse(VolleyError error){
            textView.setText("That didn't work");
    }
});
```

Figure 7.3: An example of an instrumented `volley` request. An object which extends the Request class is created and we can find the response in the overridden onResponse method.

time, so with a more appropriate profiler, we may be able to achieve better results. Secondly, we modify the set of mutation operators used to more closely reflect the changes made by software developers, as identified in our mining study, described in Chapter 4, on how developers improve non-functional properties of Android apps.

```
OkHttpClient client = new OkHttpClient();
String run(String url) throws IOException {
    Request request = new Request.Builder()
    .url(url)
    .build();
    Response resp = client.newCall(request).execute()
    String reponseString = resp.body().string();
    Log.d("AndroidHttpProfiler", "ThisClass.getAndroid");
    Log.d("AndroidHttpProfiler", reponseString.size());
    return reponseString;
    }
}
```

Figure 7.4: An example of an instrumented `okhttp` request. The execute method is called on an OkHttpClient object and returns a response. As before, we log the method name and the data received.

7.1.1 Network Usage Profiler

In order to identify the most network-intensive areas of code, we develop a profiler. This profiler identifies and instruments HTTP requests made in three popular HTTP libraries in Android (`HttpUrlConnection`, `okhttp`, and `volley`). Both `volley` and `HttpUrlConnection` are official Android HTTP libraries, whereas `okhttp` is a popular third-party library (appearing in almost 5% of all applications in the Google Play store [152]) which is in fact used as the backend of `HttpUrlConnection`. We show example requests made by each library in Figures 7.2, 7.3, and 7.4. We use the Soot[1] static analysis tool to find invocations of HTTP requests in each of these libraries and then exercise the application, logging the size of the data that is sent and received. In the case of HttpUrlConnection, the static analysis looks for the invocationof the read method on a BufferedReader which is reading the InputStream of a HttpUrlConnection object and logs the size of the lines that are read by the buffered reader. In the case of Volley, all overridden onResponse and onErrorResponse methods on Reponse.Listener objects are modified to log the size of the response. Finally, for OkHttp, we simply log the size of the body of responses that are created with the Call.execute() method. Once we have run static analysis, we can discard those applications that do not contain any

[1] `https://soot-oss.github.io/soot/`

invocations of the APIs of interest.

We then use automated testing tools to find the methods that result in the largest and most frequent usages of the network. In particular, we use the Monkey testing tool [12] to randomly exercise the application being profiled and exercise as much of the code as possible. We run Monkey with 1000 random inputs. Whilst other more advanced automated testing tools are available (e.g., Mahmood et al.'s Evo-Droid [17] and Mao et al.'s Sapienz [18]), we choose to use Monkey as it is compatible with the latest versions of Android unlike the testing tools available at the time of experimentation.

7.1.2 Novel Mutation Operator Targeting Network Usage

We introduced a new mutation operator, based on the results of our mining study, described in Chapter 4. In particular, we found that many developer-made changes would add conditional branching around statements, to only make the requests over the network when they were actually necessary. This mutation operator wrapped statements in `if` statements, with the goal of avoiding making unnecessary requests. The conditions of the `if` statements consist of comparisons between local variables and primitives. In the case that the local variable is a primitive, direct comparison can occur. In the case where the local variable is not a primitive, we use either one of the variable's fields which is a primitive, or one of its methods that returns a primitive for comparison. In order to select a variable for comparison, we use a random selection which is weighted based on the distance between the statement being wrapped and the closest use of the variable in the AST. The probability of the comparison variable V_c being equal to variable v_x given that the target statement S_t is equal to s and the number of potential target variables ($\{v_0, v_1, ...v_n\}$) is n is shown in Equation 7.1.

$$Pr(V_c = v_x | S_t = s) = distance(v_x, s) / \sum_{i=0}^{n} distance(v_i, s) \tag{7.1}$$

We use this weighting to prefer comparisons with variables that are more relevant to the statement being wrapped. Figure 7.5 shows how this operator is applied

Table 7.1: The potential operators and values that primitives can be compared with and to, depending on the type of the primitive selected for the newly created `if` statement.

Type	Operator	Value to compare to
Boolean	$\{==\}$	$\{true, false\}$
Other	$\{==, <, \leq, >, \geq\}$	$\{0, 1, 2, 3, 4, 5\}$

in practice. We select the comparison from those shown in Table 7.1, these conditions are based on those suggested by Brownlee et al. [69] for injecting shortcuts into code. They were extended to allow comparisons to integers from 0-5, rather than just 0, as these conditions are observed in real commits [135].

7.2 Research Questions

In order to evaluate the effectiveness of our proposed framework for improvement of network usage of Android applications, we pose the following research questions:

RQ1: *How much data do Android applications send over the network through* HTTP *requests*?

We want to know how much of an impact HTTP requests have on network usage in Android applications and how network-intensive the methods that our profiler identifies are.

RQ2: *How effective is Genetic improvement at reducing the network usage of Android applications?*

We want to know if genetic improvement can automatically reduce the amount of data sent and received over the network in Android applications, and what is the impact on the amount of data sent and received.

RQ3: *How expensive is it to improve the network usage of Android applications using genetic improvement?*

We want to know how long the GI process takes to find improvements. If the process takes an exceedingly long time for small improvements it may not be worth it for developers to use GI in a real-world setting.

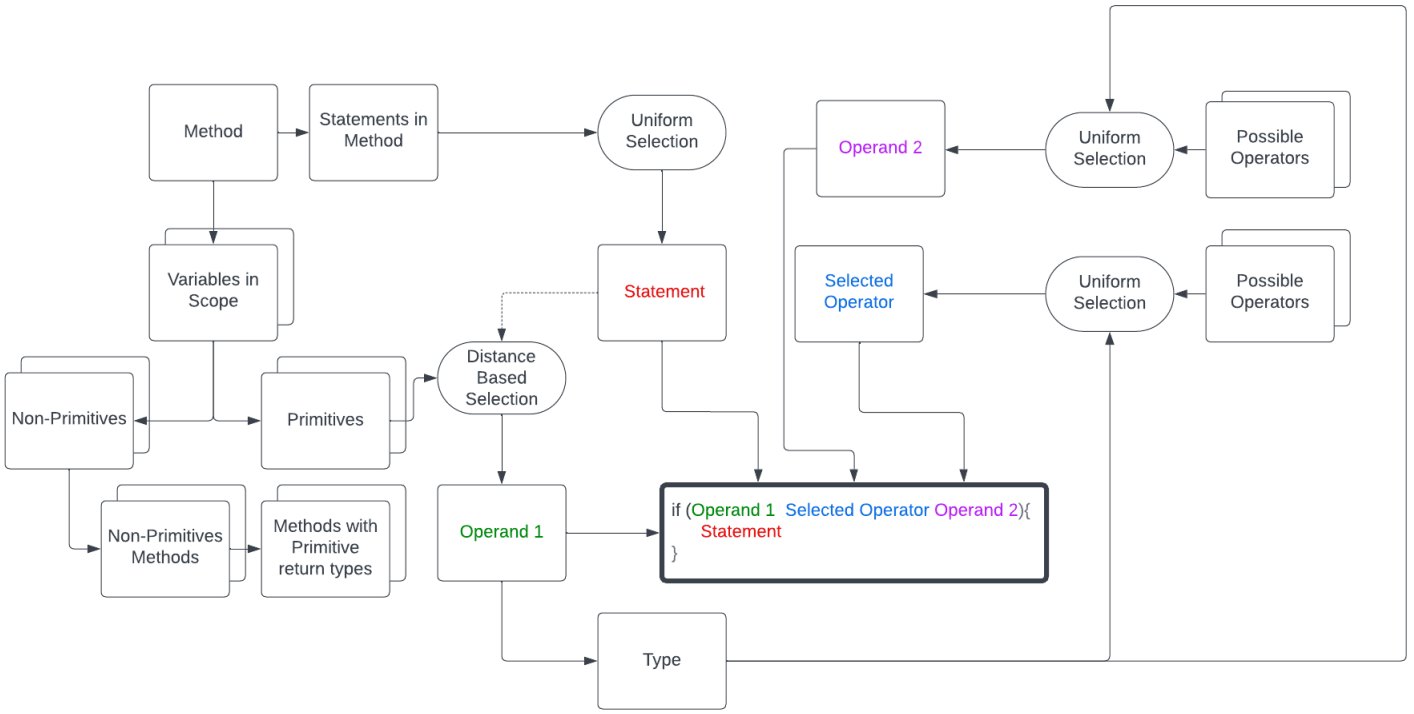


Figure 7.5: Process for creating a new `if` statement wrapper. First, a statement to be wrapped is selected from the target method. Next, either a primitive local variable or a method of a non-primitive local variable with a primitive return type is selected. This selection is based on the distance of the variable from the statement. Then an operator is selected based on the type of the primitive that was selected. Then, a value to compare to is selected, again, based on the type of the primitive. Finally, an `if` statement is constructed from the selected components and inserted into the target method.

(a) Code before mutation

```
...
Asset asset = assets.get(0);
Request request = new Request.Builder()
.url(asset.url)
.build();

Response response = client.newCall(request)
                                .execute()
...
```

(b) Code after mutation

```
...
Asset asset = assets.get(0);
Request request = new Request.Builder()
                            .url(asset.url)
                            .build();
if (asset.isNeeded() == true){
     Response response = client.newCall(request)
                            .execute()
}
...
```

Figure 7.6: An example of the 'add condition' operator, checking if the method `isNeeded` of the local variable `asset` return true. The introduced `if` statement is highlighted in bold text. This mutation avoids unnecessary HTTP requests.

7.3 Methodology

In order to evaluate our framework for the improvement of network usage of Android applications, we propose the methodology described in this section.

7.3.1 Framework for Network Usage Optimization

We use the same framework as described in Chapter 6, i.e., GIDroid. However, we modify it with the addition of our new mutation operators which specifically target network usage. Each individual program variant is represented as a patch, where each patch consists of a list of edits which is sequentially applied to the source code of the problem.

Fitness Function To evaluate each program variant i.e., whether it improves

network usage without sacrificing functionality, the corresponding patch is applied to the code, and the program is run against test cases to evaluate its fitness. We instrument the applications to log the sizes of HTTP queries, allowing us to directly measure the bytes sent and received over the APIs of interest. We then use this measurement as a fitness in our search algorithms, with the goal of minimizing network usage, i.e., variants with lower fitness measurements are considered fitter.

Mutation Operators Aside from our new mutation operator (see Section 7.1.2), we use a subset of the mutation operators used in GIDroid which were shown in our mining study to improve bandwidth, i.e., the 'caching' operators, and the 'delete statement' operator. We also use the same mutation rate as in previous work. There are two types of caching operators: one caches a variable value, while another a method call. The delete operator simply deletes a randomly selected statement. The aim of each of these operators is to avoid making unnecessary HTTP requests by either removing unnecessary invocations, storing their results and reusing them, or avoiding them when some state of the application suggests that they are unnecessary[2]

Crossover Operator We use the same crossover operator as used in GIDroid , which appends sections of individuals onto the end of others, as they have shown to be successful in improving other non-functional properties. We also use the same crossover rate as in previous work.

Search Strategy We evaluate both the Local Search algorithm and the Genetic Programming algorithm available in the GIDroid framework. Genetic Programming (GP) stochastically generates a set of patches (the population) and simulates evolution upon them to find better patches. In GP each patch is applied and, if valid, its fitness (in our case network usage) is measured. The next generation is then created through tournament selection (size 2), where two individuals are randomly selected, and the fittest is added to the new generation. Mutation and crossover are then applied to add new edits and combine individuals in the new generation. This process continues for a set number of generations where a population of improved

[2]We opt not to use the operator proposed by Li et al [57] as none of our benchmarks contained sequential requests.

patches is produced.

In Local Search (LS), we maintain a single best individual, at each step, we add or remove and evaluate its fitness. If the new individual is fitter than the existing best it becomes the best individual. This repeats for a set number of steps where we have a single best individual. We will use 400 evaluations in each run, 400 steps in LS an 10 individuals for 10 generations in GP.

7.3.2 Benchmark of Network-Intensive Android Applications

To evaluate our approach, we collect a set of Android applications that contain network-intensive methods. As we need tests to validate the patches that we produce, we begin by trying to find applications with network-intensive methods that are covered by local tests.

To identify these applications we run our profiler on 2 sets of applications. Firstly, we profile the applications identified by Pecorelli et al. [99] as being covered by tests in their study on the way in which every application available on the open-source app store FDroid [119] was tested. We eliminate those applications which do not contain any unit tests. Secondly, as the study by Pecorelli et al. [99] was performed in 2020, we also consider all applications that have been released since the study was performed and made available on FDroid. This resulted in a total of 4443 apps.

If our profiler identified any methods in an application that used one of the libraries previously discussed, we checked whether it also had unit tests that covered these methods. In the case where multiple methods were identified, the one that resulted in the most network traffic was selected for improvement. We use the data collected in this step to answer RQ1. In some cases, the methods identified were simply wrappers around HTTP requests, in these cases we instead ran GI on the methods which utilized the wrappers most often.

We also added in the application (F-Droid Client) that we previously failed to improve, but now there is a possible improvement to be made based on real developer commits Unlike the other applications being improved, this application is not the latest version but a previous version of the application, one commit before

Table 7.2: Android applications and commit sha of version targeted for improvement and links to their repositories.

Application		Repository
Adaway	Repo	`https://github.com/AdAway/AdAway`
	sha	75bee423e8635f84266c521e94cf177c1521ff6c
FDroid Client	Repo	`https://github.com/f-droid/fdroidclient`
	sha	bf8aa30a576144524e83731a1bad20a1dab3f1bc
GPS Logger	Repo	`https://github.com/mendhak/gpslogger`
	sha	5437cfff42d728111f9a0ca03dc7f52a11beafc9
Mi Mangu Nu	Repo	`https://github.com/raulhaag/MiMangaNu`
	sha	84f8773985af04e0c96d2d5290f3f1245107c39e
Materialistic	Repo	`https://github.com/hidroh/materialistic`
	sha	b631d5111b7487d2328f463bd95e8507c74c3566
F-Droid Build Status	Repo	`https://codeberg.org/pstorch/F-Droid_Build_Status/`
	sha	818ae54b2398d1b9ec7e2ccc8f620431f001b2b6
Ooni Probe	Repo	`https://github.com/ooni/probe`
	sha	26dd6c96dd7129b635f15c4d4bf956939a9cdb44

a network usage improving commit was made. We know that this improvement lies in the search space of our mutation set, thus forming a baseline for our approach. All benchmarks use developer-written tests other than F-Droid Client which we created a test suite for.

In total, this selection process resulted in the 7 applications, shown in Table 7.2. Whilst we found many applications that contained HTTP API usages, the overwhelming majority were not covered by any of the application's tests. At the time of experimentation automated tools for unit test generation for Android did not support the later Android versions and, thus could not be utilized to generate tests for the methods of interest.

7.3.3 Experimental Setup

For each of our benchmarks and each search algorithm, we perform 20 runs, as Genetic Improvement is stochastic, and statistical tests are needed to evaluate its efficacy. The results of these runs will be used to answer RQs 2 and 3. We perform all of our experiments on a cloud computer with 16GB RAM and 8-core Intel Xenon CPUs.

7.4 Results

Next, we discuss and analyze the results which we attained from our experiments.

Table 7.3: Network used by applications identified by our profiler which had network-using methods covered by unit tests, the number of KLoC in each application, and the most network-intensive method name.

Application	KLoC	Network usage (kB)	Most network-intensive method
Adaway	21.6	110.2	GitHubHostsSource.getLastUpdate
FDroid Client	88.5	237.9	FDroidApp.onCreate
GPS Logger	23.2	2.4	GoogleDriveJob.updateFileContents
Mi Mangu Nu	33.1	512.1	NineManga.getMangasFiltered
Materialistic	31.1	17.1	UserServicesClient.submit
F-Droid Build Status	7.1	1.5	FdroidClient.getRunning
Ooni Probe	32.7	147.6	MeasurementsManager.downloadReport

7.4.1 RQ1: Network Used

For each of the applications found in which network requests were covered by tests and thus suitable for Genetic Improvement, we measure the amount of network used by the applications with random inputs from the Monkey testing tool. The results attained are shown in Table 7.3.

We find that our profiler is capable of identifying methods in applications that use between 1.5 kB and 512kB of data. This data is collected over only 10000 inputs, taking a few minutes to execute. For real users, this could result in large amounts of data usage if they use the applications often, demonstrating the need for network usage reduction in Android applications.

> **Answer to RQ1:** We find that our profiler is capable of identifying methods in applications that use between 1.5 kB and 512kB when exercised with 10000 random inputs. This is only over the course of a few minutes and real usage is likely to result in large amounts of data being transmitted.

7.4.2 RQ2: Improvements to network usage

Unfortunately, we did not find any improving patches in our experiments. One possible reason is that the potentially improving mutations are too sparsely distributed in the search space. Both Genetic Programming and Local Search rely on being able to find small improvements and build upon them, so-called "exploitation". However, for this particular problem, search algorithms that can explore the

Table 7.4: Number of potentials `if` statements which could be inserted for each benchmark.

Application	Search Space Size
Adaway	11,088
FDroid Client	157,859
GPS Logger	32,457
Mi Mangu Nu	236,918
Materialistic	196,962
F-Droid Build Status	44,352
Ooni Probe	10,065

search space in a more intelligent way may be more useful. We are also limited by time. Test executions in Android are slow, often taking minutes to evaluate a single variant. If we wish to perform 1000s of evaluations and explore large areas of the search space, we require faster ways to test or validate that the variant is equivalent to the original program.

To investigate this, we calculate the number of potential `if` statements that could be added to each of our benchmarks. We find that each benchmark has between 10,000 and 250,000 potential conditions that could be inserted. We show these values in Table 7.4. With the relatively long-running tests needed to evaluate each mutant, 400 evaluations are simply not enough to explore the 10s of thousands of potential edits in the search space in a reasonable amount of time, posing the need for effective heuristic search strategies.

Answer to RQ2: We find that our approach is unable to find improvements to network usage. We believe that this is because we cannot effectively explore the very large search spaces of this problem.

7.4.3 RQ3: Cost of Genetic Improvement

As shown in Figure 7.7, running GI with the Genetic Programming Meta-Heuristic takes between 0.4 and 7.0 hours, with a median time of 4.3 hours. Alternatively, as shown in Figure 7.8 when using Local Search we find that Genetic Improvement takes between 0.3 and 7.6 hours, with a median time of 4.8 hours. The difference between the two approaches is due to Local Search producing more compiling variants which must then be tested, taking more time. This is not surprising as every

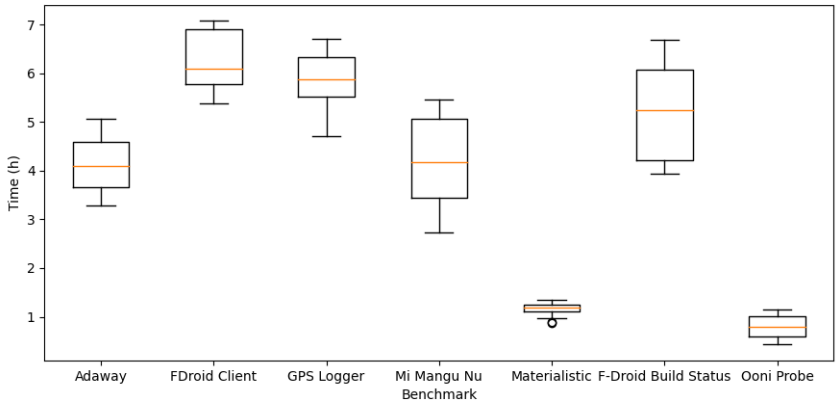


Figure 7.7: Time taken by Genetic Improvement when using Genetic Programming for each of our benchmarks

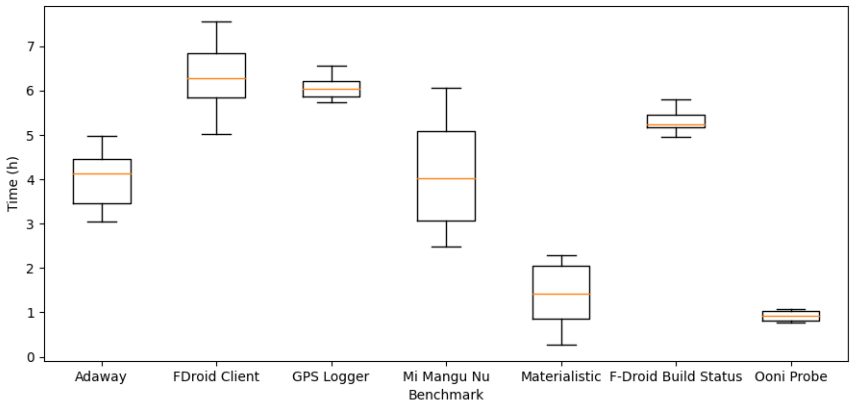


Figure 7.8: Time taken by Genetic Improvement when using Local Search

individual in Local Search is a single edit away from a variant that compiles and passes all tests, whereas variants generated in Genetic Programming may be multiple edits away. However, this may allow GP to explore the search space more quickly than LS and be more successful in future work.

> **Answer to RQ3:** We find that running GI with the GP meta-heuristic takes between 0.4 and 7.0 hours. We find that when using Local Search GI takes between 0.3 and 7.6 hours. The median time taken by GP (4.3 hours) is less than that taken by LS (4.8 hours).

7.5 Threats to validity

We present threats to validity to this work and our mitigation methods.

Using unit tests to assess whether two programs are equivalent can lead to

false positives or variants that pass all tests but are not equivalent to the original program. This threat can be mitigated through a standard code review of any patches suggested by GI by the developers of the project being improved.

Genetic Improvement is a stochastic process. This means that in some cases it can get "lucky" and find strong improvements that it wouldn't find in a normal run. We mitigate this threat by performing 20 runs for each of our benchmarks for each of our search algorithms. This gives us confidence that we know how our approach will perform in a standard run.

Whilst Madaan et al. [153] have shown that large language models can be used to improve the execution time of C++ programs, we found that we could not reproduce the changes made by developers for the improvement of the non-functional properties of Android applications (Chapter 6).

Finally, we make our tool and results publicly available so that our work can be validated and replicated. These are available at `https://github.com/SOLAR-group/NetworkGI`

7.6 Conclusions

In conclusion, we propose an approach for the identification and improvement of network-intensive methods in Android applications. We augment an approach that was previously successful in improving execution time and memory consumption, but not network usage, to specifically target network usage by only making changes that could improve network usage.

We implement a profiler, which instruments http libraries so that we can observe the size of the network traffic being sent and received. We then exercise the applications so that the http requests will be performed and the network usage can be measured.

Additionally, we implement a new mutation operator, based on real developer commits. This operator wraps statements in conditionals, to prevent unnecessary network traffic.

We identify 7 applications with network-intensive methods that are covered

by the applications' unit tests and evaluate our approach on them. We find that our approach cannot successfully explore the tens of thousands of potential changes that could be made to our benchmarks to find patches that improve network usage. In the future, work into either more efficiently exploring the search space or reducing its size may be beneficial to this approach

We do however provide the implementation of our tool and results so that future researchers can improve this approach and hopefully find success for this important problem.

Chapter 8

Conclusions

Non-functional properties are very important in Android, concerning both users and developers. They can lead to bad reviews and even deletion of apps. One approach which has shown promise for improving non-functional properties of traditional software is Genetic Improvement, a technique which automatically generates and improves patches to the source code of software with respect to a particular property. However, the application of GI in the mobile domain has not been well explored. Thus in this thesis we investigate how GI can be applied to Android apps to improve non-functional properties.

8.1 Contributions

We initially targeted frame rate for improvement, however, achieved limited success (Chapter 3). A large reason for this was the weakness of the test suites available in the applications we were improving. This remains one of the largest challenges of applying GI to Android apps. We also found that testing applications running on devices or emulators was impractical and slow.

Next, we decided to identify the types of changes that developers made and find the areas of GI that could be improved or tuned to more closely mimic real changes (Chapter 4). We identified 18 categories of commits, 5 of which had already been mimicked by existing GI techniques, and identified 4 new ways in which the GI process could be improved to more closely replicate these commits.

After this study, we made another attempt at applying GI to Android appli-

cations (Chapter 5). In this case we removed the need for emulators or Android devices by using a simulation-based testing library (Robolectric). With this simulation, we were able test new variants more quickly and were not restricted to only the applications which used the built-in test-framework. With this setup, we were able to improve the time taken to load new screens on apps, whilst navigating around, them by up to almost 30%.

Following this success, we extended this approach to improve multiple objectives simultaneously (Chapter 6). We experimented with 3 different MO algorithms and found that all were capable of finding improvements. We were able to improve memory consumption by up to 33%, and execution time by up to 35% in these apps. We were unable to reduce network usage with GI in this setting, we believe that this is due to the benchmarks used not offering much room for improvements and limitations in the kinds of changes that GI can make. We also found that SO found similar improvements to MO search when applied separately for each objective.

Finally, we attempted to refine GI specifically to improve network usage (Chapter 7). We did this by building a profiler to identify the most network-intensive methods and modifying the mutation operators to reflect the changes made by real developers as found in our mining study. This included the addition of a new operator which can wrap statements in if statements. However, unfortunately, we did not find any improvements.

8.2 Limitations & Future Work

There are still several large challenges in applying GI both generally to software and specifically to Android apps. In our approach, we are only able to evaluate a few hundred patches during an attempt at improvement. However, in the desktop domain, there are often thousands of patches evaluated [84]. Our approach is currently limited by the slow speed of Android testing and may be able to produce better improvements if it were faster and more of the fitness landscape could be explored.

The majority of previous work has only used simple operators, which only

remove, and move statements around in source code. In Chapter 6, we showed that a caching operator was able to find improvements by simply leveraging information about how many times a method is called in a particular method or class. By leveraging patterns of changes made by real developers, as proposed by Krauss et al. [154], we may be able to make edits which are more likely to find improvements. This could then reduce the size of the search space that needs to be explored, which was one of the main limitations found in this work.

Another potential future application of this work could be applying the MO techniques to single objective problems with multiple metrics. For example, improving execution time, but exploring the potential trade-offs that can be made between the time taken to launch the app and its speed once it has been initialized.

In this thesis, we explored improving multiple different objectives and found that we could find trade-offs between them. However, Mkaouer et al. [155] showed that MO-GI could find improvements and trade-offs between 8 different metrics for a single objective (code quality).

One limitation that is common to all automated patch generation techniques is having the patches integrated into code bases by the developers reviewing them. In Chapter 6, we submitted 6 pull requests with improving changes. They were all either ignored or rejected without explanation. Often, developers will reject bug fixes suggested by automated tools. For example, Bader et al [114]. deployed an automated repair tool at Meta, repairing bugs found by static analysis. Only 42 % of the patches were accepted by developers, in 9% of cases the developers actually created semantically equivalent versions of the patches which were suggested automatically. In 2022, Winter et. al. [156] deployed their tool Fixie at Bloomberg. They found that developers were reluctant to accept patches from automated tools. One suggestion for how to improve acceptance was to introduce changes at more relevant times.

With this feedback, we developed a system to give suggestions to developers when they are most relevant. We propose building a system that will take the suggestions made by Bloomberg's internal static analysis tool (RSAT) and suggest

them to developers when two conditions are met:

1. The developer has made a change to the code base, which results in one or more GitHub check failing.
2. The developer has made changes to the lines of code that the suggested patch modifies.

If these conditions are met, the developer will be looking to fix the failing check and will suggest a patch that will only affect the lines that they are interested in, thus not polluting pull requests with irrelevant suggestions.

This project is still ongoing at Bloomberg and thus is beyond the scope of this thesis. The project was recently improved by a team of Masters students, who extended the tool from a proof-of-concept, with many limitations on the types of PRs to which it could be applied to. The tool has been extended to be able to handle PRs with multiple commits and multiple suggested changes.

This is a promising area of research and hopefully in the future interesting results will be found based on the app and its deployment within Bloomberg.

For Android apps specifically, applications mostly have very small or no test suites which reduces the applicability of GI [99]. This is one of the main challenges in evaluating the techniques discussed in this thesis. This problem could be mitigated if more automated test generation tools were created that could produce tests for Android apps. Currently, much of the literature is focused on exercising the UI of applications to induce and expose crashes. However, GI may introduce bugs that do not introduce crashes, e.g. the patch in Chapter 5 which deletes text from the screen, making these techniques unsuitable. They must also be run for relatively long periods. This would be unsuitable for GI as hundreds of patches are evaluated. Instead, unit tests which both exercise the app's code and check that the app's state after execution is correct are more suitable.

8.3 Summary

In conclusion, we have shown that Genetic improvement can be successfully used to improve the nonfunctional properties of Android applications in both single and multi-objective settings. This is despite the differences between Android applications and the desktop software in which GI has been applied in the past. We have explored and overcome many of the challenges that are faced when improving Android applications with fully automated techniques. This includes automatically modifying and testing apps, measuring an app's non-functional properties, and making changes that more closely resemble those made by developers. However, there are still several challenges that must be further mitigated in the future if this technique is to be widely adopted. These challenges are primarily making changes that developers will actually accept, and speeding up the process of Genetic Improvement. To aid in this endeavor, we provide all of our results and tools as fully open source with permissive licenses to encourage future work in this direction in the hopes that the future of Android applications will be more performant. All publications, reproduction repositories, and tools produced in the completion of this thesis can be found at `https://solar.cs.ucl.ac.uk/os/android.html`

Appendix A

Classifier Training

In order to build an accurate classifier for NFP-improving commits, we investigated several different options as detailed in this Appendix. The source code and the results obtained are provided in our online repository (`https://github.com/propMiner/propMining`). For clarity, we compare three different classes. *unknown* are commits that have not been analysed, and were excluded by the keyword search. *irrelevant* commits were identified via keyword search but deemed not relevant towards non-functional properties by one or more of the examiners. Finally, *relevant* commits encompass all commits identified to deal with a non-functional property after a keyword search.

We initially attempted to generate the classifier for all four subclasses of relevant commits, execution time, memory, bandwidth, and frame rate. This yielded no satisfactory results as the class's bandwidth and framerate had no recall (not a single commit) and the other two groups yielded less than 0.1 recall (i.e., less than 1 in 10 commits found).

We also attempted to balance the data. This was done in two ways. The first was to balance the *irrelevant* commits (3,132) with the *relevant* class (229) as the drastic difference lets classifiers overfit towards the *irrelevant* class. The balancing yields large margins in the test and training sets (accuracy 0.8, with the same recall in both classes). However, when attempting to reproduce this on the entire dataset the precision in the relevant classes dropped to 0.07. In this case, the classifier misses 20% of commits but yields no advantage over keyword search as many

commits that are not relevant are identified as such. The second attempt was balancing the classes of commits to reduce the under-representation of bandwidth and frame rate (see Table 4.6). This reduced the recall to 0.02 meaning that the classifier finds only 2 in 100 commits. We also investigated algorithms specifically targeted towards unbalanced data, to no avail. All of the balancing was done by keeping all commits of the respective smallest class, and randomly removing commits from other classes until they were the same size.

Since balancing the data worsens the precision drastically, we continued training classifiers with the datasets as they were recorded (i.e., without balancing). This investigation was conducted with all combinations of text preprocessing and featurisation. In all cases we did a preliminary tokenisation and stop word removal of all commit messages. For string processing, we used:

- Lemmatisation - reducing word inflections to a word root
- Stemming - removing word endings to approximate word roots

For the featurisation of the remaining word roots we used:

- TF/IDF - term frequency / inverse document frequency essentially ranking words in the source
- Bag of words - counting words in the commit messages
- Improved bag of words - the adaption was that we reduce to words that we identified as discriminative between the *relevant* and *irrelevant* sets. E.g. when a word occurs more often in one or the other it is included, otherwise it is not used in bag of words.

Table A.1 shows the best classifier found with this. Quadratic Discriminant Analysis yields a recall of 0.68 meaning two in three commits are identified, and a precision of 0.36 meaning that about three commits have to be consulted manually to yield one relevant commit. This is much better than keyword search (13,67 commits on average) but still represents a loss of information. The best results were yielded by using Lemmatisation as a preprocessing step and an improved bag of words to create the feature vectors.

Quadratic Discriminant Analysis was the best algorithm found out of the fol-

Table A.1: Quadratic Discriminant Analysis of NFP finding two in three commits

	Precision	Recall	F1-score
Relevant	0.36	0.68	0.47
Irrelevant	0.97	0.90	0.94

lowing twenty algorithms:

- Multi-layer Perceptron Classifier
- C-Support Vector Classification
- Linear C-Support Vector Classification
- Gaussian Process Classifier
- Decision Tree classifier
- Extra Tree Classifier
- Random Forest Classifier
- Ada Boost Classifier
- Bagging Classifier
- Gaussian Native Bayes
- Multinomial Native Bayes
- Bernoulli Native Bayes
- Complement Native Bayes
- Quadratic Discriminant Analysis
- Linear Discriminant Analysis
- Stochastic Gradient Descent Classifier
- Ridge Classifier
- Passive Aggressive Classifier

We chose Quadratic Discriminant Analysis as it supports the highest recall in the relevant category while still being a reasonable filter. Multinomial Native Bayes has the highest accuracy overall (0.95) but only a recall of 0.32, meaning two out of three *relevant* commits are lost. Quadratic Discriminant Analysis is highly dependent on the feature selection, as selecting regular bag of words actually produces a recall of 1.0 for relevant commits (no loss), but with a precision of 0.09, meaning that it requires an average of 11 commits to find one relevant one. This is

similar to the results that the keyword search itself produces. Further information is captured in the GitHub repository.

In an attempt to improve the quality of our classifier, we then added the dataset from [9], which are the results discussed in the main body of this work. The results of this analysis are available in our online repository consisting of csv tables with 10 runs of a random training/test split and a final line with the averages over all runs. All of the Classifiers using TF/IDF are outperformed by classifiers using other featurisation methods. In a similar vein, lemmatisation outperformed stemming. In general our improved bag of words slightly outperforms bag of words. The exception is the best classifier Decision Tree which performs best when using TF/IDF with stemming. It achieves a recall of 0.80 (i.e. 4 out of 5 commits) with a precision of 0.72 meaning that about 3 in 4 commits will be found.

A.1 Algorithm Changes in Commits

In this Appendix, we describe the commits in the "other" subcategory in the "Different algorithm" category, found in Section 4.2.2, in more detail. We First describe those from the KM set, then those from the CM set. For each commit, we report its SHA and a description of the changes made in it.

A.1.1 KM commits

- **c9afd823e8da9393a167a89345301782ef3483b**: Change from depth first search to depth last.
- **f3974898af9299632ea9354accb61e12393308c2**: Use a look up table.
- **b0e4f59f43984867c123fbffb4f345d5cc5f3814**: Change if else statement to a switch statement.
- **85ead3bd940bd445bbaff6a4a4d30ebca6b7c7a6**: Use the spread operator.
- **22904667b8e79167339972d4682024d95cd3d169**: Use lazy initialisation.

A.1.2 CM commits

- **fc82441c9aa412be6c1448b99a34607ff98e551d**: Use the SharedPreferencesCompat apply method.
- **588c35967f3dd9c2d27bb8739c46922a9b7a1c24**: Call free methods of class

components in their free methods, rather than in the onClose method of the class.

- **1cc32362aad23dfe5d508776274e643a307a3577**: Use lzma2 compression.
- **8bc2b8d5f53b04bbaea49a52eab260e02684376e**: Use String.format.
- **adbcdddb5625d7cd49b80d5c560eb8998446183a**: Faster text rendering method, blitFrom instead of addBlitFrom.
- **1aedaa5c28fc6a3cae76c34ac03d808d5860aa1c**: A faster algorithm for calculating the position of elements on the screen.
- **a9bc9ed71a84d68bf3e1652550251320c0a38cd8**: Use getCount, rather than iterating over the whole cursor.
- **d9c393f20e4d869628d8e3531af4e63a4e7b851b**: Use Androidx workManager begin to en-queue jobs.
- **5731360f8a894e49c5383857b629e450af9fa29d**: change DOM to SAX for UML parsing.
- **893909146d6e406ca36c5f2d95b82526a4ed6167**: Use raw input (effects rooted devices only).
- **6d1fa0cf056289457c3ca7616861a3dd362caea7**: More efficient average calculation.

Bibliography

[1] Simon Kemp. Digital 2022: Mobile duopoly consolidates its grip - datareportal – global digital insights, 2022. https://datareportal.com/reports/digital-2022-mobile-duopoly-consolidates-grip.

[2] Soo Ling Lim, Peter J. Bentley, Natalie Kanakam, Fuyuki Ishikawa, and Shinichi Honiden. Investigating country differences in mobile app user behavior and challenges for software engineering. *IEEE Trans. Software Eng.*, 41(1):40–64, 2015.

[3] Venkata N. Inukollu, Divya D. Keshamoni, Taeghyun Kang, and Manikanta Inukollu. Factors influencing quality of mobile apps: Role of mobile app development life cycle. *CoRR*, abs/1410.4537, 2014.

[4] Abhijeet Banerjee and Abhik Roychoudhury. Automated re-factoring of android apps to enhance energy-efficiency. In *MOBILESoft*, pages 139–150. ACM, 2016.

[5] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. What do mobile app users complain about? *IEEE Softw.*, 32(3):70–77, 2015.

[6] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024. ACM, 2014.

[7] Cuiyun Gao, Jichuan Zeng, Federica Sarro, Michael R. Lyu, and Irwin King. Exploring the effects of ad schemes on the performance cost of mobile phones. In *A-Mobile@ASE*, pages 13–18. ACM, 2018.

[8] Cuiyun Gao, Jichuan Zeng, Federica Sarro, David Lo, Irwin King, and Michael R. Lyu. Do users care about ad's performance costs? exploring the effects of the performance costs of in-app ads on user experience. *Inf. Softw. Technol.*, 132:106471, 2021.

[9] Alejandro Mazuera-Rozo, Catia Trubiani, Mario Linares-Vásquez, and Gabriele Bavota. Investigating types and survivability of performance bugs in mobile apps. *Empir. Softw. Eng.*, 25(3):1644–1686, 2020.

[10] buildfire. Mobile app download statistics & usage statistics (2023), 2023. `https://buildfire.com/app-statistics/#:~:text=There%20are%202.87%20million%20apps,on%20the%20Google%20Play%20Store.`

[11] Android Development Team. Android testing guide, 2022. https://developer.android.com/training/testing/fundamentals.

[12] Ui/application exerciser monkey : Android developers. https://developer.android.com/studio/test/monkey.

[13] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A GUI crawling-based technique for android mobile application testing. In *ICST Workshops*, pages 252–261. IEEE Computer Society, 2011.

[14] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, pages 641–660. ACM, 2013.

[15] Wontae Choi. *Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning*. PhD thesis, University of California, Berkeley, USA, 2017.

[16] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *FASE*, volume 7793 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2013.

[17] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *SIGSOFT FSE*, pages 599–609. ACM, 2014.

[18] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *ISSTA*, pages 94–105. ACM, 2016.

[19] Michael Kerrisk. Linux time. https://man7.org/linux/man-pages/man1/time.1.html, 2019. Last accessed: February 10, 2023.

[20] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. A survey of performance optimization for mobile applications. *IEEE Trans. Software Eng.*, 48(8):2879–2904, 2022.

[21] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. A quantitative and qualitative investigation of performance-related commits in android apps. In *ICSME*, pages 443–447. IEEE Computer Society, 2016.

[22] Raffaele Montella, Sokol Kosta, David Oro, Javier Vera, Carles Fernández, Carlo Palmieri, Diana Di Luccio, Giulio Giunta, Marco Lapegna, and Giuliano Laccetti. Accelerating linux and android applications on low-power devices through remote GPGPU offloading. *Concurr. Comput. Pract. Exp.*, 29(24), 2017.

[23] Min Chen and Yixue Hao. Task offloading for mobile edge computing in software defined ultra-dense network. *IEEE J. Sel. Areas Commun.*, 36(3):587–597, 2018.

[24] Young-Woo Kwon and Eli Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM*, pages 170–179. IEEE Computer Society, 2013.

[25] Mark S. Gordon, David Ke Hong, Peter M. Chen, Jason Flinn, Scott A. Mahlke, and Zhuoqing Morley Mao. Tango: Accelerating mobile applications through flip-flop replication. *GetMobile Mob. Comput. Commun.*, 19(3):10–13, 2015.

[26] Mark S. Gordon, Davoud Anoushe Jamshidi, Scott A. Mahlke, Zhuoqing Morley Mao, and Xu Chen. COMET: code offload by migrating execution transparently. In *OSDI*, pages 93–106. USENIX Association, 2012.

[27] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri E. Bal. Cuckoo: A computation offloading framework for smartphones. In *MobiCASE*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–79. Springer, 2010.

[28] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *EuroSys*, pages 301–314. ACM, 2011.

[29] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM*, pages 945–953. IEEE, 2012.

[30] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K. Nurminen, Matti Kemppainen, and Pan Hui. Can offloading save energy for popular apps? In *MobiArch@MobiCom*, pages 3–10. ACM, 2012.

[31] Aaron Yi Ding, Bo Han, Yu Xiao, Pan Hui, Aravind Srinivasan, Markku Kojo, and Sasu Tarkoma. Enabling energy-aware collaborative mobile data offloading for smartphones. In *SECON*, pages 487–495. IEEE, 2013.

[32] Ayat Khairy, Hany H. Ammar, and Reem Bahgat. Smartphone energizer: Extending smartphone's battery life with smart offloading. In *IWCMC*, pages 329–336. IEEE, 2013.

[33] Florian Berg, Frank Dürr, and Kurt Rothermel. Increasing the efficiency and responsiveness of mobile applications with preemptable code offloading. In *IEEE MS*, pages 76–83. IEEE Computer Society, 2014.

[34] Soomin Ki, Gyuri Byun, Kyungwoon Cho, and Hyokyung Bahn. Co-optimizing CPU voltage, memory placement, and task offloading for energy-efficient mobile systems. *IEEE Internet Things J.*, 10(10):9177–9192, 2023.

[35] Paul Baumann and Silvia Santini. Every byte counts: Selective prefetching for mobile applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(2):6:1–6:29, 2017.

[36] Brett D. Higgins, Jason Flinn, Thomas J. Giuli, Brian Noble, Christopher Peplin, and David Watson. Informed mobile prefetching. In *MobiSys*, pages 155–168. ACM, 2012.

[37] Prashanth Mohan, Suman Nath, and Oriana Riva. Prefetching mobile ads: can advertising systems afford it? In *EuroSys*, pages 267–280. ACM, 2013.

[38] Xiaomeng Chen, Abhilash Jindal, and Y. Charlie Hu. How much energy can we save from prefetching ads?: energy drain analysis of top 100 apps. In *HotPower@SOSP*, pages 3:1–3:5. ACM, 2013.

[39] Yixue Zhao, Marcelo Schmitt Laser, Yingjun Lyu, and Nenad Medvidovic. Leveraging program analysis to reduce user-perceived latency in mobile applications. In *ICSE*, pages 176–186. ACM, 2018.

[40] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. Inspection and characterization of app file usage in mobile devices. *ACM Trans. Storage*, 16(4):25:1–25:25, 2020.

[41] Han-Yi Lin, Pi-Cheng Hsiu, and Tei-Wei Kuo. Shiftmask: Dynamic OLED power shifting based on visual acuity for interactive mobile applications. In *ISLPED*, pages 1–6. IEEE, 2017.

[42] Xiang Chen, Kent W. Nixon, Hucheng Zhou, Yunxin Liu, and Yiran Chen. Fingershadow: An OLED power optimization based on smartphone touch interactions. In *HotPower*. USENIX Association, 2014.

[43] Han-Yi Lin, Chia-Chun Hung, Pi-Cheng Hsiu, and Tei-Wei Kuo. Duet: an OLED & GPU co-management scheme for dynamic resolution adaptation. In *DAC*, pages 126:1–126:6. ACM, 2018.

[44] Chun-Han Lin, Chih-Kai Kang, and Pi-Cheng Hsiu. Catch your attention: Quality-retaining power saving on mobile OLED displays. In *DAC*, pages 42:1–42:6. ACM, 2014.

[45] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Multi-objective optimization of energy consumption of guis in android apps. *ACM Trans. Softw. Eng. Methodol.*, 27(3):14:1–14:47, 2018.

[46] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. Making web applications more energy efficient for OLED smartphones. In *ICSE*, pages 527–538. ACM, 2014.

[47] Bhojan Anand, Karthik Thirugnanam, Jeena Sebastian, Pravein G. Kannan, Akkihebbal L. Ananda, Mun Choon Chan, and Rajesh Krishna Balan. Adaptive display power management for mobile games. In *MobiSys*, pages 57–70. ACM, 2011.

[48] Haidong Chen, Ji Wang, Weifeng Chen, Huamin Qu, and Wei Chen. An image-space energy-saving visualization scheme for OLED displays. *Comput. Graph.*, 38:61–68, 2014.

[49] Karthik Rao, Jun Wang, Sudhakar Yalamanchili, Yorai Wardi, and Handong Ye. Application-specific performance-aware energy optimization on android mobile devices. In *HPCA*, pages 169–180. IEEE Computer Society, 2017.

[50] Andrew J. Pyles, Zhen Ren, Gang Zhou, and Xue Liu. Sifi: exploiting voip silence for wifi energy savings insmart phones. In *UbiComp*, pages 325–334. ACM, 2011.

[51] Hyukjoong Kim and Dongkun Shin. Optimizing storage performance of android smartphone. In *ICUIMC*, page 95. ACM, 2013.

[52] Hyeong-Jun Kim and Jin-Soo Kim. Tuning the ext4 filesystem performance for android-based smartphones. In *ICFCE*, volume 133 of *Advances in Intelligent and Soft Computing*, pages 745–752. Springer, 2011.

[53] Yu Lin, Semih Okur, and Danny Dig. Study and refactoring of android asynchronous programming (T). In *ASE*, pages 224–235. IEEE Computer Society, 2015.

[54] Yingjun Lyu, Ding Li, and William G. J. Halfond. Remove rats from your code: automated optimization of resource inefficient database writes for mobile applications. In *ISSTA*, pages 310–321. ACM, 2018.

[55] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Getting the most from map data structures in android. *Empir. Softw. Eng.*, 23(5):2829–2864, 2018.

[56] Jürgen Cito, Julia Rubin, Phillip Stanley-Marbell, and Martin C. Rinard. Battery-aware transformations in mobile applications. In *ASE*, pages 702–707. ACM, 2016.

[57] Ding Li, Yingjun Lyu, Jiaping Gui, and William G. J. Halfond. Automated energy optimization of HTTP requests for mobile applications. In *ICSE*, pages 249–260. ACM, 2016.

[58] Abhijeet Banerjee and Abhik Roychoudhury. Future of mobile software for smartphones and drones: Energy and performance. In *MOBILESoft@ICSE*, pages 1–12. IEEE, 2017.

[59] Luis Cruz, Rui Abreu, and Jean-Noel Rouvignac. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In *MOBILESoft@ICSE*, pages 205–206. IEEE, 2017.

[60] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. EARMO: an energy-aware refactoring approach for mobile apps. *IEEE Trans. Software Eng.*, 44(12):1176–1206, 2018.

[61] Mahmoud A. Bokhari, Bobby R. Bruce, Bradley Alexander, and Markus Wagner. Deep parameter optimisation on android smartphones for energy minimisation: a tale of woe and a proof-of-concept. In *GECCO (Companion)*, pages 1501–1508. ACM, 2017.

[62] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *GECCO*, pages 1375–1382. ACM, 2015.

[63] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David Robert White, and John R. Woodward. Genetic improvement of software: A comprehensive survey. *IEEE Trans. Evol. Comput.*, 22(3):415–432, 2018.

[64] Justyna Petke, William B. Langdon, and Mark Harman. Applying genetic improvement to minisat. In *SSBSE*, volume 8084 of *Lecture Notes in Computer Science*, pages 257–262. Springer, 2013.

[65] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *GECCO*, pages 1427–1434. ACM, 2011.

[66] Eric M. Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *ASPLOS*, pages 317–328. ACM, 2013.

[67] Eric M. Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *ASE*, pages 313–316. ACM, 2010.

[68] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *SIGSOFT FSE*, pages 306–317. ACM, 2014.

[69] Alexander E. I. Brownlee, Justyna Petke, and Anna F. Rasburn. Injecting shortcuts for faster running java code. In *CEC*, pages 1–8. IEEE, 2020.

[70] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. Darwinian data structure selection. In *ESEC/SIGSOFT FSE*, pages 118–128. ACM, 2018.

[71] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. Object-oriented genetic improvement for improved energy consumption in google guava. In *SSBSE*, volume 9275 of *Lecture Notes in Computer Science*, pages 255–261. Springer, 2015.

[72] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.

[73] Aymeric Blot and Justyna Petke. Empirical comparison of search heuristics for genetic improvement of software. *IEEE Trans. Evol. Comput.*, 25(5):1001–1011, 2021.

[74] Anne Brindle. *Genetic algorithms for function optimization*. PhD thesis, University of Alberta, Canada, 1980.

[75] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *GECCO*, pages 959–966. ACM, 2012.

[76] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168. IEEE, 2008.

[77] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *EuroGP*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149. Springer, 2014.

[78] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *ISSTA*, pages 257–269. ACM, 2015.

[79] Mahmoud Al Najar, Rafael Almar, Erwin W. J. Bergsma, Jean-Marc Delvit, and Dennis G. Wilson. Genetic improvement of shoreline evolution forecasting models. In *GECCO Companion*, pages 1916–1923. ACM, 2022.

[80] Erik M. Fredericks and Byron DeVries. (genetically) improving novelty in procedural story generation. In *2021 IEEE/ACM International Workshop on Genetic Improvement (GI)*, pages 39–40, 2021.

[81] George O'Brien and John A. Clark. Using genetic improvement to retarget quantum software on differing hardware. In *2021 IEEE/ACM International Workshop on Genetic Improvement (GI)*, pages 31–38, 2021.

[82] William B. Langdon, David Robert White, Mark Harman, Yue Jia, and Justyna Petke. Api-constrained genetic improvement. In *SSBSE*, volume 9962 of *Lecture Notes in Computer Science*, pages 224–230, 2016.

[83] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Trans. Graph.*, 30(6):152, 2011.

[84] William B. Langdon. Performance of genetic programming optimised bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. *BioData Min.*, 8:1, 2015.

[85] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3d medical image registration CUDA software with genetic programming. In *GECCO*, pages 951–958. ACM, 2014.

[86] P. Walsh and C. Ryan. Automatic conversion of programs from serial to parallel using genetic programming - the paragen system. In *PARCO*, volume 11 of *Advances in Parallel Computing*, pages 415–422. Elsevier, 1995.

[87] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *GECCO*, pages 1327–1334. ACM, 2015.

[88] William B. Langdon, Afnan A. Al-Subaihin, Aymeric Blot, and David Clark. Genetic improvement of LLVM intermediate representation. In *EuroGP*, volume 13986 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2023.

[89] James Zhong, Max Hort, and Federica Sarro. Py2cy: a genetic improvement tool to speed up python. In *GECCO Companion*, pages 1950–1955. ACM, 2022.

[90] David Robert White. *Genetic programming for low-resource systems*. PhD thesis, University of York, UK, 2009.

[91] David Robert White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Trans. Evol. Comput.*, 15(4):515–538, 2011.

[92] Eric M. Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genet. Program. Evolvable Mach.*, 15(3):281–312, 2014.

[93] Alberto Carbognin, Leonardo Lucio Custode, and Giovanni Iacca. Genetic improvement of TCP congestion avoidance. In *BIOMA*, volume 13627 of *Lecture Notes in Computer Science*, pages 114–126. Springer, 2022.

[94] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of android code smells. In *MOBILESoft*, pages 59–69. ACM, 2016.

[95] Moo-Ryong Ra, Anmol Sheth, Lily B. Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *MobiSys*, pages 43–56. ACM, 2011.

[96] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David Robert White. Gin: genetic improvement research made easy. In *GECCO*, pages 985–993. ACM, 2019.

[97] Jacoco. https://docs.gradle.org/current/userguide/jacoco_plugin.html.

[98] Espresso for UI testing. https://developer.android.com/training/testing/espresso/.

[99] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. Testing of mobile applications in the wild: A large-scale empirical study on android apps. In *ICPC*, pages 296–307. ACM, 2020.

[100] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*, pages 416–419. ACM, 2011.

[101] Mark Harman and Bryan F. Jones. Search-based software engineering. *Inf. Softw. Technol.*, 43(14):833–839, 2001.

[102] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88. ACM, 2012.

[103] Irineu Moura, Gustavo Pinto, Felipe Ebert, and Fernando Castor. Mining energy-aware commits. In *MSR*, pages 56–67. IEEE Computer Society, 2015.

[104] Mario Linares Vásquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In *ICSME*, pages 352–361. IEEE Computer Society, 2015.

[105] Yiqun Chen, Stefan Winter, and Neeraj Suri. Inferring performance bug patterns from developer commits. In *ISSRE*, pages 70–81. IEEE, 2019.

[106] Xiang Chen, Chunyang Chen, Dun Zhang, and Zhenchang Xing. Sethesaurus: Wordnet in software engineering. *IEEE Trans. Software Eng.*, 47(9):1960–1979, 2021.

[107] Kenji Yamauchi, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Clustering commits for understanding the intents of implementation. In *ICSME*, pages 406–410. IEEE Computer Society, 2014.

[108] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[109] Karl Pearson. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London Series I*, 58:240–242, 1895.

[110] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *GECCO (Companion)*, pages 1513–1520. ACM, 2017.

[111] Bobby R. Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. Approximate oracles and synergy in software energy search spaces. *IEEE Trans. Software Eng.*, 45(11):1150–1169, 2019.

[112] Fan Long, Peter Amidon, and Martin C. Rinard. Automatic inference of code transforms for patch generation. In *ESEC/SIGSOFT FSE*, pages 727–739. ACM, 2017.

[113] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811. IEEE Computer Society, 2013.

[114] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019.

[115] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *SSBSE*, volume 11036 of *Lecture Notes in Computer Science*, pages 65–86. Springer, 2018.

[116] Justyna Petke. New operators for non-functional genetic improvement. In *GECCO (Companion)*, pages 1541–1542. ACM, 2017.

[117] Mark Harman and S. Afshin Mansouri. Search based software engineering: Introduction to the special issue of the IEEE transactions on software engineering. *IEEE Trans. Software Eng.*, 36(6):737–741, 2010.

[118] Robolectric, 2019. http://robolectric.org/.

[119] F-droid - free and open source android app repository. https://f-droid.org/en/.

[120] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *ICSE (Companion Volume)*, pages 23–26. IEEE Computer Society, 2017.

[121] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50 – 60, 1947.

[122] David Freedman, Robert Pisani, and Roger Purves. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.

[123] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. Enhancing genetic improvement of software with regression test selection. In *ICSE*, pages 1323–1333. IEEE, 2021.

[124] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra B. Cohen, and Justyna Petke. Keeping secrets: Multi-objective genetic improvement for detecting and reducing information leakage. In *ASE*, pages 61:1–61:12. ACM, 2022.

[125] N. Srinivas and Kalyanmoy Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evol. Comput.*, 2(3):221–248, 1994.

[126] James Callan and Justyna Petke. Multi-objective genetic improvement: A case study with evosuite. In *SSBSE*, volume 13711 of *Lecture Notes in Computer Science*, pages 111–117. Springer, 2022.

[127] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In *PPSN*, volume 1917 of *Lecture Notes in Computer Science*, pages 849–858. Springer, 2000.

[128] Mifa Kim, Tomoyuki Hiroyasu, Mitsunori Miki, and Shinya Watanabe. SPEA2+: improving the performance of the strength pareto evolutionary algorithm 2. In *PPSN*, volume 3242 of *Lecture Notes in Computer Science*, pages 742–751. Springer, 2004.

[129] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. *IEEE Trans. Evol. Comput.*, 18(4):577–601, 2014.

[130] Android Development Team. Android context, 2022. https://developer.android.com/reference/android/content/Context.

[131] James Callan and Justyna Petke. Improving android app responsiveness through automated frame rate reduction. In *SSBSE*, volume 12914 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2021.

[132] Android Development Team. Android compilation guide, 2023. https://developer.android.com/studio/build.

[133] Niraj Tolia, David G. Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients. *Computer*, 39(3):46–52, 2006.

[134] James Callan and Justyna Petke. Improving responsiveness of android activity navigation via genetic improvement. In *ICSE-Companion*, pages 356–357. ACM/IEEE, 2022.

[135] James Callan, Oliver Krauss, Justyna Petke, and Federica Sarro. How do android developers improve non-functional properties of software? *Empir. Softw. Eng.*, 27(5):113, 2022.

[136] Shengjie Zuo, Aymeric Blot, and Justyna Petke. Evaluation of genetic improvement tools for improvement of non-functional properties of software. In *GECCO Companion*, pages 1956–1965. ACM, 2022.

[137] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. Pyggi 2.0: language independent genetic improvement framework. In *ESEC/SIGSOFT FSE*, pages 1100–1104. ACM, 2019.

[138] Oracle Development Team. Javaruntime, 2020. https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html.

[139] Michael Kerrisk. Linux process tracking. https://man7.org/linux/man-pages/man5/proc.5.html, 2022. Last accessed: February 10, 2023.

[140] Michael Auer, Felix Adler, and Gordon Fraser. Improving search-based android test generation using surrogate models. In *SSBSE*, volume 13711 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2022.

[141] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, and Porfirio Tramontana. Agrippin: a novel search based testing technique for android applications. In *DeMobile@SIGSOFT FSE*, pages 5–12. ACM, 2015.

[142] Young Min Baek and Doo-Hwan Bae. Automated model-based android GUI testing using multi-level GUI comparison criteria. In *ASE*, pages 238–249. ACM, 2016.

[143] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of android apps. In *ESEC/SIGSOFT FSE*, pages 245–256. ACM, 2017.

[144] Husam N. Yasin, Siti Hafizah Ab Hamid, and Raja Jamilah Raja Yusof. Droidbotx: Test case generation tool for android applications using q-learning. *Symmetry*, 13(2):310, 2021.

[145] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *ESEM*, pages 291–301. IEEE Computer Society, 2009.

[146] Thomas Bach, Artur Andrzejak, Ralf Pannemans, and David Lo. The impact of coverage on bug density in a large industrial software project. In *ESEM*, pages 307–313. IEEE Computer Society, 2017.

[147] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. Quality of automated program repair on real-world defects. *IEEE Trans. Software Eng.*, 48(2):637–661, 2022.

[148] Miqing Li, Tao Chen, and Xin Yao. How to evaluate solutions in pareto-based search-based software engineering: A critical review and methodological guidance. *IEEE Trans. Software Eng.*, 48(5):1771–1799, 2022.

[149] Ruihua Ji, Zhong Li, Shouyu Chen, Minxue Pan, Tian Zhang, Shaukat Ali, Tao Yue, and Xuandong Li. Uncovering unknown system behaviors in uncertain networks with model and search-based testing. In *ICST*, pages 204–214. IEEE Computer Society, 2018.

[150] Yuan Liu, Ningbo Zhu, and Miqing Li. Solving many-objective optimization problems by a pareto-based evolutionary algorithm with preprocessing and a penalty mechanism. *IEEE Trans. Cybern.*, 51(11):5585–5594, 2021.

[151] András Vargha and Harold 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(2):101–132, 2000.

[152] AppBrain. okhttp - android statistics, 2023. https://www.appbrain.com/stats/libraries/details/okhttp/okhttp.

[153] Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2023. arxiv, 2302.07867.

[154] Oliver Krauss. Amaru: a framework for combining genetic improvement with pattern mining. In *GECCO Companion*, pages 1930–1937. ACM, 2022.

[155] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empir. Softw. Eng.*, 21(6):2503–2545, 2016.

[156] Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Óskar Haraldsson, John R. Woodward, Serkan Kirbas, Etienne Windels, Olayori McBello, Abdurahman Atakishiyev, Kevin Kells, and Matthew W. Pagano. Towards developer-centered automatic program repair: findings from bloomberg. In *ESEC/SIGSOFT FSE*, pages 1578–1588. ACM, 2022.