Automated Software Engineering (2025) 32:2
https://doi.org/10.1007/s10515-024-00472-7

Multi-objective improvement of Android applications

James Callan[1] · **Justyna Petke**[1]

Received: 31 July 2024 / Accepted: 3 October 2024

Abstract
Non-functional properties, such as runtime or memory use, are important to mobile app users and developers, as they affect user experience. We propose a practical approach and the first open-source tool, GIDroid for multi-objective automated improvement of Android apps. In particular, we use Genetic Improvement, a search-based technique that navigates the space of software variants to find improved software. We use a simulation-based testing framework to greatly improve the speed of search. GIDroid contains three state-of-the-art multi-objective algorithms, and two new mutation operators, which cache the results of method calls. Genetic Improvement relies on testing to validate patches. Previous work showed that tests in open-source Android applications are scarce. We thus wrote tests for 21 versions of 7 Android apps, creating a new benchmark for performance improvements. We used GIDroid to improve versions of mobile apps where developers had previously found improvements to runtime, memory, and bandwidth use. Our technique automatically re-discovers 64% of existing improvements. We then applied our approach to current versions of software in which there were no known improvements. We were able to improve execution time by up to 35%, and memory use by up to 33% in these apps.

Keywords Android apps · Genetic improvement · Multi-objective optimization · Search-based software engineering

1 Introduction

Android applications (or apps for short) are one of the most widely used types of software (Kemp 2022). They are designed for direct user interaction, with the main

✉ Justyna Petke
j.petke@ucl.ac.uk

James Callan
james.callan.19@ucl.ac.uk

[1] Computer Science Department, University College London, Gower Street, London, Greater London WC1E 6BT, UK

Published online: 04 November 2024

entry point for the software being its UI components. Due to the small size of Android devices (phones, smartwatches, 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 and 59% of bad reviews were due to poor performance (Lim et al. 2014; Inukollu et al. 2014).

Hort et al. (2022)'s survey on Android performance optimizations lists several approaches for improving non-functional properties of Android apps. These include prefetching online resources to avoid having to wait for them when they are needed (Mohan et al. 2013; Baumann and Santini 2017) and offloading computation onto remote servers which are faster than the mobile device (Saarinen et al. 2012). Offloading, however, requires external server infrastructure to be set up and applications to be re-engineered to be utilised. Prefetching is only applicable to areas of applications that interact with the network. Other approaches (Hort et al. 2022) include anti-pattern detection, which requires manual implementation, and refactorings, which are limited to specific code fragments. We argue that an approach that does not require external resources and is more easily applicable to all applications regardless of type and structure would make developers more likely to adopt it.

Whilst existing approaches for automated improvement of Android apps are capable of improving multiple properties simultaneously, e.g., by removing unnecessary computation reducing runtime and energy use, in most cases such correlations have not been considered (Hort et al. 2022). Moreover, single-objective improvements can have negative effects on other properties. For example, during prefetching, the resource which is prefetched must be stored, which might result in higher memory use. To get the full picture of how an application is affected by an improvement, properties other than those that are direct targets for improvement should be considered. Hort et al. (2022) reveals only one work that applies multi-objective optimization to non-functional properties of Android apps. Morales et al. (2018) consider energy consumption and the number of anti-patterns. Although the authors release their framework, it is not open-source and requires external hardware for energy measurements.

Rather than targeting specific features or resources, we aim to find source code transformations. There have been a few attempts to find Android app performance optimizations with source code transformations so far. Lin et al. proposed two approaches, Asynchronizer (Lin et al. 2014) and AsyncDroid (Lin et al. 2015), for refactoring code to be executed asynchronously. However, both of these approaches require developers to identify the particular lines of code which they want to execute asynchronously and there has been no work to show the actual impact of these refactorings on performance. Lyu et al. (2018) propose an approach that moves costly database operations out of loops. Whilst this approach can improve performance, it is only applicable to methods that access databases inside loops.

The only tools for Android app performance improvement, which are both available and generally applicable to Android source code, are linters (Habchi et al. 2018). Linters contain rules which aim to identify areas of code that may cause performance issues, leaving to app developers the decisions to implement suggested changes. How-

ever, their use comes with challenges (Habchi et al. 2018), including dealing with false positives.

In order to find patches to source code, we propose to use Genetic Improvement. GI is a search-based technique that uses meta-heuristics to perform a guided search over software patches, to find those that improve a given software property. GI makes changes to source code and thus can be applied to a wide range of software types. GI has been used to improve many different properties of software, including runtime (Langdon et al. 2015; Petke et al. 2013), memory (Basios et al. 2017; Wu et al. 2015), and energy consumption (Bruce et al. 2015; Burles et al. 2015).

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 which 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 (Mesecan et al. 2022; Wu et al. 2015).

GI has been applied to Android applications a handful of times. Callan and Petke attempted to improve the frame rate of Android apps with GI, however, were unsuccessful (Callan and Petke 2021). In another work, Callan and Petke (2022a) were able to find improvements to the navigation responsiveness of Android apps. Bokhari et al. (2017) improved the energy consumption of Android apps, with a type of GI known as deep parameter optimization. To the best of our knowledge, no GI work so far has attempted to improve and find trade-offs between multiple properties of Android apps, and no approach has attempted to improve either the memory consumption or bandwidth usage of Android apps, which we target in this work.

Previous work on applying GI in the Android domain revealed several practical challenges: (1) 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 (2) tests themselves are scarce, and (3) performance fitness measurements used in the desktop domain are not accurate enough to witness performance issues in Android apps, yet users deem wait time of just 150 ms as 'laggy' (Tolia et al. 2006). We overcome these challenges. We utilise the Robolectric testing library (Robolectric Develop. Team 2023) which mimics UI behaviour, allowing for quick unit testing, without need for installation on an actual mobile or tablet device. This addresses challenge (1) of having to conduct constly runs on actual devices during GI. This simulation-based approach provides us with means of utilising performance measurement tools unavailable on Android devices, addressing challenge (3). When using GI, we validate the patches that we generate using the program's test suite and validate the best-improving final patches manually. This ensures that our patches do not disrupt the functionality of the program. However, most open-source Android applications do not have test suites (challenge (2)), and those that do are limited, achieving a median line coverage of 23% (Pecorelli et al. 2020). This meant that we had to create tests for all the benchmarks on which we ran GI.[1]

[1] At the time of our experiments, none of the automated test generation tools for Android were compatible with latest Android software, thus we had to manually create tests to evaluate our MO-GI approach.

In order to validate our proposed approach, we created a tool, GIDroid (2023), for running multi-objective (MO) GI on Android applications. We provide three fitness functions, to improve runtime, memory, and bandwidth use. GIDroid contains three MO algorithms (NSGA-II (Deb et al. 2000), NSGA-III (Deb and Jain 2014), and SPEA2 (Kim et al. 2004)). Based on work by Callan et al. (2022), who mined non-functional improvements made by Android developers, we implement in GIDroid two novel mutation operators, specifically designed to mimic human-made edits. These cache results of repeated calls, aiming to save memory use.

We selected Android apps that contain real-world non-functional-property-improving commits, in order to see if GIDroid can re-discover changes made by Android developers. Moreover, we use the latest versions of these applications, to see if we can find as-yet-undiscovered improvements. Overall, we created a benchmark of 21 versions of 7 Android apps, which we open source for future work.

GIDroid was able to find patches that improve execution time by up to 35%, and memory usage by up to 65%. Unfortunately, no improvements to bandwidth use were found. Such improvements are within GIDroid's search space, which leaves room for future work for more effective search strategies.

To sum up, we present the following novel contributions:

1. An open-source, simulation-based tool, GIDroid (GIDroid 2023), for automated multi-objective improvement of Android applications' runtime, memory, and bandwidth use.
2. A benchmark of 21 versions of 7 Android applications, including tests, for future work on performance improvement in the Android domain.
3. An evaluation of the effectiveness of 3 multi-objective Genetic Improvement algorithms at improving runtime, memory use, and bandwidth of Android applications. No GI work has targeted 3 properties before.
4. A comparison between both multi- and single-objective Genetic Improvement approaches for automated optimization of Android applications.
5. An empirical comparison of our multi-objective GI-based approach for Android application performance improvement with state-of-the-art linters.

The rest of this paper is structured as follows: Sect. 2 describes related work; Sect. 3 presents an introduction to Genetic Improvement and multi-objective optimization; Sect. 4 presents challenges of applying GI to the Android domain and our proposed framework that overcomes these challenges; Sect. 5 presents research questions we aim to answer to evaluate our approach, with Sect. 6 outlining our methodology; Sect. 7 presents our results, with threats to validity presented in Sect. 8; Sect. 9 concluding.

2 Android app performance optimization

A number of approaches have been proposed for improving the performance of Android applications. Hort et al. (2022)'s survey on this topic presents the following code-level approaches:

Prefetching: Network resources are fetched before they are needed by the application and stored locally (Mohan et al. 2013; Baumann and Santini 2017). When the application needs said resources, it can get them without having to wait for a lengthy network transaction, making the application more responsive. Prefetching can lead to increased memory and storage usage, and lead to the app not having the most up-to-date version of a particular resource. Prefetching can only optimize parts of applications that utilise network resources.

Anti-patterns: Approaches that detect patterns in source code that indicate performance defects, for example, repeated expensive memory access operations inside for-loops (Nistor et al. 2013), or incorrect wake lock usage affecting energy use (Cruz and Abreu 2017). The only tools which are both available and generally applicable to the source code of Android apps are linters (Android Development Team 2023b), PMD (PMD Development Team 2023), and FindBugs (FindBugs Development Team 2015). These tools have performance rules which aim to identify areas of code that may cause performance issues. However, often these warnings are false-positives (Habchi et al. 2018). The developer must then manually fix the issues. Existing Android linters do not provide any information on the impact of fixing the issues they detect.

Refactoring: Refactoring approaches aim to modify the source code of the application to be more performant. In Lin et al. (2014, 2015)'s work applications were refactored to execute code asynchronously, making them execute more quickly. Ayala et al. (2019) investigated three asynchronous communications methods' impact on mobile energy use. These approaches require developers to identify each line of code that they wish to execute asynchronously. Lyu et al. (2018) propose to move database operations out of loops. However, this is only applicable to limited areas of code that contain such database calls.

Offloading: This approach aims to perform the most costly computation on external servers, rather than Android devices (Das et al. 2016; Chun et al. 2011; Ding et al. 2013; Berg et al. 2014; Saarinen et al. 2012). This has the benefit of reducing the amount of energy used by the application, extending the device's battery life, and speeding up the computation to make the app more responsive. Offloading requires external hardware to function, which may not always be suitable.

Programming Languages: In the Android environment, a number of different programming languages are available to developers. The majority of Android apps are written in either Java or Kotlin, which usually compile to JVM bytecode. This bytecode is then (optionally) obfuscated and recompiled into dex code. This allows Java and Kotlin APIs to be used across both languages interchangeably and some applications even use a mixture of both languages. There is little performance difference between the two languages (Mateus et al. 2021). C/C++ can also be used to write native code. Such code is generally faster than the Java/Kotlin code and thus can be used to find performance improvements. However, changing a programming language can be time-consuming, with no upfront knowledge of the magnitude of possible performance gains.

The above works have all proved useful, but they either do not perform fully automatic improvement (Lin et al. 2014, 2015; Habchi et al. 2018; FindBugs Development Team 2015; Android Development Team 2023b), are only applicable to specific areas of code (Lyu et al. 2018; Mohan et al. 2013; Baumann and Santini 2017), or require external infrastructure (Das et al. 2016; Chun et al. 2011; Ding et al. 2013; Berg et al. 2014; Saarinen et al. 2012).

Given the shortcomings of the above-mentioned approaches, we propose to use multi-objective GI to improve several software properties in the Android domain. By using GI, we will be able to apply our approach to any source code and will not be limited to only improving code using certain patterns. GI is fully automated. Developers will only have to review the patches produced by GI once the process is finished to ensure that they do not have unintended side effects. Such patches would thus undergo a standard code review process. Furthermore, GI does not require the setup of any external infrastructure to achieve optimisations and can be performed in the local development environment of the application developer. We illustrate this in Table 1. We aim to find multiple patches, which may find trade-offs between different properties that can allow developers to choose the best patches for their particular needs, and be fully aware of the consequences that a particular patch will have on other properties. We note that prefetching, offloading, and others are complementary to GI, and could still offer benefits to applications that have been optimized using GI.

3 Background

Before we outline our proposed framework for automatic performance improvement of Android applications, we first provide a short introduction to Genetic Improvement (GI) and Multi-Objective (MO) optimization.

3.1 Genetic improvement

Genetic Improvement (GI) (Petke et al. 2018) is a search-based software engineering technique that utilises search to iterate over different versions of software in order to find improved program variants. These improvements can be bug repairs or improvements to non-functional properties like execution time or memory use. GI has already proven useful for improvement of traditional software, fixing bugs during the development of commercial software (Haraldsson et al. 2017), improving the execution time of large bioinformatics software (Langdon et al. 2015), improving compiler optimizations (Li et al. 2022), and more (Petke et al. 2018).

Each program in GI is represented as a patch to existing software. Patches are constructed from a set of edits to code, i.e., mutations, which describe modifications to the program being improved. The most common mutation operators used in previous work have been: DELETE, COPY, and REPLACE. These operations can be applied at the level of lines of source code, bytecode, or other. The vast majority of GI work operates at statement-level, applying mutation operators to nodes of an abstract syntax tree (AST).

Table 1 Comparison of existing strategies for improvement of non-functional properties of Android apps with our tool—GIDroid

Work	Properties	Source Code	Fully Automatic	Trade-offs Considered
Prefetching	Runtime	x	✓	x
Anti-patterns	Runtime, memory, energy use	✓	x	x
Refactoring-asynchronous	Runtime, energy use	✓	x	x
Refactoring-database loops	Runtime	✓	✓	x
Offloading	Runtime, energy use	x	✓	x
GIDroid	Runtime, memory, bandwidth	✓	✓	✓

Bold font indicates our approach, different to previous work

Each GI patch is applied to the original software for evaluation, measured using a fitness function. In the case of program repair, this fitness can be the number of passing tests, and for execution time improvement it could be the time taken by the tests. This fitness measurement is used to guide search through the landscape of patches to find improved software variants. Traditionally, genetic programming has been used for this purpose, though other search techniques, such as local search, have also proven effective (Blot and Petke 2021).

Although there is a lot of literature on the improvement of traditional software using GI, little is known about how the technique would fare in the mobile domain. Initial approaches have shown mixed results, with none trying to optimize multiple properties. Bokhari et al. (2017) were able to reduce the energy consumption of Android apps, using deep parameter optimization, i.e., mutating parameters within source code, not exposed to the user. Callan and Petke (2022a) were able to reduce the time taken to move between Activities, the main UI components, of Android apps. However, when attempting to improve the frame rate of Android apps, Callan and Petke did not find improving patches (Callan and Petke 2021).

3.2 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 (Srinivas and Deb 1994), 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. (2015) and Callan and Petke (2022b) used NSGA-II (Deb et al. 2000), White et al. (2011) used SPEA2 (Kim et al. 2004), with Mesecan et al. (2022) comparing four MO algorithms, with SPEA2 and NSGA-III (Deb and Jain 2014) 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 successful in previous work.

4 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 (Android Development Team 2022a), 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 (Callan and Petke 2021).

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 (Android Development Team 2022b, 2023a).

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 app functionalities or UI interatcions that take more than 150 ms are considered to be 'laggy' by users (Tolia et al. 2006). 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.

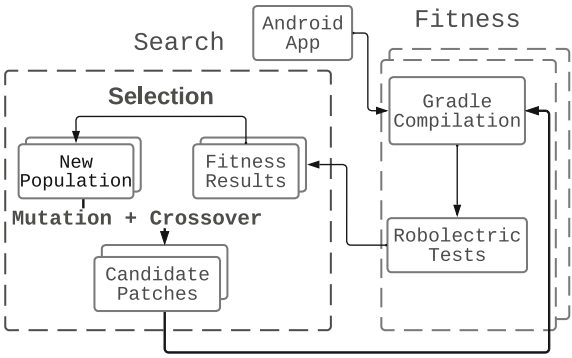

Fig. 1 GI framework for Android app improvement, with search based on a genetic algorithm. In the case of local search, only mutation is applied

In order to overcome the aforementioned challenges, we propose the GI framework shown in Fig. 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.

4.1 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 proven successful (Petke et al. 2018). We use a list of edits, rather than 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 Listing 1.

Listing 1 An example of a program variant that deletes the statement with ID 608 and then copies the statement with ID 1307 to position 320 into the block with ID 365 in the file Example.java. The pipe symbol separates individual edits.

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

4.2 Fitness

In the Fitness section in our framework (see Fig. 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 which 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 and Petke (2022a) 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. (2015) 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 (Bokhari et al. 2017; Morales et al. 2018) found energy estimates to be too noisy, thus requiring external devices for physical energy measurements. Although GI can be used to optimize energy consumption (Bokhari et al. 2017), 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 (Mohan et al. 2013). No previous attempts have been made to improve it using source-code transformations, despite such changes being made by developers (Callan et al. 2022).

4.3 Search

The Search section of our framework for Android app improvement (see Fig. 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.

4.3.1 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, insert, or replace another statement. Additionally, we investigated which other mutation operators might be beneficial in the Android domain.

Callan et al. (2022).'s work 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 (execution time, memory consumption, bandwidth use, and frame rate) 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 operations), caching is the most generically applicable strategy found, and thus, the most suitable for multi-objective improvement. Based on manual analysis of the commits from Callan et al.'s work, in which caching is used, we propose two new mutation operators. Caching could prove useful for the three properties which 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 propose a simple **In-Method Caching Operator**. 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 Fig. 2. 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 Fig. 3. 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

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
```

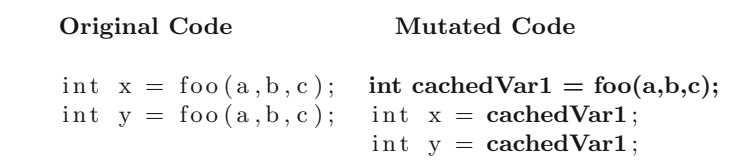

Fig. 2 An example of the In-Method Cache Operator. The resultant code stores the results of a method call foo, with parameters a, b and c. This stored result can then be used later in the same method

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
```

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.

5 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 Multi-Objective Genetic Improvement (MO-GI) optimize Android apps in the same way as real developers?

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;
  }
}
```

Fig. 3 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

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?

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 Single-Objective Genetic Improvement (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 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. Given the popularity of large language models (LLMs) to address a variety of software engineering tasks, we also ask ChatGPT[2] to find improvements on our benchmark set with known improvements.

6.1 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. (2022) found that PYGGI (An et al. 2019) and the Genetic Improvement In No time tool (Gin) (Brownlee et al. 2019) were the only GI tools that could be readily applied to new software, with more recent tool by Mesecan et al. (2022) not yet available. However, none of the aforementioned work can be run upon Android applications. Whilst Gin is compatible with most Java programs, and thus could potentially easiest to extend, it is not compatible with the Android compilation and testing environments.

In GIDroid, we implement three MO algorithms: NSGA-II (Deb et al. 2000) as it is one of the most widely used multi-objective algorithms; NSGA-III (Deb and Jain 2014), that was specifically developed for problems with 3 or more objectives in mind; and SPEA2 (Kim et al. 2004), which has recently proven successful for MO-GI in the desktop domain (Mesecan et al. 2022). 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 3.

To measure execution time we use Linux's time tool (Kerrisk 2019), we measure memory usage with the Java class Runtime's memory allocation tracking (Oracle Development Team 2020) and we use Linux's built-in process-level network traffic tracking (Kerrisk 2022) to measure bandwidth.

[2] https://openai.com/blog/chatgpt/.

6.2 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% (Pecorelli et al. 2020). It is worth pointing out, however, that testing is simply good practice in software development. Therefore, our approach will be more easily applicable to projects following this practice. 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 (Auer et al. 2022; Amalfitano et al. 2015; Azim and Neamtiu 2013; Baek and Bae 2016; Mahmood et al. 2014; Mao et al. 2016; Su et al. 2017; Li et al. 2017; Yasin et al. 2021) 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 (Mockus et al. 2009; Bach et al. 2017) 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. Callan et al. (2022) has recently conducted a study of the changes that Android developers make to improve app performance. They pose that some of those changes are within the GI search-space. For instance, moving an operation outside of a FOR loop, if only need 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 Callan et al. (2022)'s criteria to iteratively analyse the commits from their dataset that improve runtime, bandwidth, or memory use, until we reached 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. These are presented in Table 2.[3]

[3] Links to the apps are available in our repository: https://github.com/SOLAR-group/GIDroid/tree/main/Benchmark.

Table 2 Benchmarks: details of Android apps used in our study

Application version	Acronym	Commit	Type	LoC
PortAuthority 1	PA1	e0163e20d1a67c22c2f7ed0f0345206ce1a050f0	Port Scanner	4k
PortAuthority 2	PA2	e37a1a522a15773710f051d9fff5c0ce68ade5cb		16k
PortAuthority 3	PA3	3a1329297881aff069cdbc80c92de386ac952d77		5k
PortAuthority 4	PA4	adc73aac9c7dba5c61e1e18a96dfe7dd9712d100		16k
PortAuthority 5	PA5	3e6846b6a377c35780ddb49e21eeab5749381bf2		16k
PortAuthority 6	PA6	a02a0170a38ec257e1f390388e4b5d1414b3cf36		16k
PortAuthority Current	PAN	9dbc43ebe454195b1610eee9b7473a83d400d48b		6k
Tower Collector 1	TC1	956ea2213c1f7f012d6ab1388536a0c6d5202bd9	Location Collector	27k
Tower Collector 2	TC2	0632608d26667e3a1864bf436086cf9422a913cb		12k
Tower Collector Current	TCN	b069a973031823339bf62a8330086b8e9a1cda6e		
Gadgetbridge 1	GB1	c75362c5ea489247cc00b473a0ef91dbb1cc1569	SmartBand Software	106k
Gadgetbridge Current	GBN	305078f2535f5508c13b089bc68deeff7bf7b1cc		
Fosdem Companion 1	FS1	b79e29a67c29699b9b8d4ad9c09a3349ce32c59f	Schedule Browser	11k
Fosdem Companion Curr.	FSN	4d6914e2765712f86af02fec0538121d7dda197c		
Fdroid 1	FD1	e44ca193dd0adcbc5e240410aec4c681f5053dae	Repository System	77k
Fdroid 2	FD2	bf8aa30a576144524e83731a1bad20a1dab3f1bc		2k
Fdroid Current	FDN	bc6fba88fada1dcb186a40d0aead430bcc0031f8		
Lightning Browser 1	LB1	460da386ec10cb82b97bd2def2724fe41f709a88	Web Browser	69k
Lightning Browser Curr.	LBN	ca7da585bdfcfdd89f85bc2a03d6a62cec28220f		71k
Frozen Bubble 1	FB1	e9f6a51be9f7c4ad9f11d8712b06cb906e9ddf28	Game	36k
Frozen Bubble Current	FBN	c3ac715a03370389d0d649a0eb5b7b5b3005e8b8		40k

Table 3 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.)

Each edit represents one mutation operation (see Sect. 4.3.1)

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 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.

6.3 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 (Motwani et al. 2022; Callan and Petke 2022a). We set number of evaluations to 400 as, even when using simulation-based testing, the evaluation of an individual is slow, taking up to 2 min. We use the Genetic programming parameters in Table 3 as they have been used successfully in the past (Callan and Petke 2022b).

We had 2520 runs in total, taking a mean of 3 h per run, resulting in roughly 7500 h 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.

7 Results and discussion

In this section, we present and analyse the results of our experiments, answering our Research Questions (Sect. 5). 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 which we aim to minimize.

7.1 RQ1: known improvements

Figures 4 and 5 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[4] 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 applications. The result of this analysis can be found in Table 4. In 64% of benchmarks GIDroid is able to find patches containing edits semantically-equivalent to developer patches, providing at least the same percentage 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.

7.2 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 there are no known improvements.

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

Table 4 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.)

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

Each MO run was repeated 20 times

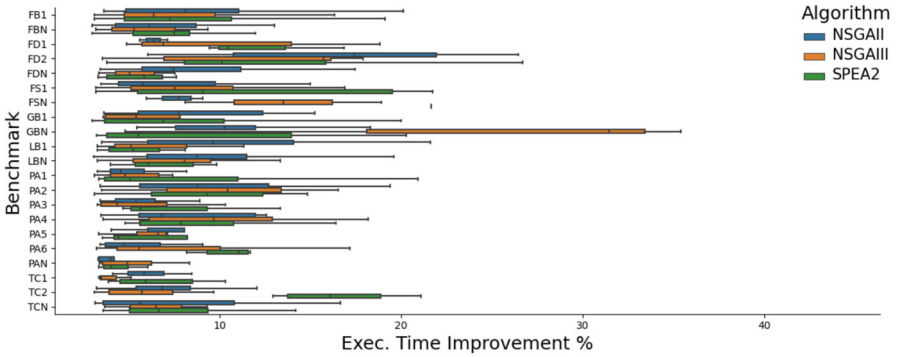

Fig. 4 Execution time improvements (%) achieved by patches generated by GIDroid using three MO algorithms on 21 versions of 7 Android apps (see Table 2). Each boxplot shows best patches from 20 runs per MO algorithm and benchmark

The performance of each algorithm on versions of software is shown in Figs. 4 and 5. We find improvements to the execution time of up to 35% (this patch improved memory use by 9% and had no effect on bandwidth usage) and to memory consumption of up to 32% (with a 3% increase in execution time and no effect on bandwidth usage). 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 (Figs. 6, 7).

We have compiled the best changes found by GIDroid in these experiments to demonstrate the capabilities of GIDroid.

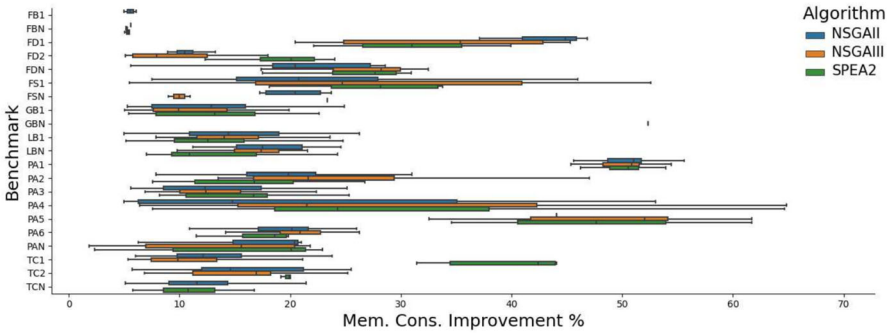

Fig. 5 Memory consumption improvements (%) achieved by patches generated by GIDroid using three MO algorithms on 21 versions of 7 Android apps (see Table 2). Each boxplot shows best patches from 20 runs per MO algorithm and benchmark

We have also issued pull request, however, these have not yet been actioned by the developers. We detail each of these patches below[5]

7.2.1 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. This improved execution time by 8% and memory usage by 24%.

Listing 2 Best patch for Port-Authority (PAN).

```
37,47d36
<
<             try {
<                 InetAddress address = InetAddress.getByName(ip);
<                 ip = address.getHostAddress();
<             } catch (UnknownHostException e) {
<                 activity.processFinish(false);
<                 activity.processFinish(e);
<
<                 return null;
<             }
<
```

[5] These can be found in our repository (GIDroid 2023) under 'bestPatch' in each of the 7 applications (PAN,FDN,TCN,FBN,FSN,GBN,LBN): in the 'Benchmark' folder.

7.2.2 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. This improved execution time by 19% and memory usage by 29%.

Listing 3 Best patch for F-Droid (FDN).

```
101,103c101
<         if (alphaAnimator == null) {
<             alphaAnimator = ValueAnimator.ofInt(0, 255);
<         } else {
---
>         if (alphaAnimator != null) {
```

7.2.3 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 by 21% but slightly increases execution time due to an extra function call.

Listing 4 Best patch for Tower Collector (TCN).

```
184a185
>             db.close();
238d239
<         SQLiteDatabase db = helper.getReadableDatabase();
265a267
>         SQLiteDatabase db = helper.getReadableDatabase();
511a514
>             db.close();
```

7.2.4 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. This improved execution time by 15%.

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.

Listing 5 Best patch for Frozen Bubble (FBN).

```
350d349
<         if (newRow[column] != -1) {
358,359d356
<             this.spriteToBack(tempBubble);
<         }
```

7.2.5 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 24% and execution time 23%.

Listing 6 Best patch for Fosdem-Companion (FSN).

```
50a51,54
>     List<Person> persons = new ArrayList<>();
>     event.setPersons(persons);
>     List<Link> links = new ArrayList<>();
>     event.setLinks(links);
69,72d72
<                 List<Person> persons = new ArrayList<>();
<                 event.setPersons(persons);
<                 List<Link> links = new ArrayList<>();
<                 event.setLinks(links);
```

7.2.6 Gadget bridge (GBN)

In the best change for the Gadget Bridge Application we remove the redundant rendering of a view that is already visible. reducing execution time by 35%

Listing 7 Best patch for Gadget Bridge(GBN).

```
336d335
<                 appListFabNew.show();
```

7.2.7 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. This improved execution time by 19%. The list is an argument decorated with @NonNull so should never be null, and in the case that is there will be no errors.

Listing 8 Best patch for Lightning Browser (LBN).

```
71d70
<                 Preconditions.checkNonNull(bookmarkList);
```

Table 5 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**

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.

7.3 RQ3: multi-objective search

In order to compare the different algorithms used in search, we consider the procedure proposed by Li et al. (2020), 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 (Ji et al. 2018; Liu et al. 2021). Due to different fitness scales, we normalise the values, though also present raw ones in our online repository, including all Pareto fronts (GIDroid 2023). Normalised hypervol-

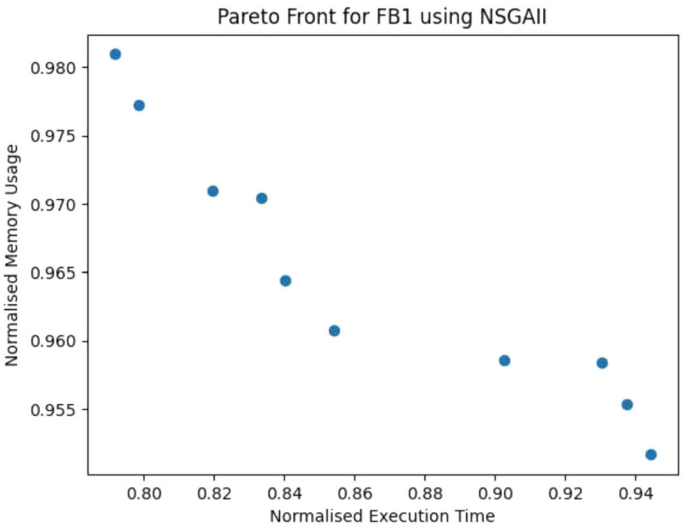

Fig. 6 Pareto Front from NSGA-II experiments on the FB1 Benchmark

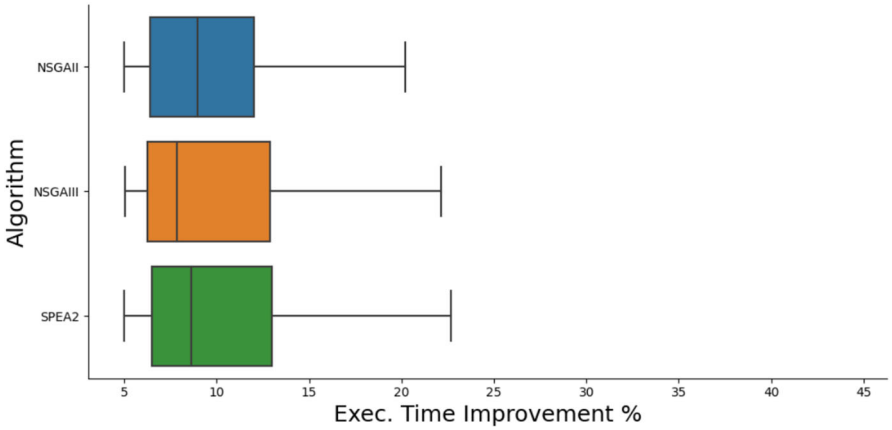

Fig. 7 Boxplot of improvements found to execution time by each algorithm

ume values are presented in Table 5. The Pareto fronts from all of our multi-objective experiments can be found in our repository (GIDroid 2023). We find that across our experiment we find patches spread across the Pareto front (see Figs. 6, 7, and 8), 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 Figs. 4 and 5. We find that the caching operators we introduced turned out to be highly effective, appearing in 26% of improving patches.

Table 6 A effect size for each algorithm on each benchmark

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)

Effect sizes larger than 0.5 show positive improvement. differences: N = negligible, S = small, M = medium, L = large

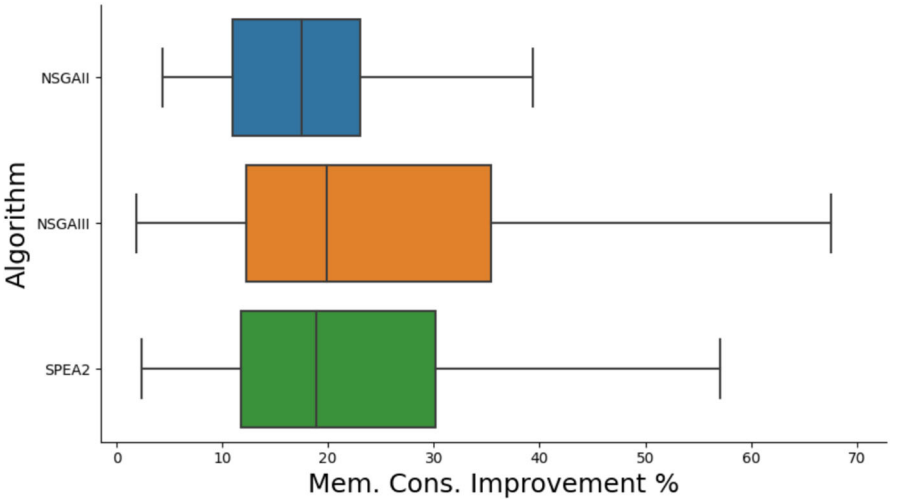

Fig. 8 Boxplot of improvements found to memory consumption by each algorithm

Table 7 Frequency in which each operator was found in the best patches of each run

Operator	% of best patches using operator
Delete	24.0
Swap	17.1
Replace	15.4
Copy	20.1
Cache	7.0
ClassCache	4.5

We find improving patches in 1092 out of 1260 experiments.

We also evaluate the effect size of the improvements found by each of the MO algorithms, as show in Table 6. We use the Vargha and Delaney A measure (Vargha and Delaney 2000) 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 (Table 7).

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.

7.4 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

Table 8 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
PortAuthority 6	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

results of this evaluation can be found in Table 8. 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.

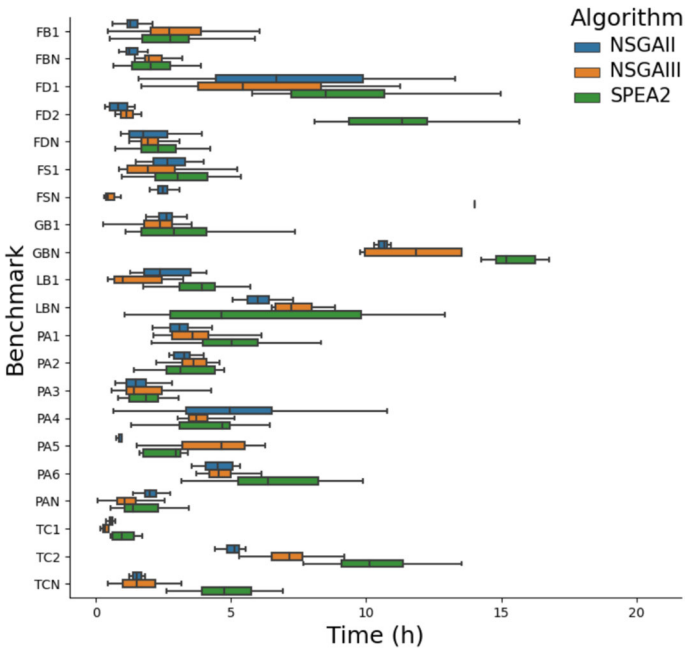

Fig. 9 Time taken by GIDroid using different MO algorithms to evolve 10 generations, each with 40 individuals

7.5 RQ5: cost of GI

In order to evaluate the applicability of our approach, we analyze its cost. Figure 9 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 and 20.6 h, with a median time across the benchmarks of 2.6 h. 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 s to execute, whereas the quickest one takes 2 s. 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 h, a maximum of 19.0 h, and a median of 3.5 h. 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 app users consider wait time of 150 ms 'laggy', which might lead to them abandoning an app, and considering the difficulty of manually optimising non-functional properties, especially in large codebases, 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.

Table 9 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

7.6 RQ6: comparison to linter and ChatGPT

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, and are available in our online repository (GIDroid 2023).

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 min 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 (Table 9).

ChatGPT, on the other hand, failed to find any improvements,[6] while GIDroid rediscovered 64% of those.

> **Answer to RQ6:** We find that our setup is more effective than linters or ChatGPT 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.

8 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. In scenarios with multiple processes or extensive I/O operations, the CPU time may not accurately represent the execution time. To mitigate such threats, we run the garbage collector before taking final memory measurements so we can consistently measure the memory usage. We also measure the runtime so increase in this due to garbage collection triggers would be detected and the changes where this had a large impact discarded. 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. Finally, we validate all improvements both with extra evaluations after experiments and manual analysis.

Furthermore, we use tests to determine whether or not a patch is valid. This does not guarantee correctness. We test areas of code in which we either know there is a performance defect, as either they have been fixed by developers previously, or a static analyser has indicated there may be a performance defect. Each test suite achieves at least 75% branch coverage. The test cases were found to be good enough that the majority of the patches validated against them were actually valid and allow us to find real improvements, thus, for GI they were certainly good enough. Moreover, during search, we exclusively utilise local unit testing, not testing patches on actual devices or emulators. We 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. 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.

[6] All responses are in our repo (GIDroid 2023), in the 'Results/ChatGPT' folder.

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 (see Table 2). Moreover, we found improvements in current app versions, which were previously undiscovered. Some of them simply removed redundant calls, an optimisation that can be applied to any software and found using the `delete` mutation operator in GIDroid (see Sect. 7.2). 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 make all our code and results freely available (GIDroid 2023), allowing other researchers and developers to use and extend our tool and validate our work.

9 Conclusions and future work

We propose to 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 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. Future work could explore the capabilities of large language models for generating non-functional property-improving patches. Although the techniques currently only perform well on relatively small programs (Madaan et al. 2023), trained on source code from programming competitions or puzzles which is short and self-contained (Puri et al. 2021). These examples do not contain the complex shared state and interaction with external components that are commonplace in Android apps.

Funding This work was supported by EPSRC Grant No. EP/P023991/1.

Data Availability All code and results are available in our repository GIDroid (2023).

Declarations

Conflict of interest Prof. Petke is a Deputy Editor-in-Chief for the Automated Software Engineering journal. There are no other conflict of interest to declare.

References

Amalfitano, D., Amatucci, N., Fasolino, A.R., et al.: Agrippin: a novel search based testing technique for Android applications. In: DeMobile@SIGSOFT FSE, pp. 5–12. ACM (2015)

An, G., Blot, A., Petke, J., et al.: Pyggi 2.0: language independent genetic improvement framework. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pp. 1100–1104 (2019)

Android Development Team: Android context. https://developer.android.com/reference/android/content/Context (2022a). Accessed 12 May 2023

Android Development Team: Android testing guide. https://developer.android.com/studio/test (2022b). Accessed 07 Feb 2023

Android Development Team: Android compilation guide. https://developer.android.com/studio/build (2023a). Accessed 07 Feb 2023

Android Development Team: Android lint tool. https://developer.android.com/studio/write/lint (2023b). Accessed 06 Feb 2023

Auer, M., Adler, F., Fraser, G.: Improving search-based Android test generation using surrogate models. In: SSBSE, Lecture Notes in Computer Science, vol. 13711, pp. 51–66. Springer (2022)

Ayala, I., Amor, M., Fuentes, L.: An energy efficiency study of web-based communication in android phones. Sci. Program. **8235458**(1–8235458), 19 (2019). https://doi.org/10.1155/2019/8235458

Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: OOPSLA, pp. 641–660. ACM (2013)

Bach, T., Andrzejak, A., Pannemans, R., et al.: The impact of coverage on bug density in a large industrial software project. In: ESEM, pp. 307–313. IEEE (2017)

Baek, Y.M., Bae, D.: Automated model-based Android GUI testing using multi-level GUI comparison criteria. In: ASE, pp. 238–249. ACM (2016)

Basios, M., Li, L., Wu, F., et al.: Optimising Darwinian data structures on Google Guava. In: SSBSE, Lecture Notes in Computer Science, vol. 10452, pp. 161–167. Springer (2017)

Baumann, P., Santini, S.: Every byte counts: selective prefetching for mobile applications. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. **1**(2), 6:1-6:29 (2017)

Berg, F., Durr, F., Rothermel, K.: Increasing the efficiency and responsiveness of mobile applications with preemptable code offloading. In: 2014 IEEE International Conference on Mobile Services (MS) Â, pp. 76–83 (2014)

Blot, A., Petke, J.: Empirical comparison of search heuristics for genetic improvement of software. IEEE Trans. Evol. Comput. **25**(5), 1001–1011 (2021)

Bokhari, M.A., Bruce, B.R., Alexander, B., et al.: Deep parameter optimisation on Android smartphones for energy minimisation: a tale of woe and a proof-of-concept. In: GECCO (Companion), pp. 1501–1508. ACM (2017)

Brownlee, A.E.I., Petke, J., Alexander, B., et al.: Gin: genetic improvement research made easy. In: GECCO, pp. 985–993. ACM (2019)
Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: GECCO, pp. 1327–1334. ACM (2015)
Burles, N., Bowles, E., Brownlee, A.E.I., et al.: Object-oriented genetic improvement for improved energy consumption in Google Guava. In: SSBSE, Lecture Notes in Computer Science, vol. 9275, pp. 255–261. Springer (2015)
Callan, J., Petke, J.: Improving Android app responsiveness through automated frame rate reduction. In: SSBSE, Lecture Notes in Computer Science, vol. 12914, pp. 136–150. Springer (2021)
Callan, J., Petke, J.: Improving responsiveness of Android activity navigation via genetic improvement. In: ICSE-Companion, pp. 356–357. ACM/IEEE (2022a)
Callan, J., Petke, J.: Multi-objective genetic improvement: a case study with EvoSuite. In: SSBSE, Lecture Notes in Computer Science, vol. 13711, pp. 111–117. Springer (2022b)
Callan, J., Krauss, O., Petke, J., et al.: How do Android developers improve non-functional properties of software? Empir. Softw. Eng. **27**(5), 113 (2022)
Chun, B.G., Ihm, S., Maniatis, P., et al.: Clonecloud: Elastic Execution Between Mobile Device and Cloud. Association for Computing Machinery, New York (2011)
Cruz, L., Abreu, R.: Performance-based guidelines for energy efficient mobile applications. In: 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22–23, 2017, pp. 46–57. IEEE (2017). https://doi.org/10.1109/MOBILESOFT.2017.19
Das, P.K., Shome, S., Sarkar, A.K.: Apps: accelerating performance and power saving in smartphones using code offload. In: 2016 IEEE 6th International Conference on Advanced Computing (IACC), pp. 759–765 (2016)
Deb, K., Jain, H.: 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)
Deb, K., Agrawal, S., Pratap, A., et al.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In: PPSN, Lecture Notes in Computer Science, vol. 1917, pp. 849–858. Springer (2000)
Ding, A.Y., Bo Han, Yu., Xiao, et al.: Enabling energy-aware collaborative mobile data offloading for smartphones. In: 2013 IEEE International Conference on Sensing, Communications and Networking (SECON), pp. 487–495 (2013)
FindBugs Development Team: FindBugs Lint Tool. https://findbugs.sourceforge.net/ (2015). Accessed 06 Feb 2023
GIDroid: a tool for multi-objective GI in Android (2023). https://github.com/SOLAR-group/GIDroid
Habchi, S., Blanc, X., Rouvoy, R.: On adopting linters to deal with performance concerns in Android apps. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018, pp. 6–16. ACM (2018)
Haraldsson, S.O., Woodward, J.R., Brownlee, A.E.I., et al.: Fixing bugs in your sleep: how genetic improvement became an overnight success. In: Bosman, P.A.N. (ed.) GECCO, Berlin, Germany, July 15–19, 2017, Companion Material Proceedings, pp. 1513–1520. ACM (2017)
Hort, M., Kechagia, M., Sarro, F., et al.: A survey of performance optimization for mobile applications. IEEE Trans. Softw. Eng. **48**(8), 2879–2904 (2022)
Inukollu, V., Keshamoni, D., Kang, T., et al.: Factors influencing quality of mobile apps: role of mobile app development life cycle. Int. J. Soft Eng. Appl. **5**, 15–34 (2014)
Ji, R., Li, Z., Chen, S., et al.: Uncovering unknown system behaviors in uncertain networks with model and search-based testing. In: ICST, pp. 204–214. IEEE (2018)
Kemp, S.: Digital 2022: mobile duopoly consolidates its grip—datareportal—global digital insights. https://datareportal.com/reports/digital-2022-mobile-duopoly-consolidates-grip (2022). Accessed 01 Mar 2024
Kerrisk, M.: Linux time. https://man7.org/linux/man-pages/man1/time.1.html (2019). Accessed 10 Feb 2023
Kerrisk, M.: Linux process tracking. https://man7.org/linux/man-pages/man5/proc.5.html (2022). Accessed 10 Feb 2023

Khalid, H., Shihab, E., Nagappan, M., et al.: What do mobile app users complain about? IEEE Softw. **32**(3), 70–77 (2015)
Kim, M., Hiroyasu, T., Miki, M., et al.: SPEA2+: improving the performance of the strength Pareto evolutionary algorithm 2. In: PPSN, Lecture Notes in Computer Science, vol. 3242, pp. 742–751. Springer (2004)
Langdon, W.B., Lam, B.Y.H., Petke, J., et al.: Improving CUDA DNA analysis software with genetic programming. In: Silva, S., Esparcia-Alcázar, A.I. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11–15, 2015, pp. 1063–1070. ACM (2015)
Li, M., Chen, T., Yao, X.: How to evaluate solutions in Pareto-based search-based software engineering: A critical review and methodological guidance. IEEE Trans. Softw. Eng. **48**, 1–1 (2020)
Li, S.S., Peeler, H., Sloss, A.N., et al.: Genetic improvement in the Shackleton framework for optimizing LLVM pass sequences. In: GECCO Companion. ACM, pp. 1938–1939 (2022)
Li, Y., Yang, Z., Guo, Y., et al.: DroidBot: a lightweight UI-guided test input generator for Android. In: ICSE (Companion Volume), pp. 23–26. IEEE (2017)
Lim, S.L., Bentley, P., Kanakam, N., et al.: Investigating country differences in mobile app user behavior and challenges for software engineering. IEEE TSE **41**, 40–64 (2014)
Lin, Y., Radoi, C., Dig, D.: Retrofitting concurrency for Android applications through refactoring. In: FSE, pp. 341–352. ACM (2014)
Lin, Y., Okur, S., Dig, D.: Study and refactoring of Android asynchronous programming (T). In: ASE, pp. 224–235. IEEE (2015)
Liu, Y., Zhu, N., Li, M.: 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)
Lyu, Y., Li, D., Halfond, W.G.J.: Remove rats from your code: automated optimization of resource inefficient database writes for mobile applications. In: ISSTA, pp. 310–321. ACM (2018)
Madaan, A., Shypula, A., Alon, U., et al.: Learning performance-improving code edits. CoRR abs/2302.07867 (2023)
Mahmood, R., Mirzaei, N., Malek, S.: Evodroid: segmented evolutionary testing of Android apps. In: SIGSOFT FSE, pp. 599–609. ACM (2014)
Mao, K., Harman, M., Jia, Y.: Sapienz: multi-objective automated testing for Android applications. In: ISSTA, pp. 94–105. ACM (2016)
Mateus, B.G., Martinez, M., Kolski, C.: An experience-based recommendation system to support migrations of Android applications from Java to Kotlin. CoRR abs/2103.09728 (2021)
Mesecan, I., Blackwell, D., Clark, D., et al.: Keeping secrets: multi-objective genetic improvement for detecting and reducing information leakage. In: ASE, pp. 61:1–61:12. ACM (2022)
Mockus, A., Nagappan, N., Dinh-Trong, T.T.: Test coverage and post-verification defects: a multiple case study. In: ESEM, pp. 291–301. IEEE (2009)
Mohan, P., Nath, S., Riva, O.: Prefetching Mobile Ads: Can Advertising Systems Afford it? Association for Computing Machinery, New York (2013)
Morales, R., Saborido, R., Khomh, F., et al.: EARMO: an energy-aware refactoring approach for mobile apps. IEEE Trans. Softw. Eng. **44**(12), 1176–1206 (2018)
Motwani, M., Soto, M., Brun, Y., et al.: Quality of automated program repair on real-world defects. IEEE TSE **48**(2), 637–661 (2022)
Nistor, A., Song, L., Marinov, D., et al.: Toddler: detecting performance problems via similar memory-access patterns. In: ICSE, pp. 562–571 (2013)
Oracle Development Team: Java's runtime class. https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html (2020). Accessed 10 Feb 2023
Pecorelli, F., Catolino, G., Ferrucci, F., et al.: Testing of mobile applications in the wild: a large-scale empirical study on Android apps. In: ICPC, pp. 296–307. ACM (2020)
Petke, J., Langdon, W.B., Harman, M.: Applying genetic improvement to MiniSAT. In: SSBSE, Lecture Notes in Computer Science, vol. 8084, pp. 257–262. Springer (2013)
Petke, J., Haraldsson, S.O., Harman, M., et al.: Genetic improvement of software: a comprehensive survey. IEEE Trans. Evol. Comput. **22**(3), 415–432 (2018)
PMD Development Team: PMD: an extensible cross-language static code analyzer. https://pmd.github.io/ (2023). Accessed 06 Feb 2023

Puri, R., Kung, D.S., Janssen, G., et al.: Project codenet: a large-scale AI for code dataset for learning a diversity of coding tasks. CoRR abs/2105.12655 (2021)
Robolectric Develop. Team: Robolectric.https://robolectric.org/ (2023). Accessed 12 May 2023
Saarinen, A., Siekkinen, M., Xiao, Y., et al.: Can offloading save energy for popular apps? In: Proceedings of the Seventh ACM International Workshop on Mobility in the Evolving Internet Architecture (2012)
Srinivas, N., Deb, K.: Multiobjective optimization using nondominated sorting in genetic algorithms. Evol. Comput. **2**(3), 221–248 (1994)
Su, T., Meng, G., Chen, Y., et al.: Guided, stochastic model-based GUI testing of Android apps. In: ESEC/SIGSOFT FSE, pp. 245–256. ACM (2017)
Tolia, N., Andersen, D.G., Satyanarayanan, M.: Quantifying interactive user experience on thin clients. Computer **39**(3), 46–52 (2006)
Vargha, A., Delaney, H.D.: A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000)
White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Trans. Evol. Comput. **15**(4), 515–538 (2011)
Wu, F., Weimer, W., Harman, M., et al.: Deep parameter optimisation. In: GECCO, pp. 1375–1382. ACM (2015)
Yasin, H.N., Hamid, S.H.A., Yusof, R.J.R.: Droidbotx: test case generation tool for Android applications using q-learning. Symmetry **13**(2), 310 (2021)
Zuo, S., Blot, A., Petke, J.: Evaluation of genetic improvement tools for improvement of non-functional properties of software. In: GECCO Companion, pp. 1956–1965. ACM (2022)