# Improving Android App Responsiveness through Automated Frame Rate Reduction

James Callan and Justyna Petke

University College London (UCL), London, United Kingdom
{james.callan.19, j.petke}@ucl.ac.uk

**Abstract.** Responsiveness is one of the most important properties of Android applications to both developers and users. Recent survey on automated improvement of non-functional properties of Android applications shows there's a gap in application of search-based techniques to improve responsiveness. Therefore, we explore the use of genetic improvement (GI) to achieve this task. We extend Gin, an open source GI framework, to work with Android applications. Next, we apply GI to four open source Android applications, measuring frame rate as proxy for responsiveness. We find that while there are improvements to be found in UI-implementing code (up to 43%), often applications' test suites are not strong enough to safely perform GI, leading to generation of many invalid patches. We also apply GI to areas of code which have highest test-suite coverage, but find no patches leading to consistent frame rate reductions. This shows that although GI could be successful in improvement of Android apps' responsiveness, any such test-based technique is currently hindered by availability of test suites covering UI elements.

**Keywords:** Genetic Improvement · Search-Based Software Engineering · Responsiveness · Android · Mobile Applications

## 1 Introduction

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 [17]. Khalid et al. [19] found that unresponsiveness was one of the most frequent reasons that users left bad reviews on mobile applications. Lim et al. [23] found that unresponsiveness was to blame 1/3 of the times when users abandoned applications.

Despite the importance of app responsiveness, there is not much research on its automated improvement. Given that the first Android version appeared only in 2008, the discrepancy between the number of approaches for software improvement for desktop vs. mobile applications is, perhaps, unsurprising. Recently Hort et al. [16] conducted a survey on Android performance optimization. It reveals that most approaches for responsiveness improvement focus on problem detection, rather than automated improvement. Among the most common techniques

in the second category are: offloading and refactoring. Offloading [29], however, requires external infrastructure, while refactoring-based approaches tend to focus on very specific improvements, like introducing concurrency to long running processes [9] or combining HTTP requests [22]. Although these transformations could indeed help improve responsiveness, we believe that the search space of mutations could be combined and extended. With increase of the space of possible code refactorings, the search space for finding improvements will inadvertently increase. That's why search-based approaches would be a good fit to explore it. Notably, Hort et al. [16] do not report any search-based approaches for automated responsiveness improvement. With this work we intend to fill this gap.

In the desktop domain, Genetic Improvement (GI) has recently shown success in improvement of various functional (e.g., bug fixing [3]) and non-functional (e.g., runtime [21] or energy consumption [7]) properties. GI uses search-based algorithms to navigate the space of patches to existing software. It uses a fitness function that guides the search. Fitness could be based on test case failures, to check whether the program behaves correctly, and/or other measure, such as runtime to measure improvement of program's execution time.

In order to apply GI to improvement of responsiveness, we must first define the fitness function. Responsiveness, however, can be difficult to quantify. Moreover, measurements such as runtime are inherently noisy. Inspired by previous work [14], we propose using the frame rate of an application as a metric for responsiveness.

In this work we apply genetic improvement to improve Android app responsiveness. In particular, we extend an existing GI framework, Gin, to work on the Android domain. Next, we use it to improve frame rate of four Android applications. We find improvement of up to 50%, though closer inspection reveals many of the patches to be invalid, due to weak test suites — used as proxies for correct program behaviour, as is common in GI work. Nevertheless, we found a valid mutation that reduced frame rate by 43%. Subsequently, we apply GI on code with high test-suite coverage. However, in this case, no consistent improvements were found.

In summary, this work provides the following contributions:

- extension of an open source GI framework for the Android domain;
- first open source framework for automated frame rate reduction of Android applications[1];
- feasibility study for application of genetic improvement for the purpose of automated improvement of Android app responsiveness.

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

The rest of the paper is divided as follows: Section 2 presents a short introduction to genetic improvement; Section 3 outlines our proposed framework for improving Android app responsiveness; Section 4 presents our methodology

[1] Available: https://github.com/AndroidGI/AndroidGI

for the empirical study; with results in Section 5; Section 6 notes threats to validity of our work; Section 7 shows related work on automated improvement of responsiveness of Android apps; while Section 8 concludes the paper.

## 2 Background

Genetic improvement uses automated search to improve existing software [27]. Typically it operates at the level of source code, though mutations to the binary, assembly, and others have been tried. GI takes existing software, mutates it, generating sometimes thousands of software variants. Each variant is represented as a list of edits to the original code. Typical mutations involve copying, deleting, or replacing a code fragment, that being either a statement (most often), line or other (e.g., a binary operator). The search space of the evolved programs is navigated using a search strategy. Although historically genetic programming has been used, recent work show that local search can be equally effective [6]. Although the technique is simple, it has already been incorporated in the industry, during development process [13].

Work on Android software improvement using GI is scarce — so far only one work on GI exists in the Android domain [7], in which 'deep parameters' (constants not exposed to developers) were modified in order to find transformations which reduce the energy consumption of an application. The framework, however, is not open source, and source code was re-factored so that the search was conducted on an external file with parameters, causing upfront cost, and limiting mutations that could be automatically applied.

In this work we investigate the power of more traditional GI to improve another non-functional property of Android applications, namely their responsiveness.

## 3 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 [12, 20, 28] 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. [11] measured the "user-perceived latency" of interactions with applications, which is the time between a user input 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.

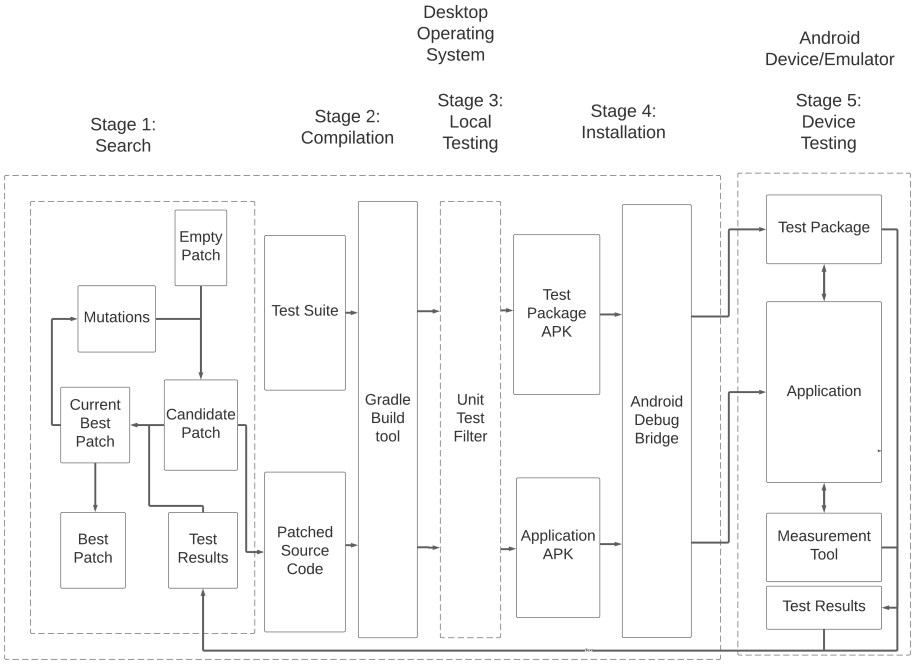



Fig. 1: Genetic improvement framework for Android applications.

To measure an applications' 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 1. The improvement process takes place across two devices: 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, 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 which 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).

## 4 Methodology

In order to investigate the effectiveness of genetic improvement for the purpose of improvement of 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.

In order to answer our research questions we implemented the framework presented in Figure 1, and run it on a selection of Android applications.

### 4.1 Framework

We realise the abstract framework presented in Figure 1 in the Gin GI tool [8]. 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 which 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.

**Fitness** There are a number of different metrics which can be used to measure frame rate. They include the frames per second (FPS), 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.

**Testing** Patch evaluation consisted of running all test cases which 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 efficiency of the GI process, we identify test cases that cover the given class for improvement, using *jacoco* [2]. Next, we use *espresso* [1] 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 generalisability of improvements found. Therefore, for Stage 3 and Stage 5 of our GI process presented in Figure 1 we use UI tests causing 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 it's improvement in Stage 5, where each test is run 10 times, and median 95th percentile frame render time 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.

**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. [6]. 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.

### 4.2 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 record median frame rate improvement. This allowed us to run statistical test 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 manual analysis of the patches to confirm their validity.

### 4.3 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 which 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. [2]

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. [26] performed an analysis of all applications from FDroid, documenting both the number of tests and the coverage of those tests.

[2] Based on manual judgement 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 [26] in descending order of line coverage. We then discarded applications which were not written in Java, those that could not be compiled, those whose tests could not be run successfully, and those which 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 flaky tests - for two reasons. Firstly, the *jacoco* test coverage plugin [2] requires all tests to pass so flaky tests could disrupt the coverage measurement. Secondly, flaky tests may produce false negatives in patch validation. If a test fails due to flakiness, rather than due to the applied patch, it will make 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 [10], 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 which could generate automatic UI test input, however none worked on the recent versions of Android we ran our experiments on. 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.

**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 which they exercise, and unit test 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 which is best covered by the whole test suite, and covered by at least one UI test, so we could measure frame rate usage.

Table 1: The number of tests cases and % line coverage for each of the selected classes

| App Name | Class Name | No. Tests | Line Cov.(%) |
|---|---|---|---|
| AntennaPod | PreferenceActivity (Exp1) | 8 | 43 |
| | MainPreferencesFragment (Exp2) | 37 | 68 |
| Gnu Cash | AccountsListFragment (Exp1) | 11 | 64 |
| | GnuCashApplication (Exp2) | 37 | 76 |
| MicroPinner | MainDialog (Exp1) | 10 | 44 |
| | MainPresenterImpl (Exp2) | 14 | 75 |
| WikimediaCommons | AboutActivity (Exp1) | 9 | 45 |
| | RecentSearchesContentProvider (Exp2) | 18 | 63 |

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 analysed to check for suitability, as described above.

Table 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.

### 4.4 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.

## 5 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, that was also covered by at least one UI test.

### 5.1 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 rate of the unpatched application and the patched application."*

Table 2: Improvements achieved in poorly tested UI classes

| Project | No. improvements 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 |

Table 3: Improvements achieved in well tested classes

| Project | No. improvements 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 |

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 2 and 3 show our results.

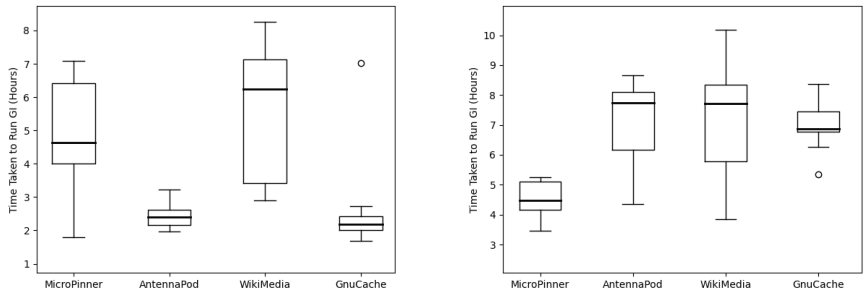
We find that only 11 out of 160 of the GI runs performed found statistically significant improvements and 8 of those were in one application. In the vast majority of cases no improvements were found and the GI execution simply returned an empty patch. In 7 cases in the first experiment, patches were found which 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 which 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 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.

### 5.2 RQ2: Types of Improvements

To understand the types of improvement which 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.




(a) Times taken for experiments on UI classes

(b) Times taken for experiments on well covered classes

Fig. 2: Boxplots of the Time Taken for Each Run on Each Project in Hours

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 users 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 [26].

In the cases were improvements found turned out to be invalid, again the classes being improved did not have adequate coverage and the test which did cover were not very robust. Some patches removed lines of text which 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 on the UI classes with lower coverage.

### 5.3 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 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 which could be built, and therefore tested, that were found. Trying to target classes which 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.

## 6 Threats to Validity

There are a number of threats to the validity of this work. Below we present these threats and the actions taken to mitigate them.

**Noise in Measurements** Whilst the fitness measurement only showed a small amount of noise when tested, these small deviations could still produce false positives for improvements. In order to mitigate this threat, we conduct repeated measurements and statistical tests on all improvements, in order to verify that the improvements are real.

**Stochastic Search** Using randomised search may result in us 'getting lucky', and finding improvements that would not be likely to be found in subsequent runs. In order to show how our approach works generally, we perform 20 runs for each experimental setup (160 runs total).

**Overfitting** As our patches are generated on a single emulator there is a chance that they will not translate to other, real world hardware. In order to test this, we validate all improvements that are found on a real device. Improvements may also be overfitted to the set of tests used during fitness evaluations. To test for this overfitting, a larger set of tests is used to validate the patches that are found, checking if any of them fail, along with how much of an impact the improvements have on other test cases.

## 7 Related Work

A recent survey [16] revealed several approaches for improving Android app responsiveness.

Offloading is a popular technique for improving the responsiveness of applications [5, 11, 12, 18, 20, 28]. When offloading, expensive computation is performed on an external server, saving both computational effort and energy usage on the actual device. The main challenge of offloading is dynamically deciding what should be offloaded. Offloading requires developers to create an external infrastructure, e.g., using cloud computing, to perform computation. This could be complex and costly for developers. In this work, we propose finding purely local changes to applications which do not require the set up of external hardware.

Pre-fetching is another technique used to improve responsiveness [4, 15, 30]. Online resources are asynchronously fetched before they are needed so that the user does not have to wait for the request to be executed. Pre-fetching is limited to improving a limited number of operations, and is not applicable to many applications.

Local transformations for improving responsiveness of applications have also been considered. Hecht et al. [14] tested the impact of repairing Android code smells on frame rate. Lin et al. [24] developed a tool to automatically refactor code into asynchronous tasks. Yijung et al [25] automatically refactored inefficient local database writes for applications in order to improve responsiveness.

None of the discovered related work considers utilising a larger set of refactorings and using search-based approaches to navigate this space.

## 8 Conclusions and Future Work

In this work we present a genetic improvement approach for improvement of responsiveness of Android applications. Even though we report negative results, our research also revealed several avenues for future research.

Whilst genetic improvement is capable of finding improvements to the frame rate of Android applications it is greatly limited by the number of 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 increasing the code coverage.

In future work we plan to extend the traditional set of operators with refactorings specialised for responsiveness. We also plan to expand our research to investigate the power of GI to improve other properties of Android applications. We also plan to use multi-objective search, as naturally improvements to responsiveness might negatively influence other software properties, such as memory consumption. We would also aim to speed up GI for Android, if we can successfully find improvements using only local tests, we can avoid having to package and install the application, greatly speeding up fitness evaluations. We believe that despite current obstacles related to testing, future automated improvement tooling for Android will benefit from search-based approaches, such as genetic improvement.

**Acknowledgements** This work was funded by the EPSRC fellowship EP/P023991/1.

## References


1. Espresso for UI testing, https://developer.android.com/training/testing/espresso/
2. Jacoco, https://docs.gradle.org/current/userguide/jacoco_plugin.html
3. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. pp. 162 – 168 (07 2008)
4. Baumann, P., Santini, S.: Every byte counts: Selective prefetching for mobile applications. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. **1**(2) (Jun 2017), https://doi.org/10.1145/3090052
5. Berg, F., Dürr, F., Rothermel, K.: Increasing the efficiency and responsiveness of mobile applications with preemptable code offloading. In: 2014 IEEE International Conference on Mobile Services. pp. 76–83 (2014). https://doi.org/10.1109/MobServ.2014.20
6. Blot, A., Petke, J.: Empirical comparison of search heuristics for genetic improvement of software. IEEE Transactions on Evolutionary Computation pp. 1–1 (2021). https://doi.org/10.1109/TEVC.2021.3070271
7. Bokhari, M.A., Bruce, B.R., Alexander, B., Wagner, M.: 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)
8. Brownlee, A.E.I., Petke, J., Alexander, B., Barr, E.T., Wagner, M., White, D.R.: Gin: Genetic improvement research made easy. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 985–993. GECCO '19, Association for Computing Machinery (2019). https://doi.org/10.1145/3321707.3321841
9. Feng, R., Meng, G., Xie, X., Su, T., Liu, Y., Lin, S.: Learning performance optimization from code changes for android apps. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 285–290 (2019). https://doi.org/10.1109/ICSTW.2019.00067
10. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: SIGSOFT FSE. pp. 416–419. ACM (2011)
11. Gordon, M.S., Hong, D.K., Chen, P.M., Flinn, J., Mahlke, S., Mao, Z.M.: Accelerating mobile applications through flip-flop replication. In: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services. p. 137–150. MobiSys '15, Association for Computing Machinery, New York, NY, USA (2015), https://doi.org/10.1145/2742647.2742649
12. Gordon, M.S., Jamshidi, D.A., Mahlke, S., Mao, Z.M., Chen, X.: Comet: Code offload by migrating execution transparently. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. p. 93–106. OSDI'12, USENIX Association, USA (2012)
13. Haraldsson, S.O., Woodward, J.R., Brownlee, A.E.I., Siggeirsdottir, K.: Fixing bugs in your sleep: how genetic improvement became an overnight success. In: Bosman, P.A.N. (ed.) Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings. pp. 1513–1520. ACM (2017), https://doi.org/10.1145/3067695.3082517
14. Hecht, G., Moha, N., Rouvoy, R.: An empirical study of the performance impacts of android code smells. In: 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft). pp. 59–69 (2016). https://doi.org/10.1109/MobileSoft.2016.030
15. Higgins, B.D., Flinn, J., Giuli, T.J., Noble, B., Peplin, C., Watson, D.: Informed mobile prefetching. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services. p. 155–168. MobiSys

'12, Association for Computing Machinery, New York, NY, USA (2012), https://doi.org/10.1145/2307636.2307651
16. Hort, M., Kechagia, M., Sarro, F., Harman, M.: A survey of performance optimization for mobile applications. IEEE Transactions on Software Engineering (TSE) (2021)
17. Inukollu, V., Keshamoni, D., Kang, T., Inukollu, M.: Factors influncing quality of mobile apps: Role of mobile app development life cycle. International Journal of Software Engineering & Applications **5** (10 2014)
18. Kemp, R., Palmer, N., Kielmann, T., Bal, H.: Cuckoo: a computation offloading framework for smartphones. In: 2nd Int. Conf. on Mobile Computing, Applications, and Services (MobiCASE 2010) (2010)
19. Khalid, H., Shihab, E., Nagappan, M., Hassan, A.E.: What do mobile app users complain about? IEEE Software **32**(3), 70–77 (2014)
20. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. Proceedings - IEEE INFOCOM **945-953**, 945–953 (03 2012). https://doi.org/10.1109/INFCOM.2012.6195845
21. Langdon, W.B.: Performance of genetic programming optimised bowtie2 on genome comparison and analytic testing (GCAT) benchmarks. BioData Min. **8**, 1 (2015)
22. Li, D., Lyu, Y., Gui, J., Halfond, W.G.J.: Automated energy optimization of http requests for mobile applications. In: Proceedings of the 38th International Conference on Software Engineering. p. 249–260. ICSE '16, Association for Computing Machinery, New York, NY, USA (2016)
23. Lim, S.L., Bentley, P., Kanakam, N., Ishikawa, F., Honiden, S.: Investigating country differences in mobile app user behavior and challenges for software engineering. IEEE Transactions on Software Engineering **41** (09 2014). https://doi.org/10.1109/TSE.2014.2360674
24. Lin, Y., Okur, S., Dig, D.: Study and refactoring of android asynchronous programming (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 224–235 (2015). https://doi.org/10.1109/ASE.2015.50
25. Lyu, Y., Li, D., Halfond, W.G.J.: Remove rats from your code: Automated optimization of resource inefficient database writes for mobile applications. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 310–321. ISSTA 2018, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3213846.3213865
26. Pecorelli, F., Catolino, G., Ferrucci, F., De Lucia, A., Palomba, F.: Testing of mobile applications in the wild: A large-scale empirical study on android apps. ICPC '20, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3387904.3389256
27. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: A comprehensive survey. IEEE Trans. Evol. Comput. **22**(3), 415–432 (2018)
28. Ra, M., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., Govindan, R.: Odessa: enabling interactive perception applications on mobile devices. In: MobiSys '11 (2011)
29. Saarinen, A., Siekkinen, M., Xiao, Y., Nurminen, J., Kemppainen, M., Hui, P.: Can offloading save energy for popular apps? (08 2012)
30. Yang, Y., Cao, G.: Prefetch-based energy optimization on smartphones. IEEE Transactions on Wireless Communications **17**(1), 693–706 (2017)