# Improving Responsiveness of Android Activity Navigation via Genetic Improvement

James Callan
University College London
London, UK
james.callan.19@ucl.ac.uk

Justyna Petke
University College London
London, UK
j.petke@ucl.ac.uk

## ABSTRACT

Responsiveness issues are one of the key reasons why mobile phone users abandon an app or leave bad reviews. In this work, we explore the use of Genetic Improvement to automatically refactor applications to reduce the time taken to move between and within Android activities, without affecting their functionality. This particular Android responsiveness issue has not previously been tackled before. With its application directly to source code, our approach can be used to complement previous work, which modifies the operating system, or focuses on detection of specific coding patterns. We present a fully automated technique for finding improvements to this responsiveness, which does not require the use of an Android device or emulator. We apply our approach to 7 real-world open source applications and find improvements of up to 30% in navigation response time.

## CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**.

## KEYWORDS

Android, Responsiveness, Mobile, Genetic Improvement, SBSE

**ACM Reference Format:**
James Callan and Justyna Petke. 2018. Improving Responsiveness of Android Activity Navigation via Genetic Improvement. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Responsiveness is an important quality of software, especially in the mobile application domain. Responsiveness relates to the ability of software to respond to user interactions quickly and smoothly. User experience is thus affected by even minor responsiveness issues. Lim et al. [4] found that in 1/3 of cases of users abandoning an application, poor application responsiveness was given as the reason.

Several approaches have been proposed to aid developers in improving app responsiveness. These include pre-fetching [2] and offloading [3], which require access to network or external hardware. The only available tool [6] that refactors software to improve responsiveness, targets loops containing SQL queries. Other approaches, e.g., [5], find a specific set of bad coding patterns, leaving developers to decide and fix detected bottlenecks.

Here we propose an approach that automatically detects and reduces a responsiveness-related delay in mobile software. In particular, we observe that one simple source of poor responsiveness is slow navigation between activities. By simply measuring the execution times of these transition methods (e.g., the *OnCreate()* method), we can easily quantify the responsiveness of activity transitions for any application. In the cases where these transitions are slow, reducing their execution time will improve responsiveness. This simple measurement allows us to employ latest search-based improvement strategies, namely Genetic Improvement [8], to automatically identify and improve navigation response time.

We thus modified an existing Genetic Improvement (GI) framework to work in the Android domain. Additionally, we implemented a new fitness strategy, that measures navigation activity response time. We evaluated our approach on 7 real-world apps.

Our results show that GI is able to find patches that improve the navigation responsiveness of Android applications by up to 30%.

## 2 GI FOR ANDROID RESPONSIVENESS ISSUES

Genetic Improvement (GI) has been proposed as a general technique for improvement of non-functional properties of software. It takes existing software and mutates it, generating hundreds or even thousands of software variants. Each evolved patch is assessed and a fitness measurement is taken based on the attribute being improved. This fitness is used to guide the search strategy. In the case of non-functional improvement, patches which fail any tests are discarded and the fitness measurement is then based on the non-functional property being improved.

To test the ability of GI to improve the navigation responsiveness of Android apps, we use the following setup: Each patch consists of a sequence of edits to the nodes of the AST tree. Each edit can be either a DELETE, COPY, REPLACE, or SWAP statement edit. We use a simple local search hill climb to select which variant to evaluate next. We begin with an empty patch and at each step a new mutation is added to the mutant for evaluation. If the new variant is more responsive than the current best it becomes the current best. After a set number of evaluations, the current best is deemed the best patch.

To evaluate each software variant, we split relevant tests into two groups. **Validation Tests** which cover the lifecycle transitions which we wish to optimise. These are used to determine validity of the mutated software variant. **Performance Tests** which only exercise the lifecycle transitions which we wish to optimise. The


*ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

**Table 1: Percentage improvement to CPU time after GI**

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

execution time of these tests is used to determine the fitness of a mutant. The performance test set is a subset of the validation set.

To avoid the high cost of installing and running tests on actual devices or emulators, we use the Robolectric testing library (http://robolectric.org/). This library implements the Android specific APIs, whose use is normally restricted to on-device tests. It allows the testing of UI elements of applications on desktop devices using JUnit, removing the expensive steps usually needed for UI testing. Crucially, Robolectric allows us to test the activity transitions of applications locally and significantly more quickly than we would otherwise be able to.

We implemented this set up by modifying Gin [1], a genetic improvement tool for Java programs, as it specifically targets methods for improvement. We use the test results and fitness evaluation data on the original software to automatically identify the most time-consuming methods that implement navigation responsiveness functionality. We target the activity in each app with the slowest navigation callbacks for later improvement.

In order to evaluate our proposed approach, we tested it on 7 real world applications. To select these applications, we checked every application in the FDroid repository. All applications with Activities written in Java (and thus compatible with Gin) and passing tests which exercised said activities with the Robolectric library were selected. Most other applications either didn't use Robolectric or had very limited test coverage of activities (below 40%).

We run our set up 20 times on each of our 7 benchmarks, to account for the stochastic nature of local search. We run the search for 200 evaluations in each run. All computation was performed on a high performance cloud computer, with 16GB RAM.

## 3 RESULTS

In order to quantify the effectiveness of our set up, we first measure the CPU time of the targeted activity transitions of both patched and unpatched applications. The CPU times of each variant of source code is measured 10 times and the median reading taken. We also perform the Mann-Whitney U test [7] on the data collected here with the null hypothesis: ***"Tests running on patched source code have the same CPU time as those running on unpatched code."*** Those program variants which did not show statistical significance at the 95% confidence level were set to 0% improvement.

The results of our experiment can be seen in Table 1. These results show that GI is capable of finding improvements to the CPU time taken by the code which navigates between activities. We find median improvements of between 4.4% and 24.1%, and maximum improvements of between 6.4% and 29.4%. We find the greatest improvement for the least responsive application.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    if (showedActivityFailedScreen(savedInstanceState)) {
        return;}
    Timber.d("onCreate()");
    super.onCreate(savedInstanceState);
    setContentView(R.layout.card_template_editor_activity);
```

**Figure 1: The most effective patch found. It removes a mostly redundant, yet expensive check in AnkiDroid.**

We analyse the patch which produced the best improvements. This was a patch found in the AnkiDroid application, reducing the CPU time from 1.55s to 1.09s. It simply removed the call to a costly check in the case where an activity is created without an application. This patch is shown in Figure 1. This will only every appear when using certain command line tools and not in normal use, therefore the high cost appears unjustified. However, the patch offers a choice between a huge optimisation, or protection in an obscure edge case. Clearly, the existing code is causing responsiveness issues, as delays of even 150ms are noticeable to users [9].

## 4 CONCLUSION

In this work we propose to use a GI-based approach to improve responsiveness of mobile apps. We applied our approach to 7 diverse mobile applications, showing improvements in time to navigate between activities of up to 30%. Our results show that significant improvements to app responsiveness can be found with negligible changes to app functionality. Unfortunately, the main bottleneck for application to other Android software is lack of test suites covering UI Activities. We plan to extend this work to be able to cover a larger plethora of software, and release our tool to help developers automatically improve responsiveness of their mobile apps.

## ACKNOWLEDGMENTS

This work was funded by EPSRC grant no. EP/P023991/1.

## REFERENCES

[1] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White. 2019. Gin: Genetic Improvement Research Made Easy. In *GECCO*. ACM, New York, NY, USA, 985–993.
[2] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. 2012. Informed Mobile Prefetching. In *MobiSys*. ACM, New York, NY, USA, 155–168.
[3] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. 2012. Cuckoo: A Computation Offloading Framework for Smartphones. In *Mobile Computing, Applications, and Services*. Springer, Berlin, Heidelberg, 59–79.
[4] S. L. Lim, P. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden. 2014. Investigating Country Differences in Mobile App User Behavior and Challenges for Software Engineering. *IEEE TSE* 41 (09 2014).
[5] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *ICSE*. ACM, New York, NY, USA, 1013–1024.
[6] Y. Lyu, D. Li, and W. G. J. Halfond. 2018. Remove RATs from Your Code: Automated Optimization of Resource Inefficient Database Writes for Mobile Applications. In *ISSTA*. ACM, New York, NY, USA, 310–321.
[7] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60.
[8] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE TEVC* 22, 3 (2018), 415–432.
[9] N. Tolia, D. G Andersen, and M. Satyanarayanan. 2006. Quantifying interactive user experience on thin clients. *Computer* 39, 3 (2006), 46–52.