# Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring

Luis Cruz
University of Porto
and HASLab/INESC TEC
Porto, Portugal
luiscruz@fe.up.pt

Rui Abreu
Instituto Superior Técnico, University of Lisbon
and INESC-ID
Lisbon, Portugal
rui@computer.org

Jean-Noël Rouvignac
ForgeRock
Grenoble, France
jn.rouvignac@gmail.com

***Abstract*—Leafactor is a tool to automatically improve the energy consumption of Android apps. It does so by refactoring the source code to follow a set of patterns known to be energy efficient. The toolset was validated using 222 refactorings in 140 open-source apps. Changes were submitted to the original apps by creating pull requests to the official projects.**

***Keywords*-Green Computing; Mobile Computing; Refactoring;**

## I. Introduction

Mobile devices are the most popular form of pervasive computing nowadays. As a consequence, users need to charge their devices often to prevent their inoperability, making battery life a major drawback. Hence, it is important to provide developers with proper toolsets to develop energy efficient apps.

Previously, we have identified code optimizations that may have a significant impact on the energy consumption of Android apps [1]. However, ascertain that code is complying with these patterns is time-consuming and prone to errors. In this paper, we introduce Leafactor — a tool to automatically refactor Android apps to improve energy efficiency. In addition, the toolset has the potential to serve as an educative tool to developers to understand what patterns to follow to develop energy efficient apps.

There are state-of-the-art tools that provide automatic refactoring for Java apps (for instance, *AutoRefactor*[1], *Walkmod*[2], *Facebook pfff*[3], *Kadabra*[4]). Although these tools are very popular amongst the Java community, they do not provide energy related rules.

Despite the fact that recent work has addressed code transformations to energetically optimize mobile apps [1]–[4], only a few approaches offer automatic refactoring in the context of mobile energy efficiency. As an example, previous work analyzed the event flow graph to optimize resource usage (e.g., GPS, Bluetooth) being able to reduce energy consumption [5]. Although this approach gives developers an insight of how to refactor the code base, it is not fully automated yet. Other studies have studied and applied automatic refactorings in

[1]http://autorefactor.org/ (March 3, 2017) – An *Eclipse* plugin to automatically refactor Java code bases.
[2]http://walkmod.com/ (March 3, 2017)
[3]http://github.com/facebook/pfff/ (March 3, 2017)
[4]http://specs.fe.up.pt/tools/kadabra/ (March 3, 2017)

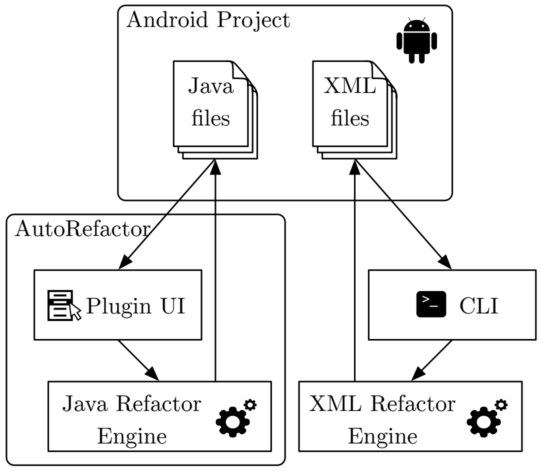

Fig. 1. Leafactor architecture diagram.

Android apps [6]. However, these optimizations are not mobile specific.

## II. Refactoring Android Apps

Leafactor statically analyzes and transforms code to implement Android-specific, energy-efficient optimizations. The architecture of the toolset is depicted in Figure 1. There are two engines: one to handle Java files and another to handle XML files. The refactoring engine for Java was implemented by leveraging the open-source project *AutoRefactor*.

*AutoRefactor* provides several common code cleanups to help delivering “smaller, more maintainable and more expressive code bases”. Eclipse Marketplace reports 1180 *AutoRefactor* installs in 2016. Under the hood, *AutoRefactor* integrates a handy and concise API to manipulate Java *Abstract Syntax Trees* (AST). The Java optimizations offered by Leafactor are also integrated in *AutoRefactor*. So far, this integration requires the use of Eclipse IDE, but we intend to deliver Leafactor as a standalone application.

Since XML refactorings are not part of *AutoRefactor*, another engine needed to be developed. At the moment, only a single XML optimization is offered — *ObsoleteLayoutParam*.

Leafactor receives a single file, a package, or a whole Android project as input and looks for eligible files, i.e., Java

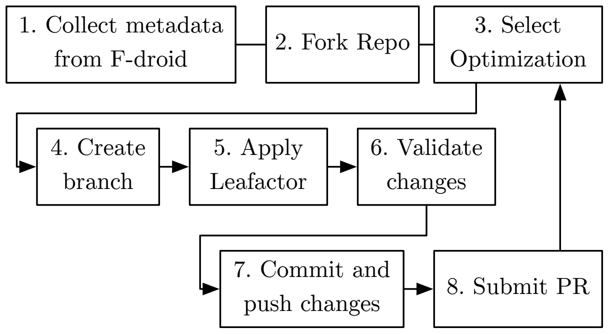

Fig. 2. Experiment's procedure for a single app.

TABLE I
SUMMARY OF REFACTORING RESULTS

| Optimization Rule | W | R | DA | VH | OLP |
|---|---|---|---|---|---|
| Total Refactors | 1 | 58 | 0 | 7 | 156 |
| Affected Projects | 1 | 23 | 0 | 5 | 30 |
| Affected Projects (%) | 1 | 16 | 0 | 4 | 21 |

Wakelock (W), Recycle (R), DrawAllocation (DA), ViewHolder (VH), ObsoleteLayoutParam (OLP)

or XML source files that are then automatically analyzed and new compilable and optimized versions are produced.

*a) Supported optimizations:* Android apps can benefit up to 5% of additional battery life in common use case scenarios; as demonstrated by our previous study [1]. The following optimizations are supported by the toolset. We do not detail these optimizations because of lack of space; for interested readers, refer to [1].

**DrawAllocation:** Allocations within drawing logic.
**WakeLock:** Incorrect wake lock usage.
**Recycle:** Failing to release singleton resources.
**ObsoleteLayoutParam:** Unnecessary layout parameters.
**ViewHolder:** List should use View Holder pattern to reuse previous processing.

## III. Empirical Results

We have analyzed 140 open-source Android apps using Leafactor. These apps are available on *F-droid* — a catalog for free and open-source apps. We have limited our search to those published before Nov 27, 2016 and whose code could be found on *Github*.

Leafactor analyzed 6.79GB of Android projects in 4.5 hours, totaling 15308 Java files and 15103 XML files. The largest project in terms of Java files is *TinyTravelTracker* (1878), while *NewsBlue* is the largest in terms of XML files (2109). Refactoring rules were applied separately.

Leafactor yielded a total of 222 refactorings, which were submitted to the original repositories as *Pull Requests (PRs)*. For a given project, a PR was created for each applied optimization, resulting in 59 PRs. This is a difficult process, since each project has different contributing guidelines. Nevertheless, by the time of writing, 16 PRs had been successfully merged for deployment. We expect this number to grow.

Table I presents the results per optimization rule. *ObsoleteLayoutParam* was found in 21% of projects (156 refactorings), followed by *Recycle* which was found in 16% (58 transformations). *DrawAllocation* and *Wakelock* only showed marginal impact.

*ObsoleteLayoutParam* and *Recycle* are frequent as they affect common Android API usage that can be found in most projects (e.g., database cursors). Leafactor is particularly interesting for these cases.

It was expected that developers were already aware of *DrawAllocation*. Still, we were able to manually spot allocations that were happening inside a drawing routine. Nevertheless, those allocations used dynamic values to initialize the object. In our implementation, we scope only allocations that will not change between iterations. Covering those missed cases would require updating the allocated object in every iteration. While spotting these cases is relatively easy, refactoring would require better knowledge of the class that is being instantiated. Similarly, *WakeLocks* are very complex mechanisms and fixing all misuses still requires further work.

In the case of *ViewHolder*, although it only impacted 4% of the projects, we believe it has to do with the fact that 1) some developers already know this pattern due to its performance impact, and 2) many projects do not implement dynamic list views. *ViewHolder* is the most complex pattern we have in terms of lines of code (LOC) — a simple case can require changes in roughly 35 LOC. Although changes are easily understandable by developers, writing code that complies with *ViewHolder* pattern is not intuitive.

## IV. Conclusion

Empirical results showed that integrating Leafactor in the development stack of Android projects will help developers ship energy efficient code. The toolset works with Java and XML files. It was able to successfully perform 222 refactorings that were combined in 59 PRs in the Github repositories of the apps. We plan to distribute Leafactor as a standalone application. At the moment, the toolset is still in development but XML and Java refactorings' source is already publicly available[5][6].

## References

[1] L. Cruz and R. Abreu, "Performance-based guidelines for energy-efficient mobile applications," in *Submitted to MOBILESoft'17*.
[2] C. Sahin, L. Pollock, and J. Clause, "From benchmarks to real apps: Exploring the energy impacts of performance-directed changes," *Journal of Systems and Software*, 2016.
[3] D. Li and W. G. Halfond, "Optimizing energy of http requests in android applications," in *Proc. of DeMobile'15*. ACM.
[4] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of android applications," in *Proc. of ICSME'14*. IEEE.
[5] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," in *Proc. of MobileSoft'16*, ser. MOBILESoft '16.
[6] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proc. of ESEM'14*. ACM.

[5]XML: https://github.com/luiscruz/android-view-refactor/
[6]Java: https://github.com/luiscruz/AutoRefactor