iPerfDetector: Characterizing and Detecting Performance Anti-patterns in iOS Applications

Sara Seif Afjehei

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

January 2019

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Sara Seif Afjehei**

Entitled: **iPerfDetector: Characterizing and Detecting Performance Anti-patterns in iOS Applications**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Charalambos Poullis

_____ Examiner
Dr. Yann-Gaël Guéhéneuc

_____ Examiner
Dr. Weiyi Shang

_____ Supervisor
Dr. Tse-Hsun Chen

Approved by _____
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2019

_____
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

iPerfDetector: Characterizing and Detecting Performance Anti-patterns in iOS Applications

Sara Seif Afjehei

Performance issues in mobile applications (i.e., apps) often have a direct impact on the user experience. However, due to limited testing resources and fast-paced software development cycles, many performance issues remain undiscovered when the apps are released. As found by a prior study, these performance issues are one of the most common complaints that app users have. Unfortunately, there is a limited support to help developers avoid or detect performance issues in mobile apps.

In this thesis, we conduct an empirical study on performance issues in iOS apps written in Swift language. To the best of our knowledge, this is the first study on performance issues of apps on the iOS platform. We manually studied 235 performance issues that are collected from four open source iOS apps. We found that most performance issues in iOS apps are related to inefficient UI design, memory issues, and inefficient thread handling. We also manually uncovered four performance anti-patterns that recurred in the studied issue reports. To help developers avoid these performance anti-patterns in the code, we implemented a static analysis tool called iPerfDetector. We evaluated iPerfDetector on eight open source and three commercial apps. iPerfDetector successfully detected 34 performance anti-pattern instances in the studied apps, where 31 of them are already confirmed and accepted by developers as potential performance issues. Our case study on the performance impact of the anti-patterns shows that fixing the anti-pattern may improve the performance (i.e., response time, GPU, or CPU) of the workload by up to 80%.

Related Publications

The following publication is waiting for the second round of review:

- S. Seif, T. Chen, N. Tsantalis, iPerfDetector: Characterizing and Detecting Performance Anti-patterns in iOS Applications, In *Empirical Software Engineering Journal (EMSE), 2019. [Major Revision]*

Dedication

To my dear husband, whose encouragement, support and affection helped me through this journey.

To my parents, who are the reasons of what I have become today.

Acknowledgments

I would like to thank my supervisor, Dr. Tse-hsun (Peter) Chen, whom without his persistent help and his guidance, this dissertation would not have been possible.

I would also like to thank my committee members, for their suggestions and comments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of mobile phone applications (i.e., apps) has increased significantly in recent years. As of 2018, there are almost six million apps on the two most popular app stores: Google Play Store and Apple's App Store (Statista, 2018). Such large number of apps makes the app stores highly competitive, since there exist many apps with similar functionalities. Hence, in addition to functional requirements, one important aspect that may affect users' perception of an app is the app's performance (Khalid, Shihab, Nagappan, & Hassan, 2015; Liu, Xu, & Cheung, 2014).

Performance issues in mobile apps usually have a direct impact on the users. Prior studies (Hu, Bezemer, & Hassan, 2018; Khalid et al., 2015) found that performance issues are one of the most common complaints that app users have. However, existing research often focuses on studying performance issues in the system applications (e.g., enterprise applications or web servers) (Chen et al., 2014; Grechanik, Fu, & Xie, 2012; Jin, Song, Shi, Scherpelz, & Lu, 2012a; Nistor, Chang, Radoi, & Lu, 2015; Nistor, Song, Marinov, & Lu, 2013). Such performance issues may not be applicable to mobile apps, since mobile apps have different characteristics compared to system applications (Syer, Nagappan, Hassan, & Adams, 2013). For example, mobile phones have limited resources (e.g., battery and network), and mobile apps are usually UI-driven applications. A recent study by Liu et al. (2014) aimed to characterize performance issues in Android apps. However, the performance issues in Android may not be generalizable to iOS apps due to the differences in the two platforms and the used programming languages (Hu et al., 2018). Moreover, most prior research on mobile app APIs or code smells focused only on the Android platform (Martin, Sarro,

Jia, Zhang, & Harman, 2017); even though iOS is the second largest mobile platform in the world[1].

Therefore, to provide a better understanding of performance issues in mobile apps, we focus our study on iOS. We conducted an empirical study on 235 performance issue reports in four open source iOS apps written in the Swift programming language (i.e., Firefox, WordPress, Wire, and Charts) to study common types of performance issues. We also manually uncovered and documented four types of performance anti-patterns that recurred in the studied issue reports. Finally, we implemented a static analysis tool, called iPerfDetector, to detect the uncovered anti-patterns in Swift files. We collaborated with our industrial partner, and evaluated iPerfDetector on three commercial and eight open source apps. In total, iPerfDetector detected 34 performance anti-pattern instances, where 32 of them we manually verified as true positives. We reported these 32 anti-pattern instances to developers, and 31 of them are confirmed and accepted by developers. Our case study on the performance impact (i.e., in terms of response time, GPU, and CPU usage) of the anti-pattern shows that fixing the anti-pattern may improve the performance of the workload (e.g., scrolling a table in the app) by 5.8% to 80%. iPerfDetector received positive feedback from both commercial and open source developers, and it is now used by our industrial partner to ensure the performance of their iOS apps.

To the best of our knowledge, this is the first study on performance issues in iOS apps.

1.1 Thesis Contributions

- We found that inefficient UI design, memory issues, and inefficient thread handling are the main causes of performance issues.
- We provided a detailed discussion on four manually-uncovered iOS performance anti-patterns.
- We also implemented a static analysis tool, iPerfDetector, to detect instances of the uncovered anti-patterns.
- iPerfDetector detected 34 anti-pattern instances in eight open source and three commercial

[1] https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

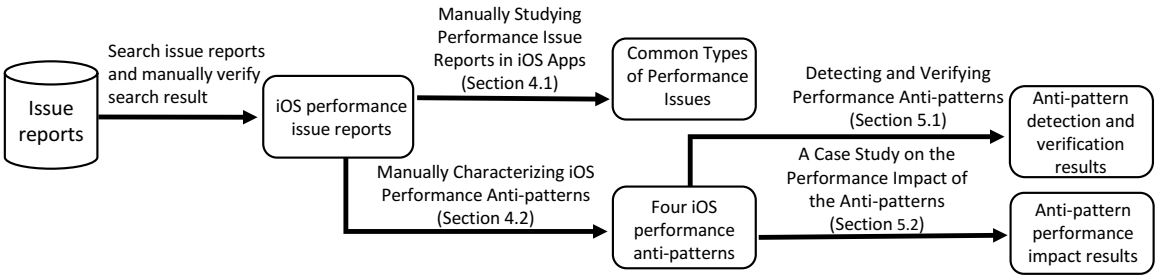

Figure 1.1: An overview of our study.

apps, where 31 of them are already confirmed and accepted by developers as potential performance issues.

- Our case study shows that fixing the anti-patterns may improve the performance (i.e., GPU usage, CPU usage and response time) of the workload by up to 80%.

1.2 Thesis Overview

Figure 1.1 shows an overview of our study.

Chapter 2 surveys related work. Chapter 3 briefly introduces Swift language and it's popularity among iOS developers. It also explains the main design pattern used for developing iOS apps and give an overview of the background of iOS app development. In Chapter 4 we discuss our manual study on iOS performance issue reports of four open source apps, and the root causes extracted from the study of issue reports. In this chapter, we also introduce the performance anti-patterns that we manually uncovered. Every anti-pattern is discussed in detail, given a description, an example of a real-world case, the approach developers take facing the issue, and the possible solutions to solve the issue. Chapter 5 embraces details of iPerfDetector on how it detects the anti-patterns, in the source code of iOS apps. Furthermore, it gives the results of running iPerfDetector on both commercial and open source applications. Finally, it provides a two step approach we took, in order to verify the detection results: examining the detected anti-patterns manualy, and asking for open source and commercial developers' feedback. A case study on how effective can fixing the anti-patterns be, is also discussed in this chapter. Chapter 6 discusses threats to validity of our study. Lastly, Chapter 7 concludes the thesis.

Chapter 2

Literature Review

In this chapter, we discuss the related work along three dimensions: 1) detecting performance anti-patterns in software applications, 2) code smells in mobile apps, and 3) empirical studies on Swift-based applications.

2.1 Detecting Performance Anti-patterns in Software Applications

Much research effort has been devoted to understanding and detecting performance anti-patterns in system applications (e.g., enterprise or database systems). According to Martin Fowler et al. (1999), a code smell is any characteristic in the code (e.g., code design or pattern) that may indicate a deeper problem. Zaman, Adams, and Hassan (2012) empirically studied the differences between performance and non-performance issues, and suggested that performance issues have their unique characteristics that would require specialized detection approaches. Nistor, Jiang, and Tan (2013) further supported the finding, where they found that, unlike functional issues, most performance issues are uncovered through code reasoning with tool supports. Jin, Song, Shi, Scherpelz, and Lu (2012b) empirically studied 109 performance issues in system applications (e.g., MySQL) and derived rules for detecting performance anti-patterns. Nistor et al. (2015); Nistor, Song, et al. (2013) proposed both dynamic and static approaches to detect inefficient memory access in loops. Chen, Shang, Hassan, Nasser, and Flora (2016); Chen, Shang, Jiang, et al. (2016); Chen et al. (2014) proposed a series of approaches that help developers detect database access performance anti-patterns

Table 2.1: Comparison between our study and prior studies in terms of the studied performance anti-patterns.

Prior Study	Anti-Pattern Category				Platform
	UI	Memory	Multi-threading	Other	
Our study	Applying UI blurring effect inefficiently, Applying UI transparency inefficiently	Retain Cycle	Accessing Data Models on the UI thread, Updating UIs on back- ground threads	-	iOS
Code Smells in iOS Apps: How do they compare to Android? (Habchi, Hecht, Rouvoy, & Moha, 2017)	-	Ignoring Low-Memory Warning	Blocking The Main Thread	Heavy Enter-Background Tasks, Download Abuse	iOS
Study and Refactoring of Android Asynchronous Programming (Lin, Okur, & Dig, 2015)	-	-	AsycnTask (used for encapsulating short running tasks) to IntentService (good choice for long-running tasks)	-	Android
Performance-based Guidelines for Energy Efficient Mobile Applications (Cruz & Abreu, 2017)	ViewHolder, ObsoleteLayout-Param, DrawAl-location, Obso-leteLayoutParam, Overdraw	Recycle, DrawAl-location	-	WakeLock, Unuse-dResources	Android
On Death, Taxes, and Sleep Disorder Bugs in Smartphones (Jindal, Pathak, Hu, & Midkiff, 2013)	-	-	-	No-Sleep Bugs, Under-Sleep Bugs, Over-SleepBugs	Android
Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices (Pathak, Hu, & Zhang, 2011)	-	-	-	OS processes, Config-uration changes, Loop bug, Immortality bug	Android
Characterizing and Detecting Performance Bugs for Smartphone Applications (Liu et al., 2014)	Wasted computation for invisible GUI	-	Lengthy operations in main threads	Frequently invoked heavy-weight callbacks	Android
An Empirical Study of the Performance Impacts of Android Code Smells (Hecht, Moha, & Rouvoy, 2016)	-	-	-	Member Ignoring Method (MIM), Internal Getter/Setter (IGS), HashMap Usage (HMU)	Android
Saving Energy on Mobile Devices by Refactoring (Gottschalk, Jelschen, & Winter, 2014)	Backlight	-	-	Data Transfer, Binding Resources Too Early, Statement Change, Cloud Computing	Android
Lightweight Detection of Android-Specific Code Smells: The aDoctor Project (Palomba, Di Nucci, Panichella, Zaidman, & De Lucia, 2017)	-	Leaking Inner Class, Leaking Thread, No Low Memory Resolver	Blocking The Main Thread	Data Transmission Without Compression, Debuggable Release, Durable Wakelock, Inefficient Data Format and Parser, Inefficient Data Structure, Inef-ficient SQL Query, Internal Getter and Setter, Member Ignoring Method, Public Data, Rigid Alarm Manager, Slow Loop, Unclosed Closable	Android
A Study and Toolkit for Asynchronous Programming in C# (Okur, Hartveld, Dig, & Deursen, 2014)	-	-	Fire & Forget Methods, Unnec-essary async/await Methods, Using Long-running Oper-ations under Async Methods , Unnec-essarily Capturing Context	-	Windows Phone

in enterprise applications.

2.2 Code Smells in Mobile Applications

Most other prior studies focus on studying code smells in Android apps. Table 2.1 further clarifies the differences and similarities between our work and prior work on mobile performance code smells (i.e., anti-patterns). Most anti-patterns discussed in the prior studies are different than the ones we studied. We classify the anti-patterns based on the three main categories that we manually uncovered when studying the issue reports. We find that although some anti-patterns may be similar to the ones that we studied (e.g., blocking the main thread), prior studies are mostly all in the Android platform and our study revealed some unique problems that were not studied before. Below, we discuss each related work in detail.

To the best of our knowledge, there is only one prior study on iOS code smells. Habchi et al. (2017) conducted a study on the Object-Oriented code smells in iOS apps and compare with that of Android apps. They found that iOS apps are less prone to Object-Oriented code smells. They also documented several iOS code smells (e.g., abuse of the singleton design pattern, ignoring low memory warning, and downloading the same files multiple times) based on reading Apple's iOS documentation, developer blogs, and Stack Overflow posts. However, most of the code smells that they found are different from the ones that we studied in this thesis, and they did not provide an evaluation of the detection results. Also, different than their study, we studied bug reports from popular open source projects to find the real-world root-causes of anti-patterns.

Hecht et al. (2016) conducted an empirical study on the impacts of three Android code smells (i.e., private getter/setter, member ignoring method, and using hashmap instead of array map). They found that fixing some code smells may reduce the number of delayed frames, improve the UI drawing performance, or improve energy efficiency in some cases. However, some of the performance problems may no longer exist in newer versions of Android. We study a broader range of anti-patterns and also provide a tool to detect those anti-patterns. Reimann, Brylski, and Aßmann (2014) compiled a list of common code smells and refactoring solutions in Android apps. Palomba et al. (2017) implemented a tool to detect Android-related code smells that are presented in the study by

Reimann et al. (2014). All of the anti-patterns they discussed are different from the ones we found in our study. Mannan, Ahmed, Almurshed, Dig, and Jensen (2016) conducted a literature survey to identify common code smells in mobile apps. They found that there was no prior study on code smells in iOS apps. They further compared the types and distributions of code smells in mobile and desktop applications. Prior studies (Gottschalk et al., 2014; Morales, Saborido, Khomh, Chicano, & Antoniol, 2017) also studied the relationship between the above-mentioned code smells and energy consumption in Android apps. Jindal et al. (2013) study wakeLocks and the scope of the code paths which should be protected by wakeLocks and look for cases of where it is over-used, which increases battery consumption, or under-usage which affects apps correctness. Pathak, Hu, and Zhang (2012) developed an energy profiler named eprof, to help developers find instances of energy bugs caused by wakeLocks. In another study, Pathak et al. (2011) also provided the taxanomy of energy bugs (ebugs) in a variety of categories, such as hardware ebugs, Software ebugs, and ebugs caused by external conditions. Under software energy bugs category, they focused on bugs caused by OS processes and configuration changes, and applications and framework ebugs. such as unnecessary loops (i.e., app repeatedly trying to connect to the remote server) and apps respawning in spite of being killed by users. These two studies also focus on Android platform.

The energy code smells introduced by these studies, are only validated in Android platform, and the results cannot be generalized to iOS platform. Moreover, we believe that our performance case study may help identify which types of code smell or anti-pattern may consume more energy (e.g., base on the CPU or GPU usage), without needing to deploy complex hardware environment.

Morales et al. (2017) proposed a search-based approach to provide refactoring suggestion to developers. They found that fixing the code smells may reduce the number of delayed frames and improve the UI drawing performance. A study by Cruz and Abreu (2017) also focuses on performance related UI intensive anti-patterns, such as ViewHolder, ObsoleteLayoutParam, and also memory issues such as Recycle and DrawAllocation, all in Android mobile apps. They selected the most common performance related suggestions given by lint, and published a benchmarking suite for energy consumption in Android.

Lin et al. (2015) implemented a refactoring tool for fixing an async programming bad practice in Android apps. The anti-patterns include improper uses of primary construct, AsyncTask, which

can lead to memory leaks, results getting lost, and waste of energy.

A study by Okur et al. (2014) focuses on uses and misuses of asynchronous constructs in Windows Phone applications. They also developed a set of tools for refactoring 2 of the common async programming bad practices. The anti-patterns studied in this thesis are specific to async constructs in Windows Phone.

The study by Liu et al. (2014) is the closest to the problems that we studied in this thesis. They manually studied 70 performance issue reports in Android apps and developed a static analysis tool to detect two manually-uncovered performance anti-patterns. In this thesis, we focus our study on another popular mobile app platform, iOS. iOS apps were rarely studied in prior research, even though they have different characteristics compared to Android apps (Hu et al., 2018). Different from the study by Liu et al. (2014), we conducted our study on 235 performance issue reports. We found that many performance anti-patterns in the studied iOS apps are related to inefficient UI design (e.g., using inefficient UI effects such as blurring) and thread handling. We also provided a case study to illustrate the performance impact of the studied anti-patterns. To the best of our knowledge, we are the first study on performance issues in iOS apps. Future studies are needed to further help mobile developers with not only Android, but also iOS app development.

2.3 Empirical Studies on Swift-based Applications

Cassee, Pinto, Castor, and Serebrenik (2018) conducted a study on most frequently asked questions about Swift programming language on Stack Overflow. They found that 14.6% of the questions are about data storage, 12.8% are related to UI actions, and around 9% are related to multi-threading issues. Their results also echo with the importance of the anti-patterns we studied. They also surveyed iOS developers and discovered that they found Swift easy to adopt. Rebouças et al. (2016) studied open-source repositories to find out to what extent Swift developers apply best practices of error handling provided by guidelines and tutorials.

In the next Chapter, we will briefly look at Swift language, and the main design pattern used for iOS developement.

Chapter 3

Background

iOS is a closed-source operating system developed by Apple for mobile devices such as iPhone and iPad. To date, iOS is one of the most widely used mobile operating systems in the world, and iOS apps generate much higher revenues compared to Android apps (Statistia, 2018). To provide better usabilities and functionalities in the app, developers usually rely on interacting with the APIs provided by the iOS Software Development Kit (SDK). Initially, iOS apps were all implemented in the Objective-C programming language. In 2014, Apple introduced the Swift programming language as an alternative for iOS app development. Compared to Objective-C, Swift offers a cleaner syntax that makes the apps easier to maintain (Apple, 2018a). Since then, Swift has become more popular among developers compared to its predecessor (RedMonk, 2018). Hence, in this thesis, we focus our study on the iOS apps that are written in Swift.

iOS apps are often developed using the Model-View-Controller (MVC) design pattern. The MVC pattern separates the app's data access and business logic from the visual presentation. The layers in MVC help abstract the underlying device differences, such as screen sizes, and simplifies app development. Below, we briefly discuss the background of iOS development based on three major software layers in an app: model, view, and controller.

Model Layer. Model layer is a set of UI independent objects, responsible for managing the persisted objects in the database. Developers often use the Core Data persistent framework that is provided by iOS SDK to manage object persistence in order to access the model layer (Guide, 2017). By using Core Data, developers can directly access or modify object states in a database through API

calls in the app's model layer, which reduces the complexity of the code (Apple, 2018d).

View Layer. View layer is responsible for displaying the user interface (UI) components and for users to interact with the app (e.g., by tapping on buttons). Developers can call subclasses of the *UIKit* class (e.g., *UILabel* or *UIButton* class) to design and organize UI components. Developers may also call APIs in the iOS SDK to customize UI components or add different UI effects (e.g., transparency or blurring). Since mobile apps are UI-driven, by default, every function call will be executed on the main thread (i.e., UI thread) (Apple, 2018b). If the main thread is blocked or delayed due to heavy computation, users may experience poor UI responsiveness.

Controller Layer. The controller layer implements the business of an app. For example, how often should the UI be updated/refreshed, or what should be stored in the model layer when users tap a button. In other words, the controller layer translates actions in the view layer to the corresponding actions in the model layer and vice versa (Apple, 2018e). To ensure that the UI component is responsive when processing the business logic, iOS provides multi-threaded APIs. Heavy operations (e.g., complex calculation) or long-running processes (e.g., access the persisted objects in the model layer) can be executed on a background thread, instead of on the main thread.

In the next chapter, we will discuss our manual study on iOS performance issue reports of four open source apps, and the root causes extracted from the study of issue reports. We also introduce the performance anti-patterns that we manually uncovered.

Chapter 4

Manual Study

In this chapter, first, we discuss our process of selecting and manually studying performance issue reports in open source iOS apps. Then we document recurrent iOS performance anti-patterns that we found during our manual study and provide a detailed discussion on the iOS performance anti-patterns we found.

4.1 Manually Studying Performance Issue Reports in iOS Apps

Motivation

In order to understand the common causes of performance problems in iOS apps, we manually study issue reports in open-source iOS apps. Understanding the common performance problems may help developers avoid the problems and inspire future research.

Approach

We conducted our manual study on open source iOS apps. We selected 21 popular open source iOS apps according to Medium (2016) and the top 20 trending open source Swift repositories on GitHub (Github, 2018). Then, we applied four selection criteria on the apps. First, the selected apps should be related to iOS mobile app development, because Swift is a general purpose language that can be used for tasks such as implementing web servers. Second, a candidate app should use an issue report system, since we need to study the issue reports to uncover the types of performance

Table 4.1: An overview of the studied apps.

System	Category	Num. of stars on GitHub	LOC	Num. of issue reports	Num. of perf bug reports
Firefox	Utilities	8K	39K	1.1K	88
Wordpress	Productivity	2.3K	85K	4.6K	82
Wire	Business	2.4K	61K	308	32
Charts	Chart Library	18K	31K	2.9K	33

issues. Third, the candidate app should contain issue reports that are related to performance issues. Finally, the app should be actively maintained (e.g. developers were active in the last 10 days, at the time of writing). Since mobile APIs are constantly evolving (McDonnell, Ray, & Kim, 2013), we want to ensure that new problems related to API usages are included in our study. We ended up with four open source iOS apps that met our criteria. Table 4.1 shows an overview of the studied apps. Mozilla Firefox client app (or simply Firefox) is a free and open-source web browser written in Swift[1]. We used the master branch of the Firefox iOS browser, which is written in Swift 4.2. WordPress is an open-source content management system (CMS)[2]. Wire is a cross-platform and encrypted instant messaging client[3]. Charts is a library for drawing various types of charts for iOS apps[4]. In general, the studied apps cover different categories, are popular (with 2.3K to 18K stars on GitHub), and are large in size (up to 39K LOC).

We used the keywords: "slow", "performance", "thread" and "memory" to search for performance-related issue reports. These keywords are commonly used in prior studies or are known as common performance problems (Chen, Shang, Jiang, et al., 2016; Jin, Song, Shi, Scherpelz, & Lu, 2012c; Liu et al., 2014; Smith & Williams, 2001). In total, we found 960 issue reports in the studied apps that contain the keywords. We manually go through all 960 issue reports to identify the relevant issue reports. We first remove all the issue reports that do not have a fix (e.g., marked as *unresolved* or won't fix). Then, we remove the false positives that are caused by the keywords we used. For example, by using the keyword "thread", we found that in some cases, developers may be referring to

[1]https://github.com/mozilla-mobile/firefox-ios
[2]https://github.com/wordpress-mobile/WordPress-iOS
[3]https://github.com/wireapp/wire-ios
[4]https://github.com/danielgindi/Charts

"issue thread". After the manual filtering process, we ended up with 235 performance issue reports for our manual study.

Table 4.1 shows the number of performance issue reports that we studied in each app. To categorize the studied performance issues based on their effect, we manually studied all of the 235 issue reports and all associated information (e.g., pull requests, code changes, and developers' discussions). We determined the categories of the anti-patterns based on their root causes. Our categorization process is as follows. We started our manual study with no specific category in mind and we took a note on the root cause of the problem. After the author manually studied the issue reports, she created the categories based on the root causes that we found, and verified the categorization.

Results

In general, we found three common types of performance issues in the studied iOS apps: inefficient UI design, memory issues, and inefficient thread handling. Note that since one issue may be assigned to more than one type (e.g., an issue report may be related to two different types of performance issues), the accumulated percentage may not be exactly 100%. There is also around 15% of the issues that do not belong to these three types. We found that such issues are often related to app-specific performance optimization, such as using more efficient data structures, reducing the number of log lines that are printed (Chowdhury, Di Nardo, Hindle, & Jiang, 2018), or optimizing database queries.

Below, we summarize the issues that we found in each type. To encourage future research on this topic, we also release our manually annotated data online (Afjehei & Chen, 2018).

Inefficient UI Design: We found that 20% of the issue reports tried to address inefficient UI design. One common UI-related performance issue that we studied happens when users scroll through a table. For example, developers may be using computationally intensive text styling in table cells (Wire bug 1751), dynamically calculating the height of each cell in a table instead of using a fixed height (Wire bug 5672), or displaying emojis in table cells (Wire bug 267). Such heavy UI-related operations in table cells may make scrolling a table sluggish. Another common problem that we saw is related to applying heavy UI effects. For example, developers may be applying UI blurring effect repetitively in a loop (Firefox bug 1221118). Since the main thread is responsible for UI

rendering, applying heavy UI effects in a repeating fashion on the main thread will make the app temporarily unresponsive (i.e., the main thread is blocked to compute the effects). In summary, many performance issues that we saw are related to inefficient usage of iOS's UI-related APIs.

4.2 further discusses the performance anti-patterns that are related to computing UI effects.

Memory Issues: We found that 34% of the studied issue reports are related to memory issues. Furthermore, about 44% of the memory related issues address memory leaks. In particular, retain cycles are the most common cause of memory leaks that we found. Retain cycle happens when two or more objects keep references to each other (i.e., a reference cycle). This may prevent the objects from being garbage collected even if they cannot be accessed elsewhere from the heap. Other types of memory problems include memory leaks (e.g., in Firefox bug 1278006, UI objects are not released even if they are not visible to the user anymore) and out of memory errors (e.g., WordPress bug 7892).

Inefficient Thread Handling: We found that threading-related issues are the most common causes of performance issues in our studied issue reports (36%). As mentioned in Chapter 3, there is a main thread (i.e., UI thread) and background threads in iOS. We found that most issues that we studied are related to inefficient uses of iOS's threading APIs. For example, developers may forget to execute heavy computation (e.g., data accesses) in a background thread, which may result in causing the UI to be temperately unresponsive (e.g., as discussed in WordPress 578 and Firefox 532).

We found that there are three main types of performance issues: inefficient UI design (20%), memory issues (34%), and inefficient thread handling (36%). We also found that most problems we studied are related to how developers use iOS-specific APIs (e.g., UI effect or threading).

4.2 Manually Characterizing iOS Performance Anti-patterns

Motivation

During our manual study of the iOS performance issue reports, we found several recurring code patterns that are common causes of the studied iOS performance issues. Hence, in this section, we document the recurring code patterns that we uncovered into four iOS performance anti-patterns.

Table 4.2: Description of the manually-uncovered iOS performance anti-patterns and their corresponding category.

Category	Anti-pattern	Description	Abbr.	No of Issues
Inefficient Thread Handling	Accessing Data Models on the UI thread	Developers access the model layer (i.e., access data) in the UI thread, which may cause bad user experiences due to unresponsive UI.	DM	17
	Updating UIs on background threads	UI updates (e.g. update label text) that are executed on a background thread, which may lead to app crashes.	UIBG	30
Inefficient UI Design	Applying UI blurring effect inefficiently	Adding computationally expensive blurring effect to complex UI components (e.g., table with multiple cells).	BLEF	4
	Applying UI transparency inefficiently	Adding computationally expensive transparency effect to complex UI components (e.g., table with multiple cells).	TREF	4
Memory Issues	Retain Cycle	The condition when two objects keep a reference to each other and are retained, which creates memory leaks (Apple., 2012).	RETAIN	22

Approach

In our study, we tried to derive the performance anti-patterns that are specifically related to iOS development and occur several times. To derive the anti-patterns, we manually analyzed the fixes in the issue reports and took notes regarding code changes (e.g., what did developers change to fix the problem). Then, we went through the note and manually summarized and extracted the code changes as the anti-pattern.

Results

Table 4.2 summarizes the manually-uncovered iOS performance anti-patterns and the number of anti-pattern instances we found. Below, we discuss each anti-pattern in detail. To provide more detailed information and breakdown of each studied iOS performance anti-pattern, we discuss each pattern using the following template:

Description. Description of when and how the anti-pattern would take place.

Example. An example of the anti-pattern from the real-world studied apps.

Examples of Developer Awareness. We summarize some of developers' discussion that we found in the issue reports. We also searched on developer forums (e.g., Stack Overflow) and documents (e.g., Apple official documents) and summarize our findings on how the anti-pattern may affect other developers.

Possible Solutions. We discuss possible solutions to resolve the anti-pattern.

Accessing Data Model on the UI thread

Description. Mobile apps that implement the MVC design pattern usually require a model layer to manage user information or application-specific data. In iOS app development, developers often use Core Data for data access (Apple, 2018g). Core Data is an object persistent framework provided by Apple that allows developers to manage the model layer and to interact with persisted objects (Guide, 2017). However, such data accesses can be time-consuming and computationally intensive. Hence, if Core Data is used inside the UI thread (i.e., the main thread), the UI may be temperately unresponsive until the data access is finished.

Example. As an example, there is a discussion on fixing this performance anti-pattern on WordPress [5].

The following function, *save*, is persisting the changed data to the model layer using Core Data (line 4).

```
1 (void)save {
2 NSError *error;
3 // saving changes to the model layer by calling managedObjectContext,
    which is the API for using Core Data
4 if (![[self managedObjectContext] save:&error]) {
5 DDLogError(@"Unresolved Core Data Save error %@, %@", error, [error
    userInfo]);
6 exit(-1);
7 }
8 }
```

[5] https://github.com/wordpress-mobile/WordPress-iOS/pull/578

```
9  ...
10 (void)remove { // will be executed on the main thread
11 ...
12 [self save]; // calling data access through Core Data
13 }
```

However, this *save* function is called inside another function named *remove* (line 12), which is executed on the main thread. As a result, the UI will be temperately unresponsive due to having heavy data accesses on the main thread (i.e., because of a chain of function calls that result in executing the *remove* function).

Developers Awareness. As mentioned in Apple's iOS development document (Apple, 2018g), any data processing should be avoided on the main thread. However, we still found that sometimes such anti-patterns may be difficult to find due to the complexity of the software design and interactions of function calls [6] (Chen, Shang, Hassan, et al., 2016; Chen et al., 2014). Namely, developers may not notice that the function that they call may eventually result in data access.

Possible Solutions. One way to address this iOS performance anti-pattern is that developers can leverage APIs that are provided by Apple to execute the data access on a background thread asynchronously (Apple, 2015). Another solution is to add a separate layer in the code to take care of *NSManagedObjectContext(s)* (i.e., call to Core Data). For example, developers may use the parent-child managed object contexts, where changes to the children objects will only be persisted when the parent object is persisted. Therefore, developers only need to make sure that calls to the parent objects are executed in background threads (i.e., developers do not need to make scatter code changes to make sure every child access is executed in the background thread). The first solution would require fewer changes to the code, but may introduce maintenance difficulties due to scattering code changes. On the other hand, the second solution may require significant code refactoring for the added abstraction layer. Developers may need to decide which solution is more suitable for the design of their app after considering the trade-offs, as indicated by WordPress developers [7].

[6] https://github.com/wordpress-mobile/WordPress-iOS/pull/578
[7] https://github.com/wordpress-mobile/WordPress-iOS/pull/578

Updating UI Controls on Background Threads

Description. We observed that sometimes developers may perform inappropriate performance optimization by moving heavy UI operations to a background thread. As an example, Firefox developers had a discussion regarding moving a UI rendering computation to a background thread in order to make the UI more responsive [8]. However, such optimization may result in some problems. As part of the iOS architectural design, updating UI (e.g., resize or refresh) on a thread other than the main thread can be problematic. Updating UI in a background thread can cause not only performance issue, but may also cause functional problems. According to Apple's development guideline, this anti-pattern may lead to problems such as missed UI updates, incorrect UI displays, or even crashes (Apple, 2017). Developers also discuss that such issues can cause delays in UI updates [9].

Example. Consider an example from WordPress [10]. In the code snippet below, there is a class function *resizeGalleryImageURL*, which resizes and scales the image (the implementation detail is omitted for better readability). The class function *resizeGalleryImageURL* is then used in another function named *formatContentString* (line 11).

```
1  // resize an image
2  public class func resizeGalleryImageURL(_ string: String, isPrivateSite
       isPrivate: Bool) -> String {
3  ...
4  }
5
6  ...
7
8  public class func formatContentString(_ string: String, isPrivateSite
       isPrivate: Bool) -> String {
9  ...
10 // will result in resizing the image
11 content = resizeGalleryImageURL(content, isPrivateSite: isPrivate)
```

[8] https://github.com/mozilla-mobile/firefox-ios/pull/1215/commits/4ba952a58bf34ecfa555a8058f7c9a20c154997d
[9] https://stackoverflow.com/questions/28137380/updating-ui-from-background-thread-swift
[10] https://github.com/wordpress-mobile/WordPress-iOS/pull/7864

```
12 return content
13 }
```

The function *formatContentString* is called in the function *sanitizeCommentContent* (line 4 in the code snippet shown below), which is then called inside the *performBlock* of *managedObjectContext* (line 13). *managedObjectContext.performBlock* is a way of executing operations in an asynchronous fashion (i.e., in a background thread, when a private managed object context is defined). As a result, this caused WordPress to crash due to resizing an image in background threads[11].

```
1  (NSString *)sanitizeCommentContent:(NSString *)string
       isPrivateSite:(BOOL)isPrivateSite{
2  ...
3  // calling formatContentString
4  content = [RichContentFormatter formatContentString:content
       isPrivateSite:isPrivateSite]
5  ...
6  }
7
8  ...
9
10 [self.managedObjectContext performBlock:^{
11 ...
12 // May result in calling resizeGalleryImageURL in a background thread,
       which led to application crashes in WordPress
13 remoteComment.content = [self
       sanitizeCommentContent:remoteComment.content
       isPrivateSite:isPrivateSite];
14 ...
15 }];
```

Examples of Developer Awareness. In September 2017, Apple released a dynamic analysis tool called the Main Thread Checker that can detect UI API calls in background threads and provides

[11] https://github.com/wordpress-mobile/WordPress-iOS/pull/7864

warnings on the Xcode console (Apple, 2017). Even though such a tool is available, during our manual study, we still found that developers may not fully utilize the tool, and thus, resulted in app crashes. As an example, we counted the number of reports that were opened after the tool was released (by comparing the release date of the tool and the issue creation date), and found that about half of the anti-pattern instances in WordPress were reported by users after the tool was released. Namely, developers did not fully utilize the tool during development, so the anti-pattern instances remained unfixed when the app was released. One reason may be that the Main Thread Checker is only enabled when running apps with the Xcode debugger (Apple, 2017). Developers may not exercise the anti-pattern instances in the code when using the debugger, so the instances may not be detected. Compared to the dynamic approach provided by Main Thread Checker, iPerfDetector provides a lightweight static approach and does not introduce overhead during testing. Another reason may be that such warnings are difficult to notice due to hundreds or even thousands of other messages on the console (Chen et al., 2017; Jiang, Hassan, Hamann, & Flora, 2008).

Possible Solutions. All UI operations should be executed on the main thread. This can be done by dispatching the UI update operation to the main thread by calling the update operations in code blocks such as *DispatchQueue.main.sync{...}* (the code inside the code block would be executed in the main thread).

Applying UI Blurring Effect Inefficiently

Description. As mentioned in Chapter 3, UI-related tasks (e.g., refresh or update) are processed on the main thread (i.e., UI thread). As a result, any time-intensive operation in the main thread may cause the UI to be temporarily non-responsive, which results in bad user experience. We found that blurring effect can be computationally expensive, especially when the blurring effect is applied in a repeated fashion (e.g., applying the effect on each cell of a table or in a loop). Applying such blurring effect inefficiently may result in noticeable delays in the UI.

Example. Consider an example from Firefox[12]. In the code snippet below, we can see that an object instance of class *UIVisualEffectView* is instantiated (defines a blurring effect in line 4) and is referred by a variable named *backgroundEffect*. In the second function *init()* (line 13), where a

[12] https://bugzilla.mozilla.org/show_bug.cgi?id=1191058

cell is initialized, we can see that *backgroundEffect* is added (line 15) to each cell in a table (line 18) when initializing the table object. Hence, iOS needs to recalculate the blurring effect for each table cell. Applying the blurring effect repetitively is computationally expensive and can result in noticeable UI delays, especially when users are scrolling the table.

```
1  // configuring the blur effect
2  lazy var backgroundEffect: UIVisualEffectView? = {
3  ...
4  let blur = UIBlurEffect(style: UIBlurEffectStyle.Light)
5  let vib = UIVibrancyEffect(forBlurEffect: blur)
6  let effect: UIVisualEffectView? = DeviceInfo.isBlurSupported() ?
       UIVisualEffectView(effect: blur) : nil
7  ...
8  return effect
9  }()
10
11 ...
12
13 override init(frame: CGRect) {
14 ...
15 imageWrapper.addSubview(backgroundEffect)
16 ...
17 // adding blur effect to every cell in the table
18 contentView.addSubview(imageWrapper)
19 ...
20 }
```

Examples of Developer Awareness. In the above-mentioned example from Firefox, developers complained that "the blur we render for each top site item is causing noticeable slowdowns in frame rate of the application"[13]. We also found many other developers complained about the potential problems of applying blurring effect inefficiently during our manual study. In our study, we found

[13] https://bugzilla.mozilla.org/show_bug.cgi?id=1191058

that 33% of the UI-related performance issues reported in Firefox are related to applying UI blurring effect inefficiently.

Possible Solutions. To fix this anti-pattern, developers sometimes choose to remove the blurring effect if the effect is not important to the UI design. For example, one Firefox developer in the above-mentioned issue report asked "just curious, how important is it to use blurring instead of a flat or alpha faded color?"[14]. Another approach to fix the anti-pattern is to calculate the burring effect once and reuse the calculated effect for every cell, instead of recomputing the effect for each cell. However, sometimes developers may apply different blurring parameters to each cell in a table. In addition, if the screen needs to be frequently refreshed, calculating the blurring effect for every UI frame may still be expensive. In such cases, removing the blurring effect may be a better performance optimization approach.

Applying UI Transparency Effect Inefficiently

Description. Similar to blurring, we found that UI transparency effect may be computationally expensive if used inefficiently (e.g., applied repeatedly, which requires more computation). Such inefficient UI design may result in unresponsive UI and bad user experience. Although this anti-pattern is very similar to BLEF, as they are both color blending effects which should not be over used, the APIs and implementation details are different. We will further discuss their differences and detection algorithms in Chapter 5.

Example. Consider an example from Firefox. The UI transparency effect is configured and returned as the variable *faviconView* (an instance of the *UIImageView* class). Then, *faviconView* is used in the *init* function of a table cell (line 14), and the transparency effect is added to each cell in the table (i.e., by calling *addSubview(faviconView)*). When the table is refreshed (e.g., when scrolling), the transparency effect will be recomputed for each cell in the table and result in unresponsive UI.

```
1 lazy var faviconView: UIImageView = {
2 ...
3 // setting the transparency level
4 faviconView.layer.borderColor = UIColor(white: 0, alpha: 0.1).cgColor
```

[14] https://bugzilla.mozilla.org/show_bug.cgi?id=1191058

```
...
return faviconView
}()

...

override init(style: UITableViewCellStyle, reuseIdentifier: String?) {
...
// adding transparency effect to each table cell
contentView.addSubview(faviconView)
...
}
```

Examples of Developer Awareness. In the studied issue reports, we found that some developers did not know the performance overhead of the transparency effect until users reported the problems. Namely, developers did not completely understand the performance profile of the app. For example, a WordPress developer was informed by a user that: "[d]rawing is improved when views that have solid backgrounds are set to opaque."[15]. Our manual observation also highlights the needs of developing research tools to help developers improve the performance of their apps.

Possible Solutions. Similar to applying blurring effect inefficiently, the approach to fix this anti-pattern is to remove the UI transparency effect when the effect is not important to the UI design. As an example, Firefox developers even mentioned that they think avoiding partial transparencies, whenever the effect is not required, would be the best way to ensure the performance of the app[16]. Other developers on Stack Overflow also discussed that the overhead to compute transparency is high and should be avoided if possible[17].

[15] https://github.com/wordpress-mobile/WordPress-iOS/issues/419

[16] https://github.com/mozilla-mobile/firefox-ios/pull/1215/commits/4ba952a58bf34ecfa555a8058f7c9a20c154997d

[17] https://stackoverflow.com/questions/9270723/what-is-the-better-way-to-set-an-uiview-backgroundcolor-to-transparent

We found that the most common performance anti-patterns in iOS are: accessing data model on the UI thread, updating UI controls on background threads, applying UI blurring effect inefficiently, applying UI transparency effect inefficiently, and object references causing retain cycles.

In the next chapter, we will discuss how we detect the anti-patterns with the tool we developed, iPerfDetector, and provide information on how we verify the detected results. We also discuss a case study on the impact of each anti-pattern.

Chapter 5

Detection

In this chapter, first, we describe how we detect the uncovered anti-patterns in iOS apps. Second, we evaluate iPerfDetector on both commercial and open source apps. Finally, we conduct a case study to measure the anti-patterns' performance impact (i.e., GPU usage, CPU usage and response time).

5.1 Detecting and Verifying Performance Anti-patterns

Motivation

We found that many of the anti-patterns that we studied are related to the complex calling relationship among the user-defined functions and iOS-specific APIs that may be difficult to find without tool assistance (Chen, Shang, Hassan, et al., 2016; Liu et al., 2014). For example, developers may not notice that, after a chain of function calls, a function on the main thread will result in data accesses. Hence, to help developers improve the quality of their apps, we create a static analysis tool to detect the anti-patterns. In this section, we want to study whether our tool can detect the anti-patterns with a high precision.

Approach

We implemented a static analysis tool, called iPerfDetector, to detect instances of the uncovered anti-patterns. iPerfDetector analyzes the abstract syntax tree (AST) of the Swift code in iOS apps

to detect the anti-patterns. iPerfDetector first uses an open source tool called SwiftAST (SwiftAST, 2018) to parse the Swift code and generate the ASTs. Then, iPerfDetector applies different detection algorithms on the ASTs to detect the manually-uncovered iOS performance anti-patterns. Below we provide more details on how iPerfDetector detects each anti-pattern. Note that since there exist many mature and widely used tools for detecting memory issues in iOS apps (e.g., Facebook's Infer (Facebook, 2017) and Apple's Instruments (Apple, 2018c)), iPerfDetector focuses on detecting other types of performance anti-patterns.

Detecting Accessing Data Model on the UI thread. Developers use Core Data APIs by calling the class *NSManagedObjectContext* that is provided by the iOS SDK to interact with the persisted objects (Apple., 2017). Hence, every data access is made through this specific class. iPerfDetector first traverses all the classes in an app searching for objects that are instances of *NSManagedObjectContext*. Then, iPerfDetector applies points-to analysis to identify all the functions that contain data accesses (e.g., calling functions such as *save*, *fetch*, *insert*, or *delete* on the *NSManagedObjectContext* object). After this step, we obtain a list of functions that contain data access calls. Then, iPerfDetector constructs an inter-procedural call graph, and traverses all the functions that are executed in the main thread. Since functions are executed in the main thread by default, iPerfDetector decides that a thread is executed in the main thread if the thread is *not executed* on background threads, such as not within in the code blocks of *DispatchQueue.global.async* and *performBackgroundTasks*. Finally, if any executed functions in the main thread may result in data access, an instance of this anti-pattern is detected by iPerfDetector.

Detecting Updating UI Controls on Background Threads. To detect this anti-pattern, iPerfDetector traverses the entire source code and identifies all the instantiated UI objects (e.g., buttons, text fields, and labels). After this step, iPerfDetector obtains the locations of all the UI objects and their corresponding functions. Then, iPerfDetector constructs an inter-procedural call graph. iPerfDetector traverses the call graph and looks for functions that are executed in background threads (e.g., called within the code block of *performBackgroundTask* or *DispatchQueue.global.async{...}*, which are the APIs provided by the iOS SDK for executing tasks in background threads). An anti-pattern

instance is detected if we see that a UI-related object or function is called within the code blocks that will be executed in a background thread.

Detecting Applying UI Blurring Effect Inefficiently. We found that there are two situations where applying blurring effects may result in bad user experience. The first situation is when an object instance of *VisualEffectView* (i.e., for setting visual effects) is added to a cell. A cell in iOS UI can be represented by any class that is extended from classes *TableViewCell* (a type of cell that repeats in a table vertically) or *CollectionViewCell* (a type of cell that repeats in a table horizontally). The second situation is when an object instance of *VisualEffectView* is added to another view (i.e., objects that are instances of *UIView*, which manages the contents on the screen) inside a loop. In these two situations, the blurring effect is computed multiple times (e.g., once for each cell) whenever the frame is updated (e.g., when scrolling a table).

To detect this anti-pattern, iPerfDetector first traverses all the source code files searching for object instances of *VisualEffectView* with the blurring effect turned on. iPerfDetector detects an instance of anti-pattern by using taint analysis. Namely, iPerfDetector checks the calls/usages of the object instance of *VisualEffectView*, and verifies whether the object instance will eventually be added as a subview to other object instances that represent cells in iOS UI (i.e., sub-classes of TableViewCell or CollectionViewCell). If the *VisualEffectView* object instance is used in a loop (e.g., *for* or *while*), an anti-pattern instance is also detected.

Detecting Applying UI Transparency Effect Inefficiently. To detect this anti-pattern, iPerfDetector first traverses all the source code files searching for objects that are instances *UIView* and have transparency configured (i.e., the alpha value is larger than 0 and less than 1). Then, similar to detecting inefficient blurring effect, iPerfDetector uses taint analysis to verify where the partially transparent object instance is called. Namely, iPerfDetector tracks the usages and calls to object instance, and an anti-pattern instance is detected if the object instance is used in classes that are of the type *CollectionViewCell* or *TableViewCell*, and the object instance is added as a subview to the cell. An anti-pattern instance is also detected if the partially transparent object instance is used in a loop.

Furthermore, in order to evaluate iPerfDetector, we apply it on both commercial and open source apps, to find out how many anti-patterns iPerfDetector can detect. Then we report our findings to developers and ask them to verify our results.

We collaborated with a company named Seeb Smart Solutions[1], and used iPerfDetector to help improve the quality of their apps. Table 5.1 shows the applications that we used to evaluate iPerfDetector. We evaluated iPerfDetector on three apps from our industrial partner. App1 is an application for travel guides (similar to TripAdvisor to some extents). App2 is a ride sharing application (e.g., similar to Uber) that is commonly used in Iran. Finally, App3 is a news app that informs users regarding technology news over the globe. In addition to commercial apps, we also applied iPerfDetector on the latest version of four additional open source iOS apps (i.e., eight open source apps in total). The categories of these apps range from social network to business and iOS libraries. These four apps are popular on Github (i.e., more than 2K stars) and are relatively large in size (more than 2.5K LOC). We did not select these four apps for our manual study in 4 due to a lack of performance-related issue reports. We applied iPerfDetector to the latest versions of the studied apps, and we focused our study on the source code files (i.e., we excluded the ones that were detected in the test files).

To verify the detection results, we took a two-step approach. First, we manually studied all of the detected anti-pattern instances to examine whether they are true positives or not. Then, we reported the detected anti-pattern instances to developers in order to receive their feedback.

Results

Table 5.2 shows the performance anti-pattern detection results. Note that the detected instances in WordPress, Wire-iOS, Firefox, and Charts are new issues that were not discovered/discussed in the manually studied issue reports. In total, iPerfDetector detected 34 anti-pattern instances in 11 studied apps. The execution time of running iPerfDetector is around three minutes or less per app, depending on the size of the app. We executed iPerfDetector on a desktop machine with 16GB of memory and 2.6GHz Intel-Core i5 CPU. Note that some anti-patterns may not be applicable in the studied iOS app (e.g., an app does not have data models or it is an iOS framework that does not have

[1] https://www.linkedin.com/company/seeb/

Table 5.1: Apps that we use to evaluate iPerfDetector.

App Name	Category	LOC	Availability	Version
Firefox	Utilities	145K	Open Source	8.0
WordPress	Productivity	84K	Open Source	8.5
Wire-iOS	Business	60K	Open Source	3.3
Charts	Library	31K	Open Source	3.0.3
Yep	Social Network	71K	Open Source	1.3
Sync	Library	8K	Open Source	3.2.3
TSWeChat	Social Network	8.5K	Open Source	1.0
SugarRecord	Library	2.5K	Open Source	1.0
App1	Travel	5 to 25K	Commercial	Latest
App2	Travel	15 to 50K	Commercial	Latest
App3	News	5 to 25K	Commercial	Latest

a UI), so we show the detection result for such anti-patterns as "N/A" in the table.

In our manual verification, we found only two false positives among all the 34 detected anti-pattern instances (i.e., a precision of 94%). The two false positives were both DM anti-pattern that were detected in Sync. We found that, in these two specific cases, Sync developers added some app-specific code to avoid data accesses in the main thread. The code checks whether the data access will be executed in the main thread, and if so, an error will be thrown. Since iPerfDetector did not consider such app-specific code in the detection algorithm, these two anti-pattern instances were detected as false positives. We reported the remaining 32 detected anti-pattern instances (after removing the two above-mentioned false positives) to developers. We contacted the developers by creating issue reports on bug tracking systems (e.g., Bugzilla or Jira), or by sending emails to developers. Overall, we received positive responses for the 31 reported anti-pattern instances (we are still waiting for responses of one reported anti-pattern instance in SugarRecord). Developers acknowledged the reported problems and expressed interest in our detection tool.

For example, a developer from Sync agreed to the reported DM anti-pattern instances and said:

> "*My recommendation is never do calls from the main thread. Another recommendation is: use NSFetchedResultsController when possible...*" [2]

[2] https://github.com/3lvis/Sync/issues/509

Table 5.2: iOS performance anti-pattern detection result. "N/A" means that the anti-pattern is not applicable to the studied iOS app (e.g., the app does not have data models or it is an iOS framework that does not have a UI). *Detected* shows the number of anti-pattern instances that are detected by iPerfDetector. *Verified* shows the number of anti-pattern instances that are manually verified to be true positives by the authors.

App Name	DM		TREF		BLEF		UIBG	
	Detected	Verified	Detected	Verified	Detected	Verified	Detected	Verified
Wordpress	2	2	0	0	0	0	0	0
Wire-iOS	2	2	0	0	0	0	0	0
Firefox	N/A	N/A	1	1	1	1	0	0
Charts	N/A	N/A	0	0	0	0	0	0
Sync	18	16	N/A	N/A	N/A	N/A	0	0
Yep	N/A	N/A	3	3	0	0	0	0
TSWeChat	N/A	N/A	2	2	0	0	0	0
SugarRecord	1	0	N/A	N/A	N/A	N/A	0	0
App1	N/A	N/A	N/A	N/A	1	1	0	0
App2	N/A	N/A	0	0	1	1	0	0
App3	2	2	0	0	0	0	0	0

The developer suggests using the *NSFetchedResultsController* API or an API wrapper that he developed to avoid the DM anti-pattern. However, after manually checking the API and the wrapper, we found that the API is only able to handle one particular case of the DM anti-pattern: when the data retrieved from the database is used directly in a UI table. Hence, even though the developers are aware of the anti-pattern, the problems still exist in the app. This also shows that iPerfDetector is able to help developers detect undiscovered problems in iOS apps.

As another example, developers from Yep responded to our email and acknowledged the reported problem:

> "*I think you're right about the anti-pattern issue, the code isn't perfect yet. This views are not in the main scenes, so the performance is tolerated [...]*"

Developers from TSWechat agreed that iPerfDetector is helping them detect problems in their app. They also mentioned how iPerfDetector helped their app detect the problems early in the development process:

> "*Yeah, absolutely yes. I didn't do anything about the performance of this project so far. So we can improve the performance of this project [based on the anti-pattern instances]*

that you [detected]." [3]

We found that in some cases, developers acknowledged that the reported anti-pattern instances are problematic, but the performance impact may not be clear. For example, a Firefox developer mentioned that:

> "*...Changing this alpha might not make a measurable impact on CPU perf, maybe it saves some power demand on the GPU though...*" [4]

Nevertheless, some developers think these anti-patterns are bad practices that one should always avoid. However, fixing the anti-patterns in a matured app can be a challenging task (e.g., may require refactoring the design of the app). For example, developers from WordPress mentioned that:

> "*In my opinion it's not a matter of whether to do it or not, but more about deciding when to do it...Moving CoreData operations to a background thread could be one of those improvements as it would allow us not to lock the main thread... we have to evaluate how moving that operation to the background (ie: making it asynchronous) would impact the rest of the code.*" [5]

We also got positive feedback from commercial apps' developers. All the reported anti-pattern instances were acknowledged as problematic by our industrial partner, and developers are in the process of fixing them. iPerfDetector is now integrated into the quality assurance process of our industrial partner to help them ensure the performance of their apps.

iPerfDetector detected a total of 34 anti-pattern instances in the studied apps, where we manually verified 32 of them as true positives. We reported the detected anti-pattern instances to developers and 31/32 are confirmed as potential performance problems (we are still waiting for developers' response for the remaining case).

[3] https://github.com/hilen/TSWeChat/issues/42
[4] https://github.com/mozilla-mobile/firefox-ios/issues/3961
[5] https://github.com/wordpress-mobile/WordPress-iOS/pull/578

5.2 A Case Study on the Performance Impact of the Anti-patterns

Motivation

To provide a better understanding of the iOS performance anti-patterns, in this section, we conduct a case study and see if these anti-patterns actually have a negative performance impact (i.e., in terms of GPU usage, CPU usage and response time).

Approach

We choose one detected anti-pattern instance from each of DM, BLEF, and TREF in the studied apps. We did not conduct the study on all the detected problems because generating a fix, in many cases, requires significant manual investigation and refactoring of the existing code. However, for each anti-pattern, the root cause is the same across all detected instances. We choose TREF and BLEF from Firefox due to its popularity and size. We choose DM from one of the commercial apps to show that the studied anti-patterns exist and have a performance impact in both open source and commercial apps. We did not include UIBG in this study since the effect of UIBG is usually related to app crashes or missing UI updates (i.e., visual performance problems from the perspective of the users). We measured GPU usage when BLEF and TREF anti-patterns were happening, because as mentioned by developers in forums and Apple Documents, color blending parts of the screen, which are common causes of sluggish scrolling in tables, are highly associated with GPU usages (Apple, 2018f; Medium, 2015). For the DM instance, we measure the CPU usage and response time, since as stated in Apple Core Data related tutorials, processing data on the main thread specifically affects CPU (Apple, 2018g).

We first measure the performance of the workload (e.g., scrolling a table) before fixing the anti-pattern. Then, we measure the performance improvement after we manually fix the anti-pattern. We used an iPhone 6 for the experiment in our case study and we used the Apple's Instrument to collect performance metrics (Apple, 2018c). In each round of the test, we take notes on 15 CPU usage samples, and calculate the average CPU usage for the round. We repeat each execution 10 times, and take the average value to minimize the effect of fluctuation during performance measurement (Chen et al., 2014). Note that the performance impact of each anti-pattern instance may vary for different

cases (e.g., some anti-pattern instances that are executed more frequently or are related to larger tables would have a larger performance impact) (Chen, Shang, Hassan, et al., 2016). Hence, our goal in this case study is to find out whether the anti-pattern has an impact on the performance in a controlled environment.

Results

For both TREF and BLEF, we measured the GPU usage since the anti-patterns are related to processing UI effect. We apply the same experimental setting to these two anti-pattern instances. Table 5.3 shows the GPU usage before and after fixing the BLER and TREF anti-patterns. The executed workload involves scrolling a table in Firefox. We set the table to contain 15 and 30 rows to study the performance impact when the size of a table increases. We monitor the GPU usage before and after removing the heavy UI effect. We wrote test cases to automate the scrolling and profile the GPU usage during test execution. We found that, when the table contains 15 rows, fixing the TREF anti-pattern instance improved the GPU usage by 5.8% (improved from 15.3% to 14.4%). On the other hand, fixing the BLEF anti-pattern instance improved the GPU usage by 72% (improved from 43.4% to 12.1%). When the table contains 30 rows, fixing the TREF anti-pattern instance improved the GPU usage by 29% (improved from 22.5% to 16%). For BLEF, we observed less improvement when fixing the anti-pattern instance in a larger table. The GPU usage was improved by 70% (improved from 50.2% to 15.3%) when BLEF is fixed.

For DM, we monitored response time and CPU usage of App3, when one anti-pattern instance is happening before and after moving Core Data access to a background thread (the operation takes less than a second). Table 5.4 shows the CPU usage before and after fixing the DM anti-pattern. It also shows the response time improvement after fixing the anti-pattern. The executed workload involves fetching 90 objects from the database. Although we cannot report the actual response time due to the confidentiality agreement, the response time of the task that contained the anti-pattern instance improved by over 80%, after the anti-pattern was fixed. The average CPU usage improved by 34.15% (from 28.9% to 19.03%).

In summary, we found that fixing the anti-pattern instances has a non-negligible performance

Table 5.3: GPU usage (percentage) changes in median, average, and standard deviation, after resolving the BLER and TREF anti-pattern instances.

	15 rows				30 rows			
	Before		After		Before		After	
	Median	**Avg±StDv**	**Median**	**Avg±StDv**	**Median**	**Avg±StDv**	**Median**	**Avg±StDv**
BLEF	43.5%	43.4%±3.02%	12.5%	12.1%±4.04%	50%	50.2%±3.19%	16%	16.5%±1.71%
TREF	13%	15.3%±4.05%	13.5%	14.4%±3.74%	21.5%	22.5%±8.19%	18%	20.1%±4.2%

Table 5.4: CPU usage changes (percentage) in median, average, and standard deviation, and response time changes (percentage) after resolving the DM anti-pattern instance. Due to the confidentiality agreement, we only show the percentage improvement for the response time.

	Before		After	
	Median	**Avg±StDev**	**Median**	**Avg±StDev**
CPU	27%	28.91%±6.42%	18.66%	19.03%±4.15%
Resp Time		-		80%

improvement. Moreover, our performance measurements show that the average values of the performance metrics are very close to the median values, which shows the there is little fluctuation in our analysis (i.e., the results are stable). iPerfDetector can help developers detect the anti-pattern instances, and developers may decide when and how to fix the problem (e.g., problems that are commonly triggered by the users may need to fixed earlier).

We found that fixing the anti-pattern instances may improve performance (i.e., in terms of response time, GPU, and CPU usage) by 5.8% to 67% in GPU usage, 34.15% in CPU usage, and by over 80% in response time of the exercised workload.

In the next chapater, we will discuss the threats to validity of our study.

Chapter 6

Threats

6.1 Internal Validity

In this thesis, we conducted a manual analysis on the performance issue reports. Similar to prior studies (Jin et al., 2012c; Liu et al., 2014), we used keywords such as "performance" and "slow" to identify performance-related issue reports. We found that some issue reports may not be related to performance issues even though they contain the keywords. Therefore, to reduce the bias in our data, we manually went through all 960 matched issue reports to identify the relevant reports. Although we tried to use more general search terms, we may still miss some performance-related issue reports. However, we have a much larger number of performance issues compared to prior performance studies (e.g., we studied 235 issue reports after our manual filter process, where Liu et al. (2014) only studied 70 issue reports).

As mentioned in Chapter 5, iPerfDetector does not detect retain cycles. Facebook's Infer focuses on using static analysis to detect null pointer dereferences and memory leaks, which makes it a great complement to our tool. Our tool is detecting performance anti-patterns that are not implemented in Infer's detection algorithm. Apple's Instruments, on the other hand, is a profiling tool that is able to show the performance information (e.g., object memory graphs and CPU usages of user events) when running an app. Therefore, developers may use Instruments to monitor the detailed activities of an app and try to diagnose the problems. However, compared to our approach, Instruments offers more flexibility in detecting performance problems, but it can not directly pinpoint developers to

the root causes (i.e., anti-pattern in the code) and requires developers to dynamically exercise the problematic code in the app. Moreover, our empirical study shows the common performance problems that happen in iOS apps and compare our findings with prior studies in Android apps. Future studies may build upon our findings and provide better techniques to help improve the performance of iOS apps.

A number of projects may still use Objective-C code in some parts of the app (e.g., not yet migrated or in third party libraries). In such cases, our tool will not be able to detect the anti-patterns. Even though our tool can only detect anti-patterns in Swift code at the moment, the studied anti-patterns are related to how developers interact with the iOS SDK (i.e., the problem can occur whether an app is implemented in Swift or Objective-C). For example, developers may be calling iOS APIs to compute heavy UI effects in tables when developing apps in either language. Hence, the anti-patterns that we studied are applicable to both languages.

As we discussed in 5.2, we conducted the case study on TREF and BLEF anti-patterns impacts, before and after removing the effects. Whereas, developers might be oliged to use these effects in their designs, and removing the effects would tamper the app's design. However, iPerfDetector generates warnings to let developers know what the performance problem is and leaves the final decision to them.

In 5.2 we only measured performance metrics that are possibly affected by the anti-patterns and not all the performance metrics, although generally in some cases fixing an issue, would help improving a perfromance metric and degrade another. However, we selected measured performance metrics according to Apple documents and tutorials.

6.2 External Validity

We found that there is a limited number of mature and open source iOS apps that contain performance issue reports. Hence, we only conducted our manual study on four open source iOS apps that met our selection criteria. Although these apps are widely used and are large in scale, the performance issue reports that we studied may not be generalizable to other apps. However, we found that

the manually-uncovered performance anti-patterns also exist in seven other iOS apps (three commercial and four open source apps), which means that these performance problems are not unique to the four manually-studied apps. We conducted a case study to measure the performance improvement when the anti-pattern instances are fixed. However, the performance improvement may vary in different cases. Various factors, such as number of cells in a table, number of iterations in loops, and how frequently users may trigger the anti-pattern instance in the code, may all affect the performance impact of the anti-pattern instance. Nevertheless, our result highlights the potential impact of the performance anti-patterns. Moreover, our tool can help detect the anti-pattern instances, and developers may decide when or how they want to fix the problem (e.g., the problems that are frequently triggered by users may need to be fixed earlier).

The performance problems that we found are mostly related to data access and UI rendering. The reason may be that we focus our study on iOS apps, so most of other problems (e.g., network-related issues) that we found are related to testing and exception handling. For example, we found some developers discussed how to test various features in an app when the network connection is slow[1][2]. In most cases, app developers do not have control of external systems (e.g., the network speed and the servers that they are communicating with). We did find some problems that are implicitly related to network, where developers are storing the received data in the database inefficiently. In such cases, we classify the problem as the data model (DM) anti-patterns. Future research should consider studying performance problems that exist in servers that are handling requests from iOS apps.

6.3 Construct Validity

We use static analysis to detect the iOS performance anti-pattern in the studied apps. Since we do not have the ground truth of the anti-pattern instances in the studied apps, it is impossible for us to compute a recall value. However, iPerfDetector was able to detect the anti-pattern instances that we manually studied in the issue reports, as well as the new anti-pattern instances that we detected in the latest version of the app.

[1] https://github.com/wordpress-mobile/WordPress-iOS/issues/9993
[2] https://github.com/wordpress-mobile/WordPress-iOS/pull/10111

During our manual study, we found that UIBG (i.e., doing UI-related operations in background threads) is a common anti-pattern (nearly 18%) among all the studied issue reports. However, iPerfDetector did not detect any such anti-pattern instance in the latest versions of the apps. We conjecture that the Main Thread Checker may have helped developers avoid such problems during app development. As we mentioned in Section 4.2, the Main Thread Checker is a dynamic analysis tool released by Apple that helps developers detect UIBG when running the app in debug mode (Apple, 2017). To verify our assumption, we conduct a small experiment on the studied issue reports in WordPress. Interestingly, we found that almost half of UIBG anti-patterns were reported by users after *the Main Thread Checker was released* for a while. Namely, developers did not fully utilize the Main Thread Checker to detect these anti-pattern instances during development, so these problems were reported by the users after the apps were released. The reason that iPerfDetector did not detect any UIBG may be that the studied apps have been used by many users for a long period of time, so most UIBG instances were already reported and fixed. Future studies are needed to study how developers interact with different bug detection tools to provide better development supports.

Finally, our tool currently analyzes the source code of an app, so if the source code is not available, we cannot detect the anti-patterns. Hence, if an app is using external libraries, which the source code is not available, we will not be able to detect the problems.

Chapter 7

Conclusion

In this thesis, we conducted an empirical study on 235 performance issue reports in four open-source iOS applications (i.e., apps) that are written in Swift. Our manual study on the issue reports found that inefficient UI design, memory issues, and inefficient thread handling are the most common types of performance issues. In particular, we found that most problems we studied are related to how developers use iOS-specific APIs (e.g., UI effect or threading). Hence, we manually derive four performance anti-patterns that are related to iOS API usage. We documented these four anti-patterns and implemented a static analysis tool, called iPerfDetector, to detect these patterns. We evaluated iPerfDetector on 11 iOS apps (eight open source and three commercial apps). iPerfDetector was able to detect a total of 34 anti-pattern instances (where we manually verified two as false positives). We reported 32 of the detected anti-pattern instances and 31 of them are accepted by developers as potential performance problems. iPerfDetector is now used by our industrial partner to ensure the quality of their apps.

To the best of our knowledge, this is the first study on iOS performance issues. Our study highlights some common problems that may be unique to iOS development. However, the current research community often only focuses on Android app development. Future studies should investigate problems in iOS development and provide further support to developers.

References

Afjehei, S. S., & Chen, T.-H. P. (2018). *ios performance issue reports.* https://docs.google.com/spreadsheets/d/1hX8IBcYIVv6x4nWfWczT5oLV0b1SMQK03q6_xC9B8eQ/edit#gid=1079185239. (Accessed: 2018-07-23)

Apple. (2012). *Practical memory management.* https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html. (Accessed: 2018-06-15)

Apple. (2015). *More core data and concurrency.* https://cocoacasts.com/more-core-data-and-concurrency. (Accessed: 2018-06-15)

Apple. (2017). *Apple threading programming guide.* https://developer.apple.com/documentation/code_diagnostics/main_thread_checker. (Accessed: 2018-06-15)

Apple. (2017). *Nsmanagedobject programming guide.* https://developer.apple.com/documentation/coredata/nsmanagedobjectcontext. (Accessed: 2018-06-15)

Apple. (2018a). *Apple guide on swift programming language.* https://developer.apple.com/swift/. (Accessed: 2018-06-15)

Apple. (2018b). *Apple UIKit.* https://developer.apple.com/documentation/uikit. (Accessed: 2018-06-15)

Apple. (2018c). *instruments.* https://help.apple.com/instruments/mac/current/#/dev7b09c84f5/. (Accessed: 2018-07-23)

Apple. (2018d). *Making core data your model layer.* https://developer.apple.com/

documentation/coredata/making_core_data_your_model_layer. (Accessed: 2018-06-15)

Apple. (2018e). *Model-view-controller.* https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. (Accessed: 2018-06-15)

Apple. (2018f). *Monitoring your app's graphics activity.* https://developer.apple.com/documentation/metal/tools_profiling_and_debugging/gpu_activity_monitors/monitoring_your_app_s_graphics_activity. (Accessed: 2018-07-23)

Apple. (2018g). *Using core data in the background.* https://developer.apple.com/documentation/coredata/using_core_data_in_the_background. (Accessed: 2018-06-15)

Cassee, N., Pinto, G., Castor, F., & Serebrenik, A. (2018). How swift developers handle errors. In *15th international conference on mining software repositories (msr 2018).*

Chen, T.-H., Shang, W., Hassan, A. E., Nasser, M., & Flora, P. (2016). Detecting problems in the database access code of large scale systems: An industrial experience report. In *Proceedings of the 38th international conference on software engineering companion* (pp. 71–80).

Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2016). Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. *IEEE Transactions on Software Engineering*, *42*(12), 1148–1161.

Chen, T.-H., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2017). Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. In *Proceedings of the 39th international conference on software engineering: Software engineering in practice track* (pp. 243–252).

Chen, T.-H., Weiyi, S., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2014). Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th international conference on software engineering (icse)* (pp. 1001–1012).

Chowdhury, S., Di Nardo, S., Hindle, A., & Jiang, Z. M. J. (2018). An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, *23*(3), 1422–1456.

Cruz, L., & Abreu, R. (2017). Performance-based guidelines for energy efficient mobile applications. In *Mobile software engineering and systems (mobilesoft), 2017 ieee/acm 4th international conference on* (pp. 46–57).

Facebook. (2017). *Facebook's infer static analysis tool.* http://fbinfer.com/. (Accessed: 2018-06-15)

Github. (2018). *Trending swift projects on github.* https://github.com/trending/swift?since=monthly. (Accessed: 2018-06-15)

Gottschalk, M., Jelschen, J., & Winter, A. (2014). Saving energy on mobile devices by refactoring. In *Enviroinfo* (pp. 437–444).

Grechanik, M., Fu, C., & Xie, Q. (2012). Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th international conference on software engineering (icse)* (pp. 156–166).

Guide, A. C. D. P. (2017). *Core data programming guide.* https://developer.apple.com/documentation/coredata. (Accessed: 2018-06-15)

Habchi, S., Hecht, G., Rouvoy, R., & Moha, N. (2017). Code smells in ios apps: How do they compare to android? In *Proceedings of the 4th international conference on mobile software engineering and systems* (pp. 110–121).

Hecht, G., Moha, N., & Rouvoy, R. (2016). An empirical study of the performance impacts of Android code smells. In *Proceedings of the international workshop on mobile software engineering and systems* (pp. 59–69).

Hu, H., Bezemer, C.-P., & Hassan, A. E. (2018). Studying the consistency of star ratings and the complaints in 1 & 2-star user reviews for top free cross-platform android and ios apps. *Empirical Software Engineering*.

Jiang, Z. M., Hassan, A. E., Hamann, G., & Flora, P. (2008). Automatic identification of load testing problems. In *Proceedings of 24th international conference on software maintenance (icsm)* (p. 307-316).

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012a). Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd acm sigplan conference on programming language design and implementation.*

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012b). Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, *47*(6), 77–88.

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012c). Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd acm sigplan conference on programming language design and implementation* (pp. 77–88).

Jindal, A., Pathak, A., Hu, Y. C., & Midkiff, S. (2013). On death, taxes, and sleep disorder bugs in smartphones. In *Proceedings of the workshop on power-aware computing and systems* (p. 1).

Khalid, H., Shihab, E., Nagappan, M., & Hassan, A. E. (2015). What do mobile app users complain about? *IEEE Software*, *32*(3), 70-77.

Lin, Y., Okur, S., & Dig, D. (2015). Study and refactoring of android asynchronous programming (t). In *Automated software engineering (ase), 2015 30th ieee/acm international conference on* (pp. 224–235).

Liu, Y., Xu, C., & Cheung, S.-C. (2014). Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering* (pp. 1013–1024).

Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D., & Jensen, C. (2016). Understanding code smells in android applications. In *Mobile software engineering and systems (mobilesoft), 2016 ieee/acm international conference on* (pp. 225–236).

Martin, W., Sarro, F., Jia, Y., Zhang, Y., & Harman, M. (2017). A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, *43*(9), 817-847.

Martin Fowler, D., Fowler, M., Beck, K., Shanklin, J., Gamma, E., Brant, J., ... Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley.

McDonnell, T., Ray, B., & Kim, M. (2013). An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the 2013 ieee international conference on software maintenance* (pp. 70–79).

Medium. (2015). *Diagnose and solve performance problems with xcode instruments.* https://medium.com/@zhenya.peteliev/diagnose-and-solve-performance-problem-with-xcode-instruments-5c25c27f21d5. (Accessed: 2018-11-05)

Medium. (2016). *21 amazing open source ios apps written in swift.* https://medium.mybridge.co/21-amazing-open-source-ios-apps-written-in-swift-5e835afee98e. (Accessed: 2018-06-11)

Morales, R., Saborido, R., Khomh, F., Chicano, F., & Antoniol, G. (2017). Earmo: an energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*(1), 1–1.

Nistor, A., Chang, P.-C., Radoi, C., & Lu, S. (2015). Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 2015 international conference on software engineering* (pp. 902–912).

Nistor, A., Jiang, T., & Tan, L. (2013). Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th working conference on mining software repositories* (pp. 237–246).

Nistor, A., Song, L., Marinov, D., & Lu, S. (2013). Toddler: detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 international conference on software engineering* (pp. 562–571).

Okur, S., Hartveld, D. L., Dig, D., & Deursen, A. v. (2014). A study and toolkit for asynchronous programming in c#. In *Proceedings of the 36th international conference on software engineering* (pp. 1117–1127).

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., & De Lucia, A. (2017). Lightweight detection of android-specific code smells: The adoctor project. In *Proceedings of the 24th international conference on software analysis, evolution and reengineering* (pp. 487–491).

Pathak, A., Hu, Y. C., & Zhang, M. (2011). Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th acm workshop on hot topics in networks* (p. 5).

Pathak, A., Hu, Y. C., & Zhang, M. (2012). Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th acm european conference on computer systems* (pp. 29–42).

Rebouças, M., Pinto, G., Ebert, F., Torres, W., Serebrenik, A., & Castor, F. (2016). An empirical study on the usage of the swift programming language. In *Software analysis, evolution, and reengineering (saner), 2016 ieee 23rd international conference on* (pp. 634–638).

RedMonk. (2018). *The RedMonk programming language rankings: January 2018.* http://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/. (Accessed: 2018-06-11)

Reimann, J., Brylski, M., & Aßmann, U. (2014). A tool-supported quality smell catalogue for android developers. In *Proceeding of the conference modellierung 2014 in the workshop modellbasierte und modellgetriebene softwaremodernisierung–mmsm* (Vol. 2014).

Smith, C. U., & Williams, L. G. (2001). Software performance antipatterns; common performance problems and their solutions. In *Int. cmg conference* (pp. 797–806).

Statista. (2018). *Number of apps available in leading app stores.* https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. (Accessed: 2018-07-23)

Statistia. (2018). *Number of apps available in leading app stores as of 1st quarter 2018.* https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. (Accessed: 2018-06-15)

SwiftAST. (2018). *Swift-ast tool on github.* https://github.com/yanagiba/swift-ast. (Last Accessed 2018-6-30)

Syer, M. D., Nagappan, M., Hassan, A. E., & Adams, B. (2013). Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *Proceedings of the 2013 conference of the center for advanced studies on collaborative research* (pp. 283–297).

Zaman, S., Adams, B., & Hassan, A. E. (2012). A qualitative study on performance bugs. In *Proceedings of the 9th ieee working conference on mining software repositories* (pp. 199–208).