Faculty of Engineering and Technology

Master of Software Engineering (SWEN)

Master Thesis

Comparative Analysis of Mobile Software Development Frameworks: React Native and Native iOS

By

Student Name: Bisan Abubaker

Student Number: 1175468

Supervised

By

Dr Adel Taweel

Dr Samer Zain

A thesis submitted in fulfillment of the requirements for the degree of Master of Software Engineering at Birzeit University, Palestine

August 5, 2020

جامعة بيرزيت
BIRZEIT UNIVERSITY

Comparative Analysis of Mobile Software Development Frameworks: React Native and Native iOS

Author: Bisan Abubaker

This thesis was prepared under the supervision of Dr.Adel Taweel and has been approved by all members of the examination committee

Dr. Adel Taweel, Birzeit University

[illegible]

Dr. Radi Jarrar, Birzeit University

Radi Jarrar

Dr. Mohammed Hussien, Birzeit University

[illegible]

Date of defense:
25 July 2020

Declaration of Authorship

I, Bisan Abubaker, declare that this thesis titled, “Comparative Analysis of Mobile Software Development Frameworks: React Native and Native iOS” and the work presented in it are my own.

I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at Birzeit University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Bisan Abubaker

Date: July 25, 2020

Abstract

Mobile applications market is divided between limited number of distinct platforms, mainly iOS and Android, which makes mobile application development problematic and difficult.

Developers are required to having knowledge about development differences and tools for both platforms, which demands more development and maintenance effort, multiplatform-skilled software engineers, increased budget and multiple programming language skills. Several approaches have attempted to address this issue, including the use of hybrid development frameworks, such as ionic **[1]** and PhoneGap **[2]**, but they were not able to provide a native user interface (UI) feeling. A newly released hybrid framework that addresses the user interface problem by providing a native UI feeling is React Native.

React Native is a cross-platform framework that enables developing mobile applications that works on both iOS and Android platforms, using a single code written in JavaScript, with native UI feeling. However, its cross-platform capabilities have not been studied enough to understand its characteristics to make an informed software development choice. This thesis evaluates the performance and efficiency of applications written in the new React Native framework and proposes a software engineering method that improves performance of React Native applications. For the former, it conducts a comparative study between applications developed using iOS and React Native with respect to execution and CPU time, memory and battery usage, frames per second and application launch time. Our focus is on the data centric apps, i.e. apps that process, read and write data to back-end server as well as local storage and files.

For the later, based on the outcome of the comparative study, a study of React Native development features was conducted to identify its comparative deficiencies to provide software development guidelines on mechanisms to improve React Native applications development.

Results are promising for React Native. They show a big similarity, on application performance, between both iOS and React Native platforms. Which means, there is no significant difference between performance in the two frameworks on the studied software features and applications, with a difference in performance ranging between 3%- 10%. However, a major performance difference was found in React Native, compared to iOS, on image processing. For this feature, iOS was found three times faster than React Native, for the studied applications. A code improvement solution is proposed to address this React Native performance issue and improve its execution time to as nearly as iOS.

الملخص

ينقسم سوق تطوير تطبيقات الهاتف المحمول بين عدد محدود من أنظمة التشغيل، بشكل أساسي الاي او اس والاندرويد، وهذا يجعل من موضوع تطوير التطبيقات للهواتف المحمولة أمر صعب ومعقد نوعا ما.

يجب على المطورين أن يعرفوا الفروقات بين الأدوات المستخدمة لتطوير نظامي التشغيل الاندرويد والاي او اس، وهذا يتطلب جهود معرفية أكثر من ناحية برمجة وصيانة بالاضافه الى مهندسين مختصين بكلا انظمه التشغيل وبالتالي زيادة في الميزانية.

هناك عدة طرق وتوجهات لحل هذه المشاكل، منها التطوير باستخدام طريقة الهايبرد مثل بيئة الايونيك والفون جاب، ولكنهما فشلتا في إعطاء واجهه شعور اصليه للمستخدمين. ظهرت بعد ذلك بيئة جديدة حلت المشكله المذكور سابقا وأعطت شعور أصلي للمستخدمين وهي الرياكت نيتف.

رياكت نيتف هي بيئة تعمل بنظام الكروس بلاتفورم بحيث تمكن المبرمجين من تطوير تطبيقات للهواتف المحمولة تعمل بنظامي الاندرويد والاي او اس بنفس الوقت باستخدام كود واحد مكتوب بلغة الجافا سكربت مع إعطاء شعور أصلي للمستخدمين، ولكن قدرات هذه البيئة لم تدرس بشكل كافي لإعطاء الخيار الأمثل لطريقه التطوير المثلى المنصوح باستخدامها.

في هذا البحث، نقوم اولا بتقييم الفعالية والأداء للتطبيقات المكتوبة في بيئه الرياكت نيتف الجديدة، ومن ثم نقوم بتقديم مقترح هندسي جديد لتحسين أداء التطبيقات المكتوبة بالرياكت نيتف. للهدف الأول، نقوم بعمل دراسة مقارنة بين التطبيقات المط،رة باستخدام الريكات نيتف والاي او اس من ناحيه ،قت التنفيذ، وقت المعالجة، الذاكرة واستهلاك البطارية، بالاضافه الى عدد الإطارات المعروضة في الثانية ووقت بدء التطبيق. تركيزنا هنا على التطبيقات التي تحتوي على بيانات كثيرة كما انها تعالج وتقرأ وتكتب هذه البيانات على سيرفرات وتخزين محلي وملفات.

للهدف الثاني، بناء على نتيجة دراسة المقارنة، قمنا بتقديم خطوط مساعدة لتحسين أداء التطبيقات المطورة ببيئة الرياكت نيتف.

النتائج إيجابية باتجاه الرياكت نيتف، وقد أظهرت تشابه كبير بين الأداء للتطبيقات المطورة باستخدام الرياكت نيتف والاي او اس، وهذا يعني أنه لا يوجد فروقات كبيرة بين هاتين البيئتين، الفروقات بسيطه ،تتراوح بين 3٪ الى 10٪ فقط. على أية حال، لقد وجد فرق أساسي بين الرياكت نيتف والاي او اس من ناحية معالجة الصور، تبين أن الاي او اس أسرع بثلاثة مرات من الرياكت نيتف، وتم تقديم طريقة هندسية لتطوير الكود المكتوب بالرياكت نيتف لتصبح سرعته قريبة جدا من سرعه الاي او اس.

شكر وعرفان

الحمد والشكر لله عز وجل الذي أنعم عليّ بنعمة العلم ووفقني لإنجاز هذا البحث ويسر السبيل إليه، أتوجه بخالص الشكر وعميق التقدير والامتنان الى الدكتور عادل الطويل الذي أشرف على هذا العمل ووقف معي في جميع مراحل إنجازه وزودني بالنصائح والإرشادات.

كما أتوجه بالشكر والتقدير إلى لجنة النقاش على ما قدموه من ملاحظات قيمة، وأشكر كل من شارك في تنفيذ التجربة وتقديم المعلومات، وكل من شجعني على إتمام هذا العمل وإكماله إلى آخر وأحسن صورة ووجه.

إهداء

إلى أعز الناس وأقربهم إلى قلبي، إلى والدتي العزيزة ووالدي العزيز، اللذان كانا دوما عونا وسندا لي، وكان دعاؤهم ممدا لي.

إلى من ساندني وخطى معي خطواتي، ويسر لي الصعاب، وتحمل معي الكثير، ووقف بجانبي الى آخر خطوة، إلى زوجي العزيز.

إلى فلذة كبدي، بنتي ماسة التي تحملت طيلة الفترة التي قضيتها في إعداد هذا البحث.

إلى عائلتي العزيزة، والدتي الثانية ووالدي الثاني والديّ زوجي، اللذان قدما كل الدعم والعون لي طيلة فترة البحث وكانا خير أهل وسند.

إلى أساتذتي وأهل الفضل علي، الذين غمروني بالحب والتقدير والنصح والتوجيه والإرشاد.

إلى كل هؤلاء، أهديهم هذا العمل المتواضع، سائلا الله العلي القدير أن ينفعنا به ويمدنا بتوفيقه.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Apps:	applications
iOS:	iPhone Operating System
RN:	React Native
JS:	Java Script
FPS:	Frames Per Second
IDE:	Integrated Development Environment
SDK:	Software Development Kit
OS:	Operating System
UI:	User Interface
MVC:	Model View Controller
XML:	Extensible Mark-up Language
JSX:	JavaScript XML
DOM:	Document Object Model
API:	Application Programming Interface
GPU:	Graphics Processing Unit

Acknowledgements

I would first like to thank my thesis supervisors, Dr Adel Taweel and Dr Samer Zain of the Master of Software Engineering program at Birzeit University. They helped me a lot whenever I ran into a trouble spot or had a question about my research or writing.

I would also like to acknowledge Dr Radi Jarrar and Dr Mohammad Hussein as the readers of this thesis, and I am gratefully indebted to their very valuable comments on this thesis.

Finally, I must express my very profound gratitude to my parents and to my spouse, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author,

Bisan Abubaker

Chapter 1 Introduction

The number of smartphone users and consequently applications are increasing at enormous speeds. In fact, the number of users has crossed over 3.2 billion users, and some studies report it at more than 5 billion users and it is expected to reach about 3.8 billion users by 2021 **[3]**. Similarly, the number of tablet users is reported at 1.35 billion users **[4]**. Smartphones are widely used at home, work and on the streets with applications covering most, if not all, of human life aspects. Approximately, 90% of the time spent on smartphones is on the applications downloaded on the mobiles from the stores **[5]**. The number of available mobile applications available for download is increasing surprisingly; it reached about 2.2 million applications for the Apple App Store, and 2.47 million applications for the Google Play Store **[6]**. In addition to that, because of the popularity and wide usage of mobile applications, the number of mobile applications downloads has reached more than 194 billion downloads **[7]**. Those applications need to be not just supported by their operating systems, but also by robust software development frameworks. However, due to the large user base for both Android and iOS, most applications are made available for both operating systems. Consequently, programmers need to develop two separate applications using different tools and programming languages for each platform in order to reach majority of mobile applications' users, even though the application itself is the same, which is a more expensive process.

Many attempts solve the problem by giving the ability to write a single code using one tool to work on both iOS and Android platforms. However, the code is written using HTML, CSS and a web browser, which are embedded into the application, such application is known as a hybrid app **[8]**. In this case, most of the code is shared between the two development platforms, but there is a shortage in achieving a native UI feeling and usually the performance of the apps developed using this method is very poor. As a result, a newly released framework called React Native born to enhance both the performance, to become closer to the native OS, and the UI feeling, to become almost as a native UI, instead of using web components. Therefore, this thesis aims to study the performance and efficiency of applications written in the new React Native framework and proposes a software engineering method that attempts to improve performance of React Native applications. The study focuses on the data centric apps, due to their performance demanding, which handle data processing, reads and writes to both a back-end server and local storage and files.

From our study, we found that the performance parameters including application launch time, CPU and memory usage, battery consumption and frames per second are very close for both iOS and React Native. The major difference was found is in the execution time. iOS was faster by three times than React Native. However, we were able to enhance the execution time to become very close to iOS by modifying the code of React Native.

1.1 *Introduction and Motivation*

After the retreat of Windows Phone **[9]**, two main dominants remain in the world of smartphones, which are iOS and Android. Normally, two separate frameworks are used for the development of mobile applications, one for the iOS, which uses Xcode with Swift or Objective-c language, and the other is the Android, which uses the Android Studio with Java or Kotlin language. Actually, this is time, resource and budget consuming process because, potentially, two separate teams with different skills would be needed to develop two separate copies of the same mobile application. New frameworks or tools have thus become available and widely popular to reduce the three main factors, which are time, resource and budget. These tools are called mobile cross-platform development frameworks **[10, 11, 12]**.

There are several cross-platform development frameworks in use **[10, 11, 12]**, including Titanium, PhoneGap, RhoMobile, WidgetPad, and Xamarin **[13]**. One of the latest and most preferred is React native **[10]**, which is a JavaScript framework for building mobile application with native look and feel for applications to work on both iOS and Android **[14]**. It was introduced by Facebook in 2015 **[15]**.

With cross-platform, a mobile application is developed once and it can run on any operating system, which utilises the concept of "learn once, write everywhere" **[16]**. In contrast, a native application is developed using a specific programming language, e.g. Java, Kotlin, Swift or Objective-C, to work on a particular operating system, e.g. Android or iOS respectively.

The decision to use a native or a cross-platform development framework is relative somehow and depends on several factors. On the one hand, native development provides higher levels of reliability and performance and delivers superior user experience. On the other hand, companies need at least twice the time and resources to launch their product on all types of mobile devices, this is because each operating system must be supported by a discrete version of the application.
Moreover, for native platform development, programmers have to be skilled in multiple programming languages, frameworks, and tools to support each of the sought native operating systems. However, for a cross platform development, such as React Native, developers do not build a "mobile web app", an "HTML5 app", or a "hybrid app", but they usually develop a mobile application, which is indistinguishable from a single-platform application, built using java or swift. It simply requires putting the building blocks of the application together using JavaScript or React **[16]**. Actually, there are several advantages for cross-platform development, including improved time to market, overall development cost, portability and maintainability and many more.

In addition to the advantages mentioned above, of using cross-platform frameworks, more specifically, there are several reasons for choosing React Native for our study. React Native is a single code base, completely free and open source, and supported by Facebook. Also, generally, it has a well-supported and faster development and application delivery than other frameworks with large community backing. Moreover, the user interface is rendered using actual native views, which enables better final user experience and better integrated solution than other solutions that simply render a web component inside a WebView. Further, React Native has excellent user experience because of the interaction with native controllers and the potential to achieve a near-native performance because of the direct access to native APIs **[16]**. As a conclusion, it is recommended to start development using React Native if the development team can learn new technology **[17]**.

1.2 *Research Objectives and Problem Statement*

The market of smartphone applications is enormous and rapidly rising. Moreover, developing applications with good efficiency and performance have become very important and really matter **[18]**. In fact, React Native is one of the newly released frameworks that has not much previous research yet. Particularly, the performance of the applications developed in it, in comparison to native ones, is not clearly determined. Therefore, understanding the performance differences between native and React Native is vital in order to decide which development path to choose.

The overall purpose of this thesis is to compare two particular features, namely efficiency and performance, of applications between native and cross-platform development. It focus, however, on iOS and React Native platforms. Specifically, the work will conduct a comparative analysis, of efficiency and performance, of applications developed in XCode framework using objective-c language as a native iOS programming language, against applications developed in React Native as a cross-platform development framework. Notably, efficiency and performance include several parameters to consider, including for example cpu usage, memory usage, application launch time, battery consumption, framerate and many more. Another purpose is to enhance React Native code development to improve its performance to become closer to native iOS resulting into a guideline for more efficient React native code writing. Our focus is on the data centric apps, such apps read and write data to back-end server as well as local storage and files.

To fulfil the above purposes, the thesis attempts to answer the following questions:

1) **RQ1**: What are the differences between native iOS and React Native, as development frameworks, on performance and efficiency of developed applications?
2) **RQ2**: How can we improve the performance and efficiency of application developed in React Native to become closer to native ones?

In our study, we have two hypothesis:

Hypothesis-1: React Native developed application features (specifically performance and efficiency) are analogous to iOS developed application features. [Task-1: Comparative study].

Hypothesis-2: Utilising code-modification engineering on React Native (as a cross-platform framework) application development would result into improvement in software features of applications (such as performance and efficiency), to become comparable to native ones. [Task-2: Code- modification engineering guideline].

1.3 *Overview of this Thesis*

The remainder of the thesis is divided into four chapters. Chapter 2 gives a background and description of both native and cross-platform development focusing on iOS and React Native. Chapter 3 discusses the pertinent literature and sources available, and identifies the research gap on the topic, to where other researchers have stopped. Chapter 4 outlines the research methodology, which was followed to collect and analyse the data. Chapter 5 summarizes the results achieved so far. Finally, chapter 6 shows a small conclusion about the research so far and the findings on the literature review, difficulties and obstacles, recommendation and future work.

Chapter 2 Background

This chapter describes a brief background about native development, specifically iOS. In addition to cross-platform development, specifically React Native. It also displays the structure for both iOS and React Native.

2.1 *Definitions*

2.1.1 Mobile Application

A mobile Application is a software program that is intended to work on smart devices such as mobiles, watches and tablets with a specific purpose **[19]**. There are several categories of mobile apps[1] such as news, sport, entertainment and games. Those Apps are found on the stores like Apple App Store or Android Google Play. They can be either paid or free for download.

In fact, the trend for mobile applications came from the first generation of iPhone in 2010, which is led to the idea of the App store. Consequently, Apple released its App store with 552 applications, 135 of them is free for download. After just one week, ten million applications were downloaded and the popularity of the word ‘apps’ has increased dramatically **[20]**.

2.1.2 Native development

Native application development is the process of writing software that works on a single platform with a specific operating system, processor and hardware. Application development results into executable files that run on a specific type of mobile devices (either iOS or Android), with a full access to the device hardware and functionalities due to the direct interaction with the operating system **[21]**.

For iOS, developers need a Mac device with Xcode IDE **[22]** installed on it. In addition, they need a physical device with compatible iOS to test the application on it.

For Android, developers need a Windows computer with Android SDK **[23]** bundled with Android Studio IDE, in addition to a physical device for testing.

[1] The terms “mobile applications” and “mobile apps”, or “applications” or “apps” will be used interchangeably throughout the thesis.

2.1.3 Cross-platform development

Cross-platform development is the process of writing software that works on multiple platforms with multiple operating systems. In other words, the same code will work on several platforms, like iOS, Android and Windows Phone **[24]**. Previously, if the application operates on a single platform it was seen as sufficient. However, to expand the users base, the application needs to work on all types of devices with different platforms. Actually, there are many different tools and approaches for this type of development which are interpreted, hybrid, cross-complied, component-based and model driven development. Each with its own advantages and disadvantages.

2.2 ***Mobile Platforms and Development***

2.2.1 iOS

iOS is a mobile operating system specific for Apple hardware. It was developed by Apple company in 2007. There are many versions of iOS, the latest version is iOS 13.2.2, which was released on the seventh of November, 2019. Actually, we will use this version in our experiment **[25]**.

2.2.1.1 OS Structure.

The type of the architecture of iOS operating system is layered architecture. It consists of four main layers. Those layers are built on top of each other. The first upper layer is the Cocoa Touch layer, which is responsible for deriving the user interface like widget and controllers, giving access to the main system functions like Camera, other apps and Contacts. The second layer is the Media Layer, which handles audio, video and graphics using several technologies like OpenGL, AV Foundation and Core graphic. The third layer is the Core Services, which is responsible about the core system services needed by the iOS application like location and networking. There are several frameworks exist in this layer, such as Cloudkit framework, Core Location and Core Motion. The last lower layer is the Core OS, which includes the low-level features that other frameworks use and kernel operations. Usually, developers will not use this layer. Examples on the technologies used in this layer are Bluetooth, External Accessory and Security Services **[26] [27]**. Figure 2-1 below shows the layers of the iOS OS.

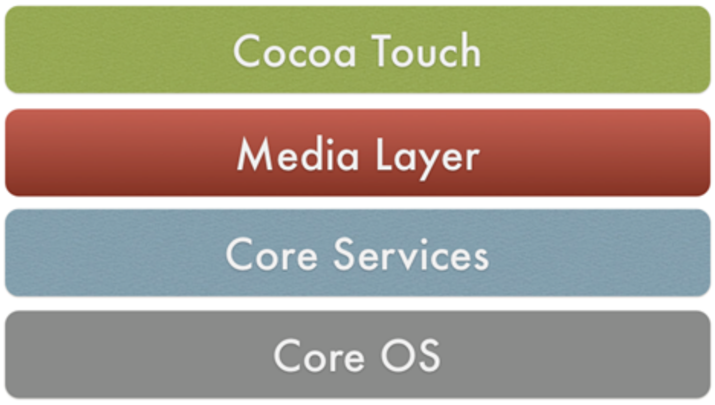


Figure 2-1: iOS Operating System Layers

2.2.1.2 App Structure.

It is recommended to use the model view control (MVC) **[28]** in the development of iOS applications in order to separate the presentation apart from the data and business logic. In fact, using MVC makes it easy to use several screen sizes with different resolutions without the need for big alteration in the code. This is because the view component, which responsible for the presentation is separated from the data and business logic. Therefore, the modification will be only in the view component. Below is the description for each part.

- Model: it is responsible for the data in the mobile app, which includes organizing, sorting and validating data. It notifies the controller when any change in the data happens. This can be done in iOS using data objects, which can be a database **[29]**.

- View: it is responsible for the presentation and user interaction, i.e. it is what the user sees and can interact with on the mobile screen. It also notifies the controller when any user action happens. We can either create custom views or use the default views provided by UIKit framework **[29]**.

- Controller: it is responsible for the management process between the model and the view, it takes data from the model and return it to the view for the presentation process. On the other hand, after user interaction, it takes the modified data from the view to the mode. Model and view do not interact with each other. Figure 2-2 illustrates the MVC architecture.

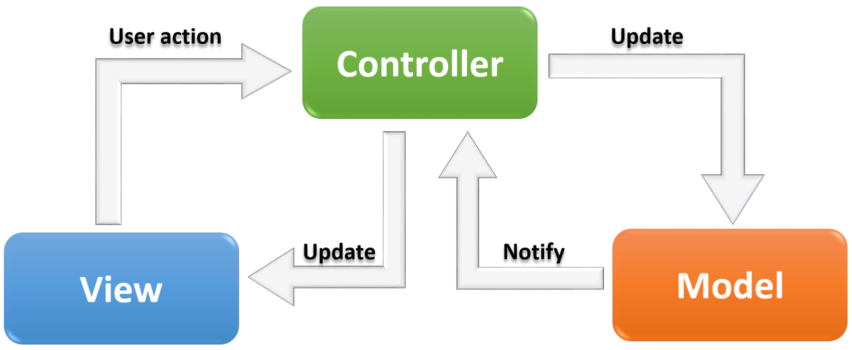


Figure 2-2: Model-View-Controller Architecture [30].

2.2.2 React.

React is a JavaScript library intended to build user interfaces, it was created by Facebook in 2013 **[31]**. The main contribution for React is to automate the update process of the UI. Previously, updating the UI of an application to reflect changes was one of the developer's responsibilities, which means that the developer must manually modify the web browser's Document Object Model (DOM) using JavaScript to update the UI of an application. However, with React, all you need to do is to inform React the current presentation of the application according to the current state. In fact, developers just notify React that the state has changed in order to trigger UI updates by making vital DOM changes.

Components are the heart of any React application. Actually, a component is a module that renders specific output. In addition, it might contain one or more component in the components' output. Examples on components are button, input field and slider **[32]**.

React uses Virtual DOM feature instead of working directly on the browser's DOM, in order to handle the process of re-rendering efficiently. Virtual DOM exists in memory and it represents the browser's DOM. Therefore, writing will not be directly to the

DOM, instead it will be written on the Virtual DOM and react will intelligently decide which changes to reflect on the browser's DOM.

2.2.2.1 Real DOM.

DOM stands for "Document Object Model". The DOM in simple words represents the user interface of the application. Every time there is a specific change in the state of the UI of an application. As a result, the DOM gets updated to represent that specific change. However, manipulating the DOM frequently directly affects performance by making it very slow.
In fact, DOM represents the document as objects and nodes. Using this way, the programming languages can connect to the page. It is an object-oriented representation of a web page that can be manipulated with a scripting language like JavaScript. Anything found in a HTML document can be accessed, changed, deleted, or added using the Document Object Model **[33]**.

2.2.2.2 Virtual DOM.

Virtual DOM is a collection of modules designed to give a declarative way of representing the DOM for any application. Actually, instead of updating the overall DOM when the state of an application is changed, virtual tree which looks like the DOM state is created. After that, this Virtual DOM will figure out how to make the DOM look like this efficiently without recreating all of the DOM nodes. Hence, the performance will be improved in comparison with the real DOM. A virtual DOM is like a lightweight copy of the real DOM. In fact, virtual DOM used in React Native, so this makes the framework with higher performance that other web and hybrid frameworks that use real DOM.

2.2.3 React Native.

React Native is built on top of React, which means that it is working on the same way React work, but it renders UI building blocks of the native (iOS or Android) platform instead of rendering HTML elements. A mix of XML and JavaScript (JSX) is used to develop React Native applications. After that, React Native 'bridge' calls the native application programming interfaces in Java language for Android and Objective-C for iOS, thus the final mobile application will render using native user interface components instead of webviews, so it will look like any other application developed natively. Moreover, applications developed using React Native gives you the ability to access platform specific features like camera, Bluetooth and GPS. In fact, there are many advantages for React Native, the main advantage which makes it better than other cross-platform techniques like ionic **[1]** and Cordova **[34]** that depends on webviews rendering, is the native UI rendering. Actually, both webview and native UI are working but with drawback on the webview performance, so it is better to use React native rather than any other cross-platform technique when the performance is a big matter. In addition, applications developed using React Native can maintain high performance without sacrificing capability, this is because React works separately from the main UI thread. Regarding the update cycle, similar to React, React Native re-renders the views when state or props change.

From developer experience point of view, if any developer will start building a mobile application using React Native for the first time, s/he will be surprised about the simplicity of the work, in addition to the strength of the developer tools and meaningful error messages. Also, the hot reload feature makes React Native unique from all other cross platform frameworks; which means in order to see your code changes all you need to do is press command + R instead of building your application again and wait until it re-runs. Additionally, React Native developer has the freedom to either use any text editor like Atom **[35]**, Sublime text **[36]** or XCode, Android studio.

From code and knowledge sharing point of view, approximately most of the code is shared between iOS and Android platforms, excepts for the pieces of functionality that requires native code then you need to dive into objective-c for iOS and Java for Android. In addition, React Native increase knowledge sharing between the team members because you can target Web, iOS and Android using only one language and a team with the same background, so they can share their knowledge with each other's **[37]**.

Chapter 3 Literature Review

This chapter describes the previous studies that are related to the topic of mobile applications and comparative studies. In addition, it highlights the gap of knowledge for our research.

3.1 *Introduction*

In this part, we mention the previous studies that are related to our field of study. The first section displays previous studies that depends on a specific set of criteria taken from practitioners and domain experts to compare either between native and cross platform approach or between more than one cross platform frameworks. Examples on criterions are: maintainability, number of line codes, ease of development, license and cost, access to device data and hardware. Those criterions were measured by using specific mobile app, then asking to put a number from a specific range to reflect the degree of criteria fulfilment with minor additions in the study like performance. The second section lists studies that uses stopwatches to measure the execution time to access device capabilities like compass, microphone, images, videos and geolocation. It also uses questionnaire and interviews to collect information about the developers experience in several frameworks and programming languages. The third section displays studies that uses several tools to measure multiple performance parameters like memory and CPU usage and response time. Power Tutor **[38]**, Terpn profiler **[39]** for android and Instruments **[40]** for iOS. The fourth section displays all papers that studied React Native for any purpose, weather it is for comparison or it is for classification purpose. Because the number of papers is very limited, we mention thesis that includes React Native in their study.

The last section includes previous studies that stated the importance of mobile application performance in both iOS and Android platforms. In addition, studies showed that most of the errors and bugs are of type performance errors. Moreover, there is a study that indicate the way Android developers fix their performance issues in the mobile apps.

3.2 *Criterion based comparative studies*

On the one hand, several studies compared cross-platform development with both native iOS and native android. Heitkötter et al. **[41]** compared the development of native applications to a number of cross-platform application development frameworks, which are web apps, PhoneGap and Titanium, based on several important criteria taken from practitioners and domain experts, that is widely used when evaluating mobile frameworks. The evaluation was made in two steps, the first one is by developing a small prototypical application for task management and evaluate the specified criterions on it textually in tables. The second one is by evaluating the degree of fulfillment for each one of the criterions from 1 which means very good to 6 which means very poor. The result was displayed in tables, two tables for each framework, one for the development like maintainability, which is measured by number of line code and ease of development, which is measured by the time it took in development and the existence of user comments, and the other for the infrastructure criteria like the license and cost, supported platforms and access to platform specific features. However, the authors did not include React Native in their study, because it was released after the study. The authors concluded that PhoneGap is applicable if a very close similarity to a native user interface look and feel can be ignored. On the other hand, other studies limited the circle by comparing several cross-platform tools and ended up with the appropriate framework for development. One of them is Dalmasso et al. (2013) **[42]** , who made an assessment that evaluate several cross-platform development frameworks, which are PhoneGap, Titanium, Rhomobile and JQuery Mobile. The authors proposed several decision criteria regarding portability concerns to take into account when choosing cross-platform development framework, like application development cost, ease of updating and time to market. Moreover, the authors defined the important requirement that must be exist in a good cross-platform framework. Also, they analysed the architecture of the cross-platform framework in general. In addition to criteria evaluation like Heitkötter et al. **[41]**, they made an experimental approach, which concentrated on a performance comparison from several sides like battery consumption, CPU and memory usage. They made the measurements on a developed android application as a test application with four specific measurement tools for android. They use the Power Tutor **[38]** app in order to measure the battery consumption, which is a

famous application to measure power consumption in android devices. In contrast, two different approaches were used to measure the CPU usage. The first one is to monitor the state change and take a CPU snapshot at each change in the state during the overall activity life cycle of the application. The states are onCreate, onStart, onStop, onPause and onDestroy. The second approach is to read the top result every one second during the overall application life cycle, then compute the average for each state. Regarding memory usage, it was measured using a plugin added to the android studio. However, they did not use a native application as a baseline to compare it with cross-platform frameworks, nor did they include iOS specific measurement tools for evaluation on iOS platform in their study. They reported that the power, CPU and memory consumption is less in PhoneGap because it does not include dedicated user interface components.
Spyros and Stelios **[43]** presented the most famous cross-platform categories, which are web apps, hybrid, generated and interpreted with a description, advantages and disadvantages for each one of them. In addition, the authors made a comparative analysis between the four mentioned approaches. They took subset of the criteria from Heitk¨otter et al. **[41]**. The criteria are market place deployment, which measure if it is easy or hard to deploy the application on the store. Second, the widespread technology, which assess if the application can be created by a common, widely used technology. Third is the data and hardware access, which measure the degree to which the application can access the data and hardware of the device. Moreover, the user perceived performance, which measures the degree of performance (low, medium, high). The measurements were taken by trial, while the final one was taken from the shared information on the web and the author's personal experience. In addition, the authors made a case study in which they developed a simple RSS feed application that will get back the latest apple's news and present the data on the mobile screen. They used Titanium as an interpreted approach in their development. The result of the comparative analysis showed that the generated application is the most favorable approach although there is no non-commercial development framework for it. Interpreted and hybrid came after the generated. Also, the case study showed that the development of the application was easy using javascript and it worked well on both iOS and android without the need to write either iOS or android specific code. After developing the application some criteria were verified on the application. But unfortunately, there is a complete dependence between the application and the

framework, so if there is a need for new feature, it must be supported by the development framework.

3.3 *TimeStamps based comparative studies*

Pålsson (2014) **[44],** studied the execution time as a performance parameter on two cross-platform frameworks, which are phone Gap as a hybrid framework and MoSync as a source code translator tool, and compared them with both native iOS and android development. The execution time measurement was done on a developed application for each cross-platform framework. Actually, the application contains a set of buttons, each button specialized with one feature, when pressing a button, a method is called to make access specific feature, a time stamp is recorded when requesting the method and another one is recorded when completion. The features under study are compass, geolocation and file systems. Several calculations were made to compare the results accurately. The author also conducted a cost evaluation to examine costs involved when adapting cross-platform frameworks. In order to do this, a questionnaire with scale from 1 to 5 was made and distributed among many developers. It contains questions about their skills in several programming languages like java, objective-c and c++. Moreover, questions about the programmer's experience in many IDE's like XCode, android studio and visual studio. Also, interviews were made to examine how the companies organize the teams to work on specific project. After analysing the questionnaire and interviews, the author reported that the choice on which approach to use is difficult and relative, and depends mostly on the skills and competences the target organisation has, and what kind of applications they are going to make. While MoSync gives better performance than Phone Gap, but Phone Gap is more flexible, so it depends. In contrast, other studies limited the circle by comparing cross-platform development with one native type either iOS or android instead of both. For example, Seung-Ho Lim (2015) **[45]** reported a study that compares between Android as a native framework and one cross-platform which is PhoneGap as a hybrid framework by developing a social network service and concentrating on the efficient utilization of device capabilities and the graphical user interface in order to evaluate the better framework in aspect of performance and development cost. The social network service application contains several screens, each one with specific features. First of all is the personal user profile. Second, friends list. Third, timeline new feeds, and Finally the messaging screen, which enables voice, image and video chatting. The measurements were made using time stamps, in the same way Pålsson **[44]** made. The author concluded that the response time as a performance parameter is less in case of native android, but unfortunately because the hybrid framework can't access the hardware, Lim could not evaluate how efficient the device capabilities were. However, the author study just android as a native framework, so we can't generalize the results to iOS.

3.4 *Tools based comparative studies*

While Seung-Ho Lim (2015) **[45]** study the response time as a performance parameter on one cross-platform, Arnesson **[46]** studied more performance parameters, which are memory and cpu usage, energy consumption, and application size in addition to the response time on two different cross-platforms types, which are phoneGap as a hybrid frame work and codename one as cross-compiler framework, in comparative with the android as native one. The goal of the paper was to show the performance variations between the three frameworks and determine which cross-platform tool has the best performance. The author made an experiment where three android applications were developed using the three frameworks respectively. Then, the performance measurements were taken using PowerTutor **[38]** and Trepn Profiler 5.1 **[39]** tools. The application contains functionalities like sort random numbers, print the prime number and write to SQL database. The author concluded that there isn't a very big performance difference between the two cross-platform frameworks but Phone Gap was the best.
Moreover, Willocx et al. **[47]** made a comparative experiment between cross platform, native ios and native android. Their experiment was a quantitative assessment of performance in mobile app development tools. They chose two cross platform tools from different categories, which are PhoneGap and Xamarin. They studied several performance parameters like cpu usage, memory usage, response time, disk space and memory consumption on a small demo application that make search according to the GPS location and return the searched values. Response time was measured in different situations, when starting the application, resuming and pausing the application using DDMS tool for android and instruments tool for iOS. However, CPU usage was measured only during the start of the application using TOP command for android and instruments tool for iOS. In contrast, memory usage was measured two times, the first one is when the application starts and the second is when the application went to the

background using ADB command in android and instruments tool in iOS. Disk space was measured by simply reading the size of the application. The final results from the tools were written in tables for the purpose of comparison. The authors concluded that cross platform tools always add performance overhead over native ones. However, this overhead is frequently acceptable for specific applications. Moreover, behavioral aspects can determine the choice of cross-platform framework. For example, phoneGap could be chose if complex GUI design is needed, while Xamarin **[13]** might be prioritized for CPU comprehensive applications. The authors repeated their study in 2016 with several enhancements. First, more cross-platform frameworks are included in their study. Actually, they compared ten cross-platform frameworks with native iOS and native android in order to draw more general conclusion. Second, a detailed summary is determined for cross-platform development framework selection. However, the authors did not include React Native as a cross-platform framework in their study.

Another previous study was by Xiaoping et al. (2018) **[48]**, who made an experiment that study the following performance parameters: building time, UI response time, memory usage, application size on different cross platform tools, which are Xamarin **[13]** as cross-complied framework, Apache Cordova as a hybrid framework, Titanium as proxy native framework and both native iOS and android, but they did not study react native. The authors developed identical mobile applications in each framework in order to compare them. Basically, the application contains two main screens, the first one is the initial screen, which is a configuration screen for setting up the required parameters. While the second screen is the main screen, which show many contents on the screen with variable sizes according to the configured parameters in the first screen. Actually, they concluded that there are significant differences in the performance characteristics of applications developed using different approaches. For example, building time, which was measured by the time it took to build the application, is less in cross-platform than it is in native. Regarding rendering time, native android and Xamarin **[13]** showed nearly constant rendering time regardless the view size. However, native iOS, Apache Cordova and Titanium showed an increasing in the rendering time with respect to the view size. Memory usage was measured with the same tools used in **[48]**, with a result of increasing required memory when the view size increased. Finally, the size of the application, which was measured by reading the application size, was found to be less in case of native android and iOS than it is in cross-platform framework.

3.5 *Emergence of React Native*

As mentioned earlier, React Native framework has not been widely studied in previous research. In fact, small number of papers studied React Native, one of the notable ones is by Majchrzak et al. (2017) **[49]**. The authors discussed the success factors and features of three cross-platform frameworks: React Native, Ionic and Fuse, outlining their potential strengths and weakness points. The Design-Science Research **[50]** methodology was used in the research, a short survey with ten questions was used to collect data about the frameworks popularity and the responses, which were analyzed to provide the first round of evaluation. A prototype application was developed with several features such as making http request, access device camera, access device contact and make phone call. They found that the most popular issues regarding the development of cross-platform applications are remote data fetching, user experience, application performance and technical implementation. In addition, Nunkesser **[51]** suggested a new taxonomy for mobile app development instead of the original classification, which is web, native, interpreted and hybrid. The new classification is endemic apps, pandemic apps and ecdemic apps. The authors used mobile OS supportability as a main criterion for the classification. Endemic apps include all applications built with IDEs and SDKs that are provided by mobile OS vendors like Apple's XCode and Google's Android studio. However, pandemic apps include applications developed using technologies that are supported by every major mobile operating system, such as HTML, CSS and JavaScript. React Native is classified as pandemic app because it uses JavaScript language. The last one is the ecdemic apps which includes applications built in a framework that uses a language that is not endemic to the mobile OS, such as Xamarin **[13]** that uses C# programming language.
Ghinea and Biørn-Hansen **[52]** studied how interpreted and hybrid apps facilitate using device native features such as camera, Bluetooth and device storage, and how communication bridges are developed and then integrated. The authors made the study as a result of a questionnaire that was distributed among several companies, with most of the respondent said that according to their experience, it is hard to integrate with device API's. React Native and Ionic frameworks were chosen for the study. Two applications were developed, one for each approach. The application is called FetchImage, it fetches an image from the device storage and return it back to the application side in order to display it in an image preview. The execution time of both

bridges was measured using performance.now( ) **[53]**. It was found that the implementation of communication bridges is fast and easy. In fact, it is five times faster in case of hybrid applications.

There are some recent theses that studied React Native, however it is based on single mobile application and compared React Native to native Android or to other cross platform tools. For example, Furuskog and Wemyss (2016) **[54]** conducted a study to assess React Native to more conventional parallel development. The authors compared the execution time of both Xamarin **[13]** framework and React Native **[11]**, they used stopwatches to measure the execution time. In fact, their study was simple because it took a hello word program. They concluded that React Native could potentially be used successfully in order to develop cross-platform applications. However, their evaluation was based on the studied example and thus limited in scalability. For future work, they suggested to conduct performance test on more complicated applications. In addition, they suggested to make the study between React native and another native platform such as android or iOS to compare between native and React Native. Another work by Danielsson (2016) **[55]** reported a study that compares between Android as a native framework and React Native as a cross-platform framework on several features like user experience and performance. An application called Budget Watch was developed, which helps the users to manage their budget. After developing the application, many users asked to use both React Native and Android version, then answer some user experience questions to see if there is a big difference between them. Then, to study the performance, the author used Android specific tool, which is Trepn profiler **[39]** to measure GPU frequency, CPU load, memory usage and battery power on both versions of the application. The answers from the users were analysed and plotted in a graph. The performance tests were repeated three times and the mean values were calculated. The author found two main results, the first one is that despite some differences, but most users could distinguish the React Native app from Android app on the studied features. Secondly, React Native application does not have as good performance as native applications. However, the performance differences are very small. For future work, the author suggested to make another thesis that compares between React Native and iOS platform because it is not included in the study.

3.6 *Performance improvement studies*

Mobile application performance is one of the top concerns for both software engineers and mobile end users, a recent study was done by Khalid et al. **[56]** . The authors studied the user reviews on a set of free and most popular iOS apps that are available on the app store. The authors used a web service called Appcomments that has the responsibility of collecting all user reviews, then parse them into app name, review title and comment. After collecting reviews, they analyzed 6K+ of them and they found that unresponsiveness and heavy resource usage are among the major reasons for the negative user reviews. In addition, Liu et al. **[57]** conducted a study of a set of performance bugs collected form 8 popular android apps. They studied in depth the type of the bug and how it occurred, then they identified common bug patterns. The authors reported that 11K+ out of 60K Android apps have suffered or are suffering from performance bugs.

For the best of our knowledge, there is no previous studies that concerned with improving the performance of applications developed using React Native framework or any other cross-platform tool. On the other hand, Linares-Vásquez et al. **[58]** analyzed real practices that are followed and actual tools that are used by developers to fix performance related bugs. The authors had surveyed 485 open source Android app and library developers. After that, they manually analyzed performance bugs and fixes in their app repositories hosted on GitHub. They concluded that developers rely on both user reviews, manual execution of app and profiling measurements tools to fix performance problems.

3.7 *Highlight the gap of knowledge*

While most of the above studies reported generally the advantages of native frameworks over cross-platforms, the counter-values and advantages of developing in cross-platform are very attractive that include improved time-to-market, overall development costs, portability and maintainability. Although the above studies mainly conducted comparative studies in attempt to understand the capabilities and features of the two different types of platforms, there has not been reported work that studies the performance of React Native code in comparison to native iOS or considers how to improve the development, in terms of application engineering, of cross-platform applications to become as much as comparable to native ones. Therefore, to address this gap, this thesis, will undertake two main research tasks, with greater focus on the first:

1) Will conduct a more detailed comparative study of a cross-platform framework, specifically React Native, and a native-platform framework, specifically iOS, on specific development engineering features, with focus on performance and efficiency (e.g. CPU usage, memory usage, application size). To the best of our knowledge, a detailed analytical comparative study on React Native and iOS, has not been reported previously, thus such study will produce useful results by its own.
2) Will develop a software development engineering guideline that will aim to improve cross-platform applications to become, as much as possible, comparable to native ones on some of the studied features in task 1 above. The method will utilise code-modification engineering, similar to the concept of code-transplantation **[59]**, techniques to improve cross-platform development. Actually, code- modification has been shown that it has potential of identifying software features and aspects and inducing code observation and replacement. To the best of our knowledge, we are not aware of any other method or technique that may provide a potential solution to this above problem.

3.8 *Summary*

As we saw from previous studies, most of the concentration is on other frameworks, such as Titanium and phoneGap, very few papers mentioned React Native in their studies. In addition, from the native point of view, most research was done on Android as a native platform not iOS. Several studies have been conducted to compare React Native to Android development **[55]**. Thus, React Native framework and iOS platform are particularly interesting for further study since they mark a new step of approaches that also introduce paradigmatic shifts. Therefore, this thesis studies React Native application development, as a cross-platform software development framework, compared to iOS application development, as a native software development framework. Specifically, it studies performance capabilities of applications developed in React Native compared to those developed in the iOS to provide a better informed software engineering decision for mobile development.

Chapter 4 Research Methodology

This chapter describes the used research methodology, including mobile application selection and software prevalent features, measure software characteristics or parameters, measurement metrics and data collections.

4.1 *Introduction*

In order to approach the problem statement and answer the research questions to reach a valid conclusion, a set of research methods have been employed. These include identifying research gaps conducted in the related work in chapter two. It also includes detailed steps to obtain results. In this chapter, we describe the approach used for designing the comparative study and how the data were collected. We describe the experimental design, the identified prevalent software features for the study and the different developed applications employed for each of the designed experiments. Also, we describe how the experiments' data were collected and results were analyzed, the tools that were used for measurement and the justification for choosing the selected tools.

4.2 *Experimental Design*

To conduct a comparative study between mobile applications, we undertook the following steps:

1- identification of software features of importance or prevalence to study: this step aims to identify and obtain the most important (prevalent) software features in mobile apps, ones that are most commonly used or developed in most mobile applications. The significance is to identify software features of value and importance to software engineers, that has high performance implications and are frequently developed and exists in mobile applications, for which to understand React Native performance implication and behavior, opposed to studying software features that are seldomly developed or exists in mobile applications or has low performance implication.

2- identification and selection of suitable mobile applications: this step aims to identify and select suitable mobile applications that have or implements the identified prevalent software features of interest.

3- Experiment design and setup: this step aims to design experiments to evaluate of the chosen applications for each of the identified prevalent software features. To measure performance for each of the prevalent software features correctly, this involves setting up an experiment for each, including choosing or developing a suitable experiment, and using the correct measurement tools and data collection method for each of the performance factors, e.g. execution time, CPU, battery usage, etc.

These are described in more details in the sections below.

4.3 *Identification of Prevalent Software Features*

To identify prevalent software features, we conducted a study on the most popular mobile apps in the App Store. As a first step, we took the top five categories that has the largest number of applications from the App Store. We found that App Store has a total of 27 categories. After categories analyzation **[60]**, we concluded that business, education, games, utilities and lifestyle are the top five categories. Next, we scanned the applications in each category in order to take the top ten rated applications, downloaded them and analyzed them well. After, we studied each of the chosen applications and identified the functional features that each application provides. Each software feature was given a unique id and since our comparative study focuses on performance, the degree of each feature's implication on performance is identified as low, medium or high. The implication of a feature on performance was estimated by how frequently it is executed (i.e. frequency of executions) within the application in a session (i.e. rate of execution), thus its increased potential implication on CPU usage and battery usage. For example, the "sign in" functional feature is executed only one time when the application is launched, while "Data retrieval from remote database server" is executed many times in any given session. Then, we calculated how frequently each of the identified functional features is used in amongst studied applications. Functional features that are most frequently used in applications (i.e. has high frequency of use) with high performance implications (i.e. high rate of execution) are the most prevalent features, i.e. the ones that are repeated in most of the applications and has high performance implications. Details of the study is shown in Appendix A. The top five prevalent features, identified from the results of the study, is shown in Table 4-1. As shown, for example, "sign in/up" feature has the highest frequency of use, which means that it repeatedly existed in almost all applications. However, it has low performance implication, since it has a low rate of execution, thus it was ignored. While, “Data retrieval from remote database server”, which has high frequency of use and high rate of execution, thus has high performance implication.

In addition, we found that “processing of images” has high frequency of use in applications and high rate of execution or high-performance implications. Additionally, it is used within or interlinked with other functional features, e.g. search and scrolling. Further, displaying text or numbers, scrolling and search were found as prevalent features.

Number	Prevalent Feature
1	Processing of local data
2	Processing of file data
3	Data retrieval from remote database server
4	Processing of images
5	Search through texts and images

Table 4-1 Prevalent Features.

4.4 *Mobile Applications Selection (Development).*

After identifying prevalent features which are the ones that are repeated in most of the applications and has high performance implications, we searched for applications that have or implements similar prevalent features from relevant Internet repositories, in both languages, but unfortunately, we could not find identical or sufficiently software-feature similar mobile applications with the required components. Thus, we developed and built the applications from scratch. For this, we have taken a similar approach to Willocx et al. **[47]** and Xiaoping et al. (2018) **[48]**. However, instead of developing only one small demo application for each platform, the aim is to develop several applications for each platform to make results more scalable, reliable and generalizable.

To study different features and capabilities of the two mobile frameworks, a set of two separate applications need to be developed, one for each framework, i.e. one developed in React Native, and one developed in native iOS. Each set of the two developed applications implements a specific software feature in order to make sure that all prevalent features are studied separately. This helps to study performance characteristics of each prevalent software feature and measure them correctly, isolating influencing independent variables or factors and/or minimizing their effect. The following five software features will be considered:

1- To develop two applications that implement “data processing”, stored and processed inside the application itself without the need for external storage (i.e. internal storage).
2- To develop two applications that implement the functional feature “file processing”, as a local storage.
3- To develop two applications that implement the functional feature “data retrieval from remote server”, such as a database server, e.g. SQLite **[61]** and MySQL.
4- To develop two applications that implement “processing of images”, such as loading and displaying of images, through searching or scrolling, using remote MySQL database.
5- To develop two application that implement a functional feature of “search”, such as searching through a list of texts, using internal storage.

These applications are described in more details below.

4.4.1 Internal Storage.

As a first step, we started with applications with static data, which means data is stored inside the application itself, i.e. it does not use external storage, local or remote. We developed two mobile applications, one of them is native iOS app using XCode with objective-c language and the other is React Native app using Atom with JavaScript language. These applications contain two main tabs. It aims to sort a dynamic list of numbers using two known sorting methods, which are insertion and merge sort. Since the general functionality on data processing, sorting function was used since it is a data-intensive function and heavily utilise the device memory. The numbers are stored statically inside the application itself; the numbers are displayed inside a list view; the numbers count will be changed to measure performance behavior on scalable list of numbers.

4.4.2 File Storage.

The second step in the research is to develop the same applications in the previous section but the difference is that data is stored inside files, which are local storage.

4.4.3 Database Storage.

In this step, we will develop applications that deal with database, which is remote storage. For study scalability, we will build more than one application for each platform, we will use MySQL database server. We will start by developing application that retrieve only texts from the database. The text will be names of cities. After that, we will develop application that retrieve names of cities with images. The last step is to add the search functionality to both text and images. After developing those applications, it will be ensured that all prevalent features are included in our study.

4.5 ***Experiment Setup.***

In this section, we describe how exactly we are going to test each prevalent feature. For each experiment, we will define the independent variable and list all the dependent variables and how will we fix them to not affect the results. In general, for all experiments, we will use the same mobile device and the same computer machine in all of the experiments below in order to remove any dependent factor of the mobile device used for testing or the computer machine used for development. Specifications of both is mentioned in section 4.6. More details about each experiment are described in the subsections below.

4.5.1 Data Processing Experiment.

In this experiment, we want to measure the effect of data processing feature on the performance of applications. The application is developed to implement number-sorting. The independent variable in this case is the list size, which means the count of numbers to be sorted. We will use power of 10 for the list size. However, there are many independent factors that may affect results, but are fixed. One of them is the sorting methodology, to overcome this factor we will use the same sorting methodology in both iOS and React Native. Another one is the internal device memory; we will use the same device with large memory (32GB) to not affect results.

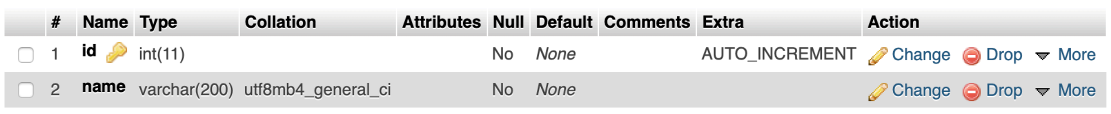
4.5.2 File Processing Experiment.

In this experiment, we want to measure the effect of file processing applications on the performance. The application implements number-sorting application. The independent variable in this case is the list size, which means the count of numbers to be sorted. We will use power of 10 for the list size. Other independent factors are fixed. For example, the type of file (.txt .pdf .docx etc.), we will use .txt file in both iOS and React Native. In addition, we will use the same sorting algorithm on both platforms to make sure that list size is the only factor.

4.5.3 Data Retrieval from Remote Server Experiment.

In this experiment, we want to measure the effect of remote data retrieval applications on the performance. The application retrieves the name of cities around the world from a database server and displays them. The used server is MySQL. The independent variable in this case is the size of the database table, we will change it by modifying the number of records in the table. Power of 10 records will be used. However, there are many other independent variables that may affect the experiment, one of them is the network connection, to eliminate its effect, we will use locally hosted server. Another factor is the type of the server used, we will use the same server, which is MySQL, for both iOS and React Native, to make sure that the experiment is being done on the same server. In addition, the maximum limit of database memory being transferred through the server. In order to solve this, we make it infinity. Also, the internal memory of the device, which is very large and the same for both platforms.

The database table consists of two basic columns, one for the id of the city and the other for the name of the city. The size of the table is 6KB, 16KB, 64KB, 400KB, 4.5MB, 36MB, 360MB, for which the number of records/rows is 10 to the power 1, 10 to the power 2, 10 to the power 3, 10 to the power 4, 10 to the power 5, 10 to the power 6 and 10 to the power 7 respectively. The structure of the table is shown in figure 4-1, below.



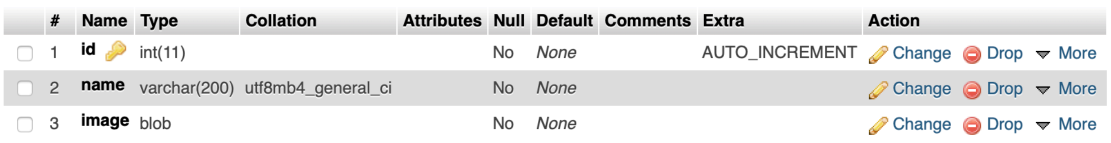
	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
☐	2	name	varchar(200)	utf8mb4_general_ci		No	None			Change Drop More

Figure 4-1 city table with text.

4.5.4 Processing of Images Experiment.

In this experiment, we want to measure the effect of processing images on the performance. The application retrieves the name of cities around the world alongwith their images from a database server and displays them. The used server is MySQL. The independent variable in this case is the size of the database table, we will change it by modifying the number of records in the table. Power of 10 records will be used However, there are many other independent variables that may affect the experiment, one of them is the network connection, to eliminate this effect, we will use a locally hosted server. Another factor is the type of the server used, we will use the same server, which is MySQL. In addition, the maximum limit of memory being transferred through the server. In order to solve this, we make it infinity. Also, the size of image is another independent factor, we will choose the same size for all images which is 31KB to neutralize this factor. Another dependent variable is the type of image (i.e. image extension) used. We found that png is the best extension for mobile apps, so we will use only png extension for all images in both iOS and React Native **[62]**.

The database table consists of three basic columns, the first one is for the id of the city and the second one is for the name of the city. While the third one is for the image of the city. The size of the table is 6KB, 16KB, 128KB, 1.5MB, 8.5MB, 81MB, for which the number of records/rows is 10 to the power 1, 10 to the power 2, 10 to the power 3, 10 to the power 4, 10 to the power 5 and 10 to the power 6 respectively. The structure of the table is shown in figure 4-2, below.



	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	**id**	int(11)			No	*None*		AUTO_INCREMENT	Change Drop More
☐	2	**name**	varchar(200)	utf8mb4_general_ci		No	*None*			Change Drop More
☐	3	**image**	blob			No	*None*			Change Drop More

Figure 4-2 city table with text and image

4.5.5 Search Experiment.

In this experiment, we want to measure the effect of search functionality on the performance. The application displays cities with their images with addition of search bar that enables to search on internal storage for specific city. The result of the search is both the city name and its image

Studied independent variables:

- Number of rows/records: each row contains <CityID, CityName, CityImage>. The number of rows (or cities) is changed x^1 to x^10.

Fixed independent Variables:

- size of image: is fixed at 31KB
- type of image: png (We found that png is the best extension for mobile apps, so we will use only png extension for all images in both iOS and React Native).

4.6 ***Performance Evaluation.***

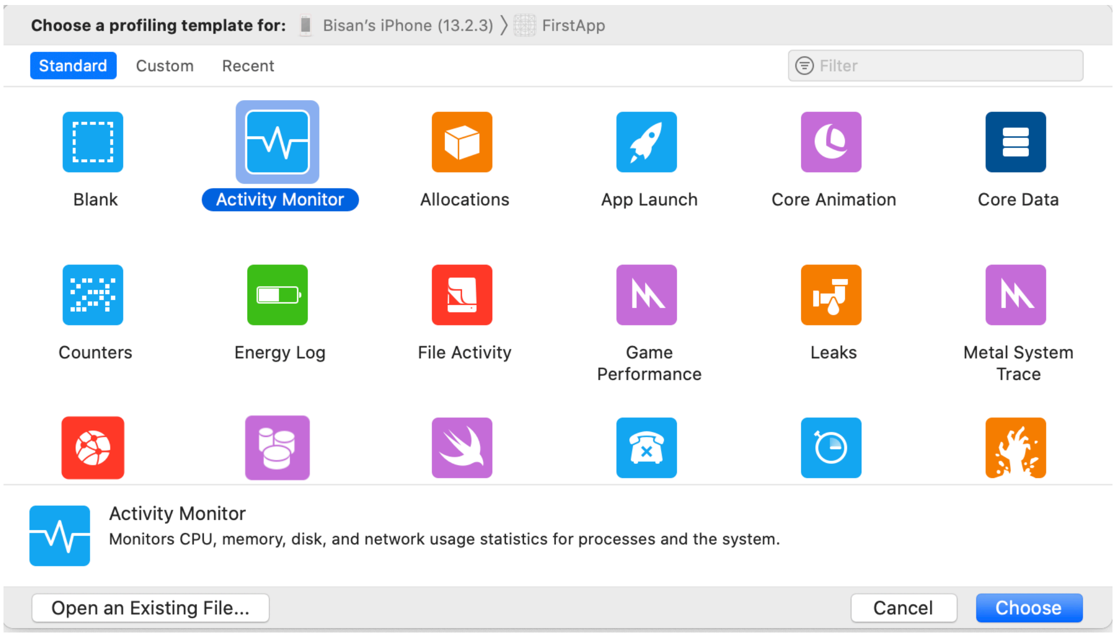
The focus of the study will be on performance and efficiency of mobile applications developed using React Native and native iOS, as mentioned in section 1.1. Performance and efficiency include several parameters to measure. However, before start conducting measurements, we must first choose a set of parameters that are considered the most important. According to Corral et al. **[63]**, there are a number of different parameters to choose from when evaluating the performance of a mobile application. They argue that execution time, memory usage and battery consumption are all useful aspects to consider in performance and efficiency assessment. Dalmasso et al. **[42]** additionally evaluated the CPU usage. In addition, Willocx et al. **[47]** studied several performance parameters including cpu usage, memory usage, response time, disk space and memory consumption on a small demo app. Also, Xiaoping et al. (2018) **[48]** studied building time, UI response time, memory usage, and application size on a very small basic mobile application.

For the purpose of this thesis, the focus will be on CPU and memory usage, execution time and battery consumption. Additional focus will be on the number of screen frames rendered per second, because it affects the rendering speed which is an important factor especially in videos and scrolling through a list view. And finally, the application launch time will be measured, because it gives a first impression about the performance of the mobile app.

In order to take measurements, several tools are reported in the literature. Some researchers used stopwatches **[44] [45] [54]** to measure the execution time. Others **[47] [48]** used Instruments tool **[40]** that is bundled with the XCode to measure other performance parameters of an iOS application. On the other hand, we found that **[46]** used Power Tutor **[38]** and Terpn profiler **[39]** to measure the performance parameters of Android application.

For the purpose of this thesis, Instruments tool will be used for making measurements. In fact, it contains many packages, each package is specialized with one or more parameters. Examples of the packages are: Activity Monitor, Time Profiler and Core Animation **[40]**. Figure 4-3 below shows the main screen of the Instruments tool with its packages. Screenshots for each package is found in Appendix D.

Regarding the measurement environment, each parameter will be measured using both the simulator and real iPhone device. In order to generalize results, more than one simulator will be used and more than one device will be used. iPhone8, iPhone11 will be used as simulators and iPhone7 and iPhone8 as a real device. The same computer machine will be used in the experiment. The specification of both mobiles and computer machine are exist in the next chapter. No other applications, in the run environment, will be active before recording to make sure that they did not affect the results. Local host server will be used as remote server to eliminate the emergence of additional factors such as network connection issues which is not always stable.



Monitors CPU, memory, disk, and network usage statistics for processes and the system.

Figure 4-3 Instruments Tool.

4.7 *Test Experiments.*

In this section, we list some test experiments we undertook in order to determine optimal values for run parameters. More details and screenshots for each test experiment are shown in Appendix C.

4.7.1 Number of Runs Test.

To determine the optimal number of runs for performance parameters, we made three test experiments on all performance parameters. The first one is with five runs, while the second is with 10 runs and the third is with 20 runs. We found that there is no major change in the obtained results when we increased the number of runs to ten or twenty, the differences were very minor. So, results are obtained and averaged for five runs for each experiment. Screenshots for the 5, 15 and 20 runs are shown in Appendix C.1.

4.7.2 Battery Level Test.

To study the effect of the battery level on the performance parameters, we made test experiments on the three battery levels (low, medium, high). Low level is less than 20%, Medium level is between 40% and 70%, High level is above 80%. We took the Application Launch Time as performance parameter and took the measurements in each of the three levels. However, we found that there is no difference in the results regardless the level of battery. So, it doesn't matter. Screenshots of this experiment are shown in Appendix C.2.

4.8 *Parameter Measurements and Data Collection.*

In this section, a brief description and the way data is collected for each performance parameter in the study is described.

4.8.1 App Launch Time.

This parameter tracks the amount of time the application needs from performing action to open the application until the application renders the first frame and then ready for use. It also tracks the application life cycle, which includes all the setup and initialization process to end up with the application in the foreground. We will use App launch package from Instruments Tool to measure this parameter. Actually, It shows the application life cycle in details and the time required for each phase to be completed. The first phase is the system interface initialization. The second one is UIkit initialization and then UIkit creation. After that, the initial frame rendering phase then the application will be in the foreground. We will repeat the test for each application five times. five times will be used because we made test experiments and concluded that five is the optimal number of runs. We will make cold and hot launch to compare the behavior in each state. Cold launch means opening the application for the first time after rebooting the device, which means that the app process does not exist in the system's kernel buffer cache. While hot launch means opening the application after it has been gone to the background **[64]**.

4.8.2 CPU Usage.

This parameter tracks the percentage of the total CPU capacity of the mobile device used by an application in a specific time interval. For our evaluation, the CPU usage at the start of the app will be measured, because it gives an interesting benchmark to compare between iOS and React Native. In addition, CPU percentage during the app usage will be measured. Activity Monitor will be used to measure this parameter. It is one of the packages that comes with the Instruments tool.

We will collect the percentage of CPU usage by each application by making five times of runs on each application.

4.8.3 Memory Usage.

This parameter tracks the amount of RAM consumed by the application during the application's operating time. It will be measured when the application starts and become in the foreground, and when it goes to the background. In addition, we will measure the usage of RAM memory used while the application is in use. We will make five runs on each application.

4.8.4 Frames Per Second.

This parameter tracks the number of frames that are rendered per second and the percentage of GPU hardware utilization. In addition, it tracks the minimum and maximum FPS, the minimum and maximum for the percentage of GPU hardware utilization. Also, the standard deviation for both FPS and GPU hardware utilization percentage. We will measure FPS by making stressed test. We will use Core Animation package from Instruments tool to measure this parameter. In order to collect data, we will make five times run for each application.

4.8.5 Battery Consumption.

This parameter tracks the usage of the device battery. For sure, mobile users don't want apps to drain their batteries. So, it is vital to check the battery usage. We will use Energy Log package in the Instruments tool to measure this parameter, we will repeat the test on each size list five times. We will make the test on both low and normal power mode.

4.8.6 Execution Time.

This parameter tracks the time it takes for the threads to be executed. We will measure the execution time for the main thread in both iOS and React Native and compare between them. In addition, we will measure the execution time for the bridge in case of React Native. We will use Time Profiler package. We will repeat the recoding on Time profiler five times to get accurate results.

4.9 ***Data Collection and Analysis.***

All the collected data above will be exported in excel sheets and tables. They will be analyzed regarding mean, medium, minimum and maximum values. They will also be plotted in statistical charts for the purpose of comparison.

4.10 ***Experiment Scenarios***

Below is a list of the repeated experiment scenarios performed in order to take measurements:

1. Made cold and hot Launch for all applications to measure launch time
2. For each software prevalent feature, run each developed respective application for changing values of the respective independent variable, e.g. list size, table size, to measure CPU usage.
3. For each software prevalent feature, run each developed respective application for changing values of the respective independent variable, e.g. list size, table size, to measure memory usage.

4. Scroll in the list for 60 seconds to measure the frames rendered per second. 60 seconds was chosen as a measurement timeframe, because the ideal rendering is 3600 frames per 60 seconds.
5. Run each application on both low and normal power mode to measure power consumption.
6. For each software prevalent feature, run each developed respective application for changing values of the respective independent variable, e.g. list size, table size, to measure execution time.

The figure below summarizes the overall phases that we will follow in our research

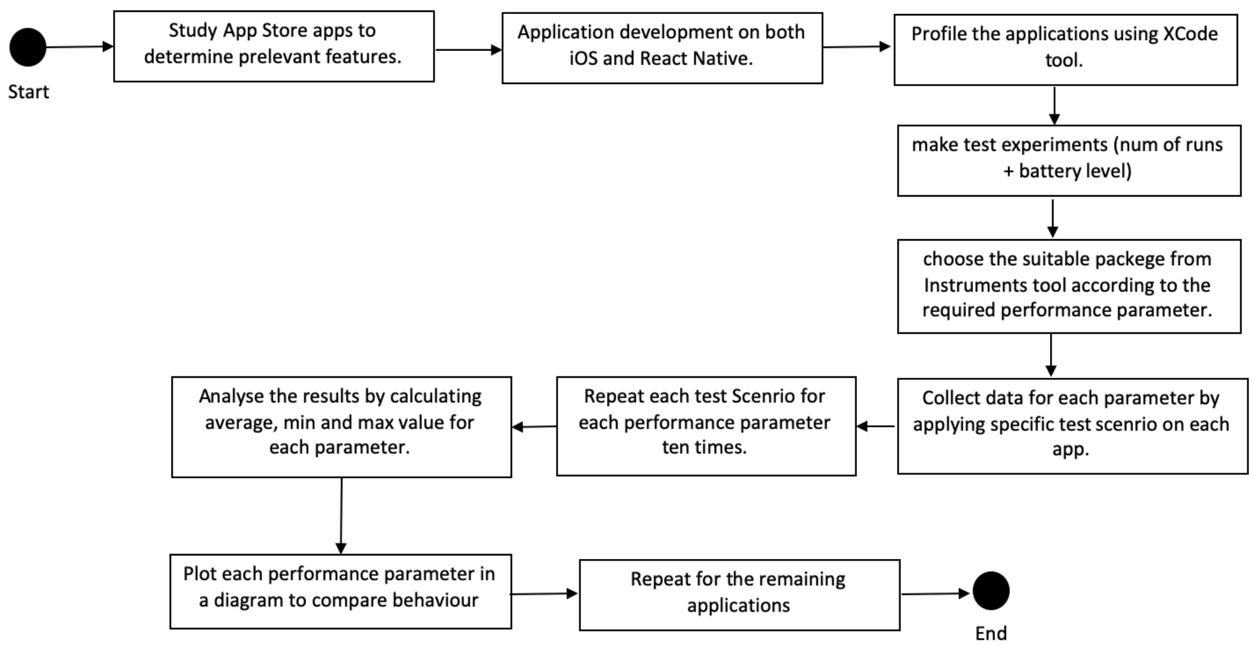

Start
Study App Store apps to determine prelevant features.
Application development on both iOS and React Native.
Profile the applications using XCode tool.
make test experiments (num of runs + battery level)
choose the suitable packege from Instruments tool according to the required performance parameter.
Collect data for each parameter by applying specific test scenrio on each app.
Repeat each test Scenrio for each performance parameter ten times.
Analyse the results by calculating average, min and max value for each parameter.
Plot each performance parameter in a diagram to compare behaviour
Repeat for the remaining applications
End

Figure 4-4 Methodology Phases.

Chapter 5 Experiments and Results

This chapter presents the results that were recorded from the run experiments as described in the previous chapter. All measurements have been conducted five times to make sure the results are correct. Also, mean, medium, minimum and maximum values are listed for each test scenario. In addition, a statistical chart is included for each measurement.

5.1 *Mobile Application Development.*

Based on the results of the prevalent features study, for which the top five prevalent features were considered, as described in chapter 4, five sets of, each with two separate, mobile applications were developed, which resulted in ten mobile applications. Five of them were written in Objective-C for the native iOS platform, and the other five were written in JavaScript for the React Native.

Each two applications, i.e. each set, were developed to implement a prevalent feature. Two of the developed apps used the application's specific data to store the list of numbers and the other two used file systems to store data. The reminder of the developed applications deals with MySQL database server. In addition, to generalize our results the applications deal with texts and images, not only numbers. Also, we developed applications that include search functionality on both text and images.

In fact, the development results showed that the two versions of the developed applications are very similar in the UI, this is because React Native renders native UI components. Screenshots of the developed applications can be found in Appendix B.

The following sections describe the categories for the top five prevalent features and the applications developed to study each feature in more details.

5.1.1 Data Processing (Application 1&2).

In order to test the data processing feature, two applications were developed, the first one is Internal Sorting iOS (Application 1), which is an iOS mobile application that has two tabs, in each tab there is a list view, which renders array of unsorted numbers that are generated randomly, the array size can be varied. Each tab is concerned with specific sorting algorithm. The first tab is for insertion sort while the second is for merge sort. After the numbers are sorted, they will be rendered in the list view. The second application is Internal Sorting RN (Application 2), which is the same as the previous one but it is the React Native version. To make sure that the two tasks (applications) measure the performance of the data processing feature, we designed them to deal and process only internal data. Sorting, as a computational process, can be one of the high CPU demanding tasks, for data processing, thus was selected. The sorting process was designed to start after pressing the sort button to make sure that it doesn't affect the launch time for the applications.

5.1.2 File Processing (Applications 3&4).

The goal of this experiment is to measure the performance of the file processing feature. In order to do this, two mobile applications (tasks) were developed, one of them is File Sorting iOS (Application 3) and the other one is File Sorting RN (Application 4). The apps are the same as the previous two, but the numbers are loaded from external files instead of an internal array. To make sure that file processing feature is measured correctly, we read from more than one external file, but only one file is opened at a time in the same experiment. The size of the file was changed each time. The experiment was repeated for several file sizes to study the effect of changing the size of the file. In addition, a button is used to load the file to not affect the application launch time.

5.1.3 Data Retrieval from MySQL (Applications 5&6).

This feature study the effect of remote data retrieval. The data is stored in a remote database, which is MySQL. In order to measure this feature effect on performance parameters, we developed two mobile applications, one of them is iOS, Database Text iOS (Application 5), while the other is React Native, Database Text RN (Application 6). Both applications make connection with MySQL database, then retrieves names of cities and displays them in a list. The size of the database table, or more precisely the number of rows in the table, is varied by changing the number of cities to be displayed. The applications were designed specifically to measure only the performance of remote data retrieval by implementing the communication part with the database after loading the views and come into foreground.

5.1.4 Processing of Images (Applications 7&8).

To measure the effect of processing images feature on performance parameters, we developed two mobile applications: one is Database Image iOS (Application 7) and the other is Database Image RN (Application 8). Both applications make connection with MySQL database, then retrieves names of cities with their images and displays them in a list. The size of the database table will be varied by changing the number of cities to be listed. The two applications were designed to measure the image processing feature, opposed to just text processing as in done in Applications 5&6. To ensure performance parameters are correctly and accurately measured, measurements are taken after the applications retrieve and add the image, at the right place in the screen, and after loading and displaying the image and come into the foreground. The image extension ".png" was used for images, since it is the most commonly used image format.

5.1.5 Search (Application 9&10).

The goal for this is to measure the effect of performance parameters on the search feature. To do this, two applications were developed which are Search iOS (Application 9) and Search RN (Application 10). The developed applications were designed to provide the ability to search from internal storage through the cities and retrieve the name and image of the cities that starts with or contains specific characters. The size of the database table, or more precisely the number of rows in the table, is varied by changing the number of cities and images to be displayed.

Each application in iOS was compared to its counterpart in React Native. Applications was compared on the following performance parameters: launch time, CPU and memory usage, frames per second and battery consumption. Both sets of applications were compared according to the test scenarios described in the previous chapter. Size of lists, files or tables, as appropriate, were variably changed in magnitude of power of 10, i.e. 10 ^ x, where the value of x changed from 1 to 7, i.e. 10 ^ 1, 10 ^ 2, 10 ^ 3, 10 ^ 4, 10 ^ 5, 10 ^ 6, 10 ^ 7, which are 10, 100, 1000, 10k, 100K, 1M, 10M respectively.

5.1.6 Run Environment.

For the purpose of this thesis, we used the run environment described below:

- A real iPhone device was used, which is iPhone7 with the latest iOS released, iOS version 13.2. Its specifications are 2.34GHz dual core CPU, 2GB RAM, 276MB wired RAM and a total storage of 32GB. The ten applications were individually installed on the device. During the experiments, all other applications were terminated to minimize or remove any potential effect on the applications under study. Additionally, the device was set to the “airplane mode” to avoid any side effects from Wi-Fi or Bluetooth or other type of external connection.
- A real iPhone device was used, which is iPhone8 with the latest iOS released, iOS version 13.3. Its specifications are 3.39GHz dual core CPU, 2GB RAM, 372MB wired RAM and a total storage of 64GB. The ten applications were individually installed on the device.
- A MacBook Pro computer machine with the latest version of XCode and Instruments, which is 11.3. In addition to Atom IDE version1.4.1 . The Mac OS is Catalina 10.15 , 2.6 GHz six core intel core i7 and 16GB of memory. All applications in the Mac were terminated during experiments, except Instruments, in order to not affect any of the parameters like CPU and memory Usage. In addition, Wi-Fi was turned off to not affect battery consumption.

5.2 ***Performance Evaluation: Results.***

This section presents and discusses obtained results from the experiments. As a first step for each experiment, we have conducted dry-runs and tested the experiment design, experiment setup and data collection method to ensure their correctness before conducting wet runs. After making sure that all things worked fine, we started running the actual experiments, i.e. wet runs. For size variability, we stopped at a size of 10M, for list, file or table sizes for respective tasks or applications, because from test runs, we found that results still increase with the same pattern, thus larger sizes would not provide any additional information.

5.2.1 App Launch Time.

Application launch time was measured in both iOS and React Native. App Launch Time package was used for both platforms. In each recording, it makes 5 seconds run and displays the amount of time needed for the application to launch. Also, the time interval for each phase in the application life cycle with the phase description was displayed. Results of making the first test scenario are shown below.

Figure 5-1 shows the average results of applying the first test scenario, which is the cold run, after applying it on all developed apps (Application 1 to 10) with variable sizes (power of ten) for the list of numbers, texts and images repeated five times for each size.

Figure 5-2 shows the average result of the hot run on variable sizes for the list of numbers, texts and images applied on all developed apps (Application 1 to 10).

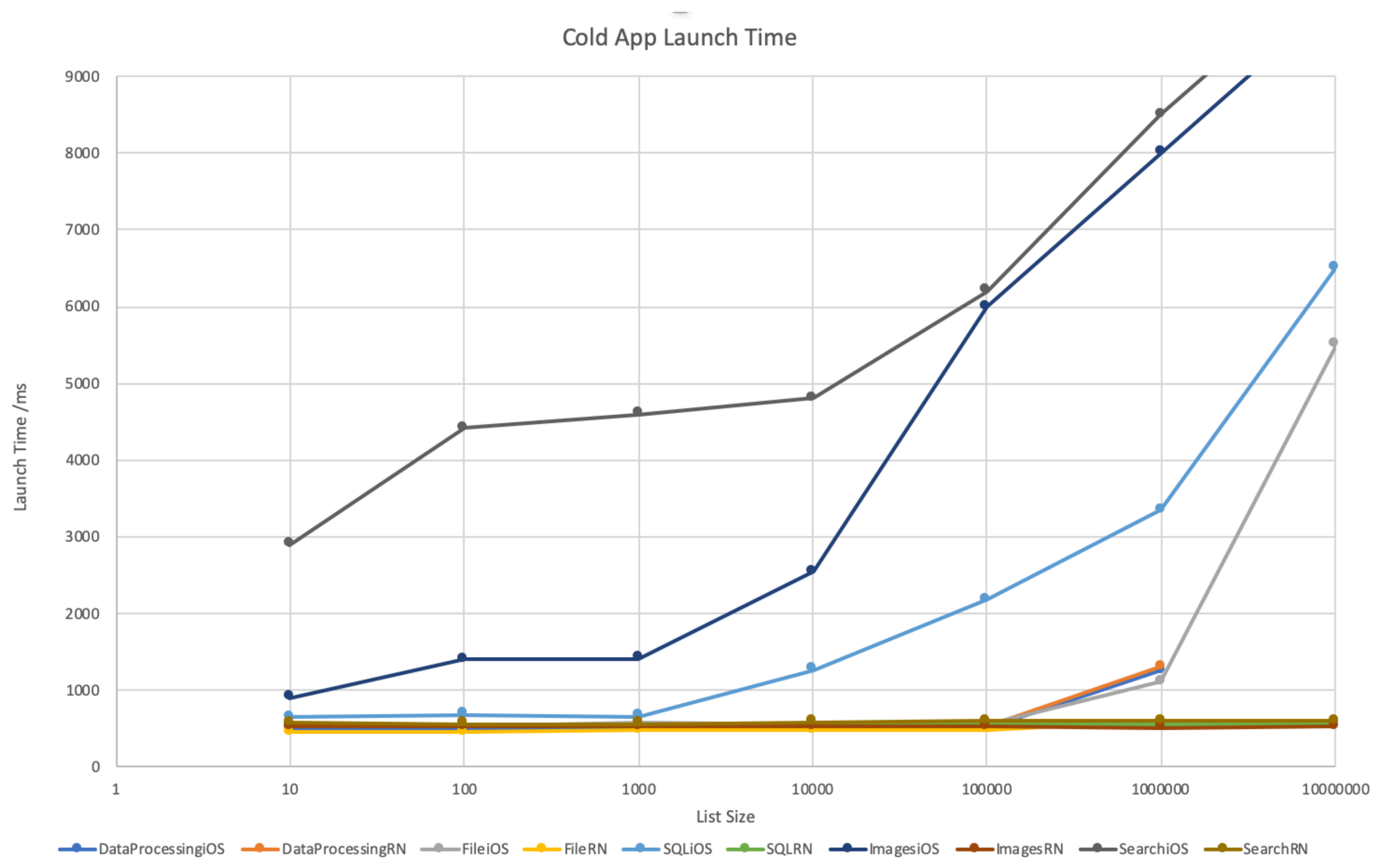


Figure 5-1 App Launch Time cold run all Apps.

As we can see, from the above figure, the first four apps have close results for both iOS and React Native for list sizes up to 100K. However, the result for iOS increased rapidly when the size become 1M, so we increased the list to become 10M but we obtained more than 5 seconds, so we stopped. In comparison to iOS, React Native application's run time still have the same value for 1M, which mean it didn't increase rapidly as iOS, we also tried 10M and 100M and it still didn't increase, so it is better than iOS on big list sizes for the first four apps. However, for the rest of apps which deal with remote database server, React Native apps have lower application launch time than iOS; there is a big difference especially in search and image apps. This because iOS is a secure platform and it needs additional time because of the handshaking made before connecting to remote server. In case of React Native, apps have less launch time because there is not as much security handshake like iOS, this leads to lower time for application launch.

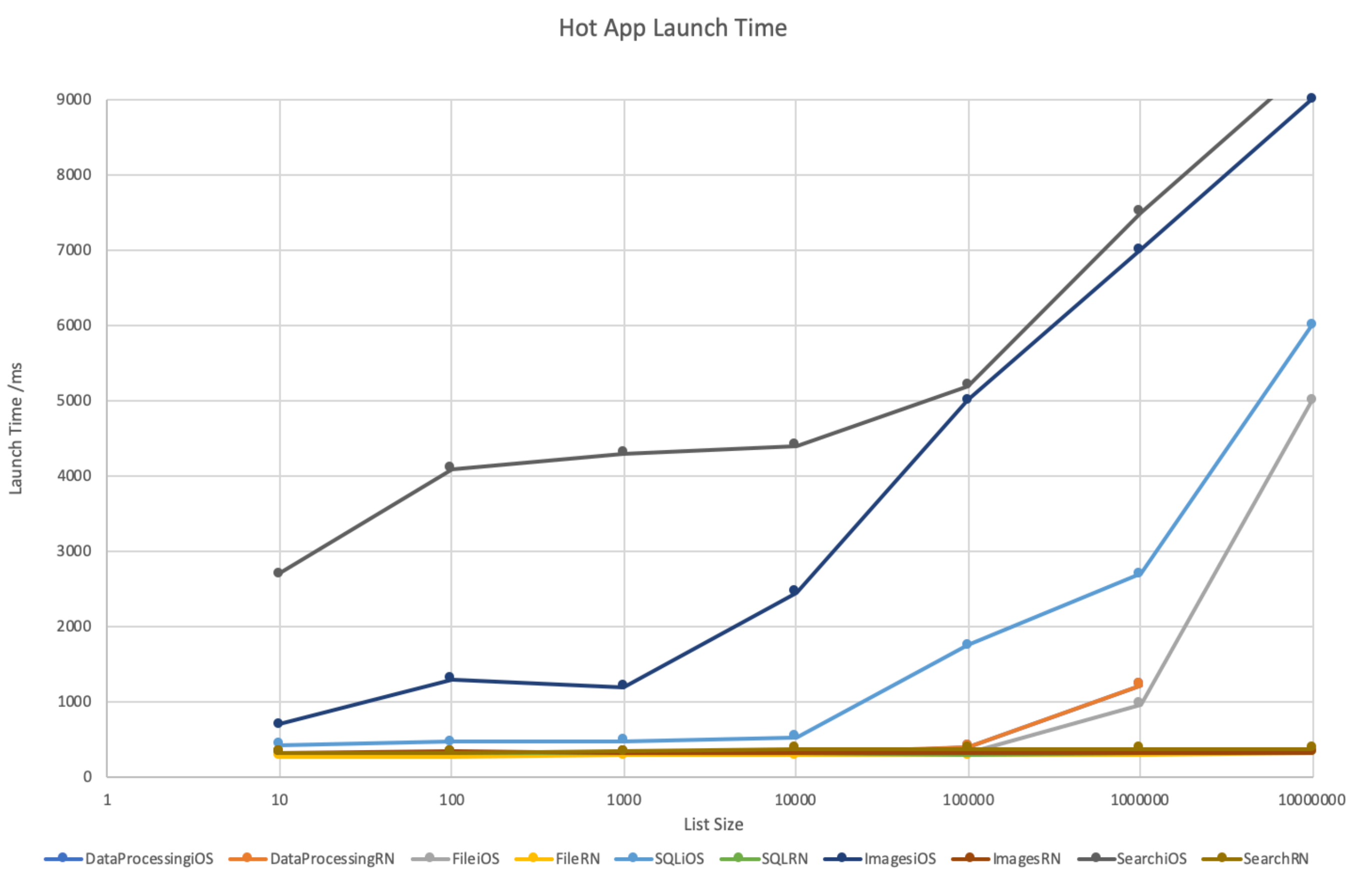


Figure 5-2 App Launch Time hot run on all Apps.

In the above figure, iOS is faster than React Native in the apps that deals with internal storage. However, React Native is faster in case of the apps that deal with remote server. In addition, we noticed that in the case of the cold run, which is in Figure 5-1, more time is required to launch the app than the hot run. This is simply because the app is launched for the first time and the app process does not exist in the system's kernel buffer cache, while in the case of the hot launch in Figure 5-2, the app is not killed but it exists in the background, so the app process still exists in memory.
As a conclusion, Application Launch Time is less in React Native than iOS for most applications. This is because of the hot reload feature which exists in React Native rather than iOS, which needs less time to start the application.

5.2.2 CPU Usage.

CPU usage was measured for both iOS and React Native platforms. It was measured by applying the second test scenario, during the usage of the app on all developed Applications with varied size for the numbers list, file and database table. The runs were repeated five times on each size. The average of CPU usage percentage for each list size is displayed in Figure 5-3 below.

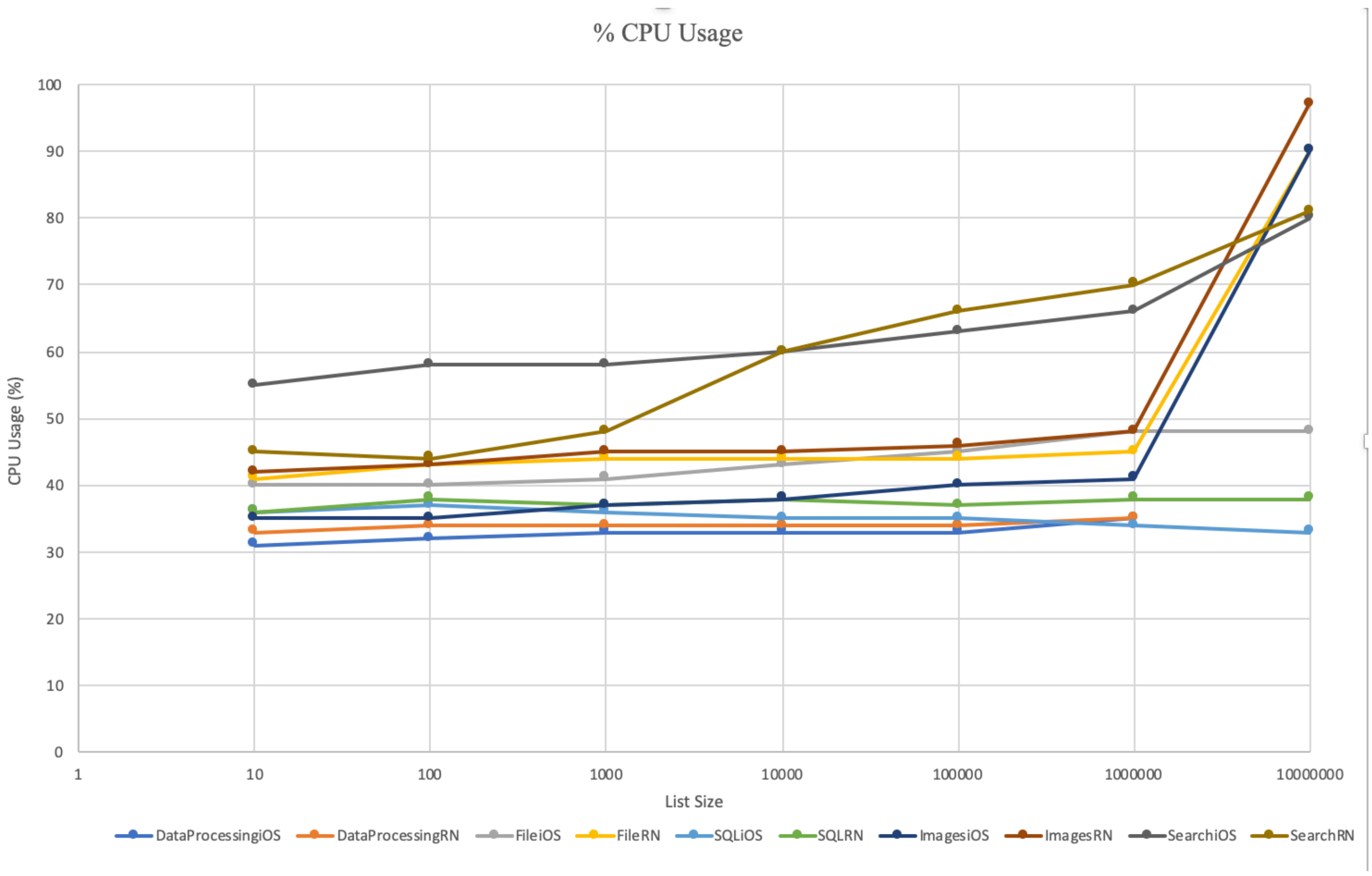


Figure 5-3 CPU Usage on all Apps.

As shown, CPU usage percentage is a little bit higher in React Native than iOS for most applications. However, the differences are minor. This is could be attributed to that iOS applications, developed natively, are making more efficient use of the CPU than RN applications. Also, this could be because React Native has more than one thread, this adds more overhead in the CPU than iOS which has only one thread. However, as shown, RN applications are relatively doing very well on CPU usage compared iOS applications, even for large size processing.

5.2.3 Memory Usage.

Memory usage was measured for both iOS and React Native platforms. It was measured by applying the third test scenario, which is to run each developed respective application for changing values of the respective independent variable, e.g. list size, table size, to measure memory usage for each one of the prevalent features. The test was repeated five times on each size, then the average of used memory was calculated and displayed in Figure 5-4.

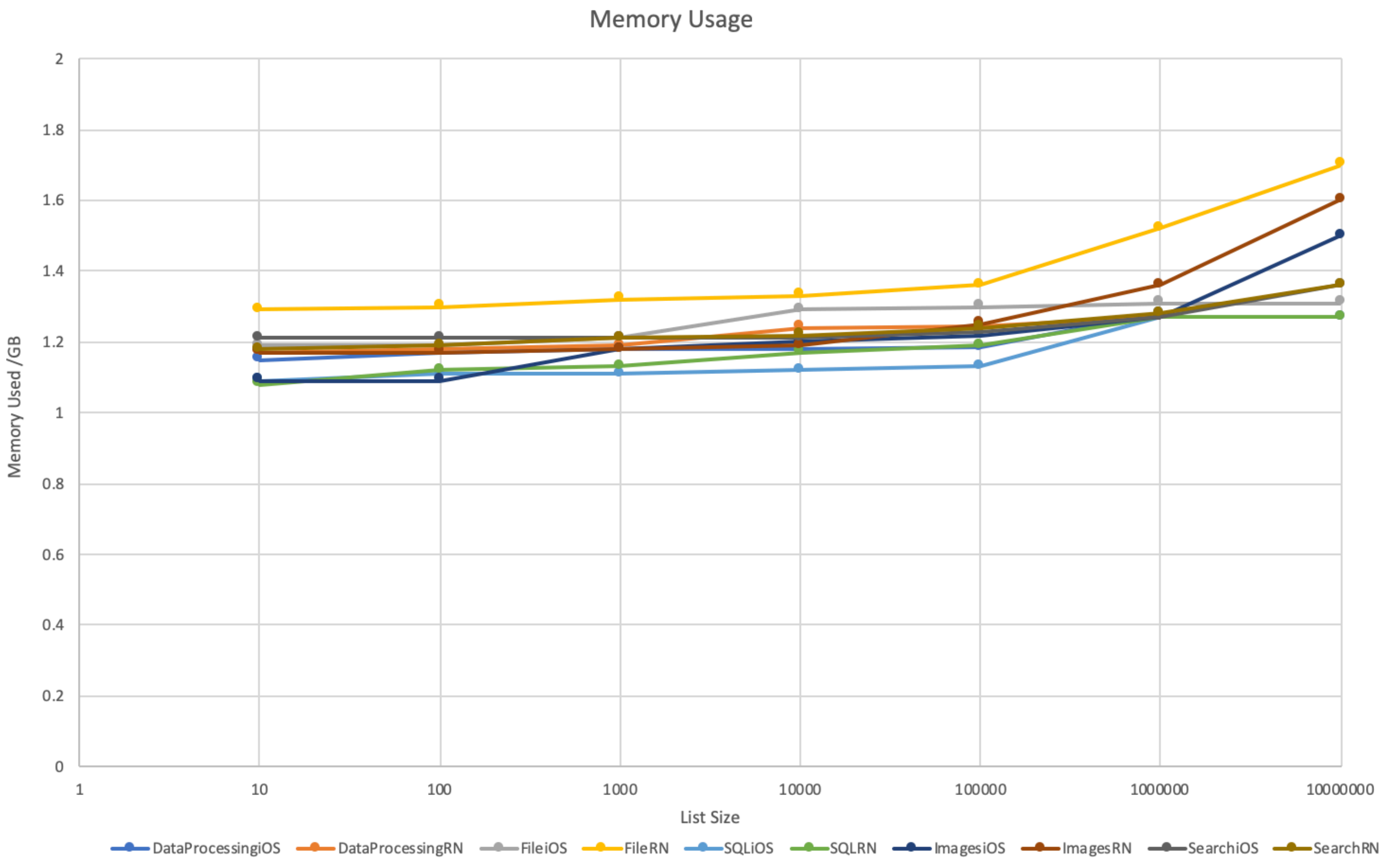


Figure 5-4 Memory used on all Apps.

As shown, memory usage percentage is a little bit higher in React Native than iOS for most applications. However, the differences are minor. This is could be attributed to that iOS applications, developed natively, are making more efficient use of the memory than RN applications. In addition, this because React Native has more than one thread and there is a communication between them, which leads to more memory usage.

5.2.4 Frames Per Second.

The numbers of frames that are rendered per second was measured on both iOS and React Native platforms. The ideal is 60FPS. It was measured on all developed applications by applying the fourth test scenario which is a stress test that aims to check the number of frames rendered when scrolling on variable list size is made. The test was repeated on each size five times and the average was calculated and displayed in Figure 5-5.

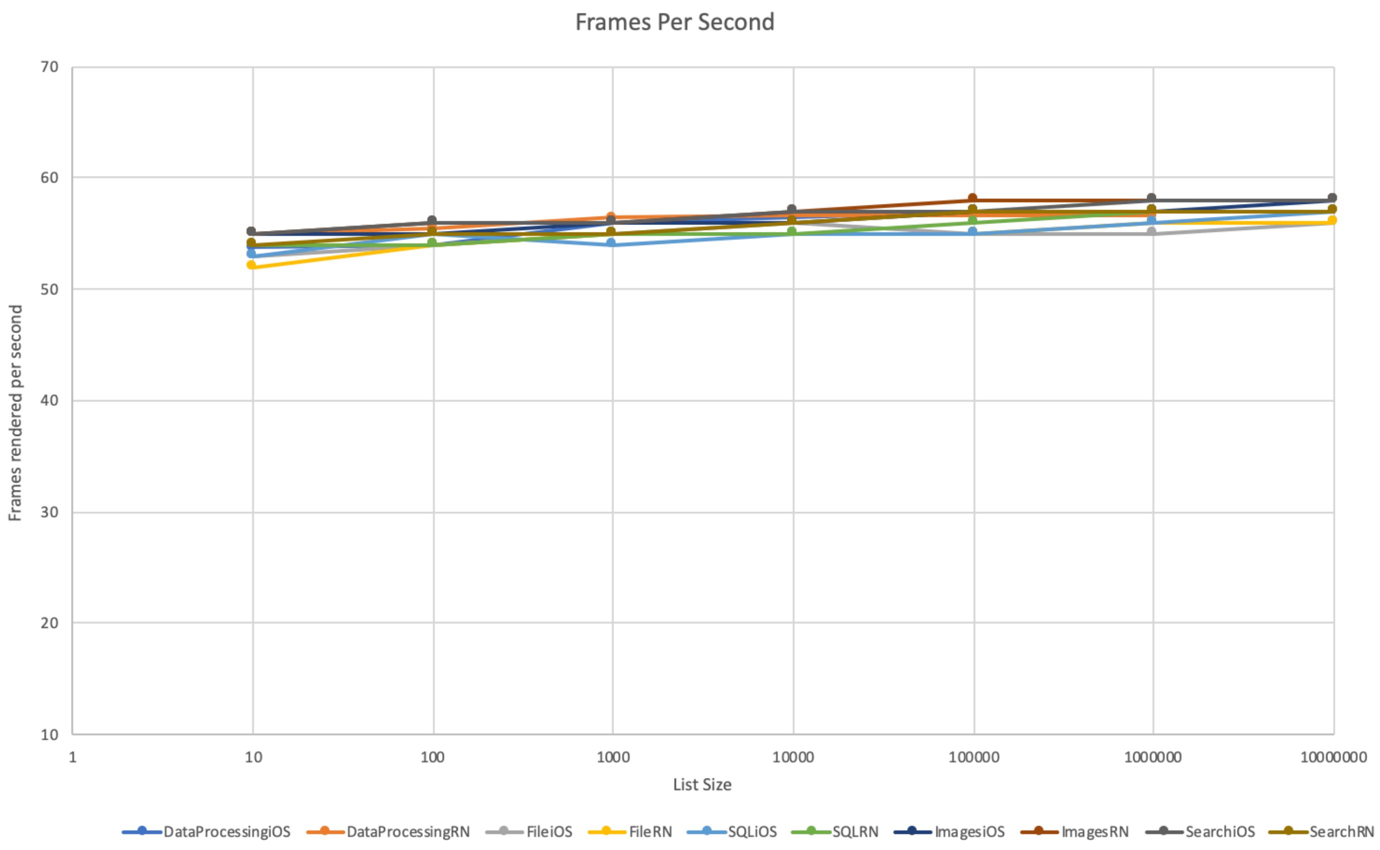


Figure 5-5 Frames Per second on all Apps.

As we can see from the above figure, the results are the same for both iOS and React Native. In case of iOS, it is the normal result we expected because it is a native platform. However, the reason that the result is the same in React Native is because of the specific thread that is responsible for only rendering, thus the logic is separated from rendering which makes the rendering to be around 55-60 FPS.

5.2.5 Battery Consumption.

Battery consumption was measured on both iOS and React Native platforms. It was measured by applying the fifth test scenario, which is running each developed respective application for changing values of the respective independent variable, e.g. list size, table size, to measure the power consumption. Energy Log was used to take measurements. The Energy Log distributes the energy into 20 levels, zero means very low power consumption and 20 means very high. We applied the test on all developed applications in normal and low power mode.

As shown, the results are very low power consumption for all applications in the low power mode and low power consumption for the applications under normal power mode. However, as shown, applications with implements image processing consume more than apps without image processing. This could be due to image rendering processing, which requires additional processing time.

5.2.6 Execution Time.

Execution time was measured on both iOS and React Native platforms for the main thread. It was measured by applying the sixth test scenario, which is for each software prevalent feature, by running each developed respective application for changing values of the respective independent variable, e.g. list size, table size, to measure execution time. Time profiler tool was used. The results are shown in Figure 5-6 below.

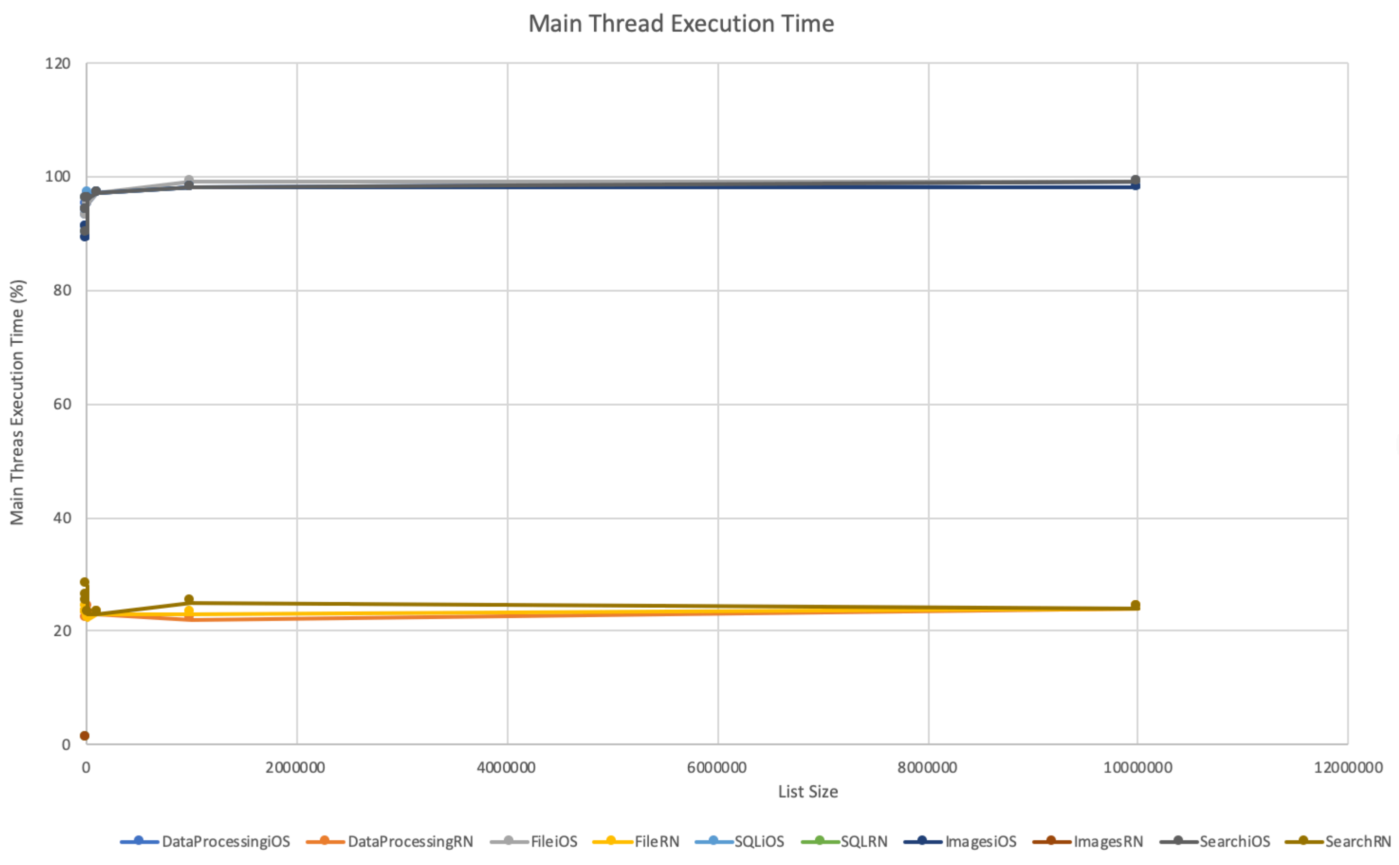


Figure 5-6 Percentage of main thread execution time all Apps.

The program structure for the iOS consists of only main thread. However, the program structure of React Native consists of the main thread, JavaScript thread and the bridge. As can be seen from the results below, the execution time for the main thread is higher for the iOS, it is almost 90-100% of the total time. However, in React native it is only around 25% of the execution time of the total time. This is because Native iOS has only one thread, which is the main thread, but for the React Native, it has also the bridge thread, which is responsible for the communication between the native side and react native side.

5.3 *Performance Improvement.*

This section presents the second task of our thesis, which is to attempt to improve the performance of React Native, where needed and possible, and to provide a guideline to help software engineers avoid React Native performance issues and develop React Native applications as comparably close as possible to the Native iOS.

Others have reported

Based on the outcome of the experiment results, as shown, applications developed in iOS and React Native are comparably similar on many of the performance parameters and changes to React Native, where possible, would not cause any significant performance difference. However, as shown in Table 5-25 to 5-30 and Figure 5-25 to 5-30, noticeably iOS outperforms React Native in Applications that implement image processing, e.g. Applications 9&10. To address this issue, this required further study and investigation of React Native programmable structure and ways to improve its performance.

As a first step, we tried to understand how React Native works under the hood and how it differed from Native iOS. We found an important difference in React Native that makes it different from Native iOS is that React Native code is being executed by more than one thread. It contains the main thread, javaScript thread, and the bridge thread. The latter thread, namely the bridge thread, has the responsibility of communication between the native side and the cross-platform side. Performance in the main thread and javaScript thread are excellent. However, performance of the bridge is not as good as the other threads. This is because the bridge is responsible for the communication between threads. Each thread is separated from the other one, and if we have to send data from one thread to another one, we need to serialize it; which adds performance overhead. In order to improve the performance of React Native App and to become comparable to Native iOS, the overhead caused by bridge communication must be reduced as much as possible, because it is a considerable process that affect performance.

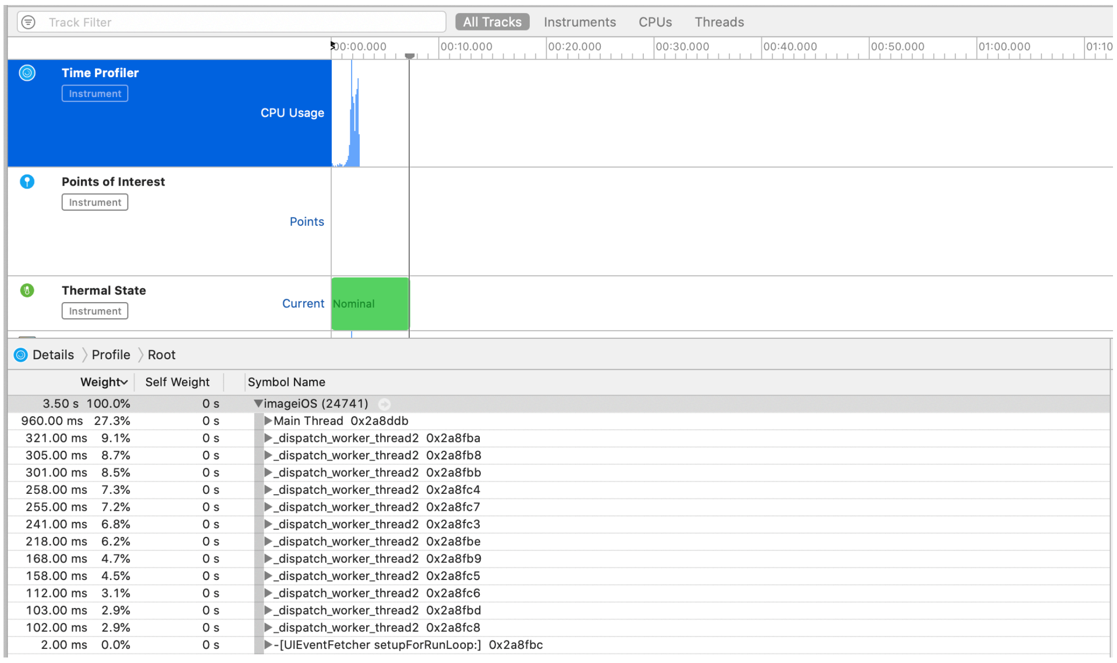
In general, applications that deal with images will communicate heavily between native and cross-platform side. This is because the bridge will serialize the data from the javaScript thread to the main thread for each image in order to render the image and this will lead to a poorer performance.

As a second step, in an attempt to improve performance, specifically execution time, we made an experiment on two developed applications that displays large number of images, one of them using iOS and the other using React Native. We measured the execution time of both applications. We found that React Native is three times slower than iOS. We tried to understand the reason behind this, we found that because of the need for passing data through the bridge for each image; this will make the application slower as the number of images increases. In other words, as the size of the data being passed through or processed by the bridge thread increases, the execution time of the respective implemented feature in the React Native application increases, potentially worsening the performance of the application as a whole.

A possible solution to this performance problem is to store images in the native side, so that there is no need for the communication between the javaScript thread and the main thread; this because the images are already in the main thread. After making this change, in an another experiment, we found that the execution time for React Native has improved to become very close to each iOS'. As shown, in Figure 5-36, before the introduced solution, the execution time was 960ms and 3.65s for iOS's and React Native's applications respectively. However, after the introduced solution, the execution time reduced to 1.17s for the improved React Native app. Below are screenshots from the results for the experiment.

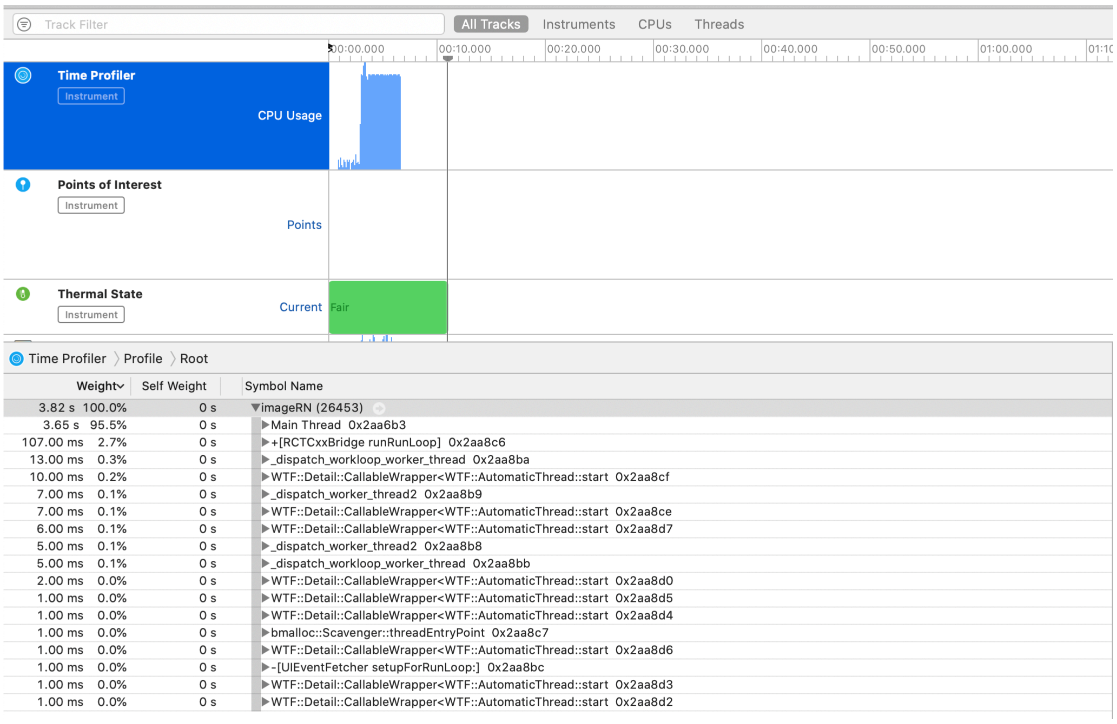
Therefore, as a guideline to React Native software developers, especially for applications that implement relatively intensive or large data communication between React Native and the Native operating system, e.g. image processing, are advised that:

1- Clear identification of software features in an Application that require intensive data processing by the bridge thread, and their performance implication on the application as a whole. Images are found as an example of such data, however, other types of data, that require serialization e.g. binary data, would need to be further studied to confirm their performance impact.
2- Communication of large data over the bridge thread is avoided or reduced.
3- Data that needed to be processed, e.g. images, by the bridge thread is advisably need to be stored, manually, in the Native side during development, to overcome the unnecessary data processing by this thread. In this case, data need to be stored in the native side



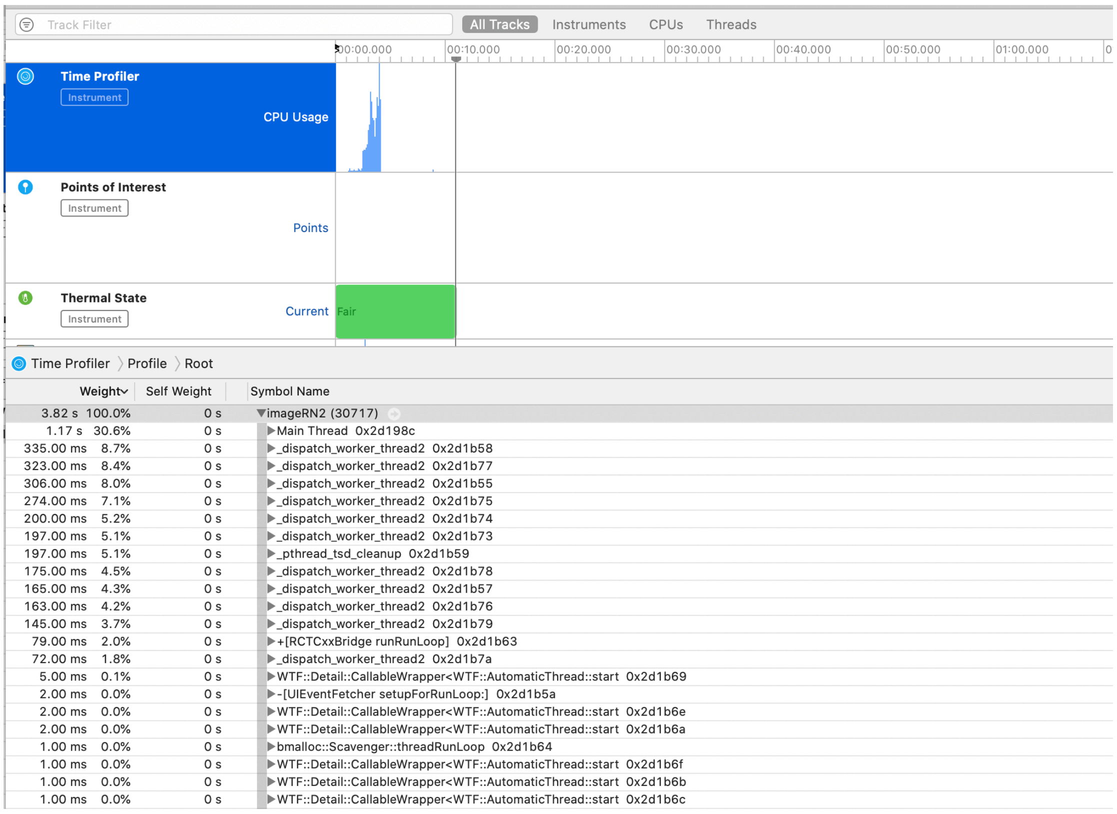
Weight		Self Weight	Symbol Name
3.50 s	100.0%	0 s	▼imageiOS (24741)
960.00 ms	27.3%	0 s	▶Main Thread 0x2a8ddb
321.00 ms	9.1%	0 s	▶_dispatch_worker_thread2 0x2a8fba
305.00 ms	8.7%	0 s	▶_dispatch_worker_thread2 0x2a8fb8
301.00 ms	8.5%	0 s	▶_dispatch_worker_thread2 0x2a8fbb
258.00 ms	7.3%	0 s	▶_dispatch_worker_thread2 0x2a8fc4
255.00 ms	7.2%	0 s	▶_dispatch_worker_thread2 0x2a8fc7
241.00 ms	6.8%	0 s	▶_dispatch_worker_thread2 0x2a8fc3
218.00 ms	6.2%	0 s	▶_dispatch_worker_thread2 0x2a8fbe
168.00 ms	4.7%	0 s	▶_dispatch_worker_thread2 0x2a8fb9
158.00 ms	4.5%	0 s	▶_dispatch_worker_thread2 0x2a8fc5
112.00 ms	3.1%	0 s	▶_dispatch_worker_thread2 0x2a8fc6
103.00 ms	2.9%	0 s	▶_dispatch_worker_thread2 0x2a8fbd
102.00 ms	2.9%	0 s	▶_dispatch_worker_thread2 0x2a8fc8
2.00 ms	0.0%	0 s	▶-[UIEventFetcher setupForRunLoop:] 0x2a8fbc

Figure 5-7 Execution time for iOS image App.



Weight		Self Weight	Symbol Name
3.82 s	100.0%	0 s	▼imageRN (26453)
3.65 s	95.5%	0 s	▶Main Thread 0x2aa6b3
107.00 ms	2.7%	0 s	▶+[RCTCxxBridge runRunLoop] 0x2aa8c6
13.00 ms	0.3%	0 s	▶_dispatch_workloop_worker_thread 0x2aa8ba
10.00 ms	0.2%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8cf
7.00 ms	0.1%	0 s	▶_dispatch_worker_thread2 0x2aa8b9
7.00 ms	0.1%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8ce
6.00 ms	0.1%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8d7
5.00 ms	0.1%	0 s	▶_dispatch_worker_thread2 0x2aa8b8
5.00 ms	0.1%	0 s	▶_dispatch_workloop_worker_thread 0x2aa8bb
2.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8d0
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8d5
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8d4
1.00 ms	0.0%	0 s	▶bmalloc::Scavenger::threadEntryPoint 0x2aa8c7
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8d6
1.00 ms	0.0%	0 s	▶-[UIEventFetcher setupForRunLoop:] 0x2aa8bc
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8d3
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2aa8d2

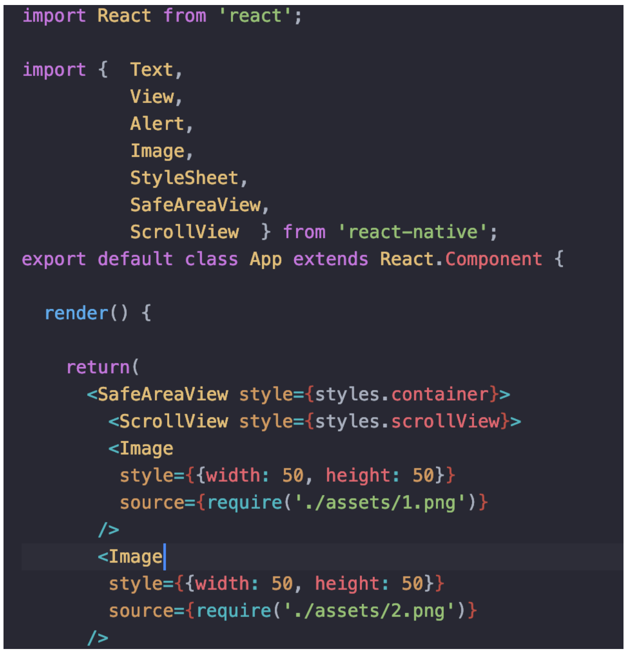
Figure 5-8 Execution time for React Native image App.



Weight		Self Weight	Symbol Name
3.82 s	100.0%	0 s	▼imageRN2 (30717)
1.17 s	30.6%	0 s	▶Main Thread 0x2d198c
335.00 ms	8.7%	0 s	▶_dispatch_worker_thread2 0x2d1b58
323.00 ms	8.4%	0 s	▶_dispatch_worker_thread2 0x2d1b77
306.00 ms	8.0%	0 s	▶_dispatch_worker_thread2 0x2d1b55
274.00 ms	7.1%	0 s	▶_dispatch_worker_thread2 0x2d1b75
200.00 ms	5.2%	0 s	▶_dispatch_worker_thread2 0x2d1b74
197.00 ms	5.1%	0 s	▶_dispatch_worker_thread2 0x2d1b73
197.00 ms	5.1%	0 s	▶_pthread_tsd_cleanup 0x2d1b59
175.00 ms	4.5%	0 s	▶_dispatch_worker_thread2 0x2d1b78
165.00 ms	4.3%	0 s	▶_dispatch_worker_thread2 0x2d1b57
163.00 ms	4.2%	0 s	▶_dispatch_worker_thread2 0x2d1b76
145.00 ms	3.7%	0 s	▶_dispatch_worker_thread2 0x2d1b79
79.00 ms	2.0%	0 s	▶+[RCTCxxBridge runRunLoop] 0x2d1b63
72.00 ms	1.8%	0 s	▶_dispatch_worker_thread2 0x2d1b7a
5.00 ms	0.1%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2d1b69
2.00 ms	0.0%	0 s	▶-[UIEventFetcher setupForRunLoop:] 0x2d1b5a
2.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2d1b6e
2.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2d1b6a
1.00 ms	0.0%	0 s	▶bmalloc::Scavenger::threadRunLoop 0x2d1b64
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2d1b6f
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2d1b6b
1.00 ms	0.0%	0 s	▶WTF::Detail::CallableWrapper<WTF::AutomaticThread::start 0x2d1b6c

Figure 5-9 Execution time for improved React Native image App.

Below are screenshots from the code and the graph for React Native before and after the code modification.



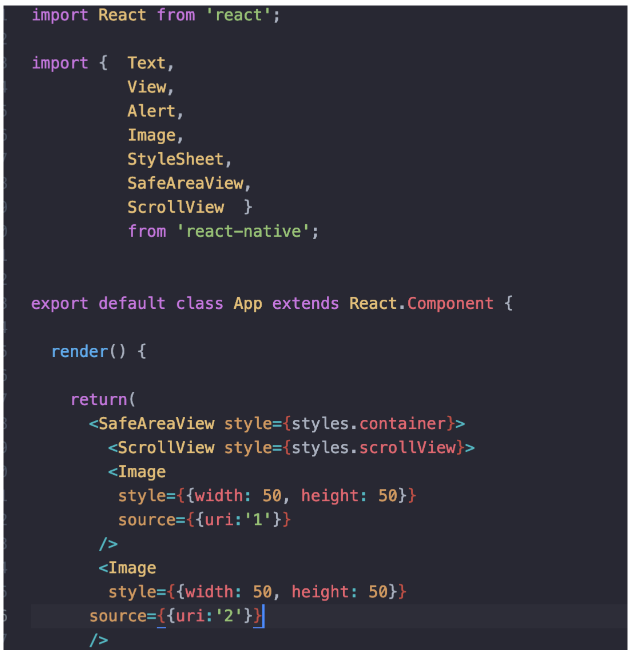
```
import React from 'react';

import {  Text,
          View,
          Alert,
          Image,
          StyleSheet,
          SafeAreaView,
          ScrollView  } from 'react-native';
export default class App extends React.Component {

  render() {

    return(
      <SafeAreaView style={styles.container}>
        <ScrollView style={styles.scrollView}>
        <Image
         style={{width: 50, height: 50}}
         source={require('./assets/1.png')}
       />
       <Image
        style={{width: 50, height: 50}}
        source={require('./assets/2.png')}
      />
```

Figure 5-10 React Native Code Before Modification.



```
import React from 'react';

import {  Text,
          View,
          Alert,
          Image,
          StyleSheet,
          SafeAreaView,
          ScrollView  }
          from 'react-native';


export default class App extends React.Component {

  render() {

    return(
      <SafeAreaView style={styles.container}>
        <ScrollView style={styles.scrollView}>
        <Image
         style={{width: 50, height: 50}}
         source={{uri:'1'}}
       />
       <Image
        style={{width: 50, height: 50}}
      source={{uri:'2'}}
      />
```

Figure 5-11 React Native Code After Modification.

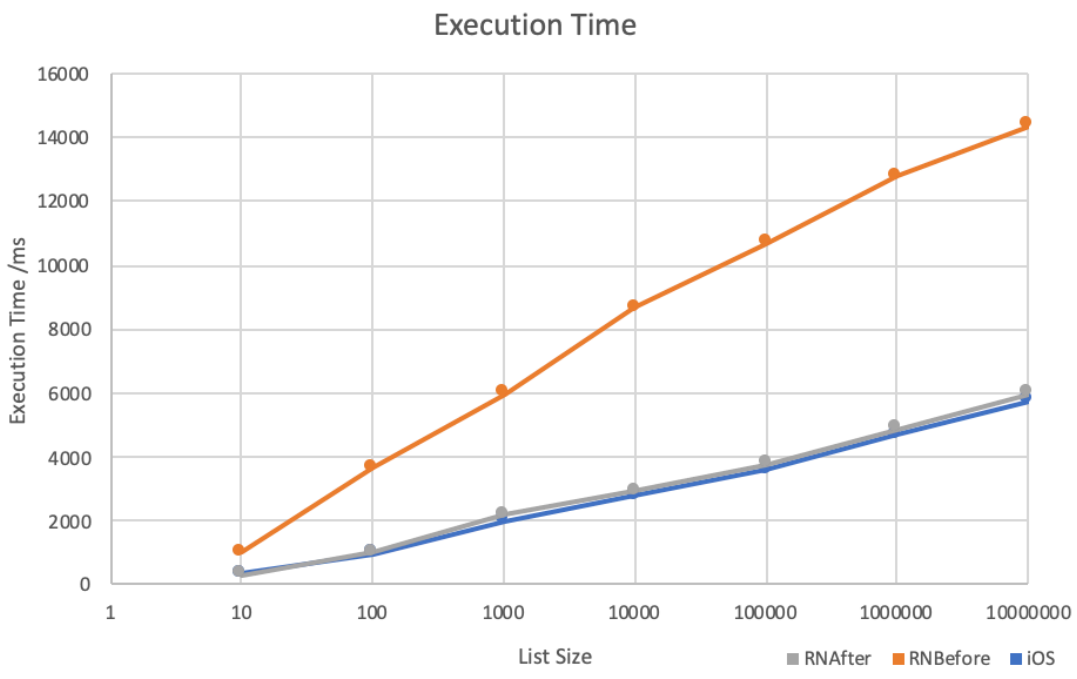


Figure 5-12 Execution Time for iOS & RN before and after modification.

Chapter 6 Conclusion

This chapter displays an overall conclusion about the study. It includes a discussion for the results obtained so far. In addition, the threats and constraints of the study. Also, it presents the difficulties and obstacles that were faced during the research. Finally, it presents the future work for the study.

6.1 *Introduction.*

This thesis studies the comparative performance of applications developed in React Native, as a cross-platform mobile software development framework, to applications in developed in Native frameworks. Specifically, it seeks to study and evaluate the different performance parameters, including execution time, and CPU and battery usage, of React Native in comparison to the performance of Native iOS and attempt to find ways to overcome any arising performance deficiencies.

Ten applications were developed to conduct this study and comparison. In addition, two applications were developed for the case of performance improvement. Performance measurements were taken for all the developed applications. The results were promising for React Native, as there was no significant difference found in the performance between applications in the two platforms.

6.2 *Results Discussion.*

In this research, prevalent mobile software features were identified, ten applications were developed, 10 main experiments were conducted, and data were collected and analysed. CPU and memory usage, frames per second, battery consumption and application run time were measured for five different prevalent software features, in six specified test scenarios using the Instruments tool. Data was collected, and results were analysed and plotted in diagrams for each performance parameter.

Below is a brief description of the results:

- For the Application Launch Time, we found that both iOS and React Native have similar results, but iOS is a little bit faster than React Native for the first four applications (Applications 1,2,3,4), that deal with file and internal memory data processing. However, we found that React Native is faster than iOS in the applications that deal with communicating with database (Application 5,6,7,8). In the Search applications (Application 9,10), we found that React Native is faster than iOS, with relatively a significant difference in Launch time, which perhaps makes React Native a more suitable choice a search functionality.

- For the CPU Usage, we found that both iOS and React Native have similar results. However, React Native apps use more CPU percentage than iOS, but the difference is minor in most of the apps.

- For Memory Usage, we found that React Native apps use a little bit more memory than iOS applications, but there is a big similarity between both frameworks.

- For the Frames rendered per second, both React Native and iOS renders around 57 frames per second which is very close to the ideal rendering rate of 60FPS.

- For Battery Consumption, both consumes the same level of battery in both normal and low power mode.

- For Execution Time, iOS outperformed React Native. React Native employs an additional thread for bridge communication for data processing, which adds an additional execution overhead. We solved this performance difference and obtained almost the same execution time by overcoming the data processing and reducing processing of bridge communication.

6.3 *Threats and constraints.*

Two main threats and constraints in our research can be identified. Firstly, ten applications were developed to evaluate performance across 5 most common prevalent software features. However, only two identical applications were developed to evaluate each prevalent software feature, one for each framework, which adds a threat on the scalability of the results. Evaluating performance across more software features and more applications would improve the generalisability of the results. Secondly, experiments were run on one computer machine and two mobile devices, however running experiments on more different types of mobile devices, with different specifications, would improve scalability of results and would eliminate differences that may arise due to operating system versions or devices.

6.4 *Difficulties and Obstacles faced throughout your research.*

In fact, many obstacles were faced during this research. First, as React Native is a new framework, the number of papers that studied react native are limited. Moreover, there is only one paper that studied the performance. In addition, there was a difficulty in the development process and it took a lot amount of time.

6.5 *Future Work.*

For future work, there are several extensions that could be made to improve the study outcome, these include:

1- The research would benefit from studying additional software features o identify exact performance deficiencies in React Native or iOS.

2- React Native was found to have a performance deficiency and, a corresponding solution was found, for large image data processing, and could benefit from investigating other large data types.

3- Although there are some limited research to study React Native in comparison to Android, as another native framework, this research could be replicated to improve our understanding of React Native development on Android.

4- For scalability and generalisability, experiments could be conducted on more prevalent software features as well as applications that examines them.

References

[1] “Ionic Framework” 17 October 2019. [Online]. Available: https://ionicframework.com/docs. [Accessed 26 11 2019].

[2] “Adobe PhoneGap” [Online]. Available: http://docs.phonegap.com/. [Accessed 21 November 2019].

[3] “Statista” 5 7 2019. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. [Accessed 4 11 2019].

[4] “Statista” [Online]. Available: https://www.statista.com/statistics/377977/tablet-users-worldwide-forecast/. [Accessed 4 11 2019].

[5] “Mobile marketing statistics compilation” Smart Insights, [Online]. Available: https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/. [Accessed 4 11 2019].

[6] M. Iqbal, “Buisness of Apps” 7 8 2019. [Online]. Available: https://www.businessofapps.com/data/app-statistics/. [Accessed 5 11 2019].

[7] “Statista” 18 9 2019. [Online]. Available: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/. [Accessed 4 11 2019].

[8] J. Cowart, “What is a Hybrid Mobile App?” Progress, [Online]. Available: https://www.telerik.com/blogs/what-is-a-hybrid-mobile-app-. [Accessed 29 11 2019].

[9] E. Spence, “Windows Phone Is Dead, Long Live Microsoft's Smartphone Dream” Forbes, 12 July 2017. [Online]. Available: https://www.forbes.com/sites/ewanspence/2017/07/12/microsoft-windows-phone-windows10-mobile-strategy/#81a1b1d172c5. [Accessed 29 11 2019].

[10] G. R, “Top Technologies Used to Develop Mobile App” Fingent, 19 December 2018. [Online]. Available: https://www.fingent.com/blog/top-technologies-used-to-develop-mobile-app. [Accessed 2 5 2019].

[11] Hermes and Dan, “Xamarin Mobile Application Development Cross-Platform C# and Xamarin.Forms Fundamentals”, apress, 2015.

[12] C. Griffith, “Mobile App Development with Ionic, Revised Edition: Cross-Platform Apps with Ionic, Angular, and Cordova”, O'Reilly Media, Inc, 2017.

[13] “Xamarin” Microsoft, [Online]. Available: https://dotnet.microsoft.com/apps/xamarin. [Accessed 1 12 2019].

[14] A. Bento, “Android and iOS” 14 April 2014. [Online]. Available: https://home.ubalt.edu/abento/315/android-ios/index.html. [Accessed 1 5 2019].

[15] Shoutem, “A brief history of React Native” Medium, 3 October 2016. [Online]. Available: https://medium.com/react-native-development/a-brief-history-of-react-native-aae11f4ca39. [Accessed 1 5 2019].

[16] Ideamotive Team, “Choosing React Native for Your Mobile Tech Stack” Idea Motive, 22 February 2019. [Online]. Available: https://ideamotive.co/react-native-development-guide/#what-is-rn. [Accessed 1 5 2019].

[17] R. Mehul, “React Native—Is it Really the Future of Mobile App Development?” Hackernoon, 13 September 2018. [Online]. Available: https://hackernoon.com/react-native-is-it-really-the-future-of-mobile-app-development-31cb2c531747. [Accessed 1 5 2019].

[18] R. O'Connor, “Why Mobile App Performance Matters” Progress, [Online]. Available: https://www.progress.com/blogs/why-mobile-app-performance-matters. [Accessed 5 11 2019].

[19] “Mobile App” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Mobile_app. [Accessed 7 11 2019].

[20] J. Agency, “Brief History of Mobile Apps” [Online]. Available: https://expertise.jetruby.com/brief-history-of-mobile-apps-286fbbf766a9. [Accessed 7 11 2019].

[21] A. Monus, “Understanding native app development - what you need to know in 2019” 19 3 2019. [Online]. Available: https://raygun.com/blog/native-app-development/. [Accessed 6 11 2019].

[22] “Xcode” Apple Developer, [Online]. Available: https://developer.apple.com/xcode/. [Accessed 6 11 2019].

[23] “Android Studio” Android Developers, [Online]. Available: https://developer.android.com/studio/. [Accessed 7 11 2019].

[24] “Cross-platform development” [Online]. Available: https://www.sapho.com/glossary/cross-platform-development/. [Accessed 7 11 2019].

[25] “iOS” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/IOS_version_history. [Accessed 9 11 2019].

[26] “Apple iOS Architecture” Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/apple-ios-architecture. [Accessed 10 11 2019].

[27] “iOS Architecture” [Online]. Available: https://intellipaat.com/blog/tutorial/ios-tutorial/ios-architecture/. [Accessed 10 11 2019].

[28] “Model-View-Controller” Apple Developer, [Online]. Available: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. [Accessed 10 11 2019].

[29] “How To Make iPhone Apps – MVC, One Pattern To Rule Them All” [Online]. Available: https://codewithchris.com/how-to-make-iphone-apps-mvc-one-pattern-to-rule-them-all/. [Accessed 10 11 2019].

[30] “Testing Apples MVC” [Online]. Available: https://medium.com/mobile-quality/testing-apples-mvc-dab15830139a. [Accessed 1 December 2019].

[31] “The History of React.js on a Timeline” RsisingStack, [Online]. Available: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/. [Accessed 12 11 2019].

[32] A. Lerner, “What is React?” FullStack React, [Online]. Available: https://www.fullstackreact.com/30-days-of-react/day-1/. [Accessed 12 11 2019].

[33] A. Haseeb, “Virtual DOM vs Real DOM” Medium, 10 Augest 2018. [Online]. Available: https://medium.com/@ahaseeb12251998/virtual-dom-vs-real-dom-angular-vs-react-framework-vs-libraries-spas-vs-mpa-s-946fceb70955. [Accessed 13 12 2019].

[34] “CORDOVA” [Online]. Available: https://cordova.apache.org/. [Accessed 26 11 2019].

[35] “ATOM” [Online]. Available: https://atom.io/docs. [Accessed 26 11 2019].

[36] “Sublime Text” [Online]. Available: https://www.sublimetext.com/. [Accessed 26 11 2019].

[37] B. Eisenman, “Learning React Native” O'Reilly Media, Inc., 2015.

[38] “PowerTutor” [Online]. Available: http://ziyang.eecs.umich.edu/projects/powertutor/. [Accessed 29 9 2019].

[39] “Trepen Profiler” [Online]. Available: https://www.apkmirror.com/apk/qualcomm-innovation-center-inc/trepn-profiler/. [Accessed 29 9 2019].

[40] “Instruments Help” Apple, [Online]. Available: https://help.apple.com/instruments/mac/current/#/. [Accessed 3 12 2019].

[41] . H. Heitk¨otter, S. Hanschke and T. . A. Majchrzak, “Evaluating Cross-Platform Development” in *Proc. 8th WEBIST*, 2012.

[42] I. Dalmasso, S. K. Datta, C. Bonnet and N. Nikaein, “Survey, comparison and evaluation of cross-platform mobile application development tools” in *Wireless Communications and Mobile Computing Conference 9th*, 2013.

[43] X. Spyros and . X. Stelios, “A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications” *ACM Proceedings of the 6th Balkan Conference in Informatics,* pp. 213-220, 2013.

[44] M. PÅLSSON, “Cross Platform Development tools for mobile” HS Universty of Skövde, [Online]. Available: http://kth.diva-portal.org/smash/get/diva2:754436/FULLTEXT01.pdf. [Accessed 12 5 2019].

[45] Seung-HoLim, “Experimental Comparison of Hybrid and Native Applications for Mobile Systems” *International Journal of Multimedia and Ubiquitous Engineering,* pp. 1-12, 2015.

[46] A. Arnesson, “Codename one and PhoneGap, a performance comparison” 2015.

[47] W. Michiel, V. Jan and N. Vincent, “A Quantitative Assessment of Performance in Mobile App Development Tools” in *IEEE International Conference on Mobile Services*, 2015.

[48] J. Xiaoping, E. Aline and . T. Yongshan, “A Performance Evaluation of Cross-Platform Mobile Application Development Approaches” *ACM/IEEE 5th International Conference on Mobile Software Engineering and Systems,* 2018.

[49] T. Majchrzak, B. Hansen and T.-M. Gronli, “Comprehensive Analysis of Innovative Cross-platform App Development Frameworks” in *Hawaii Internetional Conference on System Sciences*, 2017.

[50] V. K. Vaishnavi and W. Kuechler, “Design Science Research Methods and Patterns: Innovating Information and Communication Technology, 2nd Edition”, 2015.

[51] R. Nunkesser, “Beyond Web/Native/Hybrid: A New Taxonomy for Mobile App Development” in *ACM/IEEE 5th International Conference on Mobile Software Engineering and Systems*, 2018.

[52] A. Biørn-Hansen and G. Ghinea, “Bridging the Gap: Investigating Device-Feature Exposure in Cross-Platform Development” in *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[53] “performance.now()” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now. [Accessed 3 12 2019].

[54] M. Furuskog and S. Wemyss, “Cross-platform development of smartphone application An evaluation of React Native” 2016.

[55] W. Danielsson, “A comparison between native Android and React Native” 2016.

[56] H. Khalid, E. Shihab, M. Nagappan and A. E. Hassan, “What Do Mobile App Users Complain About?”.

[57] Y. Liu, C. Xu* and S.-C. Cheung, “Characterizing and Detecting Performance Bugs for Smartphone Applications”.

[58] M. Linares-Vásquez, C. Vendome, Q. Luo and D. Poshyvanyk, “How Developers Detect and Fix Performance Bottlenecks in Android Apps” in *IEEE International Conference on Software Maintenance and Evolution (ICSME).*, 2015.

[59] E. Barr, M. Harman, Y. Jia, A. Marginean and J. Petke, “Automated Software Transplantation” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[6 0] “42 MATTERS” [Online]. Available: https://42matters.com/stats. [Accessed 25 3 2020].

[61] 13 February 2019. [Online]. Available: https://ourcodeworld.com/articles/read/737/everything-you-need-to-know-about-sqlite-mobile-database. [Accessed 11 12 2109].

[62] “React Native Performance: Major issues and insights on improving your app’s performance” SIMFORM, [Online]. Available: https://www.simform.com/react-native-app-performance/. [Accessed 5 5 2020].

[63] L. Corral, A. Sillitti and G. Succi, “Mobile multiplatform development: An experiment for performance analysis” in *The 9th International Conference on Mobile Web Information Systems (MobiWIS)* , 2012.

[64] A. Dutta, “iOS App Launch time analysis and optimizations” 27 Augest 2017. [Online]. Available:

https://medium.com/@avijeet.dutta13/ios-app-launch-time-analysis-and-optimization-a219ee81447c. [Accessed 12 12 2019].

Appendices

Appendix A: App Store Study.

category	Number of apps	percentage
games	230271	22.3%
business	179576	10.07%
education	151152	8.82%
lifestyle	156809	8.61%
utilities	133543	6.31%

Feature	Feature ID	Performance Implication
Video conferencing	F1	high
Data retrieval from remote server	F2	high
Voice calls	F3	high
Processing of texts	F4	high
Sign in/up	F5	low
search	F6	medium
chat	F7	medium
Add/follow friends/team member	F8	medium
Record and play voice calls	F9	low
Send money be email address	F10	low
Pay on sites	F11	low
Translate unknown words	F12	low
Processing of image	F13	high
Communicate and Compete courses with others	F14	low
Access hundreds of games and activities and lessons	F15	medium
Track evolution and progress	F16	high

Enroll in courses	F17	medium
download	F18	high
Ask for free delivery	F19	low
Find artist near you	F20	high
Design with full control	F21	medium
Choose ring metrics and diameter	F22	low
View leaked passwords	F23	low
Detect wifi security problem	F24	low
Identify image	F25	medium
Send message	F26	low
Create track	F27	low
Upload image	F28	high
Open website	F29	low
Hide image	F30	medium

category	**App name**	**Functional feature**
business	Zoom	F1, F2, F3, F5
business	Adobe acrobat reader	F4, F5, F6
business	Linked In	F2, F4, F5, F6, F8
business	Skype	F1, F2, F3, F5, F6, F7, F8
business	Call recorder for me	F2, F5, F9
business	Slack	F2, F4, F5, F6, F8, F13
business	PayPal	F2, F5, F7, F10, F11
business	WhatsApp for business	F1, F2, F3, F5, F6, F7, F8
education	EWA learn English	F2, F4, F5, F12, F13, F14
education	Google classroom	F5, F14
education	Lingokids	F2, F5, F15, F16
education	Yousician	F2, F4, F5, F13

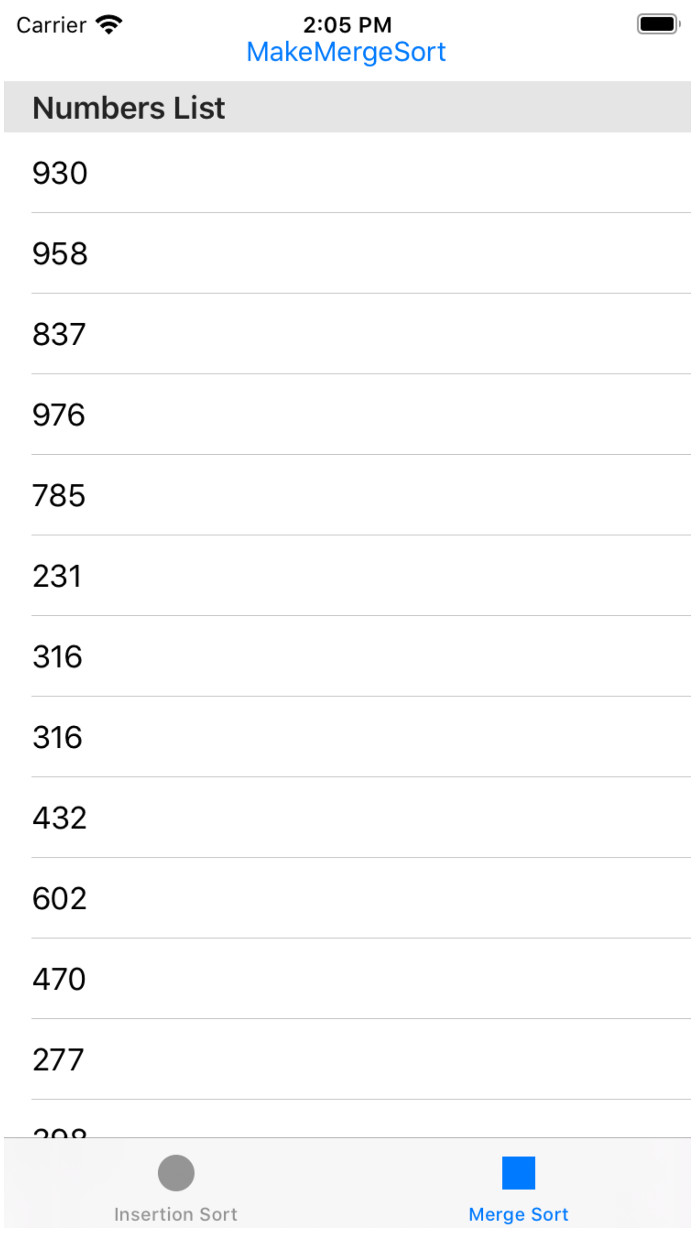
education	PictureThis	F2, F5, F7, F13, F25
education	Peak- Brain training	F2, F5, F15, F16
education	edX: courses by Harvard and MIT	F5, F14, F17, F26
education	Mondly: Learn 33 languages	F2, F5, F13
education	LinkedIn learning	F2, F5, F6, F18, F26
lifestyle	Pinterest: LifeStyle ideas	F2, F5, F13, F18
lifestyle	Tinder – Match Chat Date	F2, F5, F8, F13
lifestyle	Live wallpapers	F2, F13, F18
lifestyle	Castro	F2, F5, F13, F19
lifestyle	Perfect365	F2, F5, F13, F20
lifestyle	Décor Matters	F2, F5, F13, F21
lifestyle	Piksu	F2, F5, F8, F27, F28
lifestyle	Yoosee	F2, F5, F13
utilities	Google Chrome	F2, F5, F29
utilities	Fonts Gallery	F13
utilities	FoxFM	F4, F18
utilities	Ring Sizer	F2, F13, F22
utilities	AVG Mobile Security	F2, F5, F23, F24, F30
utilities	Collage Maker	F2, F5, F8, F13
utilities	Pokémon Home	F2, F5, F13

Feature ID	**Frequency**
F1	3
F2	27
F3	3
F4	6
F5	28
F6	6
F7	4
F8	7
F9	1
F10	1
F11	1
F12	1
F13	16
F14	3
F15	2
F16	2
F17	1
F18	4
F19	1
F20	1
F21	1
F22	1
F23	1
F24	1
F25	1
F26	2
F27	1
F28	1
F29	1
F30	1

Appendix B: Applications Screenshots.

B.1 Screenshots from Internal Sorting iOS+RN (Application 1and 2).

1. iOS.



Numbers List

930

958

837

976

785

231

316

316

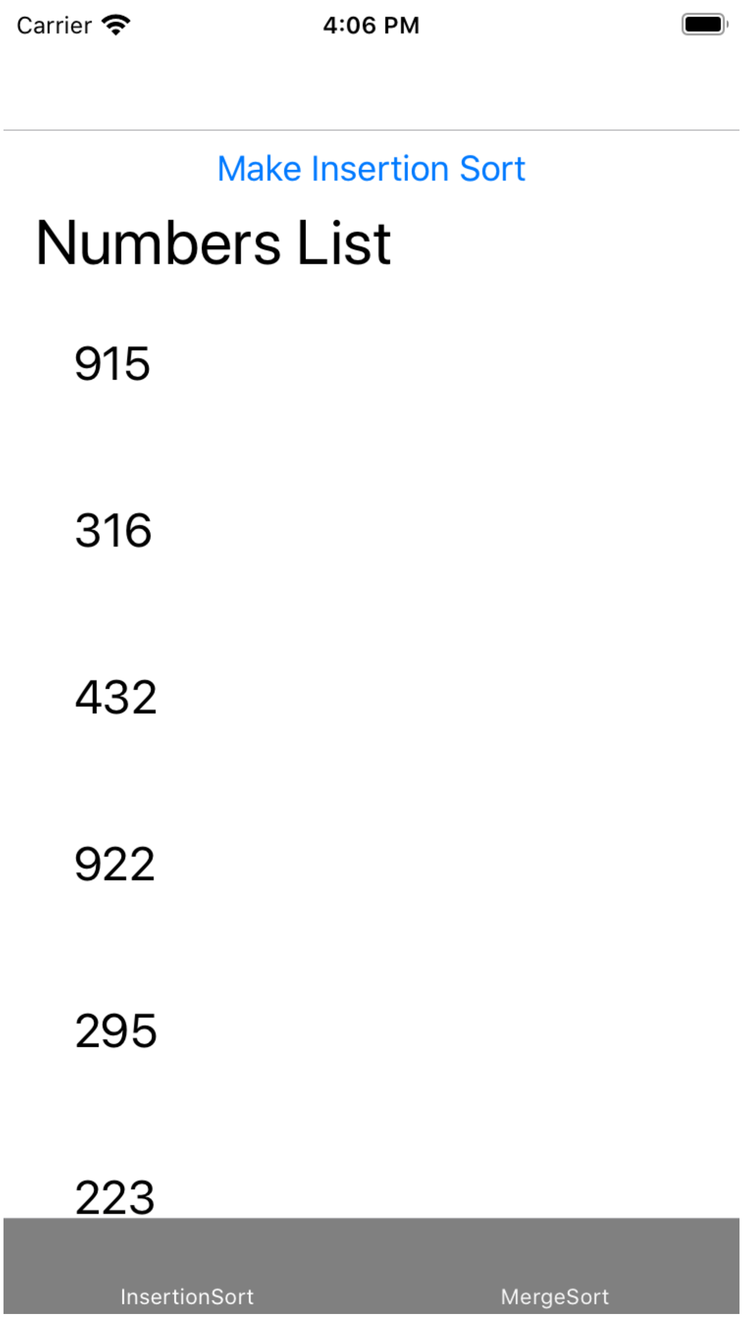
432

602

470

277

2. React Native.



Numbers List

915

316

432

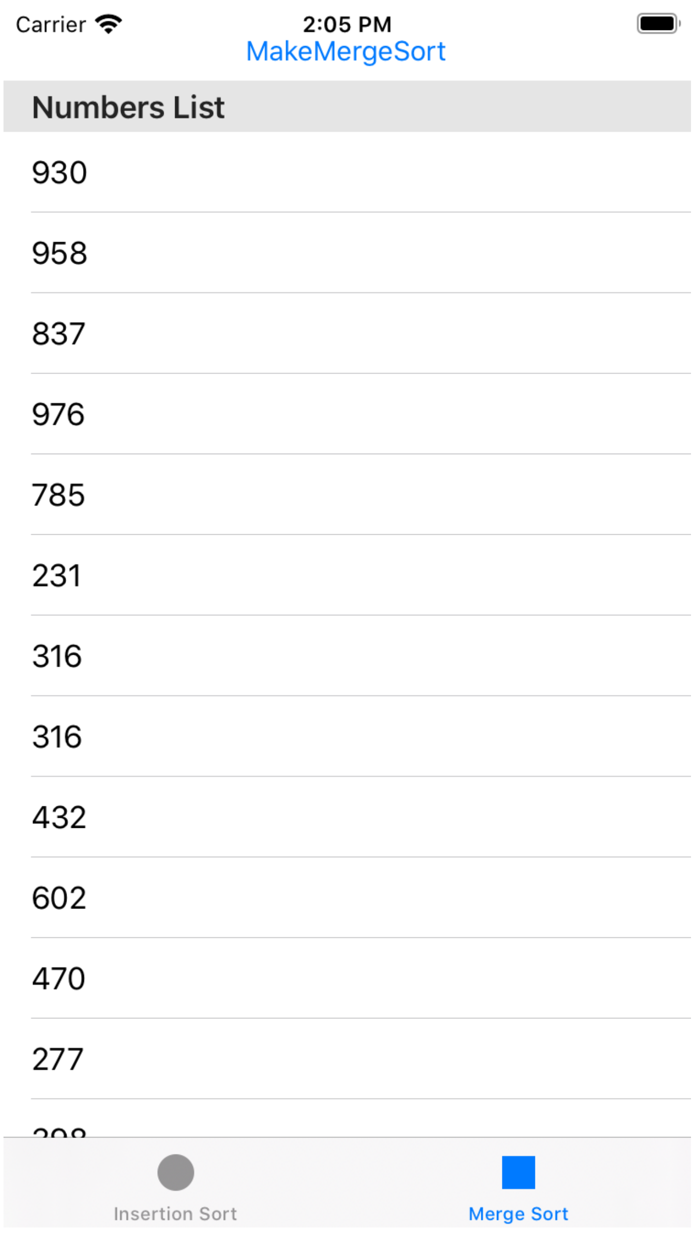
922

295

223

B.2 Screenshots from File Sorting iOS+RN (Application 3 and 4).

1. iOS.



Numbers List

930

958

837

976

785

231

316

316

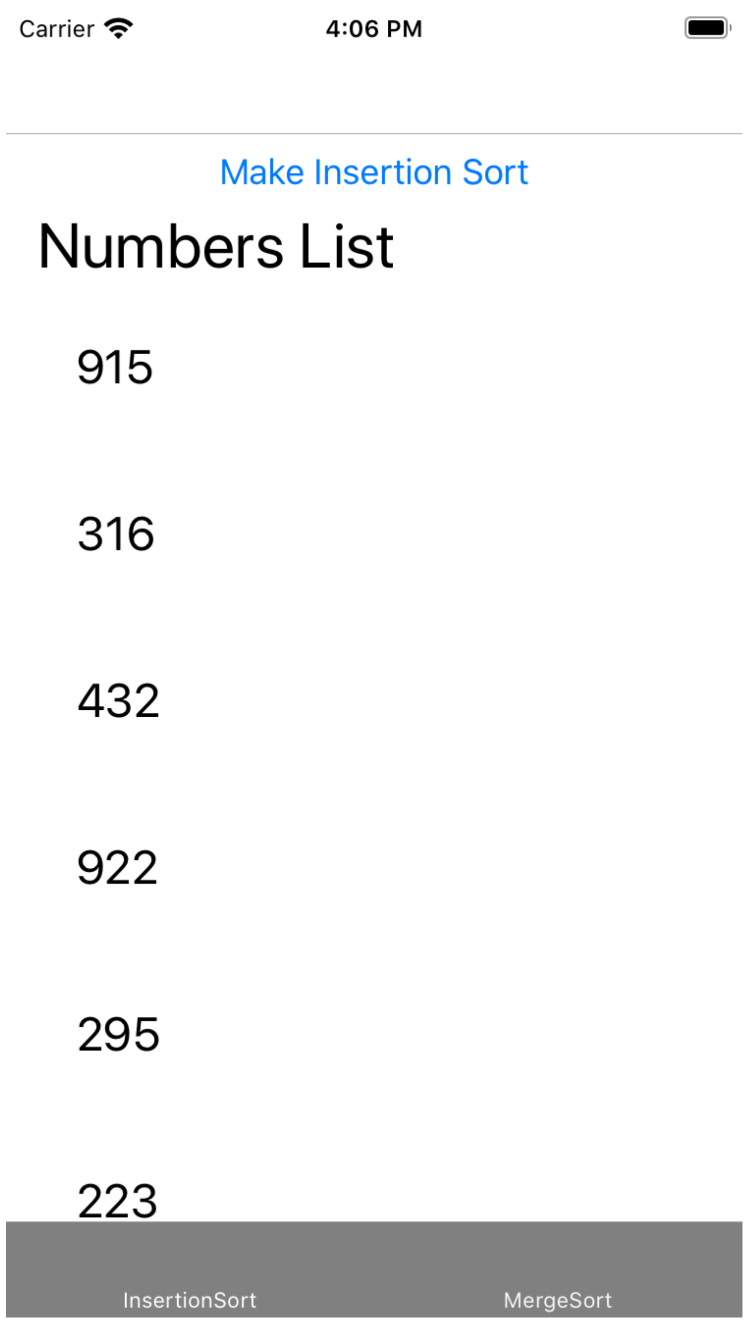
432

602

470

277

2. React Native.


Carrier 4:06 PM

Make Insertion Sort

Numbers List

915

316

432

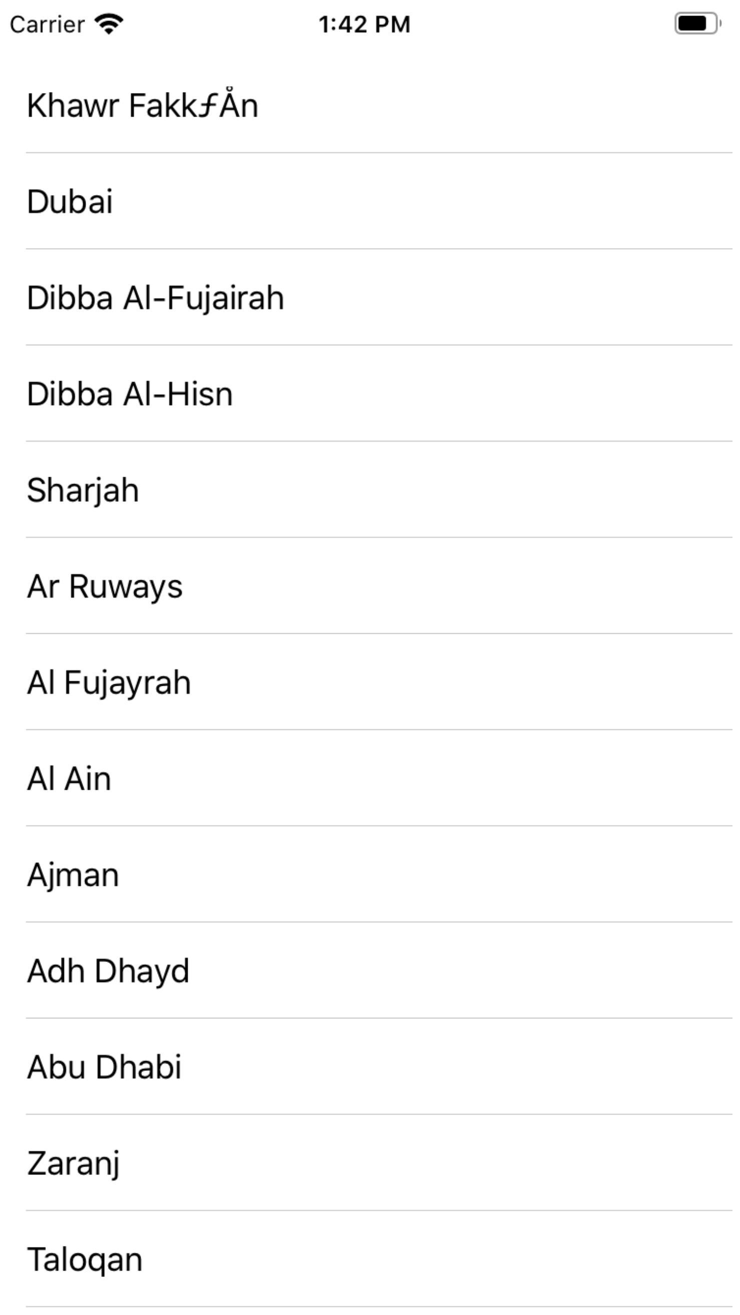
922

295

223

B.3 Screenshots from Database Text iOS+RN (Application 5 and 6).

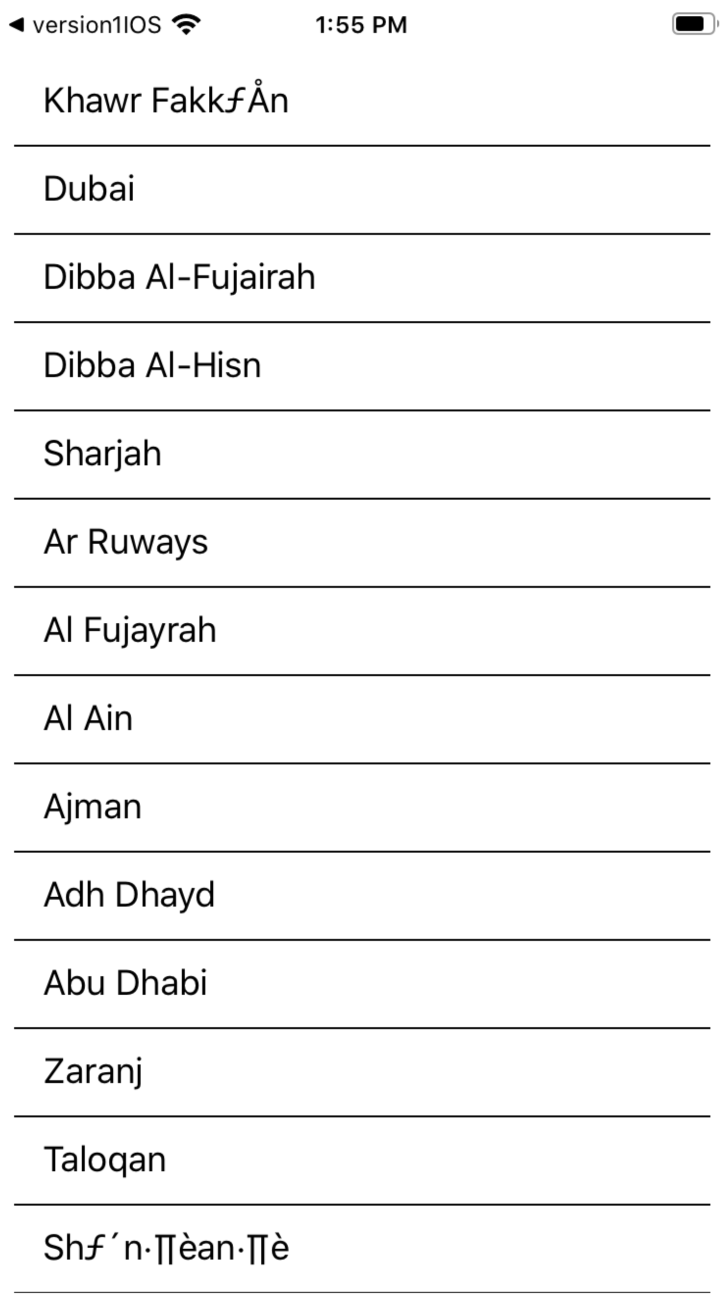
1. iOS.



Carrier 1:42 PM

Khawr Fakkƒ̊Ån
Dubai
Dibba Al-Fujairah
Dibba Al-Hisn
Sharjah
Ar Ruways
Al Fujayrah
Al Ain
Ajman
Adh Dhayd
Abu Dhabi
Zaranj
Taloqan

2. React Native.



◀ version1IOS 1:55 PM

Khawr FakkƒÅn

Dubai

Dibba Al-Fujairah

Dibba Al-Hisn

Sharjah

Ar Ruways

Al Fujayrah

Al Ain

Ajman

Adh Dhayd

Abu Dhabi

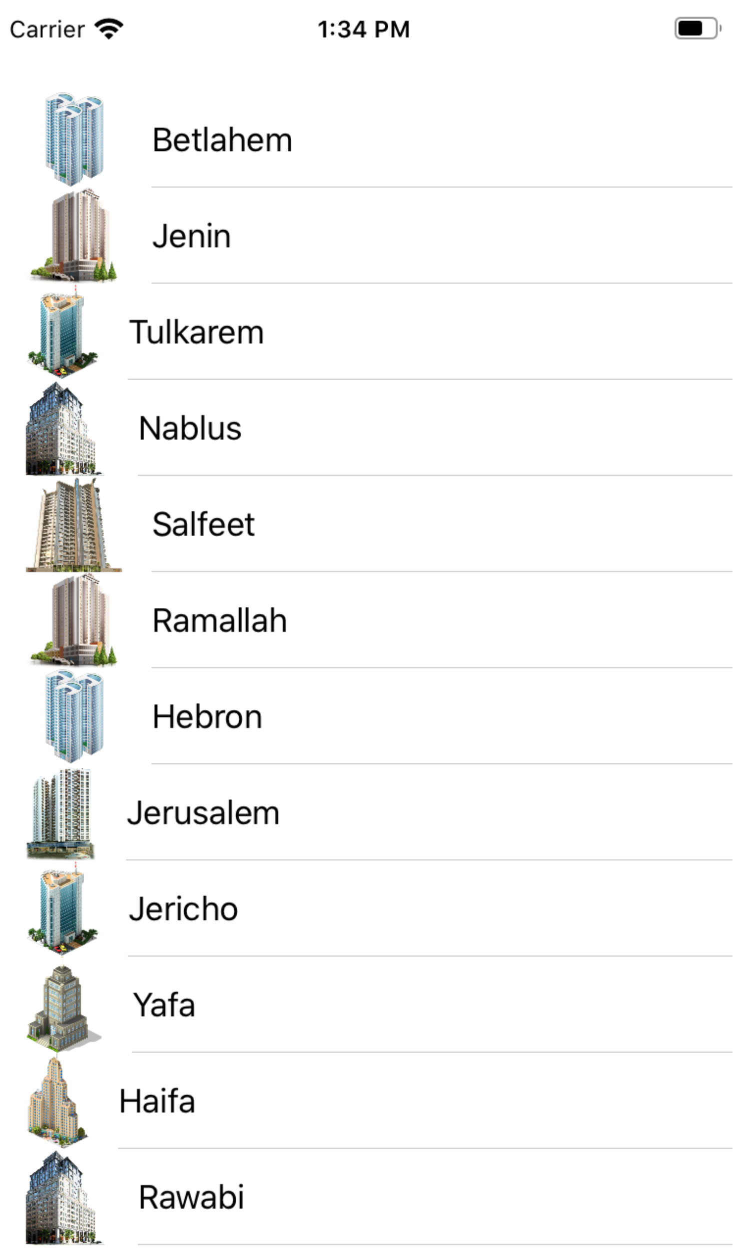
Zaranj

Taloqan

Shƒ´n·∏èan·∏è

B.4 Screenshots from Database Image iOS+RN (Application 7 and 8).

1. iOS.



Betlahem

Jenin

Tulkarem

Nablus

Salfeet

Ramallah

Hebron

Jerusalem

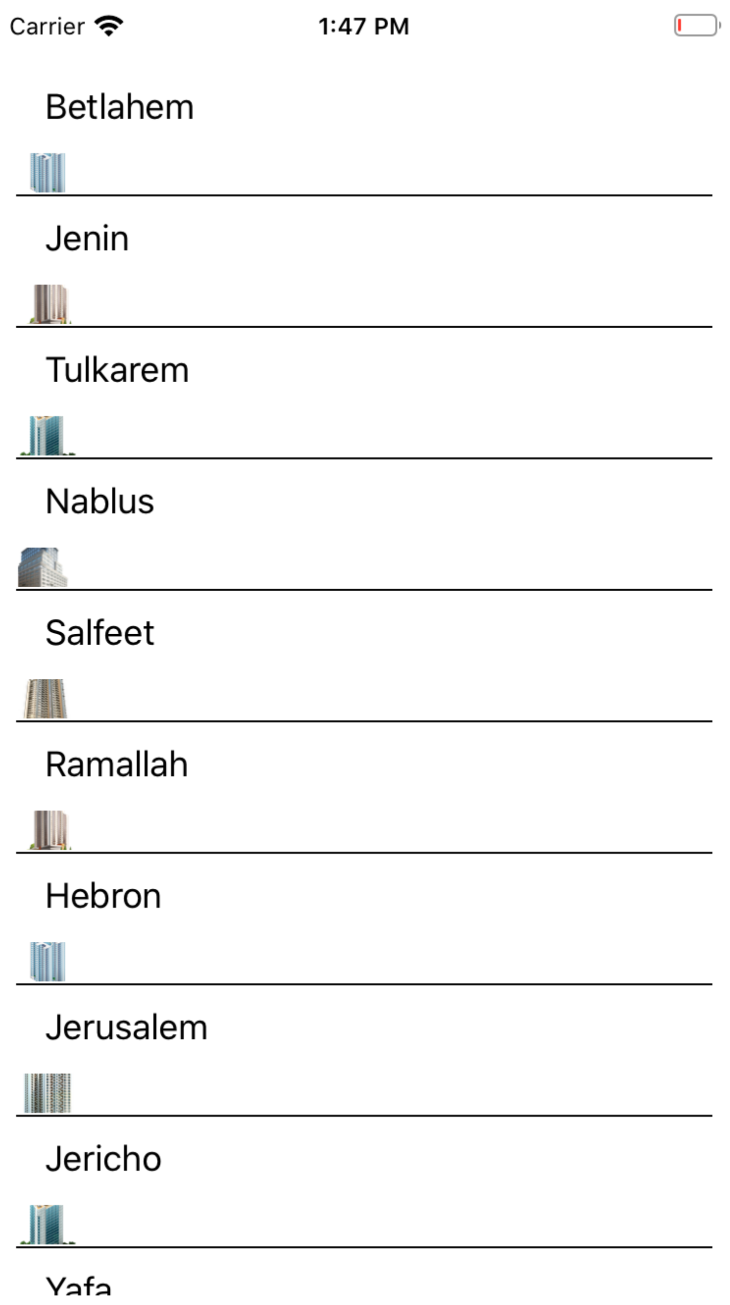
Jericho

Yafa

Haifa

Rawabi

2. React Native.


Carrier 1:47 PM

Betlahem

Jenin

Tulkarem

Nablus

Salfeet

Ramallah

Hebron

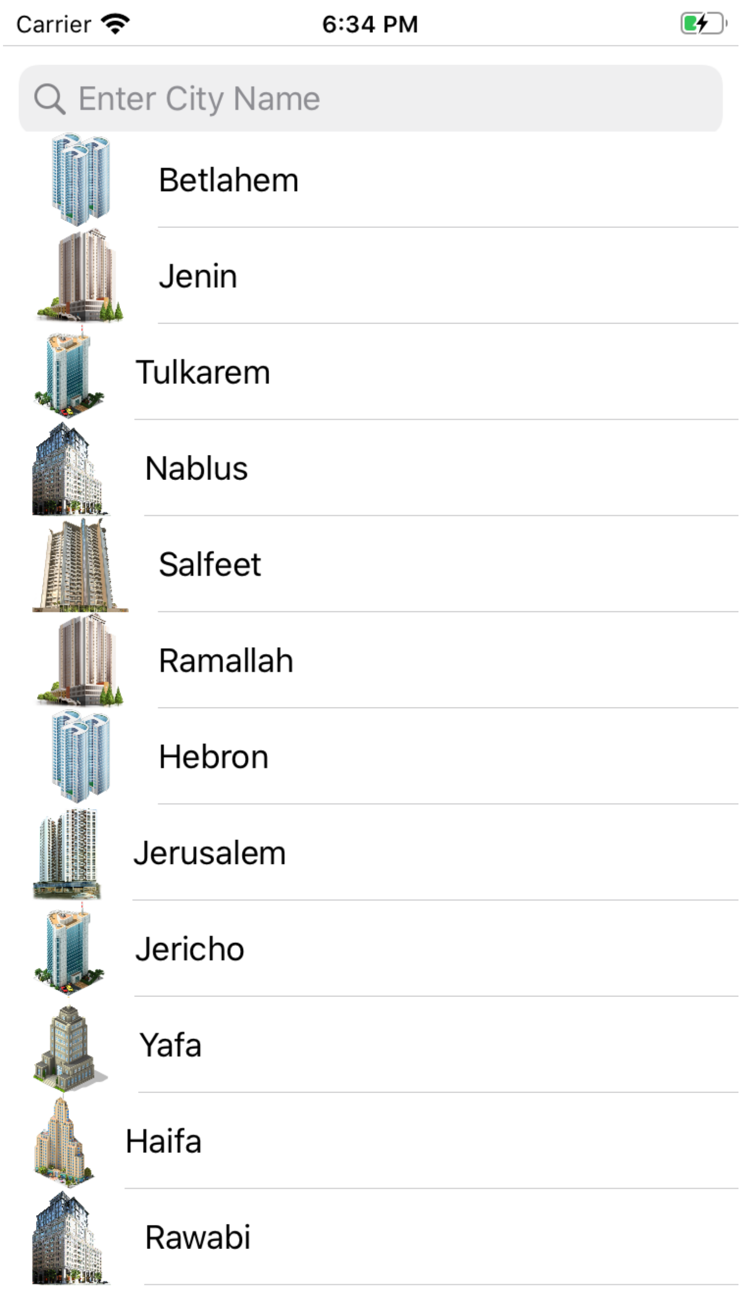
Jerusalem

Jericho

Yafa

B.5 Screenshots from Search iOS+RN (Application 9 and 10).

1. iOS.



Carrier 6:34 PM

Enter City Name

Betlahem

Jenin

Tulkarem

Nablus

Salfeet

Ramallah

Hebron

Jerusalem

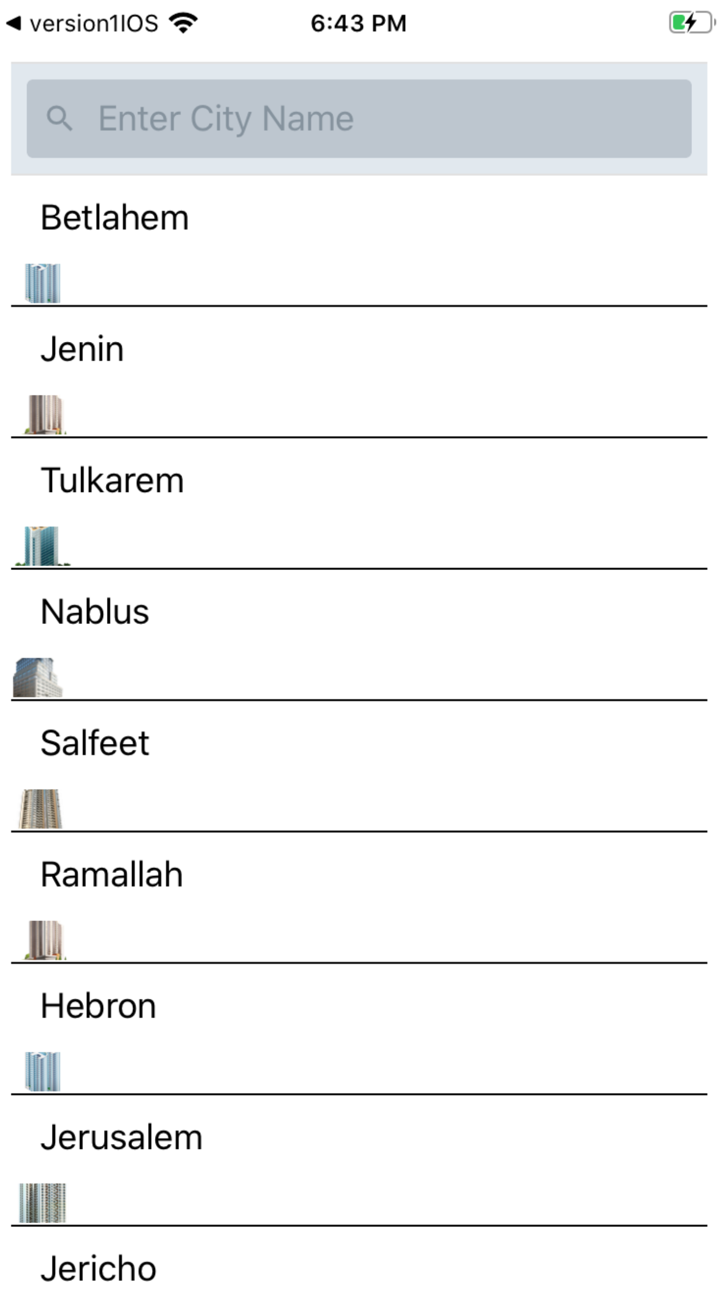
Jericho

Yafa

Haifa

Rawabi

2. React Native.


◀ version1IOS

6:43 PM

Enter City Name

Betlahem

Jenin

Tulkarem

Nablus

Salfeet

Ramallah

Hebron

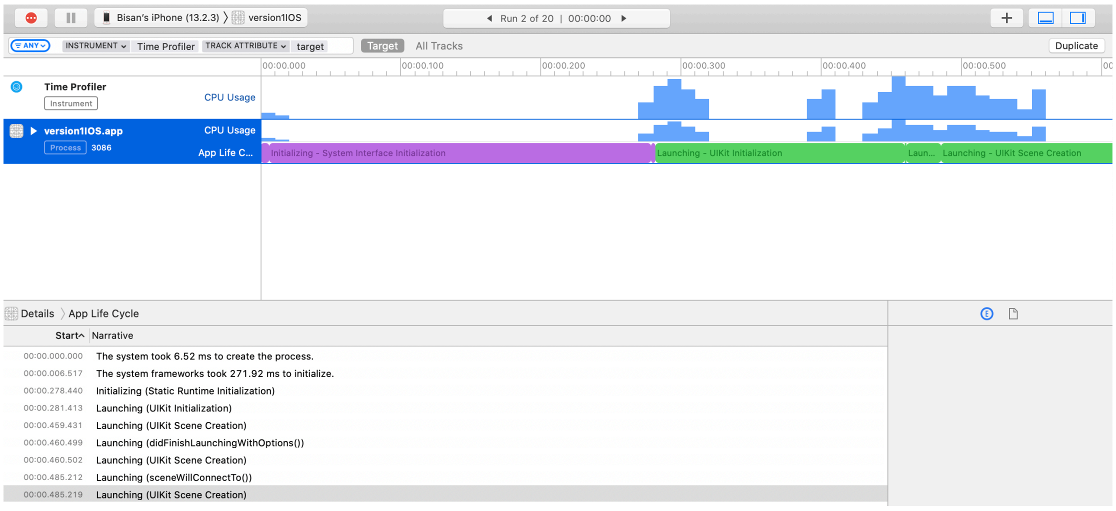
Jerusalem

Jericho

Appendix C: Test Experiments.

C.1 Screenshots from Number of Runs test.

1. iOS (Application Launch Time).



```
Start^          Narrative
00:00.000.000   The system took 6.52 ms to create the process.
00:00.006.517   The system frameworks took 271.92 ms to initialize.
00:00.278.440   Initializing (Static Runtime Initialization)
00:00.281.413   Launching (UIKit Initialization)
00:00.459.431   Launching (UIKit Scene Creation)
00:00.460.499   Launching (didFinishLaunchingWithOptions())
00:00.460.502   Launching (UIKit Scene Creation)
00:00.485.212   Launching (sceneWillConnectTo())
00:00.485.219   Launching (UIKit Scene Creation)
```

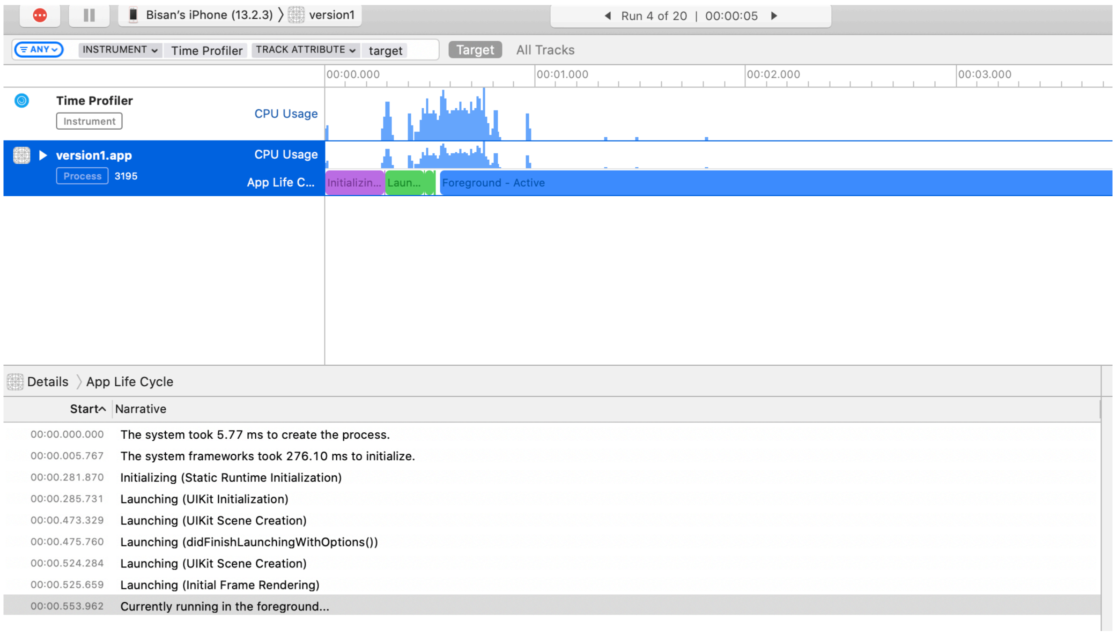
Run Number	Application Launch Time /ms
1	485
2	480
3	478
4	481
5	480

Run Number	Application Launch Time /ms
1	480
2	486
3	485
4	486
5	485
6	482
7	481
8	486
9	482
10	490

11	478
12	481
13	488
14	495
15	487

Run Number	Application Launch Time /ms
1	488
2	485
3	478
4	480
5	482
6	481
7	475
8	477
9	476
10	479
11	480
12	480
13	482
14	488
15	485
16	488
17	482
18	483
19	481
20	480

2. React Native (Application Launch Time).



Start	Narrative
00:00.000.000	The system took 5.77 ms to create the process.
00:00.005.767	The system frameworks took 276.10 ms to initialize.
00:00.281.870	Initializing (Static Runtime Initialization)
00:00.285.731	Launching (UIKit Initialization)
00:00.473.329	Launching (UIKit Scene Creation)
00:00.475.760	Launching (didFinishLaunchingWithOptions())
00:00.524.284	Launching (UIKit Scene Creation)
00:00.525.659	Launching (Initial Frame Rendering)
00:00.553.962	Currently running in the foreground...

Run Number	Application Launch Time /ms
1	550
2	565
3	553
4	553
5	557

Run Number	Application Launch Time /ms
1	480
2	565
3	553
4	553
5	550
6	549
7	542
8	559
9	555
10	554
11	550

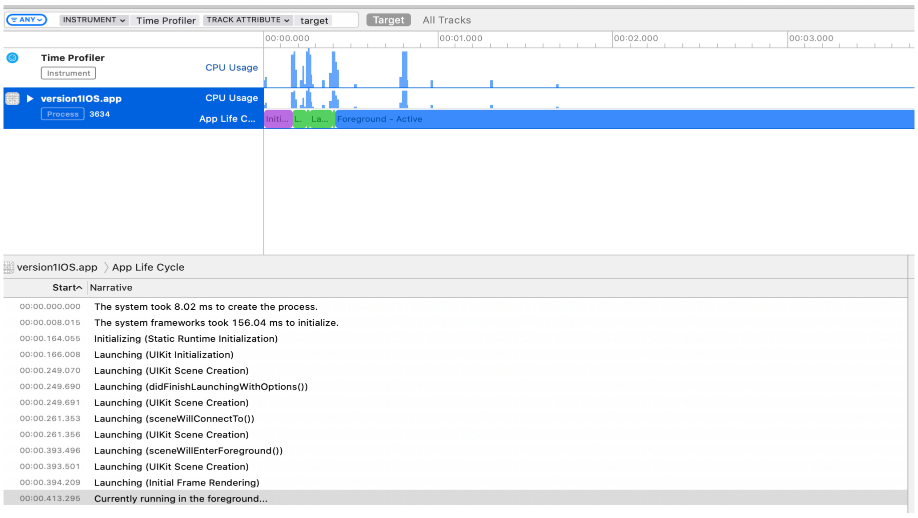
12	559
13	541
14	561
15	543

Run Number	Application Launch Time /ms
1	550
2	549
3	553
4	553
5	543
6	557
7	540
8	555
9	543
10	550
11	551
12	550
13	549
14	551
15	550
16	553
17	551
18	552
19	549
20	555

C.2 Screenshots from Battery Level test.

1. iOS (Application Launch Time).

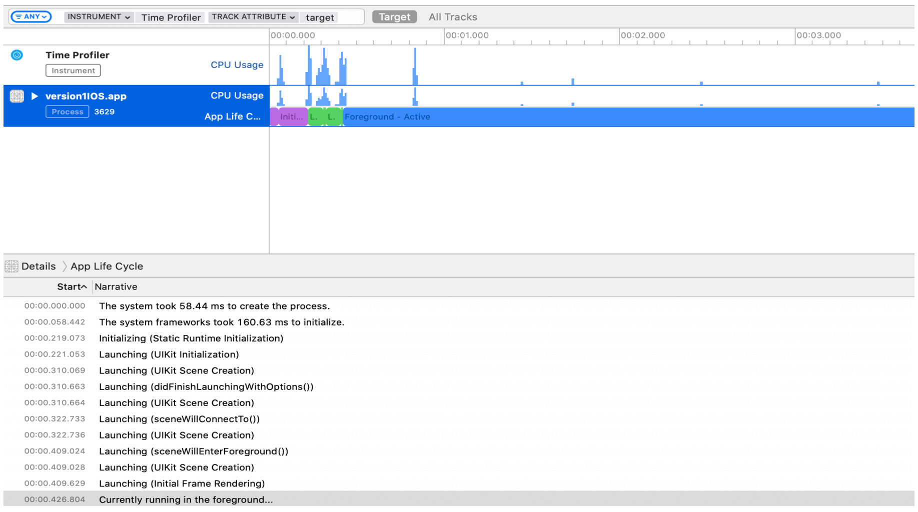
- Low Battery Level.



version1IOS.app 〉 App Life Cycle

```
       Start^  Narrative
00:00.000.000  The system took 8.02 ms to create the process.
00:00.008.015  The system frameworks took 156.04 ms to initialize.
00:00.164.055  Initializing (Static Runtime Initialization)
00:00.166.008  Launching (UIKit Initialization)
00:00.249.070  Launching (UIKit Scene Creation)
00:00.249.690  Launching (didFinishLaunchingWithOptions())
00:00.249.691  Launching (UIKit Scene Creation)
00:00.261.353  Launching (sceneWillConnectTo())
00:00.261.356  Launching (UIKit Scene Creation)
00:00.393.496  Launching (sceneWillEnterForeground())
00:00.393.501  Launching (UIKit Scene Creation)
00:00.394.209  Launching (Initial Frame Rendering)
00:00.413.295  Currently running in the foreground...
```

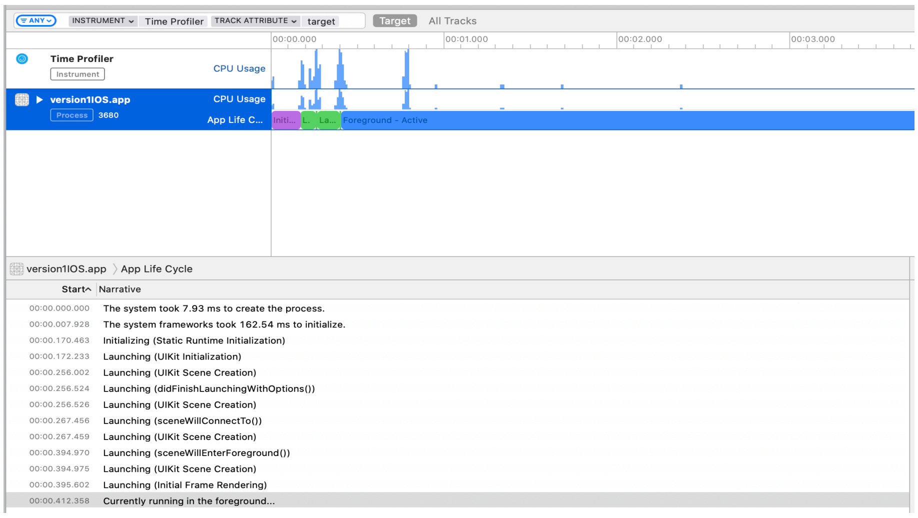
- Medium Battery Level.



Details 〉 App Life Cycle

```
       Start^  Narrative
00:00.000.000  The system took 58.44 ms to create the process.
00:00.058.442  The system frameworks took 160.63 ms to initialize.
00:00.219.073  Initializing (Static Runtime Initialization)
00:00.221.053  Launching (UIKit Initialization)
00:00.310.069  Launching (UIKit Scene Creation)
00:00.310.663  Launching (didFinishLaunchingWithOptions())
00:00.310.664  Launching (UIKit Scene Creation)
00:00.322.733  Launching (sceneWillConnectTo())
00:00.322.736  Launching (UIKit Scene Creation)
00:00.409.024  Launching (sceneWillEnterForeground())
00:00.409.028  Launching (UIKit Scene Creation)
00:00.409.629  Launching (Initial Frame Rendering)
00:00.426.804  Currently running in the foreground...
```

- High Battery Level.

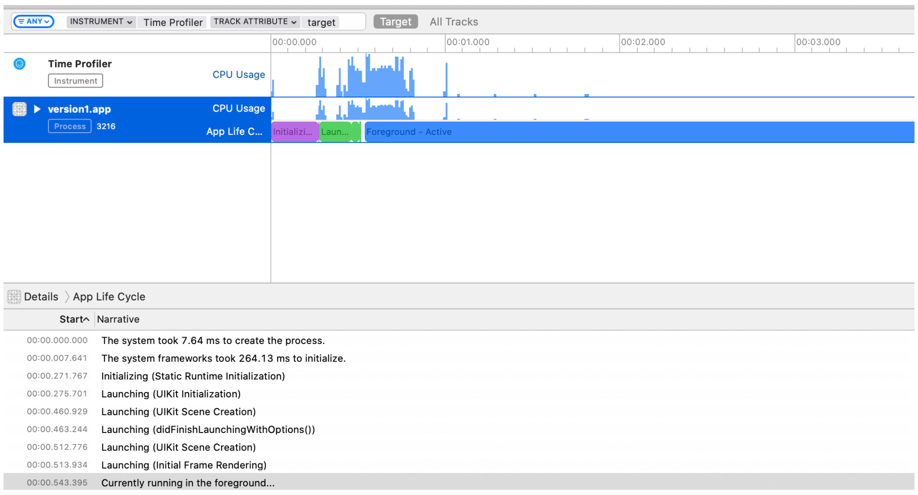


version1IOS.app 〉 App Life Cycle

```
       Start^  Narrative
00:00.000.000  The system took 7.93 ms to create the process.
00:00.007.928  The system frameworks took 162.54 ms to initialize.
00:00.170.463  Initializing (Static Runtime Initialization)
00:00.172.233  Launching (UIKit Initialization)
00:00.256.002  Launching (UIKit Scene Creation)
00:00.256.524  Launching (didFinishLaunchingWithOptions())
00:00.256.526  Launching (UIKit Scene Creation)
00:00.267.456  Launching (sceneWillConnectTo())
00:00.267.459  Launching (UIKit Scene Creation)
00:00.394.970  Launching (sceneWillEnterForeground())
00:00.394.975  Launching (UIKit Scene Creation)
00:00.395.602  Launching (Initial Frame Rendering)
00:00.412.358  Currently running in the foreground...
```

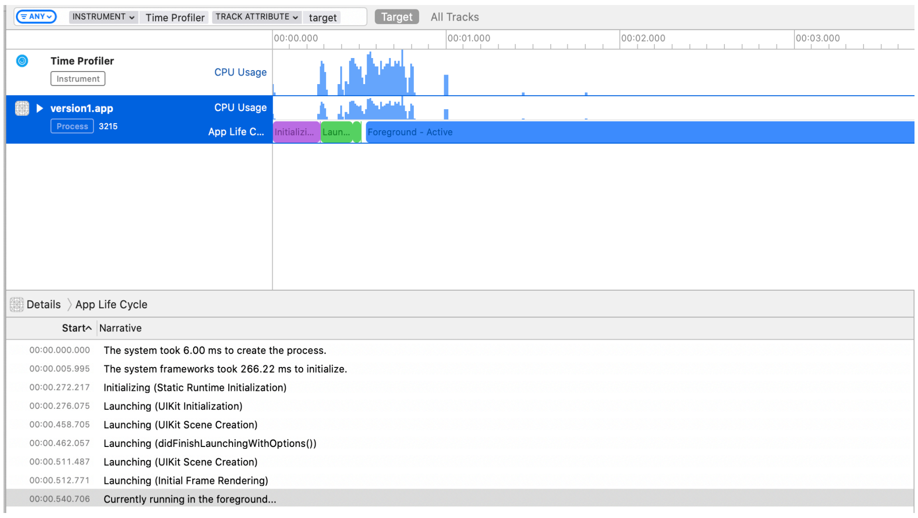
2. React Native (Application Launch Time).

- Low Battery Level.



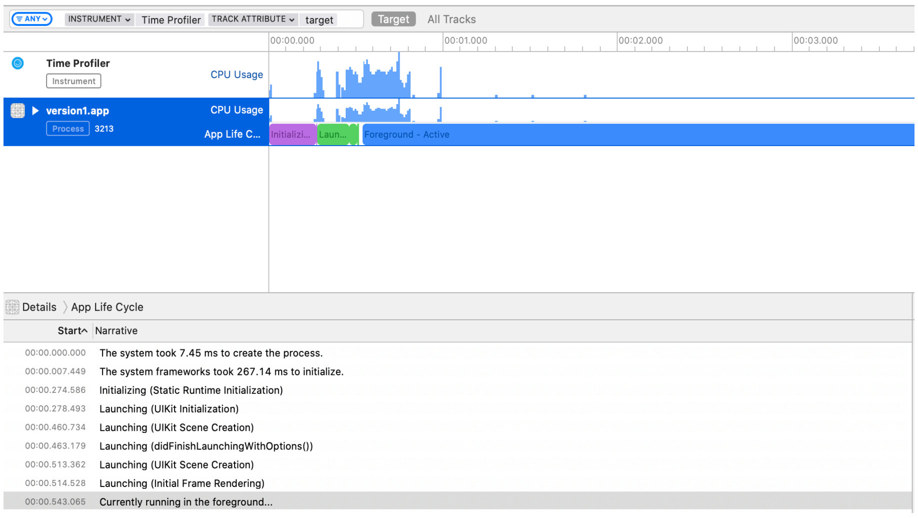
Start^	Narrative
00:00.000.000	The system took 7.64 ms to create the process.
00:00.007.641	The system frameworks took 264.13 ms to initialize.
00:00.271.767	Initializing (Static Runtime Initialization)
00:00.275.701	Launching (UIKit Initialization)
00:00.460.929	Launching (UIKit Scene Creation)
00:00.463.244	Launching (didFinishLaunchingWithOptions())
00:00.512.776	Launching (UIKit Scene Creation)
00:00.513.934	Launching (Initial Frame Rendering)
00:00.543.395	Currently running in the foreground...

- Medium Battery Level.



Start^	Narrative
00:00.000.000	The system took 6.00 ms to create the process.
00:00.005.995	The system frameworks took 266.22 ms to initialize.
00:00.272.217	Initializing (Static Runtime Initialization)
00:00.276.075	Launching (UIKit Initialization)
00:00.458.705	Launching (UIKit Scene Creation)
00:00.462.057	Launching (didFinishLaunchingWithOptions())
00:00.511.487	Launching (UIKit Scene Creation)
00:00.512.771	Launching (Initial Frame Rendering)
00:00.540.706	Currently running in the foreground...

- High Battery Level.

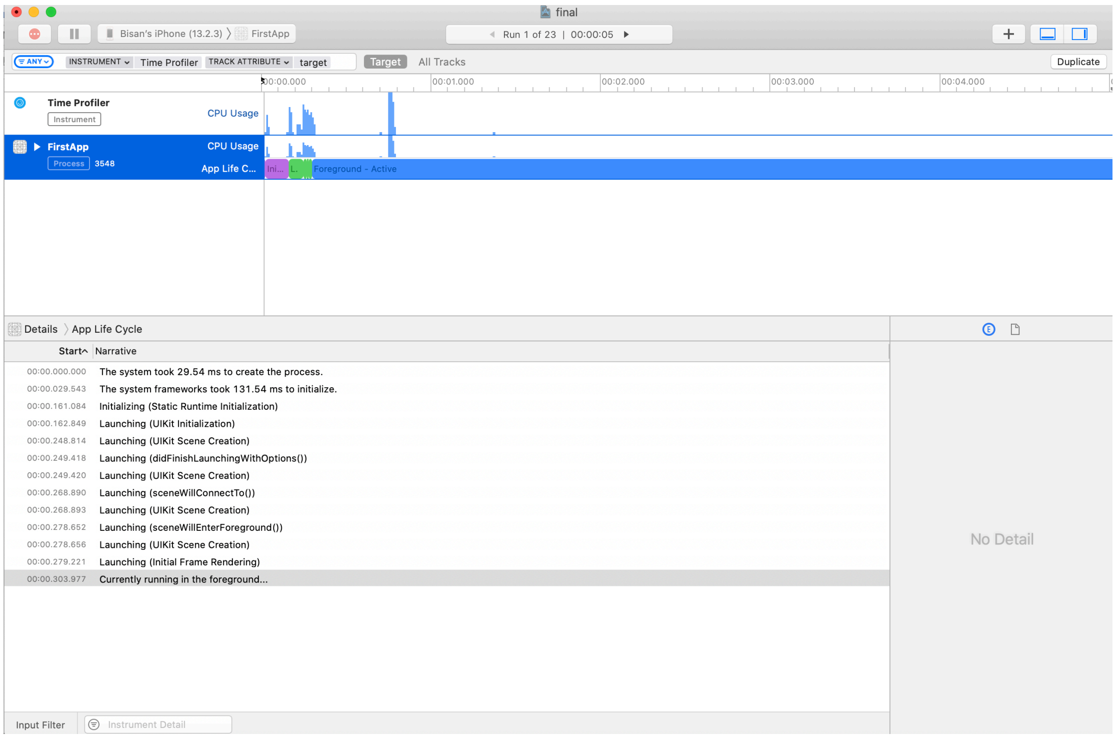


Start^	Narrative
00:00.000.000	The system took 7.45 ms to create the process.
00:00.007.449	The system frameworks took 267.14 ms to initialize.
00:00.274.586	Initializing (Static Runtime Initialization)
00:00.278.493	Launching (UIKit Initialization)
00:00.460.734	Launching (UIKit Scene Creation)
00:00.463.179	Launching (didFinishLaunchingWithOptions())
00:00.513.362	Launching (UIKit Scene Creation)
00:00.514.528	Launching (Initial Frame Rendering)
00:00.543.065	Currently running in the foreground...

Appendix D: Instruments Tool Screenshots.

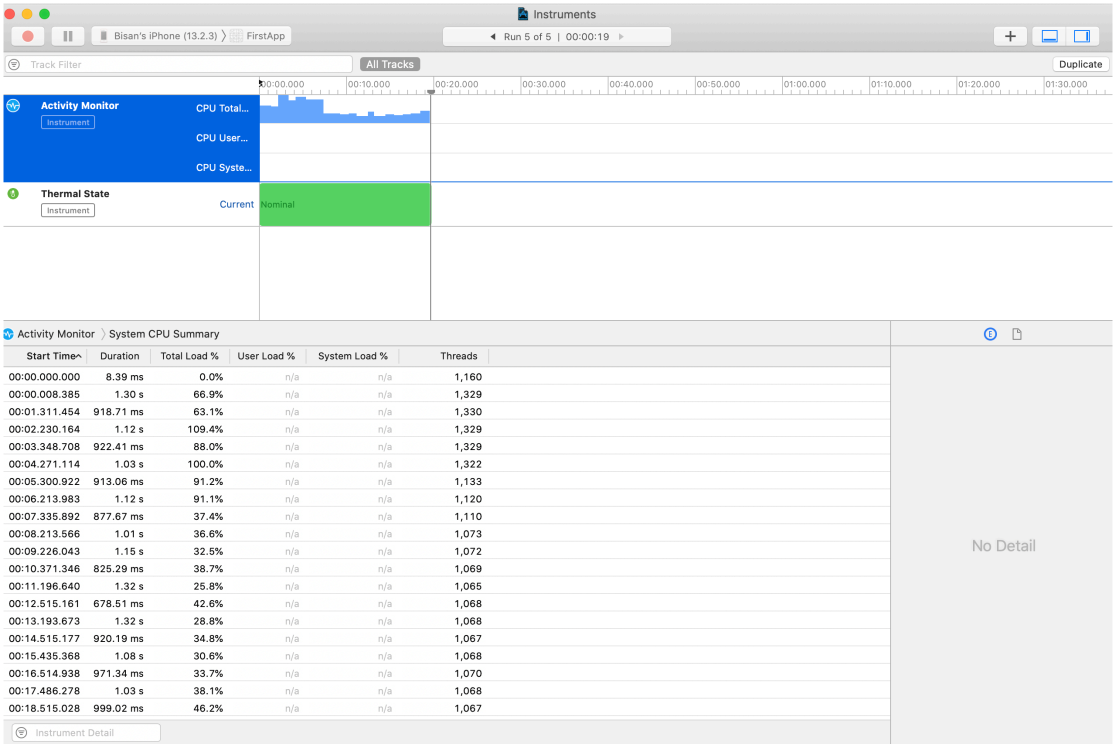
D.1 Some Screenshots from Internal Sorting iOS + RN (App 1 and 2).

1- Cold Application Launch.



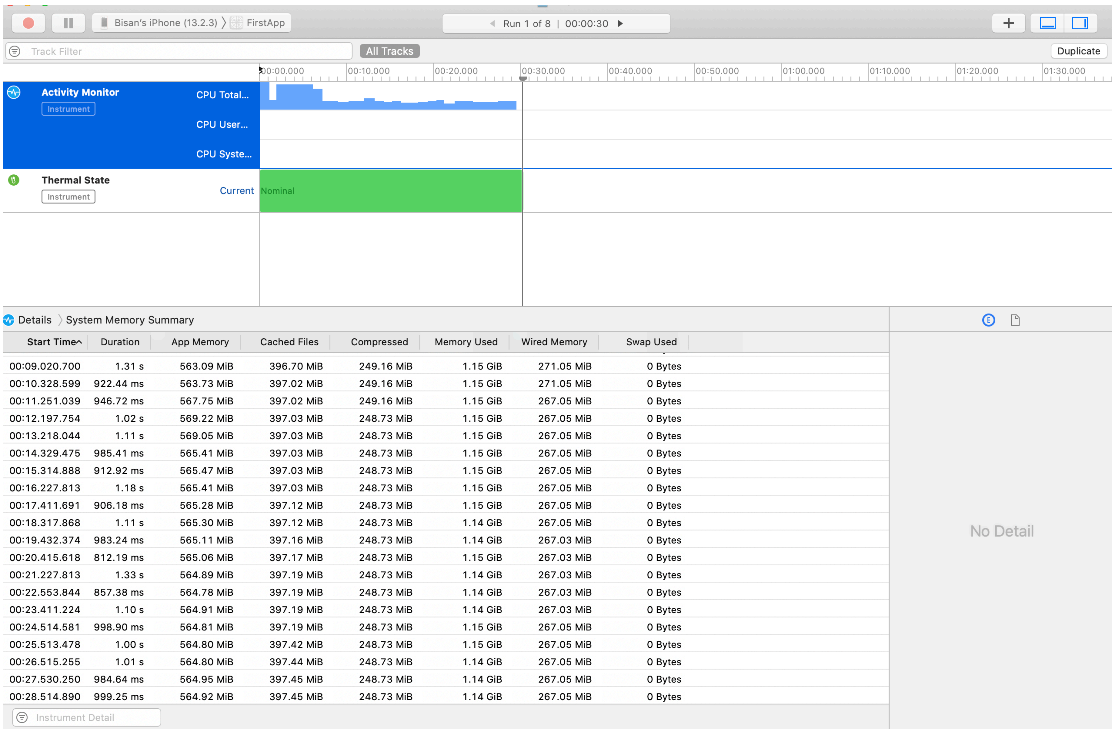
Start	Narrative
00:00.000.000	The system took 29.54 ms to create the process.
00:00.029.543	The system frameworks took 131.54 ms to initialize.
00:00.161.084	Initializing (Static Runtime Initialization)
00:00.162.849	Launching (UIKit Initialization)
00:00.248.814	Launching (UIKit Scene Creation)
00:00.249.418	Launching (didFinishLaunchingWithOptions())
00:00.249.420	Launching (UIKit Scene Creation)
00:00.268.890	Launching (sceneWillConnectTo())
00:00.268.893	Launching (UIKit Scene Creation)
00:00.278.652	Launching (sceneWillEnterForeground())
00:00.278.656	Launching (UIKit Scene Creation)
00:00.279.221	Launching (Initial Frame Rendering)
00:00.303.977	Currently running in the foreground...

2- CPU Usage.



Start Time	Duration	Total Load %	User Load %	System Load %	Threads
00:00.000.000	8.39 ms	0.0%	n/a	n/a	1,160
00:00.008.385	1.30 s	66.9%	n/a	n/a	1,329
00:01.311.454	918.71 ms	63.1%	n/a	n/a	1,330
00:02.230.164	1.12 s	109.4%	n/a	n/a	1,329
00:03.348.708	922.41 ms	88.0%	n/a	n/a	1,329
00:04.271.114	1.03 s	100.0%	n/a	n/a	1,322
00:05.300.922	913.06 ms	91.2%	n/a	n/a	1,133
00:06.213.983	1.12 s	91.1%	n/a	n/a	1,120
00:07.335.892	877.67 ms	37.4%	n/a	n/a	1,110
00:08.213.566	1.01 s	36.6%	n/a	n/a	1,073
00:09.226.043	1.15 s	32.5%	n/a	n/a	1,072
00:10.371.346	825.29 ms	38.7%	n/a	n/a	1,069
00:11.196.640	1.32 s	25.8%	n/a	n/a	1,065
00:12.515.161	678.51 ms	42.6%	n/a	n/a	1,068
00:13.193.673	1.32 s	28.8%	n/a	n/a	1,068
00:14.515.177	920.19 ms	34.8%	n/a	n/a	1,067
00:15.435.368	1.08 s	30.6%	n/a	n/a	1,068
00:16.514.938	971.34 ms	33.7%	n/a	n/a	1,070
00:17.486.278	1.03 s	38.1%	n/a	n/a	1,068
00:18.515.028	999.02 ms	46.2%	n/a	n/a	1,067

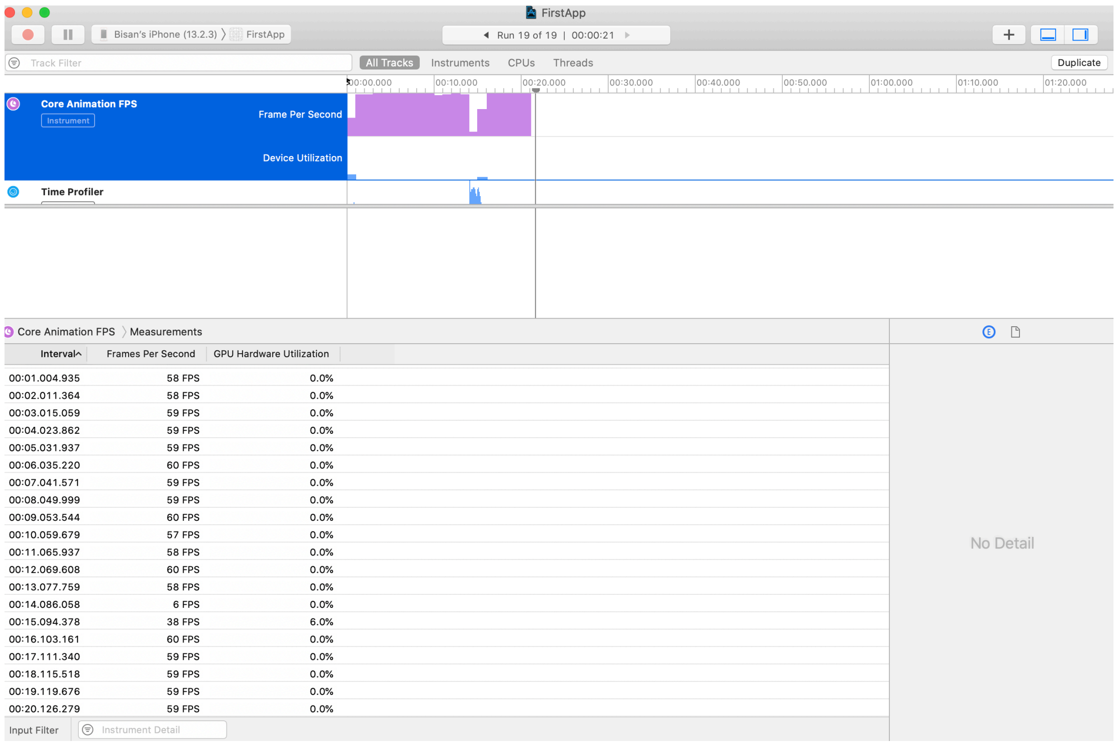
3- Memory Usage.



Details 〉 System Memory Summary

Start Time	Duration	App Memory	Cached Files	Compressed	Memory Used	Wired Memory	Swap Used
00:09.020.700	1.31 s	563.09 MiB	396.70 MiB	249.16 MiB	1.15 GiB	271.05 MiB	0 Bytes
00:10.328.599	922.44 ms	563.73 MiB	397.02 MiB	249.16 MiB	1.15 GiB	271.05 MiB	0 Bytes
00:11.251.039	946.72 ms	567.75 MiB	397.02 MiB	249.16 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:12.197.754	1.02 s	569.22 MiB	397.03 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:13.218.044	1.11 s	569.05 MiB	397.03 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:14.329.475	985.41 ms	565.41 MiB	397.03 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:15.314.888	912.92 ms	565.47 MiB	397.03 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:16.227.813	1.18 s	565.41 MiB	397.03 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:17.411.691	906.18 ms	565.28 MiB	397.12 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:18.317.868	1.11 s	565.30 MiB	397.12 MiB	248.73 MiB	1.14 GiB	267.05 MiB	0 Bytes
00:19.432.374	983.24 ms	565.11 MiB	397.16 MiB	248.73 MiB	1.14 GiB	267.03 MiB	0 Bytes
00:20.415.618	812.19 ms	565.06 MiB	397.17 MiB	248.73 MiB	1.15 GiB	267.03 MiB	0 Bytes
00:21.227.813	1.33 s	564.89 MiB	397.19 MiB	248.73 MiB	1.14 GiB	267.03 MiB	0 Bytes
00:22.553.844	857.38 ms	564.78 MiB	397.19 MiB	248.73 MiB	1.14 GiB	267.03 MiB	0 Bytes
00:23.411.224	1.10 s	564.91 MiB	397.19 MiB	248.73 MiB	1.14 GiB	267.03 MiB	0 Bytes
00:24.514.581	998.90 ms	564.81 MiB	397.19 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:25.513.478	1.00 s	564.80 MiB	397.42 MiB	248.73 MiB	1.15 GiB	267.05 MiB	0 Bytes
00:26.515.255	1.01 s	564.80 MiB	397.44 MiB	248.73 MiB	1.14 GiB	267.05 MiB	0 Bytes
00:27.530.250	984.64 ms	564.95 MiB	397.45 MiB	248.73 MiB	1.14 GiB	267.05 MiB	0 Bytes
00:28.514.890	999.25 ms	564.92 MiB	397.45 MiB	248.73 MiB	1.14 GiB	267.05 MiB	0 Bytes

4- Frames Per Second.



Core Animation FPS 〉 Measurements

Interval	Frames Per Second	GPU Hardware Utilization
00:01.004.935	58 FPS	0.0%
00:02.011.364	58 FPS	0.0%
00:03.015.059	59 FPS	0.0%
00:04.023.862	59 FPS	0.0%
00:05.031.937	59 FPS	0.0%
00:06.035.220	60 FPS	0.0%
00:07.041.571	59 FPS	0.0%
00:08.049.999	59 FPS	0.0%
00:09.053.544	60 FPS	0.0%
00:10.059.679	57 FPS	0.0%
00:11.065.937	58 FPS	0.0%
00:12.069.608	60 FPS	0.0%
00:13.077.759	58 FPS	0.0%
00:14.086.058	6 FPS	0.0%
00:15.094.378	38 FPS	6.0%
00:16.103.161	60 FPS	0.0%
00:17.111.340	59 FPS	0.0%
00:18.115.518	59 FPS	0.0%
00:19.119.676	59 FPS	0.0%
00:20.126.279	59 FPS	0.0%