Handelshögskolan
Karlstad Business School

Pelle Grönlund

A comparison study between mobile cross platform frameworks Flutter and React Native

Information Systems

Bachelor's thesis

Term: Spring 2023

Supervisor: Ala Sarah Alaqra

Karlstad Business School
Karlstad University SE-651 88 Karlstad
Phone: +46 54 700 10 00
E-mail: handels@kau.se kau.se/en/hhk

Abstract

When developers today choose a cross platform framework as the base for their coding, they have a lot of different cross platform frameworks to choose from. To aid in this decision researchers have compared and tested the different frameworks against each other to look at different aspects ranging from how much computing power the finished applications created by the framework require from the devices they are running on to how easily accessible the frameworks are to install and use.

One of the new frameworks on the market is Flutter and has only been included in some previous studies. The goal of the thesis is to add knowledge to existing research and to see if the performance has changed in the last few years, by comparing Flutter to React Native that has been used in a lot of previous studies.

The objective of the study is to look at the differences in performance usage by the two different frameworks (Flutter and React Native) and to give a recommendation for future development based on the results of the performance usage.

To answer these questions two identical mobile applications was created, one using Flutter and one using React Native and then these two applications will run through a test where the CPU, Memory and disk space usage will be recorded and compared against each other.

Notable findings in the study include that the pilot test confirmed that CPU consumption always spiked after an on-screen interaction with the applications happened, something only two other studies mentioned. A suggested way to solve this problem will be added by looking at peak CPU consumption after interactions. The test results themselves found that Flutter generally consumed less memory overall than React Native. Flutter also had a lower CPU consumption overall however the spikes of CPU consumption after an interaction were typically higher for the Flutter applications. Lastly Flutter consumed slightly more disc space on the target device.

Innehåll

Table of Figures

Table of Tables

Introduction

Problem area and background

When deciding to choose a program or framework for any work in today's IT world we, one way or another compare the different options before us on different criterions to evaluate what is the best option to use for our project. There is a lot of different aspects to look at, some may value the look and feel of a program more than they value the ease of use or the amount of code required to build a working product in the different programs/frameworks. One criterion often looked at when deciding which framework to use when designing cross platform applications today is how much computing power the finished product will require of the system running the application. So naturally a lot of testing have been done previously comparing the different frameworks available for programmers but since these tests take time to perform and the industry is an ever changing entity not all frameworks have been tested because they might have come recently to the market.

One of these frameworks is Flutter. The first stable version of flutter was released to the market in December of 2018 (Flutter, n.d) and is an open-source toolkit for creating cross platform applications. It's backed by google and has since its release become one of the most used frameworks on the market today, when asked in a survey 42% (Vailshery, 2022) of the software developers that took the survey had used Flutter. Since Flutter is still relatively new not a lot of previously performed studies have included this framework in their research, which is explored furthermore in the Literature overview chapter. The second biggest is React Native (released in 2015) being used by 38% (Vailshery, 2022) of the developers in this survey. React Native has amongst others been compared in several studies such as the ones performed by Huber and Demetz (2019) and Dorfer et al.(2020).

Purpose and target group(s)

The purpose of this study is to add knowledge to existing frameworks (frameworks which have already been tested) to help developers in choosing which framework when creating mobile applications. This is done by comparing Flutter to React Native. This is because Flutter is relatively new to the market and have only limited amounts of testing performed on it. So to compare it to one of the more tested frameworks React Native, would give developers a fair comparison and view of Flutter, which even since its new to the market

quickly have grown to become one of the biggest of the cross-platform frameworks on the market. In a survey made by Statista Flutter had been used by 42% of the developers worldwide who answered the survey.

Formulating the problem:

Flutter is a new framework supported by Google that released in December of 2018. Because of it being a new framework to the market it has only been included in some previously performed studies comparing the performance of different cross-platform frameworks. Flutter has since its release grown to be one of the biggest frameworks used by developers to create cross-platform applications.

1. What are the differences in the performance usage between Flutter and React Native?
2. What is the recommendation based on the performance results for future development?

To compare these frameworks, previous research has looked at the raw performance usage of different resources like CPU-, memory- and disk space consumption. As will be explained in Chapter 2 Literature overview. These resources are typically looked at since higher usage of these will result in the applications requiring more performance by the user device to run, if these are requiring too many resources from the target device it might lead to several potential users not being able to use the product as a result.

Scope

The study will limit itself to focus on the development of applications and not for usability or user experience. Furthermore, the study will limit itself to only testing Flutter and React Native which both are cross-platform frameworks. It will not include any native frameworks during testing, neither will it attempt to compare the results to other cross-platform frameworks directly. This study aims to serve as another source for developers to look at when deciding which framework to choose for their applications development.

Ethical considerations

The study will perform tests on the applications created and because of this no user studies be performed, neither will any gathering of personal data occur. The study will adhere to Karlstad university policies and guidelines.

Literature overview

The literature chapter will first look at what the programming frameworks used in the study, and in which integrated development environment (IDE) we will use to code the applications and to collect CPU- and Memory-consumption from the applications. An IDE is a software system that combines the common activities of writing a software such as editing/writing source code, debugging the application, building executables and it also usually also includes features such as syntax highlighting, autocompletion and automatic indentation to help the developer code and stay organised (Codecademy, n.d). Furthermore, the different keywords used to find related work within the specified topic will be explained. Finally, the major part of this chapter deals with related work and what studies have been performed previously.

Frameworks and keywords

The first framework we will look at is Flutter. Released in December of 2018 (Flutter, n.d) Flutter is the framework on the market but has since taken a great portion of the market share when it comes to cross-platform mobile application development. Flutter is a free and open source framework from Google (Flutter, n.d) which offers the possibility to create applications for mobile, web and desktop from one single codebase. Flutter is an object oriented language (Flutter, n.d) meaning it makes use of classes, methods and variables but it also uses imperative programming concepts such as conditions and loops (Flutter, n.d). Flutter is written in Dart which is similar to the syntax used by Java and C# (Ford, n.d).

The second framework is React Native which was released back in 2015 by Facebook. React Native have been used to create several of the biggest applications we can find today, such as Facebook, WordPress, Discord, Microsoft Teams and many more (React, n.d). React Native is written in JavaScript which in and of its own leads to it being very accessible to a lot of developers since JavaScript is widely known and used. A feature of React Native worth noting is that it can be used together with existing native code on an application that already exists. It accomplishes this by wrapping the native code into react components and then interacting with the native APIs (React, n.d).

The two different IDEs that will be used to create the mobile applications are Visual studio code (https://code.visualstudio.com/) and Android studio (https://developer.android.com/studio). Visual studio code will be used to create and compile the React Native application since it's one of the most used IDEs on the market. When

creating the Flutter application android studio will be used instead of Visual studio code because of the easier installation process when coding a Flutter application. Android studio is also a very popular IDE on the marked and equally as well known as Visual Studio Code amongst coders worldwide. In the end the use of the different IDEs doesn't affect performance and will not affect the outcome of the application tests since it's only a coding environment.

Android Studio Profiler (Android Studio Profiler [ASP], 2023) is a feature of the Android Studio IDE which allows developers to test and track the runtime performance of an application. The profiler can track an applications CPU- and memory consumption amongst other data. The Android Studio Profiler (ASP, 2023) access the CPU and memory consumption of the application on the target device through allowing USB debugging on the device and by using the physical device to run the applications instead of an emulator. USB debugging is a feature that can be accessed by enabling the developer options on the device.

To perform the literature overview, a few different keywords were used when searching for related work and other information. The keywords used were:

1. Cross-platform development
2. Mobile application development
3. Cross-platform frameworks
4. Performance comparison of cross-platform mobile applications

Tools and websites that were used to find the theses, books and articles mentioned in this thesis were Google Scholar, Karlstad University Library, Association for Computing Machinery (ACM) Digital Library, IEEE, ScienceDirect and Google.

Related work

This chapter will focus on the previously performed studies and similar work that have been made, on the topic of performance testing mobile applications created by cross platform frameworks. It will start by looking at what options there are to building cross-platform applications and which frameworks have been tested previously. Later it will focus on what performance parameters have been used by other related work and why these are important to look at. The chapter will also look at common application features and user interface (UI) interactions that are normally tested. Finally, it will revisit the research questions.

Mobile development frameworks

When performing studies on cross-platform frameworks there are some different approaches that researchers could take. One way is to compare different cross-platform frameworks against each other based on different criteria, but the key part here is that they are only using cross-platform frameworks. This approach has been used by IŞITAN and KOKLU (2020) and by Palmieri et al.(2012). IŞITAN and KOKLU (2020) choose to test React Native, Flutter, Nativescript and Xamarin. Palmieri et al.(2012) had four different frameworks mainly since their study was performed a lot earlier already in 2012. Palmieri et al.(2012, p.181-183) chose to test Rhodes, PhoneGap, DragonRad and MoSync.

Another common way is to add a single or a few native languages to the frameworks being tested, this gives the researchers a chance to look at the differences between native and cross-platform approaches. Researchers that choose this direction include Dorfer et al. (2020), Willocx et al (2016), Huber and Demetz (2019), Biørn-Hansen et al.(2020) and Barros et al.(2020).

Dorfer et al.(2020) tested the cross-platform framework React Native against its native counterpart Android. Dorfer et al.(2020) choose Android for their native application because it offers uniform interfaces that allows compatibility with several different android devices and software versions (Dorfer et al. 2020, p.190). They reasoned their choice of React Native as their cross-platform framework based on the popularity of the framework on GitHub and Stack Exchange (Dorfer et al. 2020, p.191). In contrast to Dorfer et al.(2020) Willocx et al.(2016) used a broader selection of frameworks, they had several both cross-platform and native frameworks. Willocx et al.(2016) tested Ionic, Famo.us, Intel App framework, JQuery Mobile, Mgwt, Sensa Touch 2, Adobe AIR, Titanium, NeoMAD and Xamarin. Willocx et al.(2016, p.40-41) selected their frameworks based on the availability on the community developed application called PropertyCross. PropertyCross is a community driven initiative which has described an application on their website which the community then have developed in over 20 different cross-platform frameworks/tools. Back to a more commonly used approach Huber and Demetz (2019, p.43) selected Android as their native framework, for their cross-platform framework they chose React Native and Ionic. They based their selections of the cross-platform frameworks on the reusability of code since both frameworks are based on web technologies (Huber & Demetz, 2019, p.42). Moving along Biørn-Hansen et al.(2020, p.3008) used the Ionic, React Native, Nativescript, Flutter, Android and MAML /

MD2 as their frameworks. Their choices are based on the different approaches to development that the frameworks can make use of, these being Hybrid, Interpreted, Model-Driven Software Development, Cross-compiled and Native. Lastly Barros et al.(2020) used Flutter and React Native as their cross-platform frameworks. For their native frameworks they used Java to produce the android native application and Swift to make the iOS native application. They chose Flutter because of it being newer and a promising candidate to the market, React Native since it's one of the biggest in use at the time of their study. They then used Swift as their native candidate to iOS and Java for their android native candidate (Barros et al. 2020, p.2). Additionally Barros et al.(2020, p.2) had four frameworks, they created one for implementation for each framework. They made sure the application had the exact same GUI, to avoid influences on the processing time from other sources than the code itself (Barros et al. 2020, p.3).

Most previous work has used several different frameworks, some staples are emerging as we compare them to each other. For example, React Native have been used in all of the above mentioned studies except for the two studies performed by Palmieri et al.(2012) and Willocx et al.(2016). Palmieri et al.(2012) performed their study back in 2012 and since React Native didn't release until 2015 it is easy to understand why it wasn't included in their study. Willocx et al.(2016) on the other hand probably didn't test React Native because of their slightly different approach to how they selected their frameworks that they were going to test. As previously mentioned, instead of creating applications in each of the frameworks Willocx et al.(2016, p.40) decided to use a community developed application called PropertyCross. Willocx et al.(2016, p.40) have chosen this approach because they themselves do not have to dedicate lots of time to learn and create the application in all of the different frameworks. Furthermore the applications created by the community are created by experienced coders which Willocx et al.(2016, p.40) argues will result in better quality code and a more representative measurement of the framework. Furthermore Flutter is more and more commonly included in the more modern studies such as the studies by Barros et al.(2020), Biørn-Hansen et al.(2020) and IŞITAN and KOKLU (2020).

Common application features and typical UI interactions

All the studies mentioned in this overview all created identical applications to perform their studies on. But what features and UI interactions that were built into the applications vary

quite a lot so this subchapter will go through how each study build their applications and later look at similarities between them.

The applications created by Dorfer et al.(2020, p.190-191) were basic restaurant finder applications. The apps consisted of 2 UI components. The first component is a map showing the location of the device running the application and showing restaurants nearby that device, the second component is a scrollable list containing all the restaurants as separate list entries also showcasing some various details of the restaurants. The reason Dorfer et al.(2020, p.191-192) created the applications with these specific components were to use the location-based services which uses techniques like GPS, Cell-ID or Wi-Fi and these services and techniques are often used by other applications like weather-, sports or navigation applications.

Willocx et al.(2016, p.40) used a community developed application called PropertyCross. PropertyCross is a community driven initiative which has described an application on their website which the community then have developed in over 20 different cross-platform frameworks/tools. Since Willocx et al.(2016) used this approach the specific features of the application will not be discussed in this paper.

Huber and Demetz (2019, p.43) created a basic mobile app consisting of a single screen with a scrollable list of contacts. The information displayed on each contact is its name and phone number. They chose this type of application since it's a commonly used UI component in many applications we use today. The main reason for these lists being popular is because of the limited size of the phone screens and a scrollable list solves that problem in many different cases.

Barros et al.(2020, p.2-3) developed one implementation for each of their four frameworks. Barros et al.(2020, p.2) used a slightly different approach and only developed functionality to call an application programming interface (API), download data from the API, display the first five items to the application and save one of them to a local database in one go. Barros et al.(2020, p.3) made sure they used the same logic in all the four applications. Barros et al.(2020, p.2) built their applications this way to later be able to automate their testing on the applications.

Biørn-Hansen et al.(2020, p.3009) created applications for each of their frameworks with 4 different menu options each leading to a separate page for every specific task/feature. Biørn-Hansen et al.(2020, p.3010) built features to test the presumably most common use cases at the time. The features Biørn-Hansen et al.(2020, p.3011) built into their applications are

features to test the accelerometer, the contacts, the file system and the geolocation. Biørn-Hansen et al.(2020, p.3011) describes more in-depth how they will achieve this testing with each feature.

Palmieri et al.(2012, p.180) used a more abstract approach to comparing their frameworks, instead of creating applications and performing tests on those applications Palmieri et al.(2012, p.180) decided to compare the frameworks based on different criteria. Palmieri et al.(2012, p.180) based their study on criteria such as which mobile operating systems each framework supported, which tool licenses they offered, what programming languages offered to developers, availability of API's, accessibility to native API's and finally which IDE's available to develop applications on.

Lastly IŞITAN and KOKLU (2020, p.276) also built one application for each of their chosen frameworks, their application simply generated and displayed a 1000 item long list with unique items. IŞITAN and KOKLU (2020, p.276) achieved this by building a loop that print "Mehmet" and a number each time the loop is initialised. Every pass of the loop increases the number by one to make sure each list item is unique.

To summarize, a majority of the studies created their own applications to test except for Palmieri et al.(2012) and Willocx et al.(2016). Willocx et al.(2016) still used applications to run their own tests on but decided to save time by using applications already developed on PropertyCross. This saves time but also limited Willocx et al.(2016) options of which framework they could potentially test. Palmieri et al.(2012) used a more abstract approach and only looked at the availability of different criteria for the application, which gives them incomparable results to the other studies but at the same time it shows there are different ways to approach the problem of selecting a framework to work with.

Comparing performance in mobile apps

When comparing performance in mobile applications the specific parameters that the researchers look at are usually determined and standardised by the older studies. For example, most studies commonly look at CPU consumption and memory consumption for their applications like we can see in the studies performed by Dorfer et al.(2020), IŞITAN and KOKLU (2020), Willocx et al.(2016). Biørn-Hansen et al.(2020) and Huber and Demetz (2019) also chose to study the CPU and memory consumptions during their testing.

To gather data the different studies had several different devices that they used. Dorfer et al.(2020) used a Samsung Galaxy S10E device and a Samsung Galaxy S8 device.), Biørn-Hansen et al.(2020, p.3012) also used physical devices but never specifically mention which ones they used. Barros et al.(2020, p.3) used an iPhone 7 as the iOS device and a Samsung Galaxy S8 as their Android device. Willocx et al.(2016) used several devices including an iPhone 4, iPhone 6, Sony Experia E3, Motorola Nexus 6 and a Nokia Lumia 925. Huber and Demetz (2019, p.44) used a LG Nexus 5. Going against the trend of using physical devices IŞITAN and KOKLU (2020, p.276) instead chose to use an emulator of the Google Pixel 3 device. Also because of their way to gather study results Palmieri et al.(2012) didn't use any test devices at all.

How the researchers performed their studies varied a bit, but a common way was to automate their tests to be able to run them several times without errors or differences. Dorfer et al.(2020) also set up automated tests with UI-Automator (UI-Automator, n.d) which is a framework for creating automated UI testing across system and installed applications on android devices. Huber and Demetz (2019, p.44) automated their testing by building a python script that opened the app, started the measuring tool, performed a few swiping interactions and then closed the recording and app. Finally, it uninstalled the application. Barros et al.(2020, p.2) Automated their testing by having the applications perform their functionality 100 times on startup for each of their applications. IŞITAN and KOKLU (2020) also only run a single loop in their program, it runs 1000 times and no manual input is needed so it's also an automated process.

Another way to execute the testing is to manually interact with the application, this was done by Biørn-Hansen et al.(2020, p.3012) who notes that this was a very time consuming process for their study since they tested several functionalities on several different devices. Willocx et al.(2016) never mention if there testing process are manual or automatic, but based on how Willocx et al.(2016, p.4) describe how each of the applications pages are accessed to gather different pieces of data the process can be assumed to be manual. Finally the last study is Palmieri et al.(2012) who never gathered any data from testing but instead looked at what features the manufacturer claimed to be supporting.

Moving on the focus will now be on what the studies found out through their testing. Beginning with results that Dorfer et al.(2020, p.193-194) gathered shows the average CPU and memory usage of all the 10 tests. Dorfer et al.(2020, p.193) found that the CPU usage of the applications were always higher on the application produced by React native on both

phones. For the memory consumption the results were similar to the CPU usage where the React native app consumed more than the application created by the Android SDK however on the Samsung Galaxy S10e the memory usage settled in better during the duration of the test. Lastly the results of the battery consumption showed that on both phones the React native application used 6%-8% more battery than the Android SDK application did and Dorfer et al.(2020, p.195) concludes that this is because of the increase in CPU and memory usage. When Willocx et al.(2016, p.6) analyse their results they find that both the android and windows applications allocates significantly more RAM than the iOS counterparts. Willocx et al.(2016, p.6) also find that android applications allocated more RAM to devices that had more RAM available to them. They see a similar situation when looking at the cross-platform applications compared to the native ones. The only frameworks that breaks these results are the JavaScript based ones which uses more memory on low-end iOS devices (Willocx et al. 2016, p.6). Willocx et al.(2016, p.7) restrains from drawing any conclusions about their frameworks CPU consumption. This is because they realise that in order to do so the sample size of the intervals gathering the CPU consumption has to be so small it will gather huge amounts of data. Instead Willocx et al.(2016, p.7) choose to only discuss the behaviour visible through their results. When Willocx et al.(2016, p.7) look at their results form disc space consumption they find that the native applications requires less space than the cross-platform applications.

Huber and Demetz (2019, p.44) measured both their CPU consumption and memory consumption in 1 second intervals. Huber and Demetz (2019, p.42) notes that the interactions in the application could have big impact on the gathered results whilst using intervals. Huber and Demetz (2019, p.43) aims to avoid this by only using continuous swiping during their testing. Huber and Demetz (2019, p.44) finds that both the cross-platform frameworks they tested consume more CPU than the native counterpart overall. Huber and Demetz (2019, p.46) get a similar result when they analyse the memory consumption. Biørn-Hansen et al.(2020, p.3030) concludes that the model-driven framework is the one that most closely resembles the native in its consumption of resources. Biørn-Hansen et al.(2020, p.3031) have an interesting finding when looking at RAM consumption of their applications. Biørn-Hansen et al.(2020, p.3031) finds that Flutter has an average score when looking a PreRam and RAM scores they have given, but when the researchers looked at the ComputedRAM they found that Flutter had a lower result than any other framework. The authors concluded that this indicates that Flutter has in their study lower peaks of memory usage when performing tasks,

whilst still having a high overall consumption. Barros et al.(2020, p.2) decided to look at the time to complete specified actions on their applications and compare those to eachother. Barros et al.(2020, p.3) found in both the android applications and the iOS applications the native apps performed the fastest in all categories. Furthermore, on android both React Native and Flutter had similar response times in all categories except for the REMOTE category. This category was measured using the time it took the application to make a request and receive all the data. In the remote category Barros et al.(2020, p.3) found that React Native was faster than Flutter. When testing the on their iOS device Barros et al.(2020, p.4) got similar result but React Native outperformed Flutter in all categories and React Native even outperformed the native application when storing the data locally.

IŞITAN and KOKLU (2020) is the last study in this literature overview that performed their own testing. IŞITAN and KOKLU (2020, p.277) used Android Studio to collect their data and when looking at CPU consumption, they found that Nativescript and Xamarin had the lowest peaks of CPU consumption throughout the test. Xamarin and Nativescript used 62% and 70% of total available CPU respectively whilst React Native used the most at 79% and Flutter second most at 75%. However, the task completion times showed that Nativescript was the fastest to complete the task at 14 seconds, Flutter cam second at 28 seconds, Xamarin third at 36 seconds and last finished React Native taking a total of 51 seconds to complete the task. IŞITAN and KOKLU (2020) found similar results when looking at the memory consumption with Xamarin using the least amount of resources and React Native used the most. Finally Palmieri et al.(2012, p.186) didn't study the performance of the applications but they concluded through their research that the Rhodes framework had the edge over the others since it provided both web based services and a model-view-controller (MVC) framework.

Revisiting the RQs

In the literature review we see some different approaches to testing and evaluating performance of frameworks but the most common is to create applications, include the desired features and create either automated on manual tests to run. Later researchers commonly compare the results on a point-by-point basis and these commonly used approaches will help design this study's method, it will also help select what performance parameters to test during the study.

Since most of the studies covered in this overview used CPU and memory consumption as their main qualities when testing the performance of the frameworks, this study will include these as well. Furthermore, three of the studies chose to include the disc space consumption. Disc space consumption could be a valuable part of selecting which framework to work with this will also be included in the study when answering the first research question "*What are the differences in the performance usage between Flutter and React Native*".

The second research question "*what is the recommendation based on the performance result for future development*" build on the knowledge gathered and will aim to help developers and researchers alike to choose how to further test the frameworks and to make informed decisions on which frameworks to chose for their own work. Giving recommendations based on the results is also common throughout all the mentioned studies.

Method

Study Design

During this thesis a quantitative study will be performed. The quantitative research is conducted through the gathering and analysis of measurable data (Patel and Davidson, 2011, p.13-14). The thesis will use quantitative research approach since numerical data (Bhandari, 2020) such as memory usage, CPU usage and disc space will be gathered from the device used to test the applications created. After the quantitative study is completed a comparative analysis of the gathered data will then be used to compare and analyse the results. Comparative research method focusses on comparing already existing pieces of data (Williams, 2021). The results of the comparative analysis will be used to answer the thesis research questions.

Some of the advantages of a quantitative research approach is the efficiency when gathering numerical data since computing equipment make it possible to process and analyse data quickly (Williams, 2021). Other advantages are the possibility to directly compare the results statistically to other studies (Bhandari, 2020), and the possibility of replicating the study is easy and accessible since it uses standardized protocols for collecting data (Bhandari, 2020). The gathering of quantitative data is another big advantage since this allows unbiased results to be communicated to others (Williams, 2021).

There are also a few notable disadvantages of using a quantitative research method, mainly a narrow focus of the data collection since it focuses on predetermined measurements which could lead to the oversight of other relevant observations (Bhandari, 2020). Furthermore, imprecise measurements or inappropriate sampling methods could lead to a negative structural bias, and it might also lead to faulty conclusions (Williams 2021).

The study will use React Native and Flutter to create two simple and identical mobile applications that will be later installed on an android mobile device. The mobile device used will be a One plus Nord 2T 5G and it's on this mobile device the testing will be performed.

Data Collection

To collect data from the device Android Studio Profiler (ASP, 2023) will be used and it's a tool included in the Android Studio IDE. ASP has also been used in previous studies by IŞITAN and KOKLU (2020) and Biørn-Hansen et al.(2020) to collect CPU and memory

consumption. The mobile device will be connected to a computer through USB and the computer will run Android Studio Profiler (ASP, 2023) and through a series of structured tests on the application the CPU and memory usage will be measured through the Android Studio Profiler (ASP, 2023). Also, the study will look at the disc space required to install and use the applications. To verify the validity of the data collection we make sure to use the Android Studio Profiler (ASP, 2023) since it's a commonly used and accepted method to collect data from a system. To make sure we have good reliability on our data the tests will be run several times and an average of the measured results will be used as our final data. This is a commonly used way to collect data that we can find in many other studies like the one performed by Hubert and Demetz (2019, p.44).

Before the tests were conducted a pilot test was conducted to validate the test procedure and to check that all the required data was able to be collected in a correct and useful way. The pilot test also aims to validate the statements made by Huber and Demetz (2019) and Willocx et al.(2016) that using intervals may not be show the CPU consumption in a correct way, unless extremely small interval timings were used.

The test will be manually performed by following a set number of steps that will be performed on the test device. These steps will include actions typically performed on a smartphone application such as scrolling, clicking on buttons, and inputting text/numbers into input fields in the application. The test will be conducted several times on each application to make sure an average of the data is allowed to be collected and to maintain reliable results.

Pilot Test

A preliminary pilot test was conducted to make sure the data collection through Android Studio profiler would work as intended as well to ensure that the data collected reflect the performance of the applications.

For this pilot test Android studio Profiler (ASP, 2023) was used together with the Flutter demo application that is automatically created when you create a new program with Flutter (Flutter, n.d).

The demo application is a simple one-page application with a header and a button in the middle of the screen together with a text saying, "you have pressed the button 0 times". Each time you press the button in the demo application the zero in the text updates and it simply tracks how many times the button was pressed.

The pilot test concluded that ASP (2023) worked as intended and all the required data would be able to be gathered through the program. A change to what data was being gathered happened after the pilot test, due to it showing that the CPU usage of the application always had a noticeable peak after each interaction. If this was not taken into consideration the interval data could possibly present a slightly misleading result since these peaks did not always happen at the set intervals. This confirms what Huber and Demetz (2019) and Willocx et al.(2016) noted during their studies. Since the CPU peaked after an interaction this data was deemed necessary to collect and could be one way to better show the true CPU consumption of the application. There was no similar difference in the memory consumption of the application after interactions and for that reason there is no similar chart for memory consumption after interactions in the results.

The results of the pilot test, the preparation for the pilot test and the setup for the testdevices can be found in appendix 1.

Test Design Step-by-Step

The test that will be conducted is set up by 16 numbered steps that will be followed each test to make sure every test is done correctly and the same way. The test steps have been set up to test commonly used interactions of mobile applications. These interactions are scrolling, clicking, and typing which are all very commonly appearing interactions any user would perform on almost any given modern mobile application.

Application test steps

1. Start the application on the test device.
2. Start Android Studio Profiler.
3. Start the profiler recording.
4. Scroll to the bottom of the application.
5. Scroll to the top of the application.
6. Repeat step 4 and 5 two more times.
7. Select input field “Weight” for the “Arnold Press” exercise and input “25” as a value.
8. Select input field “Repetitions” for the “Arnold Press” exercise and input “10” as a value.
9. Scroll down to the bottom of the application.
10. Select input field “Weight” for the “Squat” exercise and input “65” as a value.

11. Select input field “Repetitions” for the “Squat” exercise and input “8” as a value.
12. Press the floating action button named “Info”.
13. Press the “Cancel” button within the popup alert generated from the “info” button click.
14. Press the floating action button named “Info”.
15. Press the “Submit” button within the popup alert generated from the “info” button click.
16. Stop the profiler recording.

After each test is completed the results of the recording from the Android Studio Profiler (ASP, 2023) were analysed and data was gathered about what amount of percentage of CPU power was used by the application at one second into the test to see how the applications performed during the startup. Then data was collected at five second intervals throughout the test session so at five seconds into the test, ten seconds, 15 seconds and so on until the end of the session. In this form the memory consumption of how many megabytes (MB) was used by the application as well. Data about the peak CPU usage after each interaction was also gathered and lastly from the test device the amount of storage space each application used was also collected.

Test Applications

To conduct the tests two mobile applications were created, one using Flutter (Flutter, n.d) and one using React Native (React, nd.). The two different applications were tested will have an identical look and feel as well as the same functionality and code structure to ensure both applications are tested on the same basis.

The content and functionality of the applications are built to function as a workout application to help me register and save information about what weights were used and how many repetitions were performed during the workout. Based on this context the following application specifications were created.

Application Specification

Components:

- Header.
- Scrollable list.
- List items.

- Text fields to enter data.
- Floating action button which shows an alert dialog.

Features:

- Scroll through existing workouts.
- Select workout and enter data about weights used and number of repetitions.
- Show alert dialog with information about selected workout.

Applications Designs used in the test.

The look of the two applications (see Figure 1 & 2). They include an orange header element with the text “Exercises” and a light grey body element containing a scrollable list with 19 different exercises. Each exercise has two input fields, one for weights and one for repetitions for the user to fill in. Lastly, we have a floating action button at the bottom right corner of the application which will display a alert dialog (see Figure 3 & 4) with a header, a main body of text and two buttons, one for “cancel” and one to “submit” but at this point in time the functionality of the both buttons are the same. The two buttons simply close the dialog.

Figure 1 React Native application

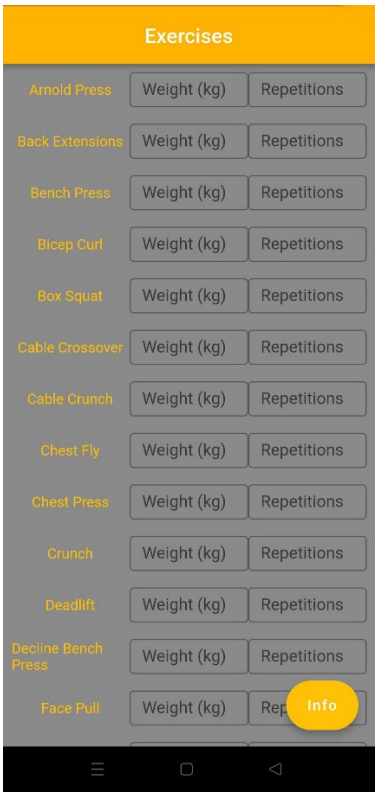

Figure 2 Flutter Application

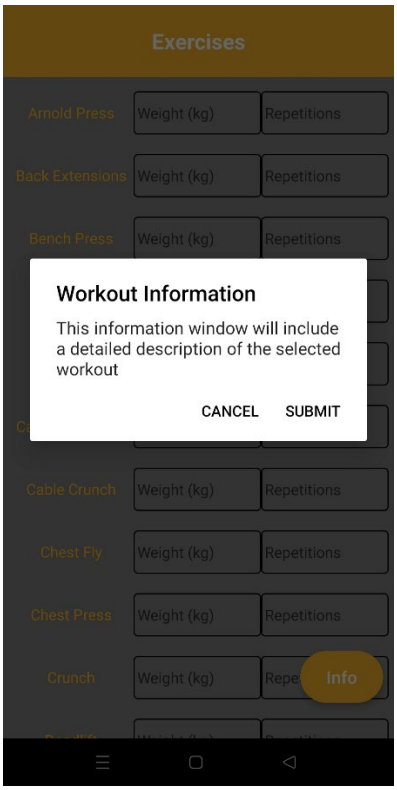

Figure 3 Alert dialog in React native application.

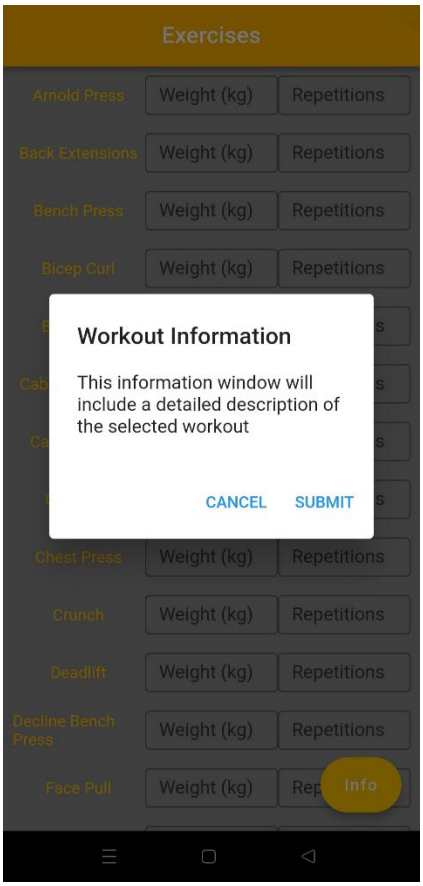

Figure 4 Alert dialog in Flutter application.

These applications have been created to allow the user to perform all the different functionalities specified in the "Features" list under the "Application Specification" heading.

Data Analysis

A comparative analysis will be used to compare the gathered data from the tests. This analysis will look at the relationship between the two frameworks and primarily the focus will target the differences of the collected data. The analysis will use a point-by-point structure (Walk, 1998) where data from both framework applications will be looked at and compared the data collected one by one (first looking at CPU usage, then on memory and so on).

The data will be structured and displayed in separate diagrams, one showing CPU usage, one for memory usage and a third for disc space usage. The diagrams will also be separated for each framework, three showing the data for the application created by the React Native framework and three showing the data for the application created by the Flutter framework.

Results

Two identical applications were created, one using React Native and one using Flutter. Before the formal tests were run a pilot test took place. The results of the pilot test can be found in Appendix 1 and the memory consumption during the set interval timings one second, five seconds, ten seconds and so on can be seen in table 11. The result from the CPU consumption of the pilot test can be found in table 12 and this resulted in a change to the data gathering during the formal tests. In table 12 we can see that during the intervals 1 second, 5 seconds and 20 seconds the results are 0% in all 3 of the cases. However, at the intervals 10 and 15 seconds the results are higher since these intervals were much closer to screen interactions which caused the CPU usage to peak. Since this sometimes happens at the selected intervals and sometimes it doesn't a third table was included in the formal test displaying all peaks of CPU consumption immediately after each screen interaction.

After the pilot test was completed, the applications were created and the formal testing begun. These test sessions included a set number of steps described in the methodology chapter. Each application then went through 10 separate tests to collect the data.

The results produced inside Android Studio Profiler can be found in Appendix 2 figures 22-41 but the results will also be summarised and displayed below in tables and line diagrams below.

React Native application results.

Firstly, the React Native application was tested and the results of the memory consumption can be found in table 1 below. The data is displayed in megabytes (MB).

Table 1 Memory consumption in MB during set intervals of each test on React Native application

Interval (sec)	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
1	221,2	206,6	224,5	222,9	221,2	219,6	219,4	219,6	219	206,6
5	229	222,8	217,4	250,2	229	230,1	240	231,5	231,4	213,8
10	239,1	217,4	240,4	232,9	239,1	243,2	230,8	236,5	238,6	238,7
15	239,9	220,6	239,2	247,3	239,9	231,7	232	239,2	239,3	232

20	236	207,8	239,7	249,2	236	234,8	231,7	237,4	234,1	234,8
25	237	208,4	228,9	249,4	237	216,2	224,3	230,4	231,5	229,8
30	230,8	209,4	231,9	240,5	230,8	242	233,4	232,1	234,5	237,8
35	230,6	207,8	227,6	243,8	230,6	243,7	235,9	244,7	234,5	232,9
40	231,5	207,9	227,7	224,2	231,5	244,1	236	244	234,6	233,1
45	233,4			224,3	233,4					
50	233,8									

The data in Table 1 is summarized in the diagram in Figure 5 below. The Y axis in the diagram displays how many MB is used by the application and the X axis displays the time intervals.

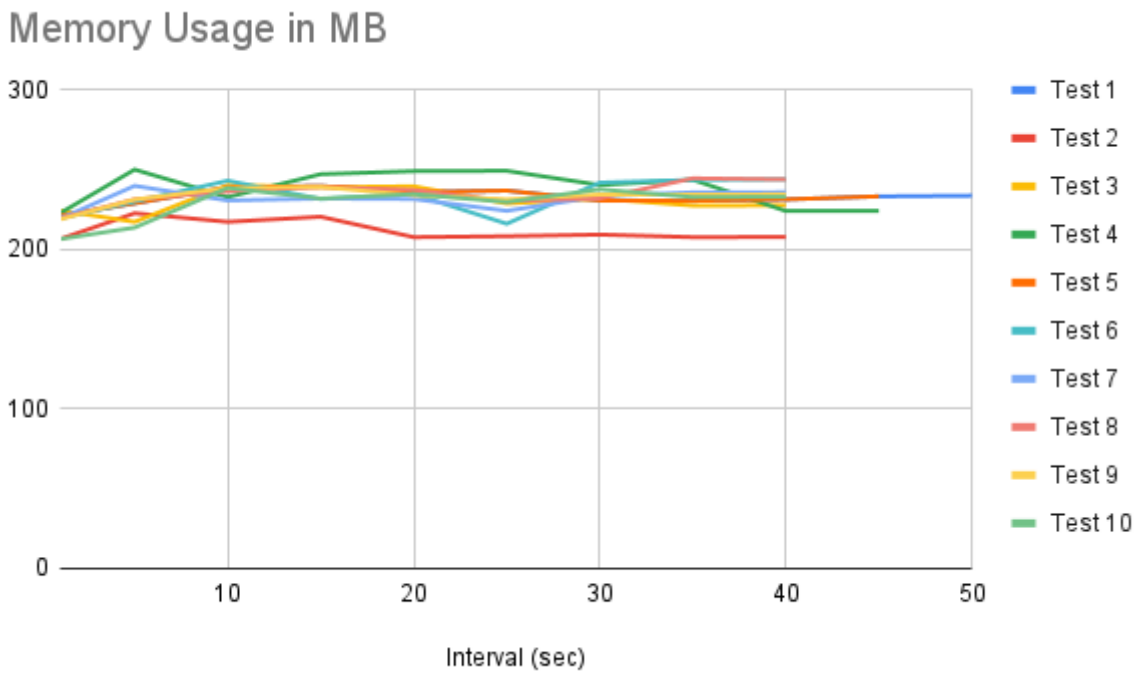

Figure 5 Results of the memory consumption from React Native applicaiton testing, displayed in a line diagram

The results of the CPU consumption during the tests can be found in table 2. The data is displayed in what percent (%) of the total available CPU the application is using.

Table 2 CPU consumption in percent (%) of total available CPU during set intervals of each test on React Native application

Interval (sec)	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
1	3	1	3	1	3	1	2	3	3	3
5	20	13	3	25	3	25	9	22	17	4
10	2	3	20	5	8	13	3	11	10	16
15	1	16	2	2	8	3	7	3	3	3
20	5	1	17	2	25	21	22	22	4	16
25	4	1	3	4	7	3	2	3	16	8
30	2	1	3	4	2	3	12	5	3	10
35	18	1	3	2	3	3	2	8	1	5
40	2	1	2	3	2	4	2	3	5	3
45	3			2						
50	2									

The data in Table 2 is summarized in the diagram displayed in Figure 6 below. The Y axis in the diagram displays what percentage (%) of total available CPU is used by the application and the X axis displays the time intervals.

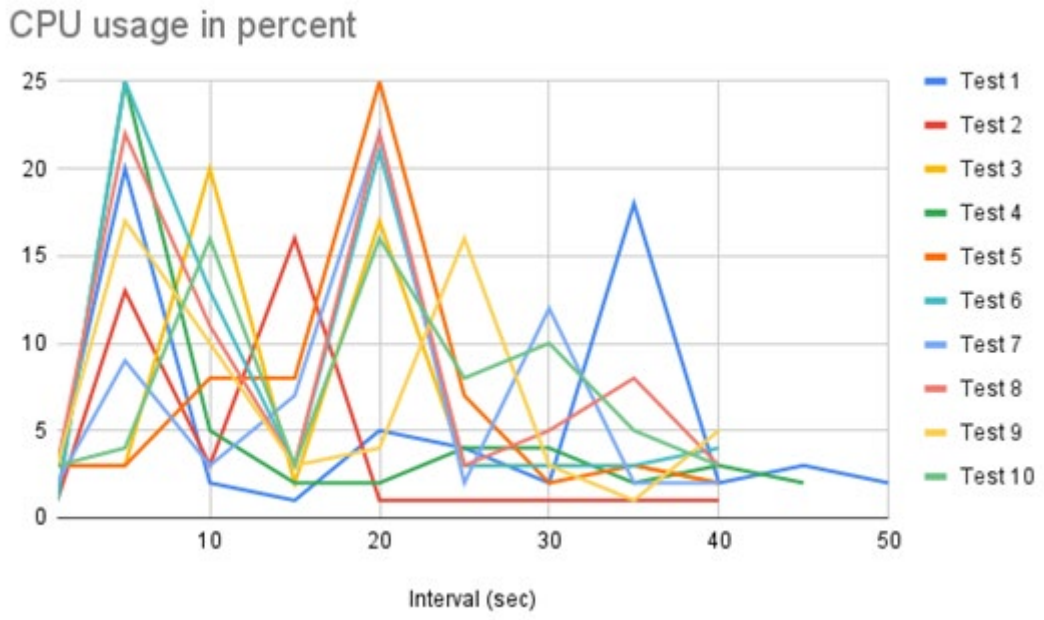

Figure 6 Results of the CPU consumption in percent (%) from React Native testing, displayed in a line diagram.

Table 3 shows the peak CPU consumption in percent after each interaction in the React Native application.

Table 3 Peak CPU consumption in percent (%) after each interaction during the test.

Interaction	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
1	20	17	23	26	21	25	21	22	28	26
2	19	16	22	24	23	25	24	20	23	26
3	20	20	21	20	21	16	25	22	20	26
4	23	22	20	21	19	16	23	22	26	28
5	23	17	17	26	20	18	21	18	21	26
6	20	19	22	16	17	20	25	22	19	23
7	6	12	7	4	8	10	9	10	8	9
8	6	7	7	6	8	9	10	8	5	7
9	24	22	19	21	22	21	21	32	24	20
10	10	23	22	19	25	21	22	25	25	4
11	3	18	16	12	13	25	14	12	7	6
12	27	10	5	10	7	24	6	9	6	8
13	18	5	7	14	25	6	23	22	21	19
14	18	18	18	16	15	21	16	15	16	16
15	21	16	20	19	20	16	18	21	19	23

The data in Table 3 is summarized in the diagram displayed in Figure 7 below. The Y axis display the peak CPU consumption in percent (%) and the X axis displays the interactions.

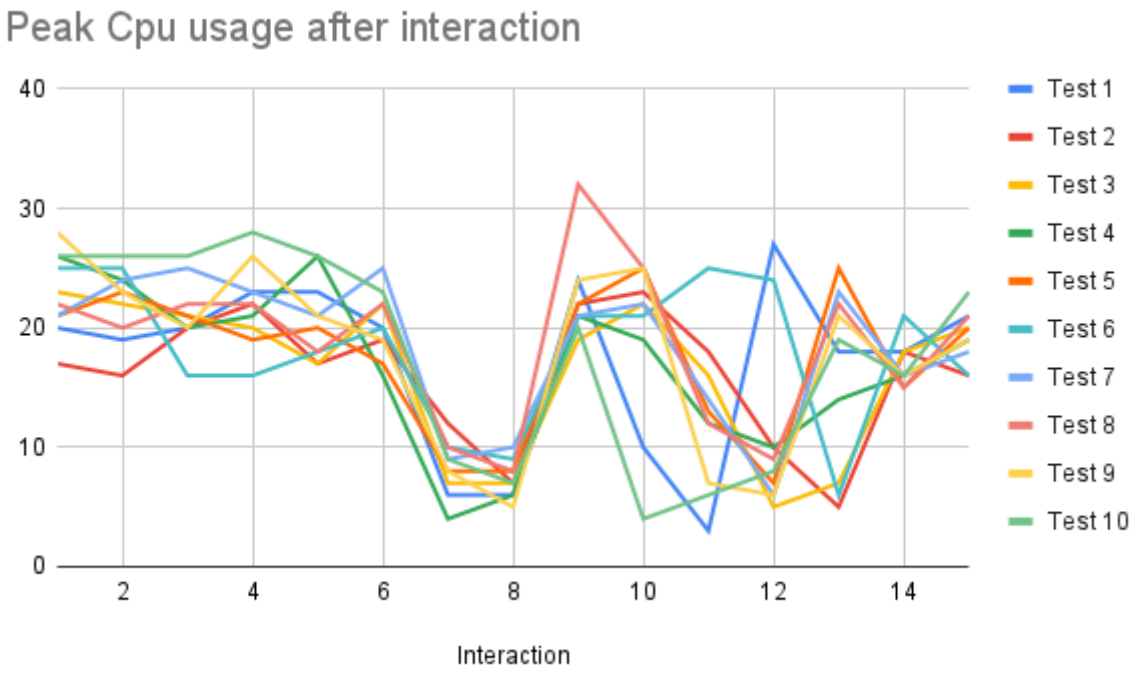

Figure 7 Peak CPU consumption in percent (%) after each interaction during the React Native application test.

Flutter application results.

After the React Native application was tested the testing on the Flutter application was completed as well. In table 4 you find the memory consumption from the Flutter application during the tests.

Table 4 Memory consumption in MB during set intervals of each test on the Flutter application

Interval (sec)	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
1	185,4	185,1	185,5	209,5	187	185,9	186,7	185,7	187,5	186
5	195,9	204,2	198,8	220	199	197,9	205,2	207,9	207,1	206,6
10	211,2	212,3	209,9	225	206,3	227,1	203	215,1	214,6	212,9
15	210	208,6	207,7	223,5	211	214	211,8	205,5	214,1	209,8
20	214,9	217	210,2	221,9	213,2	206,5	206,1	226,1	208,1	215,1
25	222	217,7	223,3	216,3	217,9	219,6	210,1	217,9	218,8	218
30	229,2	227,9	237,2	231,2	221,6	218,4	218,8	232,3	221,4	221,8
35	223,5	226,6	226,4	229,2	228,4	224,7	224,5	231,9	245,1	225,1
40	223,2	226,4	224,8	229,9	227,1	224,2	226,5	231,9	224,6	224,4
45			224,3				226,2			

50										

The data from x can be seen in a line diagram in x. The Y axis in the diagram displays how many MB is used by the application and the X axis displays the time intervals.

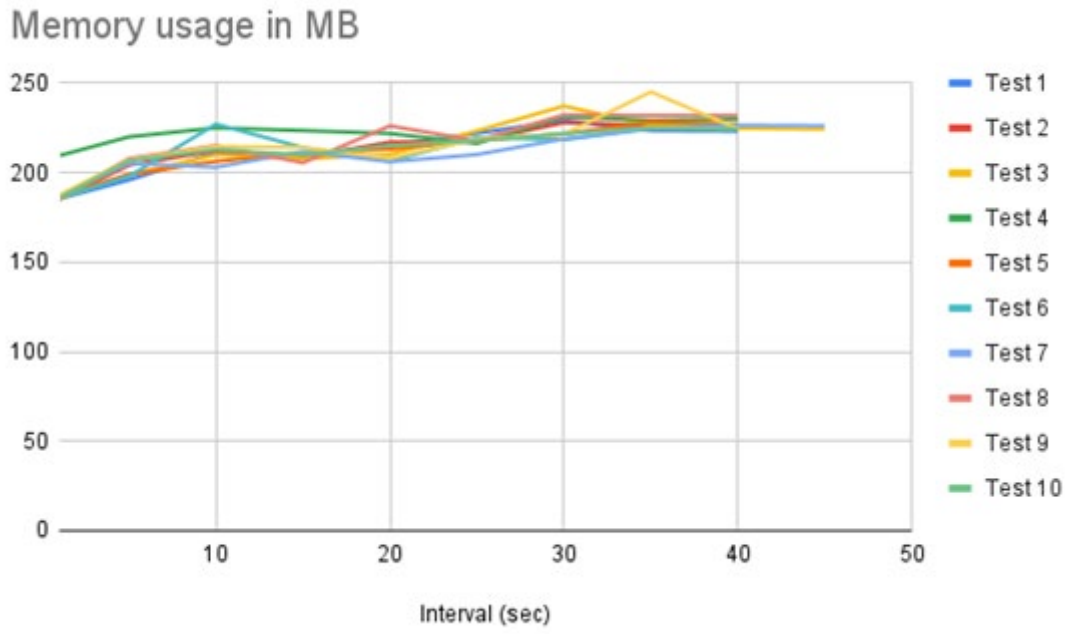

Figure 8 Result of tje memory consumption from the FLutter application testing, displayed in a line diagram.

The results of the CPU consumption in percent (%) of total available CPU in the Flutter application is displayed in table 5.

Table 5 CPU consumption in percent (%) during set intervals of each test on the Flutter application

Interval (sec)	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
1	0	0	0	0	0	0	0	0	0	0
5	0	7	25	5	0	0	13	6	6	8
10	5	6	25	5	23	6	0	5	8	15
15	8	5	0	1	0	1	0	1	0	12
20	2	1	1	27	3	1	0	17	4	0
25	9	0	0	0	1	2	0	27	1	0
30	2	0	23	6	0	0	8	12	18	0
35	0	0	3	0	0	14	20	0	1	14
40	0	0	0	0	0	0	0	0	0	0
45			0				0			
50										

The data from Table 5 can be seen in a line diagram in Figure 9. The Y axis in the diagram displays what percentage (%) of total available CPU is used by the application and the X axis displays the time intervals.

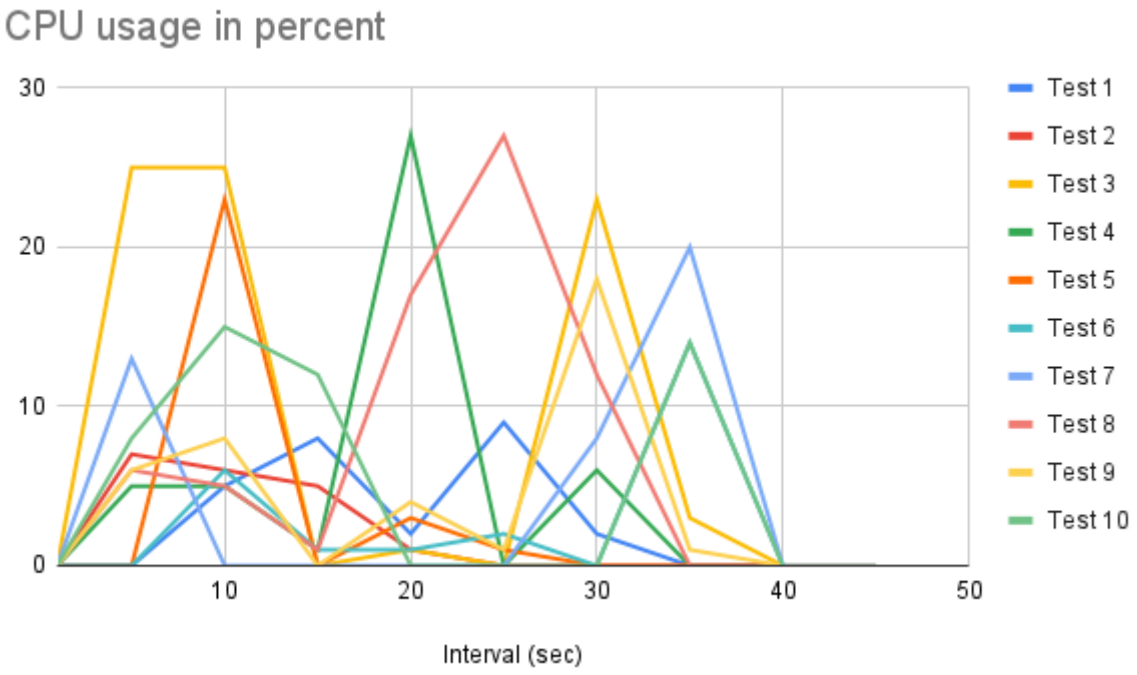

Figure 9 Results of the CPU consumption in percent (%) from the Flutter application testing, displayed in a line diagram.

Table 6 shows the peak CPU consumption in percent after each interaction in the Flutter application.

Table 6 Peak CPU consumption in percent (%) after each interaction on the Flutter application

Interaction	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
1	19	24	25	19	18	21	25	22	19	27
2	14	19	24	18	18	27	26	33	30	34
3	16	6	23	19	26	27	26	27	14	26
4	17	22	29	18	23	24	24	29	26	21
5	22	33	25	22	30	28	20	18	20	22
6	25	18	18	21	25	21	18	26	23	19
7	23	22	17	17	20	20	21	17	17	17
8	20	15	18	19	14	15	19	15	18	12
9	17	25	20	27	22	26	22	28	27	21
10	25	12	22	10	19	18	16	17	19	22
11	29	17	1	17	10	20	15	11	16	14

12	18	26	8	11	28	28	19	27	18	11
13	28	21	23	23	30	22	20	33	28	16
14	13	12	20	12	14	14	15	15	21	15
15	24	16	16	15	20	18	15	13	21	12
16	22	16	19	16	13	14	15	30	14	25
17	15		16	12						14
18	20									
19	21									

The data from Table 6 can be seen in a line diagram in Figure 10. The Y axis display the peak CPU consumption in percent (%) and the X axis displays the interactions.

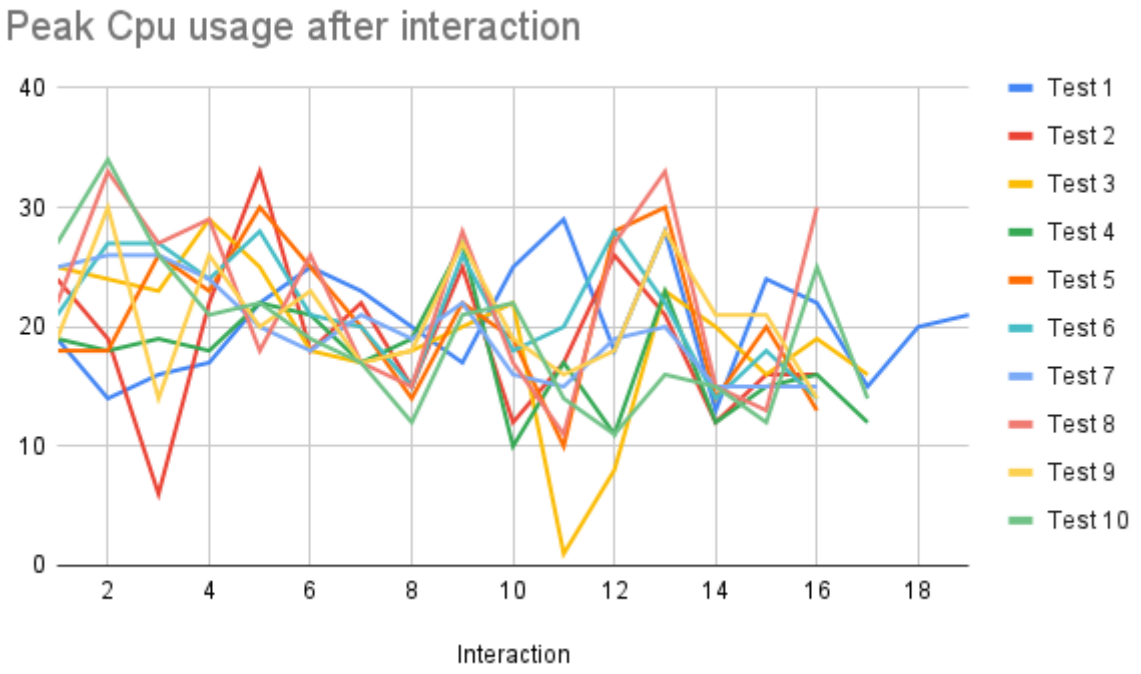

Figure 10 Results of the peak CPU consumption in percent (%) after each interaction from the Flutter application testing, displayed in a line diagram.

Average application results

After all tests on both applications were completed an average of the React Native tables were collected as well as a average of all the Flutter tables. These averages were put together in three different tables to display differences in the two applications. In table 7 both the average memory consumption of the React Native application and the Flutter application is displayed.

Table 7 Average memory consumption in MB from both applications at each given interval

Interval (sec)	React Native	Flutter
1	218,06	188,43
5	229,52	204,26
10	235,67	213,74
15	236,11	211,6
20	234,15	213,91
25	229,29	218,16
30	232,32	225,98
35	233,21	228,54
40	231,46	226,3

The data from Table 7 is displayed in a line diagram in Figure 11 where the Y axis displays the memory consumption in MB and the X axis displays the interval timings.

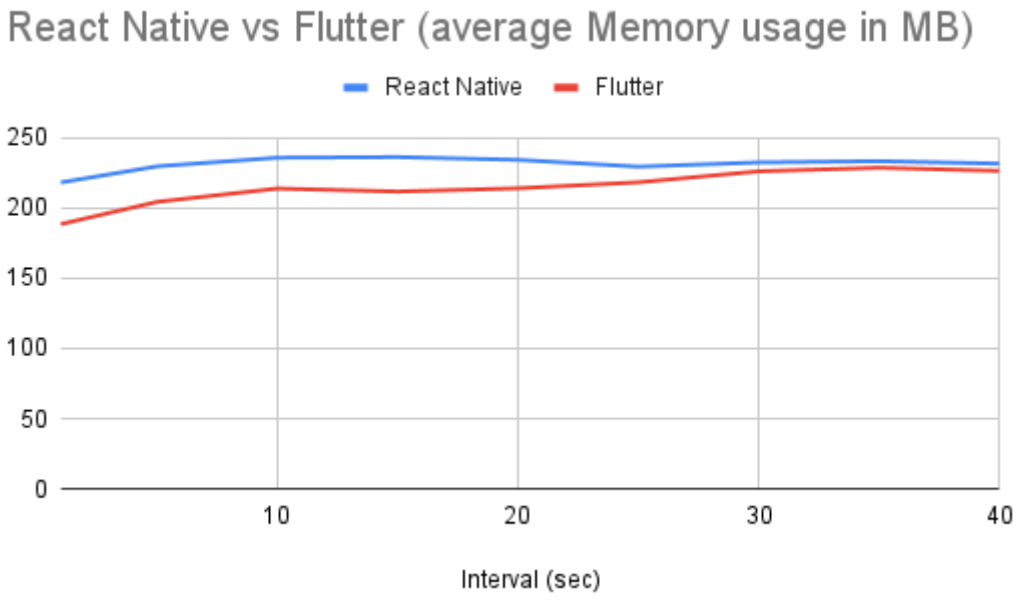

Figure 11 Average memory consumption in MB from both applications.

The average CPU consumption at each given interval was also collected and is displayed in table 8.

Table 8 Average CPU consumption in percent (%) from both applications

Interval (sec)	React Native	Flutter

1	2,3	0
5	14,1	7
10	9,1	9,8
15	4,8	2,8
20	13,5	5,6
25	5,1	4
30	4,5	6,9
35	4,6	5,2
40	2,7	0

The data from Table 8 is also displayed in a line diagram in Figure 12, where the Y axis displays the CPU consumption in percent (%) and the X axis displays the interval timings.

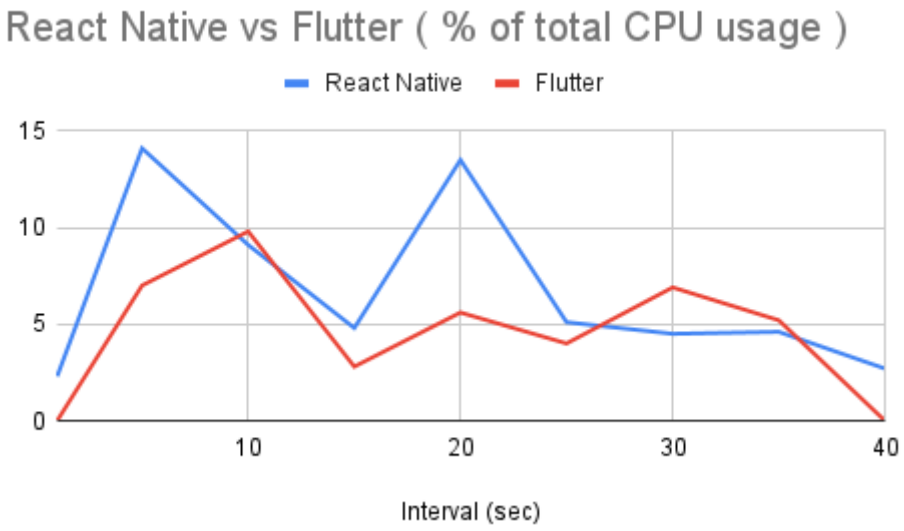

Figure 12 Average CPU consumption in percent (%) from both applications.

Lastly data about the average of the peak CPU consumption after interactions were collected and these are displayed in table 9.

Table 9 Average peak CPU consumption in percent (%) from both applications

Interaction	React Native	Flutter
1	22,9	21,9
2	22,2	24,3
3	21,1	21
4	22	23,3

5	20,7	24
6	20,3	21,4
7	8,3	19,1
8	7,3	16,5
9	22,6	23,5
10	19,6	18
11	12,6	15
12	11,2	19,4
13	16	24,4
14	16,9	15,1
15	19,3	17

The data from Table 9 is also displayed in a line diagram in Figure 13, where the Y axis displays the CPU consumption in percent (%) and the X axis displays the interactions.

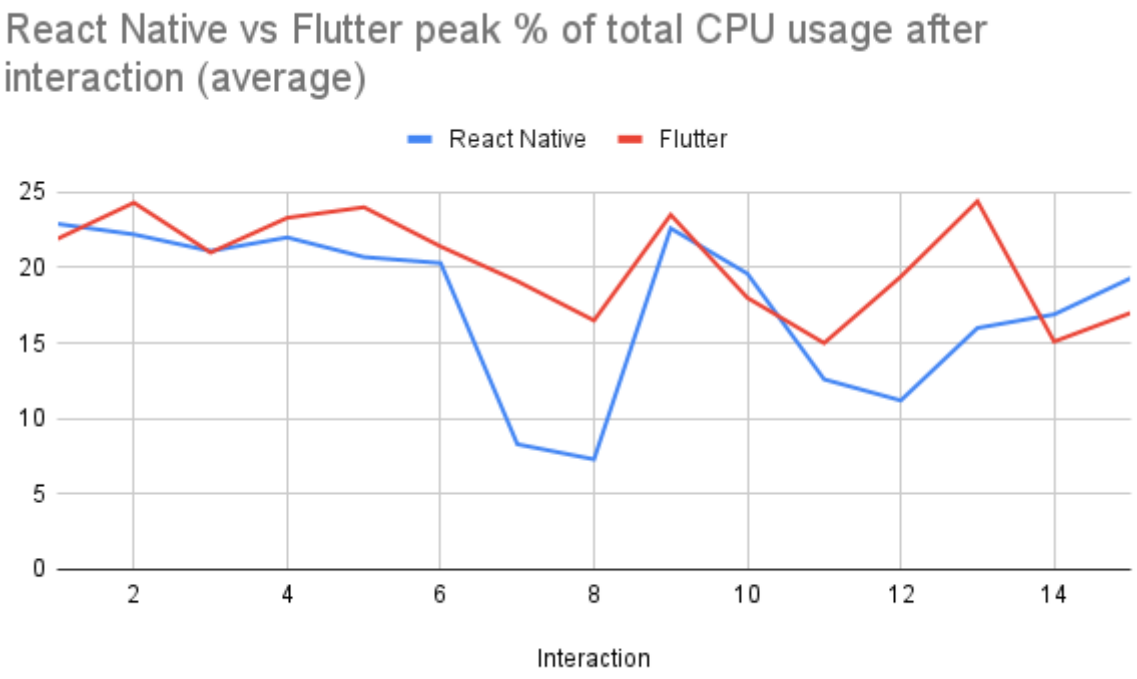

Figure 13 Average peak CPU consumption in percent (%) from both applications.

Disc space results

The Disc space required by both the applications after installation on the test device is shown in table 10.

Table 10 Disc Space in MB used by both applications.

Application	Disc Space (MB)
React Native	122
Flutter	129

The data from Table 10 is shown in a diagram in Figure 14.

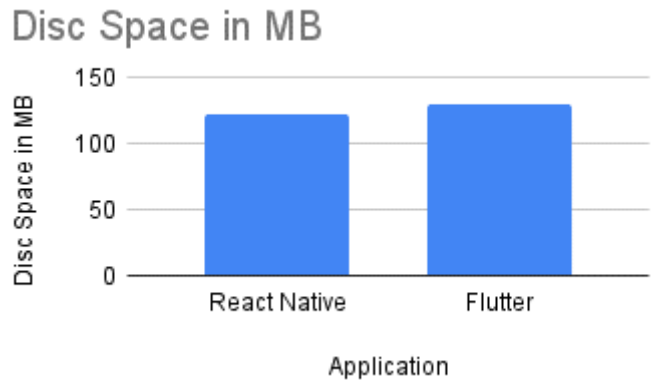

Figure 14 Disc space in MB used by both applications.

Discussion and Analysis

The main part of this chapter will focus on analysing and discussing the results and findings of the study, how they compare and what differs between the results from the Flutter and React Native applications. This will also briefly look into limitations of the study and finally it will look at what future work could be performed to further enhance the study.

The chapter will split itself into four sections and work by a point-by-point structure as described in the Method chapter. Firstly, the memory consumption of the applications will be looked at, secondly the CPU consumption, thirdly Disc space consumption and finally these results will be compared to the results found by other studies.

RQ1

- What are the differences in the performance usage between Flutter and React Native?

This heading will focus on what differences in performance there is between the two applications.

Memory consumption

The testing of the React Native application shows that throughout all the tests the memory consumption of the applications starts somewhere between 206,6 MB to 224,5 MB used (see Figure 5) and quickly as the test begins the memory consumption rises slightly to around 229-236 MB used on average but sometimes it spikes upwards to 250 MB used which can be seen several times in test 4 (see Table 1, Test 4 at intervals 5, 15,20 and 25 for example).

On the Flutter application we see a similar pattern however it usually starts a lot lower consistently around 185,5 MB used (see Figure 8 & Table 4). The Flutter applications then also consumed more and more memory the further the test until it reached a plateau around 30 seconds into the test. The Flutter application increased in value just as the React Native application but did so at a slower pace and it plateaued at a lower value using at most around 225 MB to 232 MB at most.

This data tells us that the application created by the Flutter framework used less memory resources than the React Native applications throughout the entire test however the further into the test we came the closer the two applications came to each other as seen in Figure 11 showing the average memory consumption of both the applications.

CPU consumption

The data from table 2 and figure 6 are used when looking at the CPU consumption from the React Native application. We can see that the least amount used during the test was 1 % of the total available CPU of the test device and at most the React Native applications used 25 % at the given intervals.

The data from the Flutter application can be seen in table 5 and in figure 9. These results show that the Flutter application didn't use any CPU power during several instances throughout the test however it had a few slightly higher peaks at the given intervals with the highest being at 27%.

In figure 12 the average CPU consumption of both the applications can be seen and from these data the Flutter applications looks slightly better in the beginning and the middle of the test but later it used slightly more than the React Native application.

However, as the pilot test showed gathering CPU consumption data at intervals didn't show the full picture since the UI interactions might not happen at that given second. When looking at the CPU consumption the extra data of the peak CPU consumption after each interaction is a much more interesting piece of data, since this shows how stressful each interaction was for the applications.

Table 3 and figure 7 show this peak CPU consumption for the React Native application and here we can se that during the scrolling interactions (interaction 1-6 during the test) the CPU consumption was around 17-26 % of the total available CPU of the test device. Then during interaction seven and eight which was selecting and entering the weights used and amounts of repetitions into the text fields, the CPU consumption of the application always dropped to around 4-10 %. Interaction nine and ten which then was a scroll down to the end of the exercise list made the application again use upwards 19-25 % of CPU with test 8 peaking at 32% of the total available CPU used. The next two interactions then again were inputting data into the text fields for the squat exercise saw the CPU consumption drop to again around 3-18 %. The spread here is a bit bigger probably due to the values being inputted slightly faster after the scrolling ended due to the manual testing nature of the study. Lastly the last few interactions were dedicated to opening and closing a popup window which increased the CPU consumption upwards to around 16-22 % with some few tests giving a lot lower results and one giving a slightly higher result.

For the Flutter application a similar trend can be seen in table 6 and figure 10 but it is not as noticeable as the React Native results. The Flutter applications generally had higher peaks after each interaction especially during the inputting of data into the text fields where the Flutter application didn't drop down as far as the React Native application did. The Flutter application only dropped down to 17-20 % on average instead of the 4-10% that the React Native application had.

We can see this difference in the peak CPU consumption in table 9 and figure 13. As the figure 13 shows React Native overall having slightly lower peaks and especially during interactions with text fields, we can see a significant drop in CPU consumption from the React Native application.

Disc space consumption

The disc space consumption of the two applications can be seen in table 10 and figure 14. The React Native application required 122 MB of disc space after installation on the test device whilst the Flutter application requires 129 MB of disc space. There is no major difference here which makes sense since both applications are created using the same elements and the small difference, we can see is probably due to Flutter having some premade styling to their elements to make them feel more modern from the get-go. For example, when looking at figure 1 and 2 we can see that the header element has a shadow under the header bar to make it pop more and small premade styling differences like this probably make up the installation difference. This is something that could be taken into consideration when building bigger applications however styling like this is probably made by a React Native developer in the end any way since its useful ways of making the application look and feel modern so in the end the disc space required will not make too much of a difference.

The results of the findings show that the two applications created by the two frameworks (React Native & Flutter) consume a similar number of resources during the test with the Flutter application overall using less memory and CPU during the entire length of the test. The React Native application on the other hand uses less disc space on the target device and it has slightly lower peaks of CPU consumption during scrolling interactions on the application, and considerably less CPU during interactions with text fields in the application.

This leads to the performance usage of the two applications being decently similar with the Flutter application coming out ahead of the React Native application by requiring / using less memory from the test device and the React Native application having lower peak usage of

CPU power. The React Native application could also potentially have less overall CPU consumption, but the result of the performed testing is limited by the set intervals and when looking at the peak CPU consumption the React Native application has lower peaks. The React Native application also requires less disc space to be installed.

Related work

During this study the results show that Flutter and React Native uses similar amounts of resources, but Flutter was consuming slightly less than React Native overall in both the memory and CPU cases. However, Flutter did have slightly higher peak consumption after an interaction and also consumes a little more disc space than React Native. These results show similar tendencies as the results produced by IŞITAN and KOKLU (2020) since they also found the two frameworks having similar numbers, it also matches their study since in both studies Flutter just slightly outperformed React Native. The other two studies performed in 2020 by Barros et al.(2020) and Biørn-Hansen et al.(2020) also showed similar results however in their cases it was React Native consuming slightly less resources than Flutter.

There is an argument to be made based on this study's results that Flutter has improved over the years in certain areas such as its memory consumption, but more testing would have to be conducted to validate this statement.

RQ2

What are the recommendations based on the RQ2,

- What is the recommendation based on the performance results for future development?

Since both frameworks use similar amounts of resources it's hard to immediately recommend one option over the other. Instead based on these performance results the choice of which framework to use for future development comes down to what developer values the highest and what needs and requirements they have. If the developer wants an application that consumes less memory then the Flutter application is the better choice. If the developer instead values a lower peak CPU consumption the React Native application will be the better choice. The React Native application is also the better choice if the developer is creating a big application since it uses slightly less disc space at this small application there could potentially scale increasingly the bigger the application gets.

Limitations and Future work

Limitations of this study include the number of features included in the applications. Due to time constraints, and to make sure all features are tested only a few were implemented for this study. Another limitation of the study results is the interval timings when the data was gathered for the CPU and memory consumption, In the study the intervals was set to every five second to save time and limit the amount of data being gathered since it requires a lot of time to study the graphs. The number of frameworks tested is also limited by the time constraints of the thesis.

Future work would possibly expand and add more features to the application and recreate the test again to see if the same results are acquired if more complicated features is added. Looking a few years ahead a replication study could also be performed to validate if the results found in this study is still correct, or if they have changes given that new features and updates to these frameworks happen regularly. Future work could also add on to the study by creating the application described in other frameworks and testing those.

Conclusion

This study created and performed testing on two identical mobile applications created using Flutter and React Native. The testing showed small but noticeable differences in the consumptions of CPU, memory, and Disc space. The study also suggested and tested a new way to collecting data about the CPU consumption on mobile applications, by measuring the highest peak usage after each interaction. It later answered the two research questions "*What are the differences in the performance usage between Flutter and React Native?*" and *"What is the recommendation based on the performance results for future development?"*. It also discussed similarities and differences to the produced results compared to the results of previous studies and their findings.

Bibliography

The reference system used is APA 7.

Android Studio Profiler (2023) *Profile your app performance* https://developer.android.com/studio/profile

Android UI-Automator (n.d) *Write automated test cases with UI Automator.* https://developer.android.com/training/testing/other-components/ui-automator

Barros, L., Medeiros, F., Moraes, E., & Júnior, A. F. (2020). Analyzing the Performance of Apps Developed by using Cross-Platform and Native Technologies. In SEKE (pp. 186-191). https://ksiresearch.org/seke/seke20paper/paper122.pdf

Bhandari, P. (2020) *What is Quantitative Research? | Definition, Uses & Methods*. Scribbr. https://www.scribbr.com/author/pritha/page/7/

Biørn-Hansen, A., Rieger, C., Grønli, T. M., Majchrzak, T. A., & Ghinea, G. (2020). An empirical investigation of performance overhead in cross-platform mobile development frameworks. Empirical Software Engineering, 25, 2997-3040. https://link.springer.com/article/10.1007/s10664-020-09827-6

Codecadamy. (n.d) *What is and IDE?* https://www.codecademy.com/article/what-is-an-ide

Dorfer, T., Demetz, L., Huber, S.(2020) *Impact of mobile cross-platform development on CPU, memory and battery of mobile devices when using common mobile app features.* ScienceDirect. https://www.sciencedirect.com/science/article/pii/S1877050920317099

Flutter official documentation (n.d) *FAQ* https://docs.flutter.dev/resources/faq

Ford, S. (n.d) The Dart Language: When Java and C# Aren't Sharp Enough. https://www.toptal.com/dart/dartlang-guide-for-csharp-java-devs

Huber, S & Demetz, L.(2019) *Performance analysis of mobile cross-platform development approaches based on typical ui interactions.* ResearchGate. https://www.researchgate.net/publication/335065311_Performance_Analysis_of_Mobile_Cross-platform_Development_Approaches_based_on_Typical_UI_Interactions

IŞITAN, M., & KOKLU, M. (2020). *Comparison and evaluation of cross platform mobile application development tools*. International Journal of Applied Mathematics Electronics and Computers, 8(4), 273-281. https://dergipark.org.tr/en/download/article-file/1419794

Palmieri, M., Singh, I., Cicchetti, A. (2012) *Comparison of cross-platform mobile development tools.* 16th International Conference on Intelligence in Next Generation Networks, 179-186: IEEE. https://ieeexplore.ieee.org/document/6376023?fbclid=IwAR3IgVTFRHrEEjckjG99WlS8UEuy0UGx_GWJpyebTAiBT4DUNklHeZaep5g

Patel, R., Davidson, B. (2011) *Forskings-metodikens grunder.* (4th ed.). Studentlitteratur.

React Native (n.d) *React Native Learn once, write anywhere.* https://reactnative.dev/

Vailshery, Ls. (2022) *Cross-platform mobile frameworks used by developers worldwide 2019-2021.* Statista. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/

Visual Studio Code (n.d) https://code.visualstudio.com/

Walk, K. (1998) *How to Write a Comparative Analysis.* Harvard University. https://writingcenter.fas.harvard.edu/pages/how-write-comparative-analysis

Williams, T. (2021) *Why is Quantitative Research Important.* GCU. https://www.gcu.edu/blog/doctoral-journey/why-quantitative-research-important

Willocx, M., Vossaert, J., Naessens, V.(2016) *Comparing performance parameters of mobile app development strategies. https://dl.acm.org/doi/pdf/10.1145/2897073.2897092*

Appendix 1

Pilot Test

To perform the pilot test there were a few different applications that needed to be installed and some setting changes on the test device (the mobile phone) was required.

Applications that needed installation:

1. Flutter
2. Android Studio
3. Visual Studio Code
4. Android Studio Profiler

Along with the installation of these applications I also needed to start developer mode on my mobile device which is made through accessing a hidden menu in your mobiles system settings. To access this menu, you need to find the serial number of your device in the settings, usually under a “About phone” heading. Then you need to tap the serial number 7 times to activate the developer mode on your device. After this is activated, I headed over to advanced settings, developer options and enabled the USB debugging on the device to be able to use the phone to test the applications instead of using an emulator (mention pros and cons of emulator vs real phone).

Pilot Test Results:

Total test time: 28 sec.
Peak memory usage: 219,1 MB at 11,581 sec into test.
Peak CPU usage: 19% at 11,222 sec into test.
Storage usage of application: 123 MB.

Table 11 Memory consumption during pilot test

Interval (Seconds)	1	5	10	15	20
Memory Usage (MegaByte)	185,7	197	207,9	209,1	204,4

Table 12 CPU consumption during pilot test

Interval (Seconds)	1	5	10	15	20
CPU Usage (Percent of total available)	0%	0%	15%	14%	0%

The results show that if an interaction doesn’t happen at the given intervals or close to a given interval the CPU usage will fall to a very low usage sometimes even zero as seen in table 12. This could lead to results from the intervals not showing the full picture of the CPU

consumption. As a solution to this problem a table will be added during the full tests showing the peak CPU consumption after each interaction during the test.

Appendix 2

Test Results Flutter

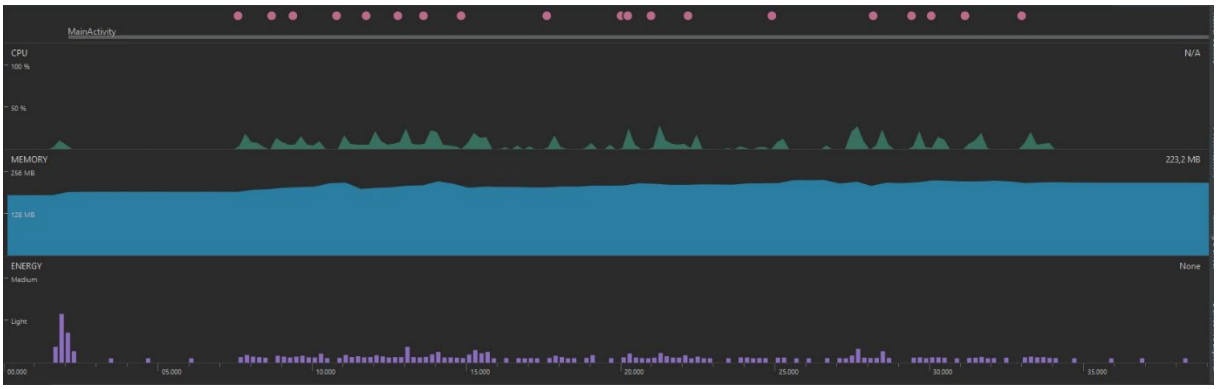

Figure 15 CPU and memory consumption from Flutter test 1

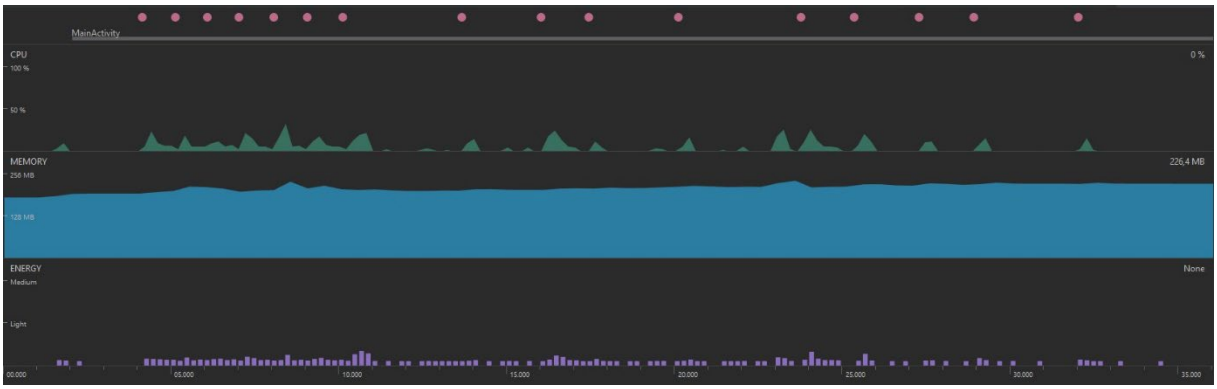

Figure 16 CPU and memory consumption from Flutter test 2

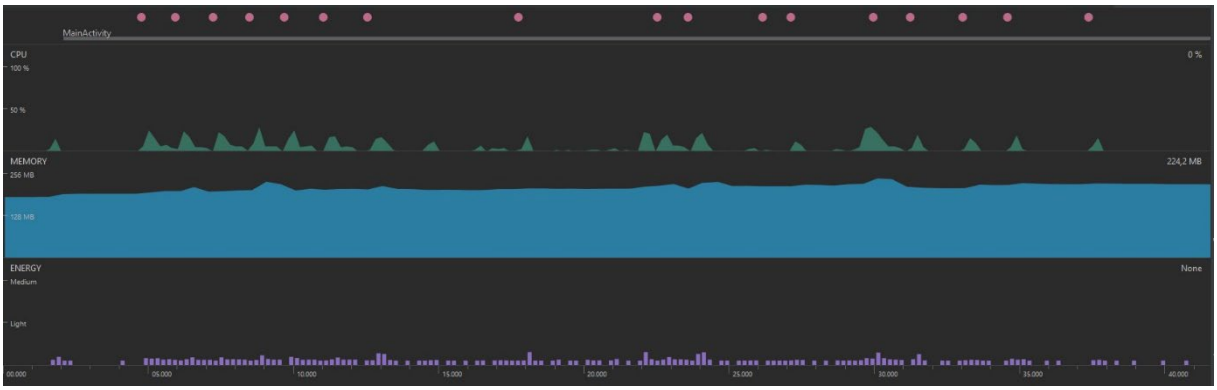

Figure 17 CPU and memory consumption from Flutter test 3

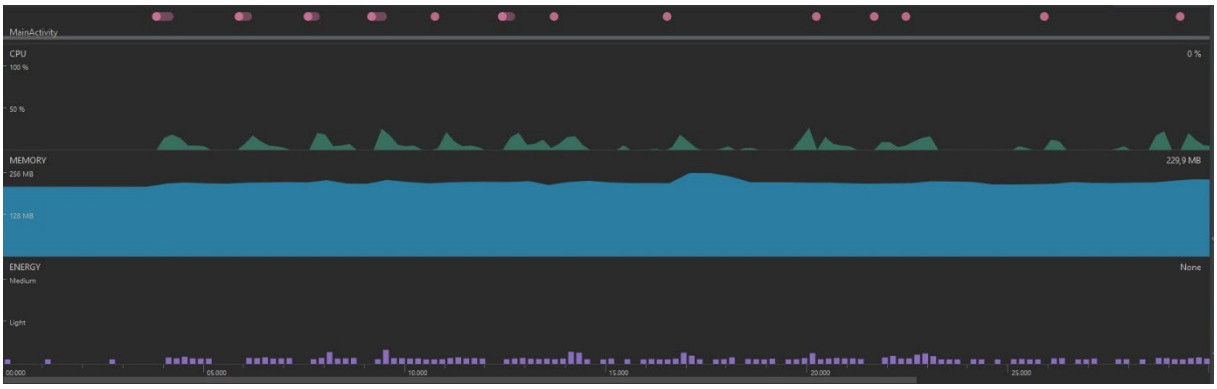

Figure 18 CPU and memory consumption from Flutter test 4

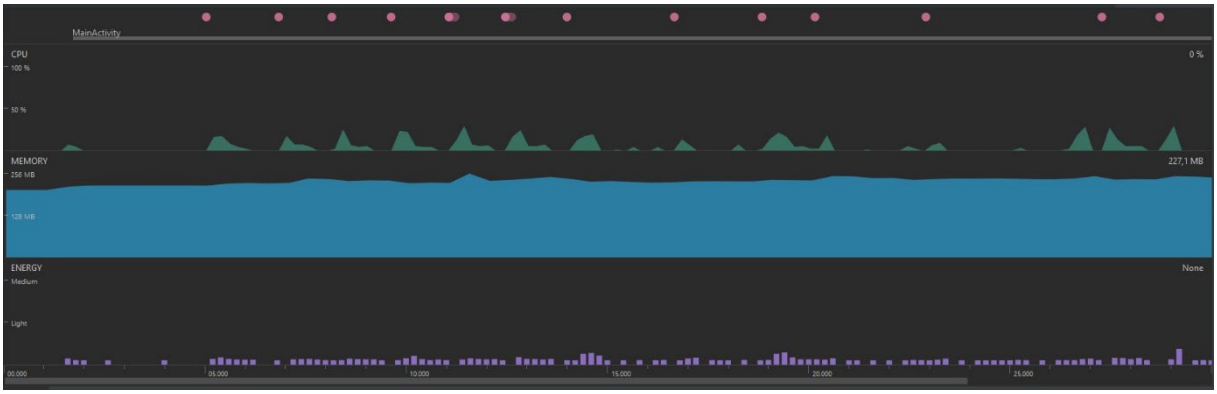

Figure 19 CPU and memory consumption from Flutter test 5

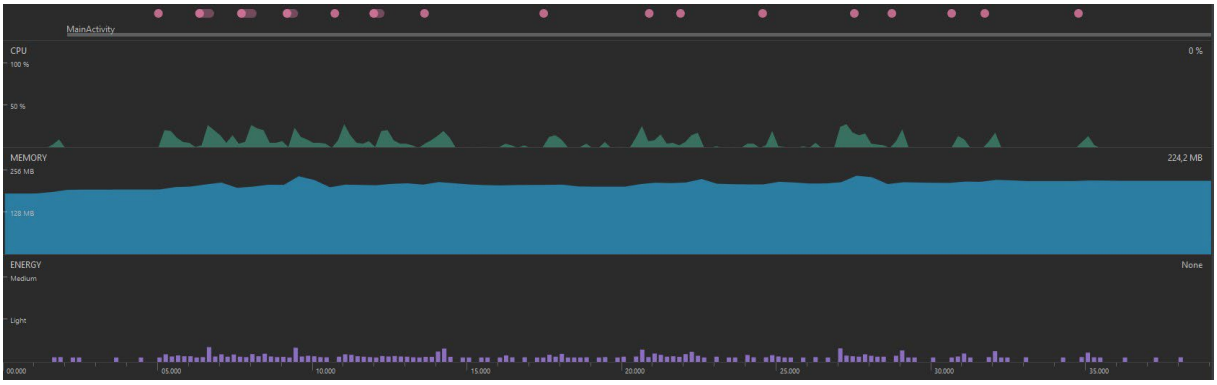

Figure 20 CPU and memory consumption from Flutter test 6

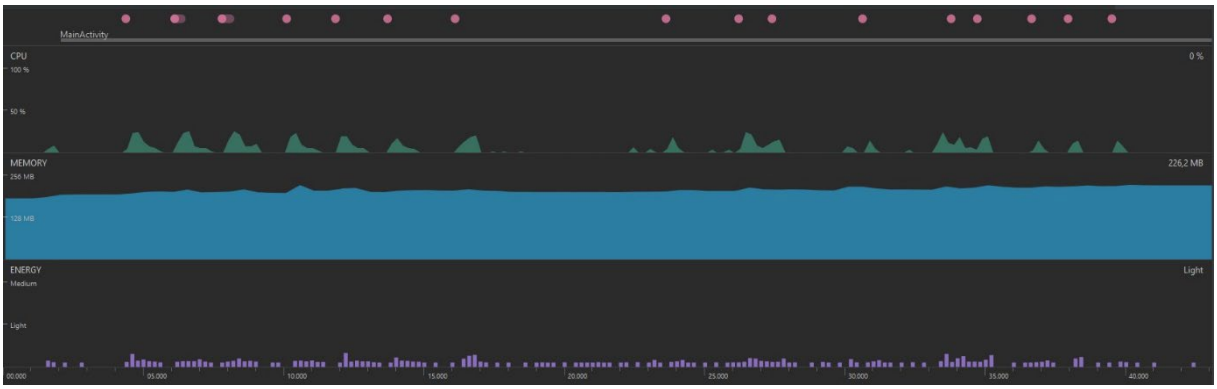

Figure 21 CPU and memory consumption from Flutter test 7

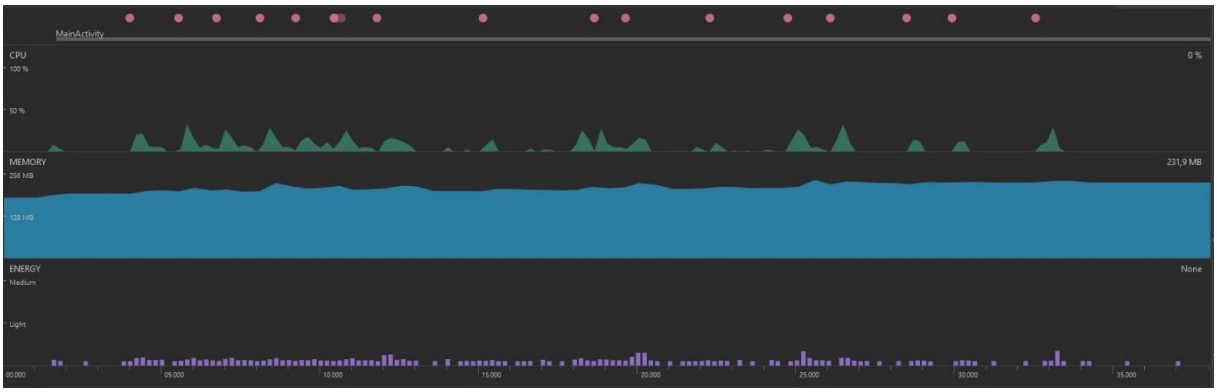

Figure 22 CPU and memory consumption from Flutter test 8

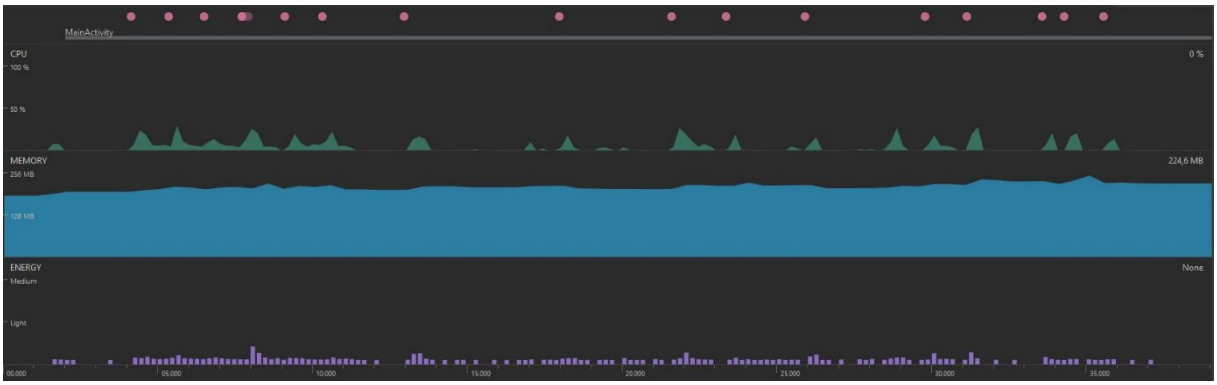

Figure 23 CPU and memory consumption from Flutter test 9

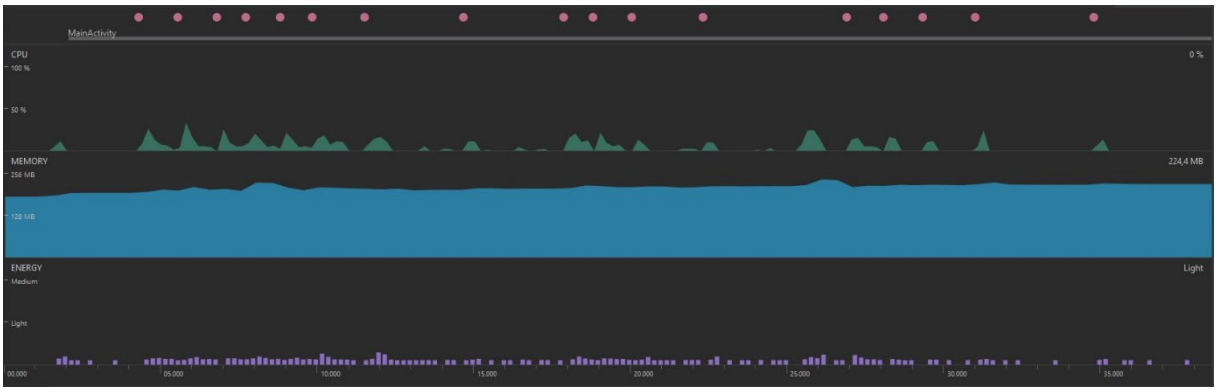

Figure 24 CPU and memory consumption from Flutter test 10

Test Results React Native

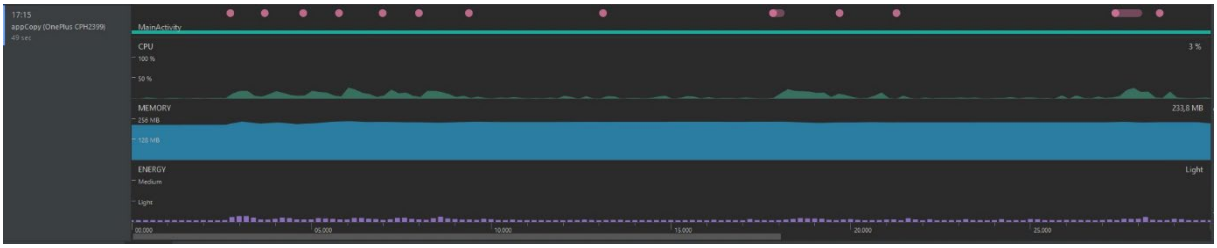

Figure 25 CPU and memory consumption from React Native test 1

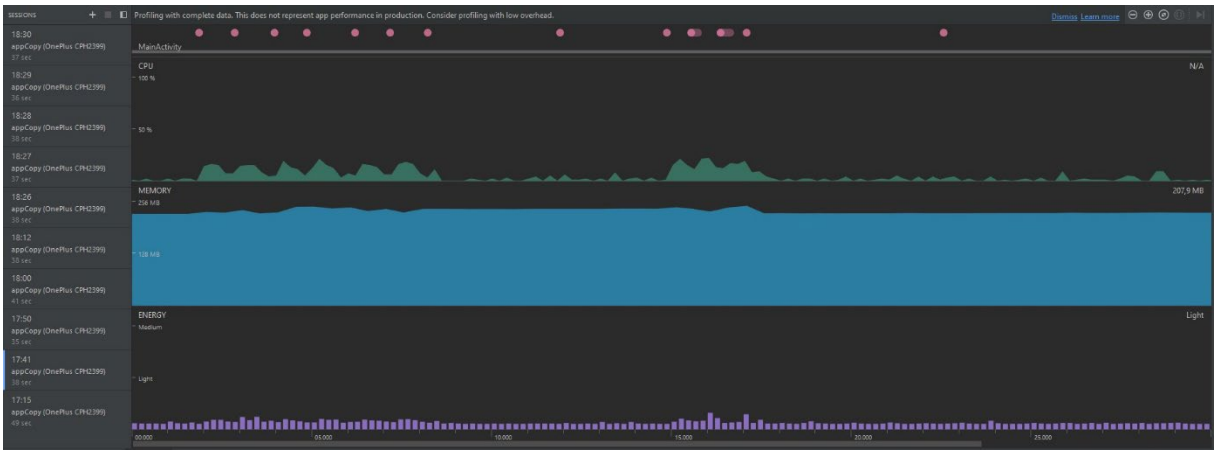

Figure 26 CPU and memory consumption from React Native test 2

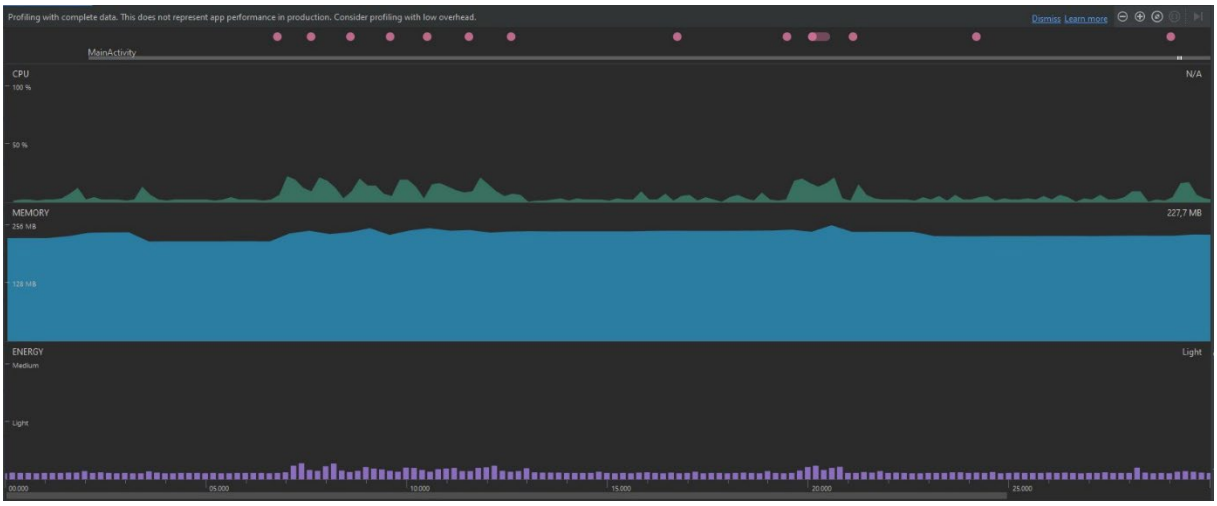

Figure 27 CPU and memory consumption from React Native test 3

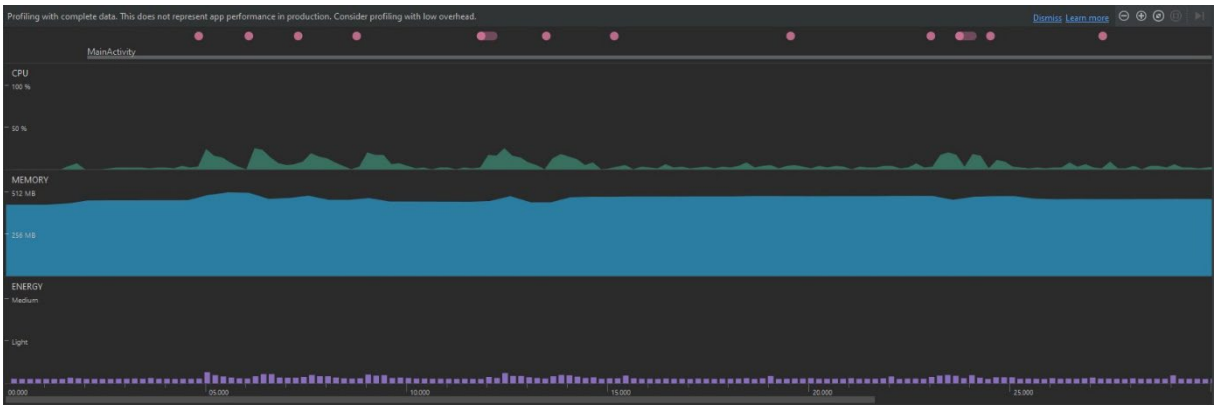

Figure 28 CPU and memory consumption from React Native test 4

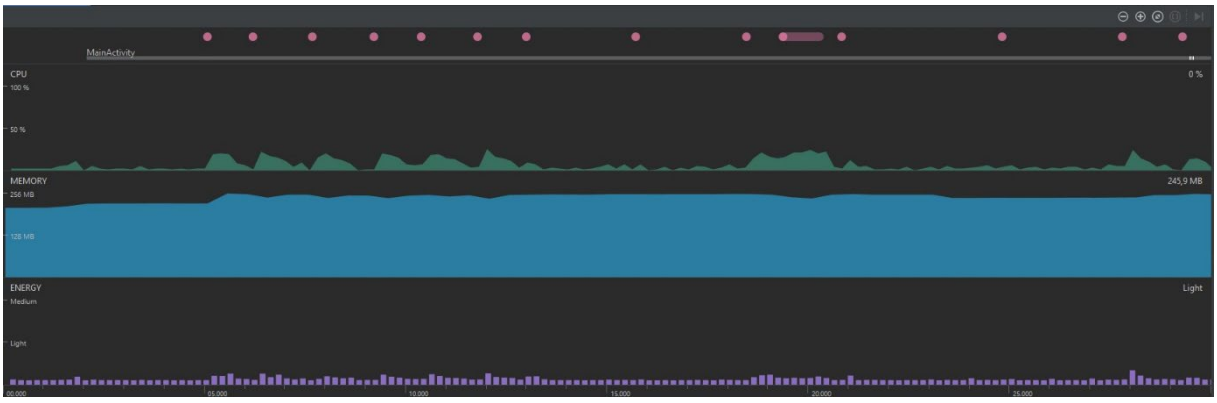

Figure 29 CPU and memory consumption from React Native test 5

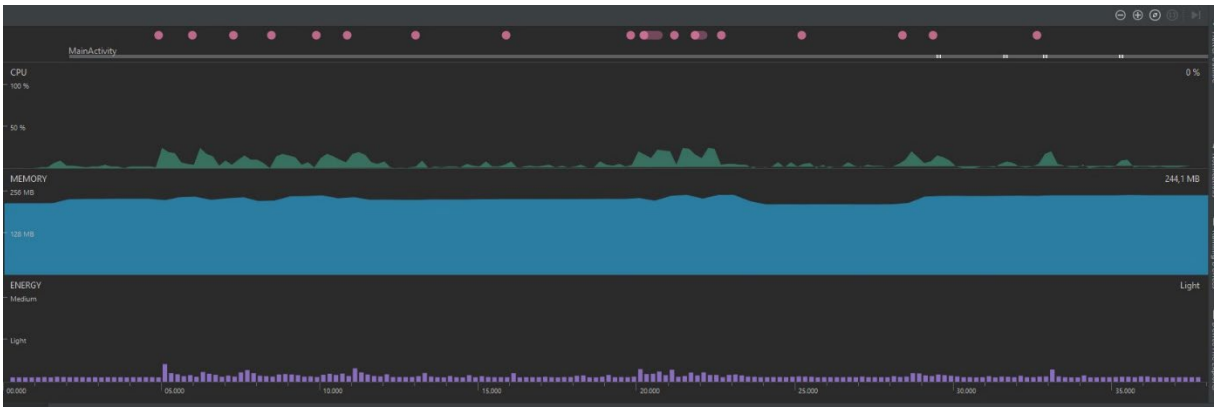

Figure 30 CPU and memory consumption from React Native test 6

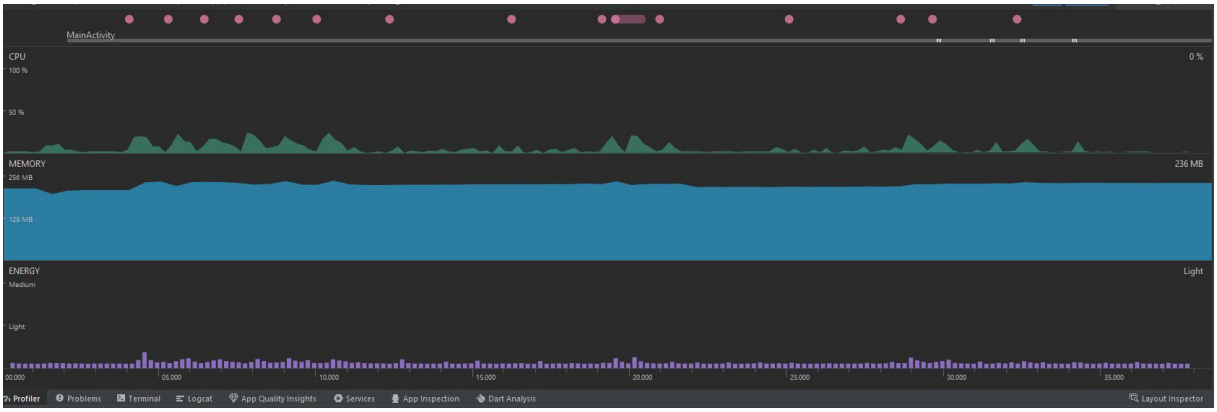

Figure 31 CPU and memory consumption from React Native test 7

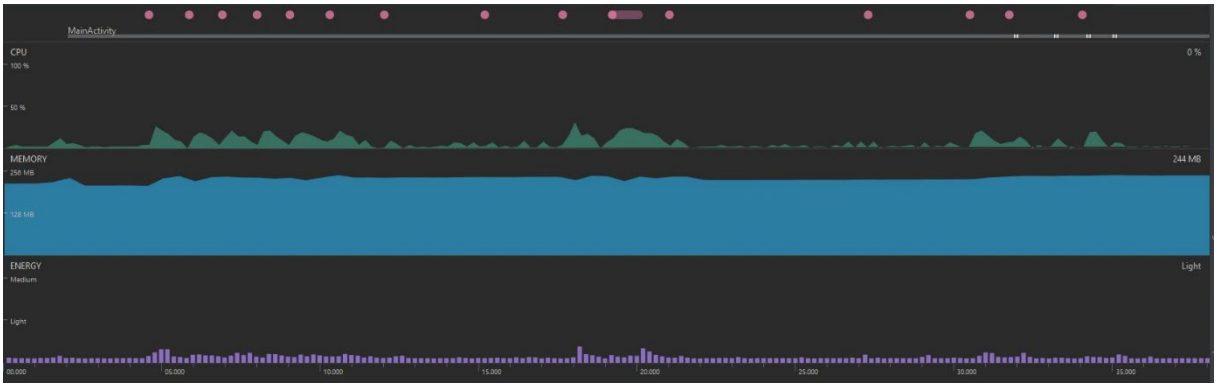

Figure 32 CPU and memory consumption from React Native test 8

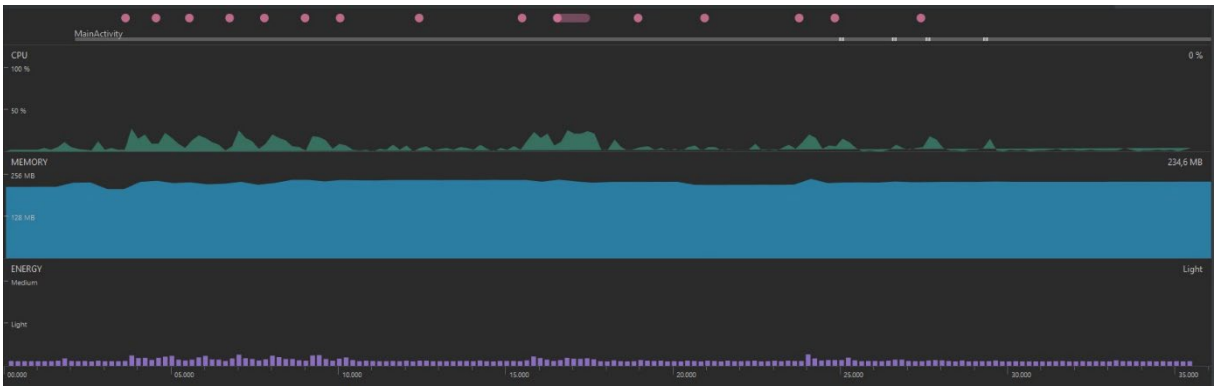

Figure 33 CPU and memory consumption from React Native test 9

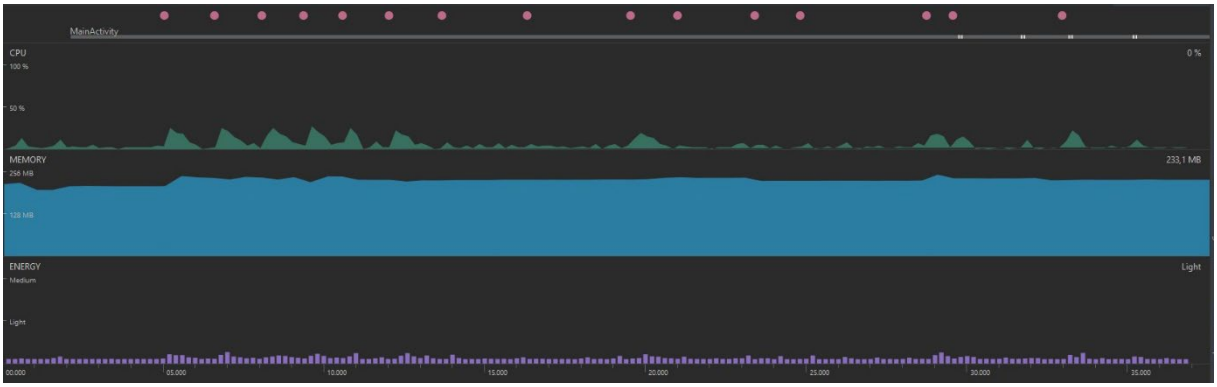

Figure 34 CPU and memory consumption from React Native test 10