PERFORMANCE BREAKDOWN OF REACT NATIVE FRAMEWORK COMPARE TO NATIVE APPLICATIONS

Nilesh Kadam¹, Nirupma Singh²
Ajeenkya DY Patil University¹,²

Abstract

Creating a mobile application often requires the developers to create one for Android and one for iOS, the two leading operating systems for mobile devices. The two applications may have the same layout and logic but several components of the user interface (UI) will differ and the applications themselves need to be developed in two different languages. This process is gruesome since it is time consuming to create two applications and it requires two different sets of knowledge. There have been attempts to create techniques, services or frameworks in order to solve this problem but these hybrids have not been able to provide a native feeling of the resulting applications. This study evaluate the React Native framework which helps developers to build cross platform application on a single code base with native feeling.

Keywords: React Native, Mobile Development, Cross Platform development, Performance Management.

INTRODUCTION:

Since the release of the first smart phone, the use and demand of mobile applications has increased rapidly. Many businesses have been established by providing a service through an application but some businesses want an application in order to prove that they are contemporary or because their competitors have one. This has led to a huge amount of applications being created but there is one significant problem, the app needs to be supported by both Android and iOS. Even though the application itself is the same, the developers still need to develop two applications which requires unnecessary time and skills. In the summer of 2015, Facebook released a framework called React Native which is used to build an application in React and then compile it to either Android or iOS. The aim of this research is to evaluate the React Native framework and how well the applications created by the framework correlates to natives. By developing an application in React Native, the research will evaluate the development potentiality and the simplicity for beginners to create their own app. Moreover, this paper will analyse the performance of a React Native application by comparing it to a similar one in Android. The result of these two aspects will result in a solid and fair conclusion whether or not React Native is a framework worth investing in or if it is not able to replace writing application in the native approach.
React Native Market Demand

Nowadays, new start-up companies along with big companies are switching to React Native. Companies like Uber, Netflix, Instagram, Airbnb are all using React Native providing big market share. Main reason for companies to switch their application to React Native is because it reduces their cost. Everything coming online nowadays makes companies to build app for their services but for start-up companies it is really hard to manage cost for hiring developers with two different skill set so by building their application in React Native reduces their cost. According to 2018 survey, use of React Native increase to 67%. Over the past two years, Airbnb have built an incredibly strong integration into their apps to enable complex native features such as shared element transitions, parallax, and geofencing as well as bridges to their existing native infrastructure such as networking, experimentation, and internationalization. They have launched a number of critical products for Airbnb using React Native. React Native enabled them to launch Experiences, an entirely new business for Airbnb, as well as dozens of other features from reviews to gift cards. Many of these features were built at a time where they simply did not have

PROBLEM STATEMENT:

While React Native promises performant applications, there is still some overhead involved compared to truly native applications. The application’s code runs inside of a JavaScript virtual machine which handles all the app logic, orchestrates the UI, makes simple computations, etc. After all of this has been done, the results need to be sent over to the native thread, where the UI can finally be rendered with new results. Time spent doing this work adds up, and could potentially have been done faster in native code. Most mobile phones today run their display at a refresh rate of 60 Hz, which also sets a target for how often applications should render a new frame in order to appear responsive and smooth. Even though a React Native application may be less performant than a truly native application, it won’t matter appearance-wise as long as it manages to hit 60 frames per second (FPS). Another important factor is latency, or the time it takes from user input until the device has responded in some way. In the history of the computer science field, many studies have been conducted on the impact of latency in user interfaces. It is important that latency stays as low as possible, or the user will be distracted from their task at hand and may get frustrated. For React Native this means that there is not much room for additional latency as compared to the native platform, and ideally a React Native application should respond as quickly as a native application. Naturally there are other factors to performance than the framerate and latency of an application. Other problems include battery life drain, cpu usage, memory usage and disk
space requirements. In this study we will be focusing on latency, framerate and tasks that cause the framerate to drop below 60 FPS. Dropping below 60 FPS causes extra latency, which means that the user will experience a slowdown and the app will appear unresponsive. Not much research exists on React Native’s performance specifically yet. This study aims to shed some light on the subject by testing and comparing common UI components in mobile apps between truly native apps and React Native apps. We measure how long these components take to render as well as how long they take to react to user input both in native and React Native apps. We also compare various React Native components to see which is most suitable for a certain situation. Common components to be tested include basic text views, lists of data, interaction with basic mobile UI components such as buttons and scrollable views.

**PERFORMANCE:**

By using the profiling tool Trepn Profiler the two applications could have different aspects of performance tested. The tests were done by using the same mobile device in order to establish that the results of the two applications did not vary due to the device. Furthermore, since the application does not require any internet connection, the mobile device was put in flight-mode which would remove any results being influenced by the traffic at the given time. In order to furthermore minimise surrounding influences, the test were performed every other time. First a test were done using the Android application and after that the React Native application were tested and so forth. Lastly, before every tests, the application had its non-initial data removed and was terminated. There are many parts of an application which can be analysed when measuring performance but this paper focused on the following data points:

1] GPU frequency which displays a value in Megahertz (MHz) and is a measurement of the speed at which the GPU operates (GPU is the unit which performs the graphical calculations and handled signal from the phone to the screen and contrariwise).
2] CPU load which provides a percentile value of how much of the CPU (which is the unit that executes programs) is used by the application.
3] Memory usage which returns how many Megabytes (MB) is required by the application.
4] Battery Power which provides a value in milliwatts (mW) of how much power is drained by the profiled application.

Performance breakdown of React Native Framework compare to Native Applications.

Furthermore, there are three cases of the application which was tested for performance. The first test was performed when the application was started and as idle for 30 seconds. Thereafter there was two user cases done while measuring the performance where the application had already been started and the user navigated through it. The first case was when the user navigated to Budgets and created a new category. The category received the type value "Test" and the value 500. When the category was created, the user was redirected to the lists of budgets. Thereafter the user edited the budget and removed it. When the budget was removed and the list of budgets was updated, the test was done and the profiling was
stopped. The screenshots with the values from the user road map of the first user case can be seen in figure 1.

![Graph of GPU frequency over time](image)

**Figure 1.1: Road map of first user case**

![Screenshot of another user case](image)

**Figure 1.2: Road map of second user case**
The second user case was when the user navigated to Transactions and created a new expense. Thenceforth the user typed in details for the new expense and took a photo which was attached as a receipt. The user consequently saved the transaction and the test was done as the profiling was stopped. The screenshots with the values from this user road map can be seen in figure 1.2

RESULT:
First test
The first tests when the applications were idle was carried out for 30 seconds and three runs for each application was captured. The average of the three sets of data were calculated and are presented below in form of graphs. Firstly as can be seen in figure 2.1, the GPU frequency dropped significantly when the application was started and displayed for the user. When the application was switched out and run in the background, the frequency incremented and the frequency of the Android application rose to roughly 340 MHz before declining to 320 MHz along with the React Native application. This is the frequency measured throughout the rest of the test, except a small dip for the Android application after about 4.5 seconds. The average GPU frequency of the Android application was 315.78 MHz and respectively 315.58 MHz for the React Native application.

Figure 2.1: GPU frequency of idle applications

The graph of the CPU load in figure 2.2 is somewhat more difficult to read due to the jagged results but it can be noted that both applications had to a high degree the same CPU load except the two distinctive spikes after 7.6 and 7.8 seconds for the React Native application. The average value for Android was 57.60 % and 57.89 % for the React Native application.

Figure 2.2: CPU load of idle applications

Performance breakdown of react native framework compare to native applications.
Performance breakdown of React Native Framework compare to Native Applications.
Furthermore, the memory usage of the applications had the same pattern with an increase of usage at the initialisation of the applications but with a steady decline afterwards. The clear difference is the somewhat lower usage of the Android application throughout which required a memory of 1607 Megabytes while the React Native application had a memory usage of 1616 MB in average.

![Figure 2.3: Memory usage of idle applications](image)

Lastly, the power consumption also had the same pattern of the applications. The consumption was immense at the beginning when the applications were started but descended over time. Noteworthy are the accretions of the React Native application after 1 second and the Android application after 4.5 seconds. The average power consumption was 1362 milliwatts and 1350 milliwatts for the Android application and the React Native application.

![Figure 2.4: Power consumption of idle applications](image)

**Second test**

The second test is described in above section and consists of the user adding a new budget and subsequently deleting it. In order to perform these actions, the tests took roughly less than 12 seconds for both applications to execute. The main actions performed during the test are presented below.
The GPU frequency of the applications when handling budgets are displayed below in figure 2.6. When the user navigated to the list of budgets, the frequency dropped to 200 MHz for the Android application and 240 MHz for the React Native application. The frequency once again dropped when the scene for adding a budget was rendered after approximately 3 seconds. Afterwards the Android application had a stable frequency at 320 MHz while the frequency of the React Native application had some drops. Subsequently, when the user saved a budget and was directed to the list of budgets the frequency once again dropped and continued to increase and decrease throughout the rest of the test as editing a budget and removing a budget was done, resulting in the previous scenes re-rendering with a few minor changes. The average frequency was 291 MHz for both the Android and the React Native application.

Performance breakdown of React Native Framework compare to Native Applications

![GPU frequency](image)

**Figure 2.6: GPU frequency when handling budgets**

The load of the CPU for the Android and the React Native application are presented in figure 2.7. As the earlier test, the graph is uneven but shows a high CPU load at the beginning of the test with an enormous drop after the list of budgets are displayed and the button for adding a new budget has not yet been pressed. Subsequently the load declined while the user was editing the values for the fields but increased when the user saved, edited and deleted that budget. Furthermore, the graph displays the CPU load of the Android application being slightly less than the React Native one which is also confirmed with the Android application having an average load of 36.14 % and 41.64 % for the application created in React Native.
The memory usage in the second test which can be seen below in figure 2.8 was similar to the first test as both applications had a similar usage throughout the whole evaluation. The most distinctive differences are the increase of the usage for the React Native application at the beginning of the test and the overall greater usage with an average memory usage of 1620 Megabytes for the Android application and 1672 Megabytes for the React Native one.

The power consumption differed from the first test regarding the initial drainage since it had to be initialised in the first test while in the second test the application was already running when the measurements began. The graph in figure 2.9 displays increases of power consumption at key events and performed actions. Initially the drainage increased when the list of budgets were rendered and after 3 seconds when a budget was to be added. Subsequently, drops and inclines of consumption was registered and when the budgets as added after 7 seconds the drainage of the Android application was very high when the new budget was saved and the list of budgets was displayed. Finally, a higher drainage can be viewed at the end of the test where the React Native application had a spike at 10.5 seconds as the scene for editing a budget was displayed and the Android application had a higher consumption after 11 seconds as the budget was deleted. To summarise, the average power consumption of the Android application was 1749 milliwatts while the React Native application had a consumption of 1803 milliwatts.
CONCLUSION:

React Native is a new and interesting framework which have been praised by developers since it might be the technique which they have wished for and might change the way we create applications. This research have evaluated the framework by focusing on the Android compilation and compared the performance of two applications and to a small extent the development process when developing in React Native.

The two applications were performance tested in regards to GPU frequency, CPU load, memory usage and power consumption. Three different cases were tested and all measurements concluded in the same result, a React Native application does not have as good performance as a native Android application. However, the differences in the tests were small and the React Native application was able to challenge the Android application in a notable way.

REFERENCES:


