Análise de Desempenho de Aplicações Móveis Nativas, Cross-Compiladas, Híbridas e Interpretadas

Performance analysis of native, cross-compiled, hybrid and interpreted mobile applications

Breno de Castro Nogueira *

Resumo

O desenvolvimento de aplicações móveis está se tornando cada vez mais importante devido a popularidade dos dispositivos móveis. As abordagens de desenvolvimento são nativa, híbrida, interpretada e cross-compilada. Nesse trabalho, nós selecionamos as ferramentas de desenvolvimento mais populares para cada abordagem (Java, Ionic, React Native e Flutter, respectivamente) e as utilizamos para desenvolver um grupo de aplicativos. Nós medimos o desempenho dos aplicativos em relação ao consumo de CPU e RAM, tempo de execução e tamanho do aplicativo. Nossos resultados indicam que, nem sempre, o desenvolvimento nativo tem o melhor desempenho. Em muitos casos, outras abordagens podem chegar perto do desempenho nativo.

Palavras-chave: desenvolvimento móvel, ferramentas cross-platform, análise de desempenho.

Abstract

Mobile application development is becoming increasingly important due to the popularity of mobile devices. The most common mobile application development approaches are native, hybrid, interpreted and cross-compiled. In this work, we select the most popular development tool for each approach (namely Java, Ionic, React Native and Flutter, respectively) and use them to develop a group of applications. We measure applications performance with respect to CPU and RAM consumption, execution time and application size. Our results indicate that native development does not always perform best. In many cases, other approaches may come close to native performance.

Keywords: mobile development, cross-platform tools, performance analysis.

Aprovado em: 15/07/2019 Versão Final em: 18/07/2019

*Trabalho de conclusão de curso apresentado ao curso de Bacharelado em Sistemas de Informação da Universidade Federal Fluminense como requisito parcial para conclusão do curso. Graduando do Curso de Sistemas de Informação-UFF; bcnogueira@id.uff.br
1 INTRODUCTION

Nowadays the number of mobile device users has exceeded the number of computer users. As reported by Google (GOOGLE, 2016), 67% of its users access their account from computers, and more than 80% of its users access their account from mobile devices (smartphones and tablets), of which 27% use only smartphones. Android and iOS (NALDI, 2016) are the main operating systems of these devices, with Android having around 80% of market share, while iOS has almost 20% of market share.

With this growing number of mobile device users, the demand for mobile applications is constantly increasing. Each mobile device operating system offers different native development tools that vary in code base. For example, Android uses Java or Kotlin, while iOS uses Objective-C or Swift. Developing native applications for different operating systems with different programming languages brings many problems from a business point of view, like increased cost and time to develop new applications (HEITKÖTTER; HANSCHKE; MAJCHRZAK, 2013; FERREIRA et al., 2018).

A recent trend is cross-platform development, which allows developers to develop only one application that runs on different operating systems, hence reducing time and cost. Cross-platform development tools must provide native-like user interface, good performance, security, low power consumption and access to built-in device features (HEITKÖTTER; HANSCHKE; MAJCHRZAK, 2013; Dalmasso et al., 2013). There are many available options for cross-platform development tools with different approaches (IONIC, 2019a; NATIVE, 2019; FLUTTER, 2019; PHONEGAP, 2019; XAMARIN, 2019; TITANUM, 2019).

Cross-platform development approaches have been categorized in many ways. In this work, we consider a four category approach (THAKARE et al., 2014; HEITKÖTTER; HANSCHKE; MAJCHRZAK, 2013): (i) Web approach — web applications are remote web pages displayed on mobile device browsers with native-like UI Components; (ii) Hybrid approach — hybrid applications combine the advantages of web and native applications, with all web code locally stored and API support to access low-level
hardware resources; (iii) *Interpreted approach* — interpreted applications have a built-in virtual machine; and (iv) *Cross-compiled approach* — cross-compiled applications consist of pure native code, just like a native application. For further explanation, please refer to Section 4.1.

Figures 1, 2 and 3 depict the popularity of cross-platform development frameworks on Google Trends for each category. As can be seen, Ionic, React Native and Flutter are the most popular frameworks that generate hybrid, interpreted and cross-compiled applications, respectively. We ignore web applications in this comparison because the application is simply a web browser that fetches a web page from a remote server and displays the content. This fits more into web development for mobile devices than into mobile application development, as there is no mobile application development tool in this category.

![Figure 1: Popularity of hybrid frameworks on Google Trends.](image)

Source: Google Trends, 2019.

So many options of development tools bring concern about performance, an important aspect in mobile application development projects when deciding upon a development tool. Intuitively, native applications should outperform other approaches. However, this is not necessarily true due to the maturity of each development tool, particularly in such a competitive market. Besides, application performance analysis is a difficult multivariate problem that depends on application features, evaluation metrics and development tools and approaches.
Performance analysis of mobile applications has been extensively studied in the literature (GUERRA, 2018; FERREIRA et al., 2018; GRZMIL et al., 2017; AHTI; HYRYNSALMI; NEVALAINEN, 2016; JIANG, 2016; Willocx; Vossaert; Naessens, 2015), mainly comparing between native development tools with cross-platform development tools (Willocx; Vossaert; Naessens, 2015; AHTI; HYRYNSALMI; NEVALAINEN, 2016; GRZMIL et al., 2017). However, cross-platform development is a heterogeneous category, and to the best of our knowledge, all performance studies so far assumed outdated taxonomies that do not distinguish between interpreted and hybrid approaches (JIANG, 2016). This leads to an incomplete set of comparisons that sometimes include tools within the same category (FERREIRA et al., 2018; GUERRA,
In this paper, we evaluate the performance of native, cross-compiled, interpreted and hybrid mobile applications for the most popular development tools in the market. For that, we develop a group of four applications using Java, Flutter, React Native and Ionic. To develop these applications we had little or virtually no knowledge depending on the tool. These applications take into account important commonly used features: file reading, file writing, file download and CPU intensive algorithm. We use CPU and RAM usage and time to completion as comparison metrics. Our target platform is Android due to its biggest market share and ease of obtaining and preparing development and testing environments. Other operating systems have obstacles, such as iOS and its need for specific Apple hardware.

This performance evaluation is relevant for many reasons. Our results show that the common intuition that native applications always perform better in all aspects is false. Mobile application performance has many facets and depends on development approach and tool, evaluation metric and application features. This study provides, thus, for wiser development tool choice in mobile application development projects from a performance point of view.

The rest of this document is structured as follows. Section 2 presents the related works. Section 3 contains explanation about each mobile application development approach. Section 4 describes the evaluation methodology used in this paper. Section 5 shows the results of the experiments and contains a discussion about it. Section 6 gathers the conclusions.

2 RELATED WORK

There are many work addressing cross-platform theme. Some approach development tools taxonomy (THAKARE et al., 2014; EL-KASSAS et al., 2017). Others evaluate development aspects of each tool (MAZARICO; CARRERA, 2015; ANGULO; FERRE, 2014; HEITKÖTTER; HANSCHKE; MAJCHRZAK, 2013). And finally, others eval-
uate applications performance (GUERRA, 2018; FERREIRA et al., 2018; GRZMIL et al., 2017; AHTI; HYRYNSALMI; NEVALAINEN, 2016). We discuss them through the rest of this section with particular focus on application performance, main subject of this work.

2.1 Taxonomy

Thakare (THAKARE et al., 2014) categorize cross-platform development frameworks into web, hybrid, interpreted and cross-compiled. El-Kassas, Abdullah, Yousef and Wahba (EL-KASSAS et al., 2017) go further and separate into six categories based on framework model and architecture.

2.2 Development Aspect

Mazarico and Carrera (MAZARICO; CARRERA, 2015) developed an application with PhoneGap and Xamarin evaluating the implementation process and conducted a research with a test group to compare the resulting applications. Angulo and Ferre (ANGULO; FERRE, 2014) describe a study where two independent teams develop two different versions of a mobile application, one using native development and that other using with Titanium, focusing on user experience analysis. They concluded that a good level of user experience can be achieved if the cross-platform framework is chosen carefully. Heitkötter, Hanschke and Majchrzak in (HEITKÖTTER; HANSCHKE; MA-JCHRZAK, 2013) compiled a set of criteria to compare cross-platform development approaches from a development perspective, concluding that cross-platform is a good option due to its low entry barriers.

2.3 Application Performance

Grzmił, Skublewska-Paszkowska, Lukasik and Smolka (GRZMIL et al., 2017) analyzed the performance of an application developed with Xamarin and Java/Swift. They analyzed the time each application took to execute selected tasks. Those tasks were computing performance, file access, image downloading and geolocation. Applications
developed with native tools achieved better performance. However, in some cases Xamarin allows for significant development time reduction without deterioration in user experience.

Ahti, Hyrynsalmi and Nevalainen (AHTI; HYRYNSALMI; NEVALAINEN, 2016) presented an evaluation framework to compare different cross-platform development frameworks. This evaluation framework is validated comparing PhoneGap with native development tools for Android and Windows Phone. The results revealed that the evaluation framework was able to capture differences between the various development tools. They concluded that PhoneGap has weak user experience and interface.

Jiang (JIANG, 2016) evaluated the performance of native, cross-compiled and hybrid applications developed by a junior programmer for iOS and Android using Swift/Java, Xamarin and Cordova. From an implementation process point of view, he considered usability, accuracy, expandability, integration, power consumption and RAM usage (regarding the device in which the development occurred). Runtime metrics are performance (start up time and execution time for each functionality), security, maintainability and battery consumption. Jiang concluded that there is not a big difference in terms of development processes and products between native, cross-platform and hybrid approaches. As expected, native approach development time is doubled comparing to cross-compiled and hybrid approaches, but in terms of performance native had a better result, consistent with what Grzmil, Skublewska-Paszkowska, Lukasik and Smolka also concluded.

Willocx, Vossaert and Naessens (Willocx; Vossaert; Naessens, 2015) present a quantitative performance analysis of an application developed with both native Android and iOS and two cross-platform development tools, Ionic and Xamarin. Their goal was to compare cross-platform development tools against native development. They measured launch time, pause and resume time, time to open another page of the application, RAM consumption, CPU usage and disk space. All tests were conducted on low-end and high-end devices. They concluded that both cross-platforms introduce a performance penalty. However, the additional overhead can be acceptable from the
user’s perspective, especially when using high-end devices. Like Guerra, they also point out that the decision of which framework to use can be driven by behavioural aspects, for example, one tool may be suitable for CPU intensive application but not a good choice when advanced graphical user interface design is required.

Guerra (GUERRA, 2018) compares different cross-platform development tools used for mobile development, namely React Native, Ionic, Weex and Flutter. For each development tool, he measured code quality and execution times, concluding that there is not one tool that excels in every aspect. Moreover, some cross-platform development tools require knowledge of native programming in order to achieve full performance.

Ferreira, Peixoto, Duarte, Torres, Júnior, Rocha and Viana (FERREIRA et al., 2018) developed an application with camera usage, image processing, access to accelerometer and GPS and communication with Web services. He developed one version for each of the following development tools: Java, Swift/Objective-C, PhoneGap, Sencha Touch and Titanium. He concluded that some resources and functionalities could not be implemented in all versions, Titanium being the framework that presented more difficulties in the development of complex resources. GPS, accelerometer and camera features were developed in all versions. Applications developed with cross-platform frameworks consumed almost as much RAM as those developed with native tools, but lost in execution time.

Except for Ferreira (FERREIRA et al., 2018) and Guerra (GUERRA, 2018), all other work on performance evaluation ignored interpreted applications in their analyses. However, Ferreira ignored cross-compiled applications, and Guerra ignored native applications. Moreover, Ferreira focuses mainly on development tools capability to access low-level device resources, not exploring different application features such as intensive CPU algorithms and file operations. In this work, we distinguish among native, cross-compiled, hybrid and native applications and conduct an extensive performance evaluation that covers many application features.
3 MOBILE APPLICATION DEVELOPMENT APPROACHES

There are different approaches in mobile development. Among them there are 5 that are more common. Native development and four cross-platform approaches: Web, Hybrid, Interpreted and Cross-Compiled approach. Figure 4 illustrates the type of application generated by the different development tools.

This section contains further explanation about each of those approaches.

![Figure 4: Mobile App Technology Stacks.](image)


3.1 Web App

Web application approach is developed using web technologies like HTML, CSS and Javascript. It is not installed in the device and is accessed via browser. Web application differs from websites because typically offers less information than a website and is developed with an interface that is closer to application’s appearance.

3.2 Native App

In native approach the application is built for a specific platform. Each platform accepts different programming languages. Android offers the possible to use Java or Kotlin and iOS offers Objective-C or Swift. The developer uses compilers which is a program that transforms the source code written in the source language (high-level
programming language) into the target language (lower-level language like the assembly language or machine code). Native applications take full advantage of the software and the operating systems' features. These applications can also directly access hardware devices such as GPS, camera and microphone.

3.3 Cross-Compiled App

In cross-compiled approach the same application is compiled just like a native application and a platform-specific version of the application is created for each target platform. The advantage of this approach is that the generated applications achieve high overall performance because of the generated native code and provides all the features of native application including its native interface components. The major drawback of this approach is that the user interface cannot be reused. Also, many of the features cannot be reused as the method to access those features for example camera access; geolocation etc is different for each platform.

3.4 Hybrid App

Hybrid approach combines web approach with native development. It is primarily developed using web technologies such as HTML and Javascript. Hybrid applications run in a full-screen and full-powered browser, known as webview, that is invisible to the user. Modern Web Views offer many built-in HTML5 APIs for hardware functionality such as cameras, sensors, GPS, speakers, and Bluetooth, but sometimes it may also be necessary to access platform-specific hardware APIs(IONIC, 2019b).

In hybrid approach, instead of your entire UI depending on the native platform, only certain native device features, like the Camera, are platform-dependent. This allows your application to run anywhere the web runs. This also means that your UI layer can be shared between all platforms. So with hybrid approach you have one codebase running anywhere. The disadvantage of this approach is that hybrid applications also suffer from platform specific behaviour of JavaScript and threading model incompatibilities with JavaScript(THAKARE et al., 2014).
3.5 Interpreted App

In interpreted approach the developer can use several technologies and languages to implement application logic. The interpreter translates the developed source code to executable instructions in runtime with a dedicated engine. The end users interact with platform-specific native user interface components. Interpreted application provides the look and feel of native application.

4 EVALUATION METHODOLOGY

In this paper we evaluate different development approaches. We compare CPU and RAM consumption, execution time and application size. For each development approach a framework was chosen based on its popularity. For hybrid development we use Ionic, for interpreted development we use React Native, for cross-compiled development Flutter and, finally, for native development we use Java. Java was the only tool that we had previous knowledge. We had a short contact with Ionic before working with it in this paper. And with React Native and Flutter we had our first experience in this work.

For each framework we develop four different applications: Sort Application, Read File Application, Write File Application and Download File Application. For each application the CPU and RAM consumption, execution time and size of the application were measured.

Section 4.1 describes the development tools used in this paper. Section 4.2 details the developed applications. Section 4.3 explains what measurements were taken and how they were taken.

4.1 Development Tools

4.1.1 Java

Java is one of the most popular programming language today (TIOBE, 2019). Java is the base for practically all types of networked applications and is the global standard
for developing and distributing applications, games, Web content, and enterprise software (JAVA, 2019). To work with Java for Android it is necessary to have the Android Software Development Kit (SDK). SDK provides a selection of tools required to build Android applications. It is possible to use an emulator to test applications, monitor your device, and do a host of other things. While it is possible to use the SDK tools by command line, the most common method is to use an integrated development environment (IDE). Android SDK also provides an IDE, Android Studio, where the work gets done and many of the tools are now best accessed or managed. In summary, the Android SDK can be broken down into several components: Platform-tools, Build-tools, SDK-tools, The Android Debug Bridge (ADB), Android Emulator.

4.1.2 Ionic

Ionic (IONIC, 2019a) is an open-source cross-platform development tool. Ionic framework is a library of UI Components, which are reusable elements that serve as the building blocks for an application. Ionic components are built with web standards using HTML, CSS, and JavaScript. Ionic framework allows developers to use the same code base for multiple platforms. Every Ionic component adapts its look to the platform on which the app is running on. By making subtle design changes between the platforms, users are provided with a familiar app experience. Additionally, deciding which platform to use in certain scenarios is entirely configurable.

Ionic applications are rendered using Web Views, which are a full screen and full-powered web browser. Modern Web Views offer many built-in HTML5 APIs for hardware functionality such as cameras, speakers and GPS, but sometimes it may also be needed to access platform-specific hardware APIs. In Ionic applications, hardware APIs can be accessed through a bridge layer, typically by using native plugins which expose JavaScript APIs.
4.1.3 React Native

React Native (NATIVE, 2019) is an open-source cross-platform development tool developed by Facebook, first released in 2015. It uses Javascript and is based on React, Facebook’s Javascript library for building user interfaces, but instead of targeting the browser, it targets mobile platforms. React Native uses a Javascript Virtual Machine to run all Javascript code. In case of Android, React Native bundles the JavaScript-Core along with the application, which increases the application size. React Native apps have direct access to all Native APIs and views provided by the platform OS.

With React Native it is possible to use either Javascript and native implementation. If a functionality does not exist in React Native, native code can be used to give the best user experience. React Native offers multiplatform development following the idea that large parts of the application code can be shared across platforms, although some platform-specific code is needed. (HANSSON; VIDHALL, 2016). Because of that its objective is not to be a "write once, run anywhere" framework, instead it’s aiming to be "learn once, write anywhere".

When the application is run for the first time, React Native CLI spawn a node packager/bundler that bundle the JS code into a single main.bundle.js file. Now, whenever the React Native application is launched, the first item to be loaded is the native entry point. The Native thread spawns the JS VM thread which runs the bundled JS code. The JS code has all the business logic of the application. The Native thread now sends messages via the RN Bridge to start the JS application. Now, the spawned Javascript thread starts issuing instructions to the native thread via the RN Bridge (Figure 5). The instructions include what views to load, what information is to be retrieved from the hardware, etc.

4.1.4 Flutter

Flutter (FLUTTER, 2019) is a mobile app SDK for building apps for iOS and Android from a single codebase. Flutter includes a modern react-style framework, a 2D
rendering engine, ready-made widgets, and development tools. The Flutter framework is organized into a series of layers, with each layer building upon the previous layer (Figure 6). The upper layers of the framework are used more frequently than the lower layers.

The engine’s C and C++ code are compiled with Android’s NDK. The Dart code (both the SDK’s and yours) are ahead-of-time (AOT) compiled into a native, ARM and x86 libraries. Those libraries are included in a “runner” Android project, and the whole thing is built into an APK. When launched, the app loads the Flutter library. Any rendering, input or event handling, and so on, is delegated to the compiled Flutter and app code. This is similar to the way many game engines work.
4.2 Applications

We have decided to develop applications that consume important device resources such as CPU and RAM. We also developed applications that execute common tasks between application, such as file handling. In this section we explain about each application and its operation.

**Sort application** sorts an array using bubble sort algorithm. This array is read from a file so every loop of each approach sorts the same array. When the application is initialized the array is read from the file and save into a variable that stays unchanged. To start sorting it is necessary to push a button on the screen. For every loop the original array variable is cloned to a new variable and this new variable is sorted. These steps of cloning and sorting are repeated for a certain number of times.

**Read file application** reads the content of a text file located in the device storage. The same file is used for every approach. The reading starts only after pushing a button in the view. This reading is made without flush and in a synchronous way so for each loop the application can only continue after finishing reading the file. It reads the entire content at once and the result is not stored in any variable and is not shown on the view. After reading the file content it repeats reading procedure for a determined number of times.

**Write file application** is a simple application that writes on a text file that is saved in the device storage. To start writing it is necessary to push a button on the screen. After starting the execution the entire content is written at once to a new file. This writing is made in a synchronous way so for each loop the application can only continue after finishing writing the file. After finishing writing the new file the application repeats this process creating a different file for each loop for a certain number of times.

**Download application** is a simple application that downloads a PDF file from the
internet. To start the download the user has to push a button in the initial screen. After pressing the button the application downloads the file in a synchronous way. The file is saved directly to the device storage without reading or editing it. After finishing these steps the application repeats them for a certain number of times. For each repetition the file is saved with a different name, resulting in a different file but with the same content.

4.3 Measurement

To measure the consumption of CPU and RAM a separate application (REDONDO, 2017) is used to not influence the tasks to be executed by each application. This application monitors and records the CPU and memory usage on Android devices. It is possible to monitor specific applications and save its data to a csv file. This application only works until Android 6.0.

Time measurement was made inside each application. Simple methods were used to get starting time and finishing time of each execution loop and place inside an array. After all task loops were executed the application saved the time array in a csv file.

We also compare the size of the installer that is generated and installed application and compare them between approaches.

5 EXPERIMENTS

5.1 Setup

To run the experiments we use an Android emulator. The emulator used is Genymotion (GENYMOTION, 2019) Personal Edition 3.0.2. The tests were executed on a high-end device, a Samsung Galaxy S7 with Android 6.0 - API 23, 4096 MB of memory and 4 CPU, provided by Genymotion.

Java applications were developed using Android SDK 23. All Flutter applications were developed using Flutter version 1.5.4-hotfix.2. React Native sort application was developed using version 0.57.5 and the others with version 0.59.9. Ionic sort application
was developed using version 3.9.2 and the others with version 3.9.5. The difference in the versions is due to the moment in which the applications were developed.

5.2 CPU, RAM and Execution Time Comparison

For these experiments the confidence interval was 99.9% and the significance level was 0.1%.

It is important to note that CPU and RAM consumption was measured during the whole time the application was running, that is, during all the repetitions. On the other hand, execution time was measured for each step.

5.2.1 Sort Application

This application sorts an array with 2500 elements using bubble sort. Sorting steps were repeated 900 times. The same array was used for all repetitions and all approaches. Figure 7 depicts the results.

Java was the approach that gained by far the best performance in RAM consumption, reaching an average of just over 5 MB. Flutter spent a little over 15 MB and React Native spent on average 30 MB. Ionic achieved the worst performance by spending at least 20 times more than Java and a little more than 3 times React Native.

In terms of CPU consumption, all approaches performed similarly, with results hovering around 25 %. Java achieved the best performance, followed by Flutter and Ionic, and finally React Native. Considering the margin error this result is a technical draw.

In time measurement Java has achieved a much better result than any of the other approaches. Its average time was less than 10 ms. Flutter came in second with an average between 400 ms and 500 ms. Next comes Ionic with average around 900 ms, followed by React Native with average over 1100 ms.

In this application the clear victory was of Java because the average consumption of RAM and the execution time was much better than its rivals. In second place was Flutter, due to its average consumption of RAM and run time. In the last two
places were React Native and Ionic respectively. Although Ionic has a slightly smaller execution time than React Native, its RAM consumption was much higher, which led to it being ranked last.

5.2.2 Read File Application

In this experiment we used a 5 MB text file that contains 5 million ’a’ characters. Reading was made synchronously. The reading was repeated 900 times for Java, Flutter and Ionic. With React we had a RAM memory problem. RAM continued to scale reading after reading until the application crashed. Because of this, it was only possible to repeat reading file 90 times. Figure 8 shows the results.

As previously mentioned, in RAM perspective React Native scaled continuously
and the Garbage Collector did not seem to work. You can see this by comparing the amount of RAM spent by React Native with the other approaches and the fact that React Native executed only 90 loops against 900 from the others. Java had a slight advantage compared to Flutter and so had the best performance. But the result can be considered a technical draw. And third place was Ionic.

React Native was ranked first in CPU consumption comparison consuming less than 25% of CPU. Flutter had a small advantage compared to Ionic. Both consumed around 25% of CPU. Here Java had the worst performance, consuming on average between 25% and 27.5%. Considering the margin of error the result can be considered a technical draw.

Due to the memory problem, React Native application had a running time well
above the others. Java had the best performance, taking on average less than 50 ms to perform each repetition. Flutter comes next with execution time just under 100 ms. Ionic is in third having taken on average between 200 ms to 300 ms.

Java and Flutter took turns in the positions of the different metrics. Flutter consumed less CPU and Java had a faster execution and both tied in memory consumption. So both can be considered as good options for this scenario. Ionic performed better than Java in CPU consumption but was well behind both RAM consumption and runtime and so appears as third place. React Native was last because it could not finish the task completely and and RAM consumption grew uncontrolled.

5.2.3 Write File Application

This experiment writes 5 MB text file and saves it to device storage. It writes 5 million 'a' characters at once, without any loop. Writing was made synchronously. The writing was repeated 900 times for all approaches. Figure 9 illustrates the results.

In RAM consumption measurement, Java obtained a better result, followed by the Flutter behind by a small difference, both consuming less than 100MB. Ionic consumed almost double, averaging 200 MB. React Native exceeded 800 MB of consumption. Despite the high RAM consumption of React Native, it was possible to perform the same amount of repetitions in all approaches.

In CPU consumption, Java obtained the best result, consuming on average a little more than 20 %. Ionic appears in second consuming around 22.5 %. Flutter and React Native had a technical draw considering the margin of error. Both consumed around 25

Ionic got the best execution time getting an average slightly below 150 ms. Java came next with just over 200 ms. React Native ranged from 300 ms to 350 ms. And Flutter had the worst performance, having time averages over 350 ms.

In an overall analyses Java again has the best performance. Although it took longer than Ionic to run each loop, its performance in RAM and CPU consumption was better among all approaches. Ionic had the second best performance. It was the approach
with the best processing time, lost only to Java in CPU consumption and was third in RAM consumption. Flutter came in third because despite having one of the two best uses of RAM, it had the worst execution time and was one of the two worst in CPU consumption. Although React Native has been able to run all 900 repetitions, its result of RAM consumption was much worse than the other approaches, being 4 times greater than the third place, so it was in the last position.

5.2.4 Download Application

In this experiment the application downloads a file from local network and saves it to the device storage. The downloaded file has a size of 5 MB. The download was made in a synchronous way and is repeated 100 times.
We configured a Virtual Machine with Ubuntu and Apache with a PHP code to provide a local link for download. We executed this virtual machine alongside Genymotion Android emulator.

We were not able to make React Native application download the file synchronously. Because of this React Native did not participate in this experiment.

Figure 10 details the results.

In RAM consumption comparison Java had the best result, consuming less than 10MB. Flutter consumed 3 times more RAM, more than 30MB. Ionic consumption passed 80 MB, at least almost 3 times more than Flutter and 8 times more than Java.

Java had also the best result in CPU consumption, consuming less than 10%. Flut-
ter again in second with around 15% of CPU consumption. Ionic consumed more than 30%, more than 2 times the consumption of Ionic and 3 times more consumption than Java.

Flutter had the best average execution time for each step, less than 200 ms. Java took more than two times more, almost 500 ms. Ionic, again, had the worst result, taking almost 800 ms per step, 4 times more than Flutter.

Although Flutter download faster, the best end result was Java. Java had better CPU and RAM consumption and the second best execution time. Flutter came in second with good execution time but worst CPU and RAM consumption than Java. Ionic came in third in all graphics.

5.3 Application Size Comparison

<table>
<thead>
<tr>
<th>Application</th>
<th>Java</th>
<th>Flutter</th>
<th>Ionic</th>
<th>React Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>1.52</td>
<td>5.88</td>
<td>3.82</td>
<td>7.97</td>
</tr>
<tr>
<td>Read</td>
<td>1.52</td>
<td>5.88</td>
<td>4.04</td>
<td>28.79</td>
</tr>
<tr>
<td>Write</td>
<td>1.44</td>
<td>5.43</td>
<td>3.83</td>
<td>28.79</td>
</tr>
<tr>
<td>Download</td>
<td>1.44</td>
<td>6.15</td>
<td>3.85</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Java</th>
<th>Flutter</th>
<th>Ionic</th>
<th>React Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>1.47</td>
<td>15.99</td>
<td>4.04</td>
<td>16.25</td>
</tr>
<tr>
<td>Read</td>
<td>1.46</td>
<td>15.99</td>
<td>4.04</td>
<td>47.76</td>
</tr>
<tr>
<td>Write</td>
<td>1.40</td>
<td>15.56</td>
<td>3.84</td>
<td>47.76</td>
</tr>
<tr>
<td>Download</td>
<td>1.40</td>
<td>17.89</td>
<td>3.87</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Java installer and installed application size is smaller than all other approaches and the difference between them is very small. Ionic have the second smallest installer and installed application size but is 2 times bigger than Java’s. The difference between them are also very small. Flutter comes after Ionic, its installed application size is 3 times bigger than Java’s. The difference between the installer and the installed
application size is big, after installed the application gets almost three times bigger than the installer. This might happens because of x86 and Dart Libraries included in a “runner” Android project. In last is React Native, which comparing to all other approaches has big installer and installed application size. This happens because React Native bundles JavaScriptCore along with the application. It is possible to notice that Sort application is not that big compared to Flutter. And is considerably smaller than Read and Write applications, this must happen because Sort application did not need any additional lib to run while Read and Write file need a lib to access file and storage.

5.4 Discussion

In our experiments, we observed that Java was almost always the best. Java lost to Flutter in one experiment, tied with other approaches in three, and won the others. We noticed that Flutter was the cross-platform approach that came closer to Java in most experiments.

React Native was expected to outperform Ionic, but from our experiment, we observed that Ionic had better results. React Native had the overall worst results.

6 CONCLUSIONS

With the growing popularity of mobile devices, mobile development is becoming more and more important. There are many different tools available in the market for developing an application. Each tool fits into a different approach category, the most common among those approaches are: native, hybrid, interpreted and cross-compiled. In this work we selected, for each approach, the most popular development tools: Java, Ionic, React Native and Flutter.

We developed four applications using each of those tools. Those applications focused in intensive CPU consumption, file operations and download. Our experiments consisted in executing those application and collection CPU and RAM consumption, execution time and application size. From the results we were capable to evaluate
performance among different tools. The results shows that though Java is still better at performance other approaches are getting closer to it.

This result is important to show that cross-platform are evolving and are getting closer and closer to native performance. This paper aims to end some myths that some approaches are better than others. Myths that are often created by the developers themselves.

As future work, we intend to develop applications with other characteristics and use different development tools.
REFERENCES


MAZARICO, Carlos Sirvent; CARRERA, Marc Campillo. Comparison between Native and Cross-Platform Apps. Bachelor in Computer Science — Linnaeus University, February 2015.


### APPENDIX A  LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 1</td>
<td>Popularity of hybrid frameworks on Google Trends.</td>
<td>3</td>
</tr>
<tr>
<td>FIGURE 2</td>
<td>Popularity of interpreted frameworks on Google Trends.</td>
<td>4</td>
</tr>
<tr>
<td>FIGURE 3</td>
<td>Popularity of cross-compiled frameworks on Google Trends.</td>
<td>4</td>
</tr>
<tr>
<td>FIGURE 4</td>
<td>Mobile App Technology Stacks.</td>
<td>9</td>
</tr>
<tr>
<td>FIGURE 5</td>
<td>React Native Architecture</td>
<td>14</td>
</tr>
<tr>
<td>FIGURE 6</td>
<td>Flutter Architecture Overview</td>
<td>14</td>
</tr>
<tr>
<td>FIGURE 7</td>
<td>Sort Results</td>
<td>18</td>
</tr>
<tr>
<td>FIGURE 8</td>
<td>Read File Results</td>
<td>19</td>
</tr>
<tr>
<td>FIGURE 9</td>
<td>Write File Results</td>
<td>21</td>
</tr>
<tr>
<td>FIGURE 10</td>
<td>Download File Results</td>
<td>22</td>
</tr>
</tbody>
</table>
### APPENDIX B  LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Installer size in MB</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>Installed application size in MB</td>
<td>23</td>
</tr>
</tbody>
</table>