Performance Analysis of Mobile Cross-platform Development Approaches based on Typical UI Interactions

Stefan Huber and Lukas Demetz
University of Applied Sciences Kufstein, Andreas Hofer-Straße 7, 6330 Kufstein, Austria

Keywords: Android, Ionic/Cordova, Mobile Cross-platform Development, Performance Analysis, React Native.

Abstract: The market for mobile apps is projected to generate revenues of nearly $ 190 billion by 2020. Besides native development approaches, in which developers are required to maintain a unique code base for each mobile platform they want to support, mobile cross-platform development (MCPD) approaches can be used to develop mobile apps. MCPD approaches allow building and deploying mobile apps for several mobile platforms from one single code base. The goal of this paper is to analyze the performance of MCPD approaches based on UI interactions. For this, we developed three mobile apps, one native and two developed using MCPD approaches. Using an automated test, we measured CPU usage and memory consumption of these apps when executing one selected UI interaction, that is, swiping through a virtual scrollable list. The results indicate that the CPU usage of the two apps developed using MCPD approaches is about twice as high compared to the native app, the memory consumption is even substantially higher than in the native app. This papers confirms results of previous studies and extends the body of knowledge by testing UI interactions.

1 INTRODUCTION

The market of mobile platforms is dominated by Google Android and Apple iOS (Statista, 2018). Revenues generated by mobile apps are forecast to reach nearly $ 190 billion by 2020 (Statista, 2019). Developers who want to develop mobile apps are presented with several ways for doing so. A common way to develop mobile apps is to use native development approaches. This requires developers to build and maintain a unique code base for each mobile platform they want to support (e.g., Google Android and Apple iOS). To circumvent this problem of multiple code bases, there exists a plethora of mobile cross-platform development (MCPD) approaches, which allow building and deploying mobile apps for several mobile platforms from one single code base.

These approaches use different techniques that allow to deploy the same code base to several platforms (Majchrzak et al., 2017). Ionic/Cordova, for instance, uses a native WebView component to display the mobile app, while React Native pursues an interpretive approach, in which JavaScript is used to render native user interface (UI) components (Biørn-Hansen et al., 2018).

Techniques used by these MCPD approaches impose different requirements on mobile devices. When executed, they put a higher load on mobile devices, especially when compared to a native implementation (Dalmasso et al., 2013; Willocx et al., 2016). So far, research regarding performance of cross-platform development approaches, focused mainly on programmatic performance of rather compute-intensive apps (Ajayi et al., 2018, for instance, analyzed run-time differences of sorting algorithms) leaving out the analysis of typical UI interactions (e.g., swipe gestures). Mobile apps are, however, rather interactive apps in which users interact with the app via the user interface (Vallerio et al., 2006). Thus, interactions with the UI are an important aspect of mobile app usage and should not be neglected in performance analyses. To investigate possible differences with respect to performance and load on mobile devices when performing UI interactions, this paper strives to answer the following research question, *How do mobile cross-platform development approaches differ with respect to performance and load on mobile devices when performing typical UI interactions?*

To answer this research question, we present the results of a quantitative performance analysis of one typical user interaction based on a common UI component: continuous swiping through a virtual scrollable list. The analysis is based on three implementations, a native mobile app used as a baseline, an app

Huber, S. and Demetz, L.
Performance Analysis of Mobile Cross-platform Development Approaches based on Typical UI Interactions.
DOI: 10.5220/0007838000400048
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 40-48
ISBN: 978-989-758-379-7

using React Native and an app using Ionic/Cordova. All three apps implement the same UI interaction. The results indicate that the CPU usage of the two apps developed using MCPD approaches is about twice as high compared to the native app, the memory consumption is even substantially higher than in the native app. Our results confirm results of previous studies and extend the body of knowledge by testing UI interactions. Mobile app developers can use these results as guidance for selecting MCPD approaches.

The remainder of this paper is organized as follows. In Section 2 we present research related to this study. We start by presenting different MCPD approaches (Section 2.1). Afterwards, we highlight related research regarding performance measurement (Section 2.2) and resource usage (Section 2.3). Section 3 presents the applied research procedure. We start by presenting the tested implementation (i.e., the mobile app) along with implementation details for each mobile cross-platform development approach, the test case, and the measurement. Afterwards, we present the results of this measurement in Section 4. These results as well as limitation of this paper are discussed in Section 5. Finally, Section 6 concludes this paper and provides possible avenues for future research.

2 RELATED WORK

In this section, we present literature related to this study. We start with a general overview of MCPD approaches (Section 2.1). Afterwards, we present literature analyzing the performance of such approaches. They fall under two broad categories user-perceived performance (Section 2.2) and resource usage (Section 2.3).

2.1 Mobile Cross-platform Development Approaches

Mobile cross-platform development approaches help developers to create and maintain one code base and to deploy this single code base to multiple mobile platforms, such as Google Android and Apple iOS. There is a multitude of such approaches, which can be broadly divided into five categories: hybrid, interpreted, cross-compiled, model-driven, and progressive web apps (Biørn-Hansen et al., 2018). In the following, we shortly present these categories.

Hybrid. The idea of hybrid approaches is to use web technologies (e.g., HTML, CSS and JavaScript) to implement user interfaces and behavior, that is, a website. Within the mobile app, this website is then displayed in a WebView component, which is a web browser (Latif et al., 2016) embedded inside a native UI component. Apache Cordova, formerly Adobe PhoneGap, leverages this approach.

Interpreted. In the interpreted approach, developers also use JavaScript to build an app. In contrast to the hybrid approach, developers do not build a website, but use JavaScript to render platform native UI components (Dhillon and Mahmoud, 2015; El-Kassas et al., 2017). Examples for this category are Facebook React Native and Appcelerator Titanium.

Cross-compiled. The idea of the cross-compiled approach is to use a common programming language to develop a mobile app. This source code is then compiled into native code that can be executed on a mobile platform (Ciman and Gaggi, 2017a). A prominent example for this category is Microsoft Xamarin.

Model-driven. When building apps using a model-driven cross platform approach, developers use a domain specific language. The approach then provides generators that translate the app written in domain specific language into native code that can be executed on a mobile platform (Heitkötter and Majchrzak, 2013). An example for this category is MD^2.

Progressive Web Apps. Apps of this category are web apps that are served by a web server and are accessed by a URL. Compared to standard web apps, progressive web apps provide more sophisticated functions (e.g., offline availability). The web app itself is developed using standard web technologies, such as HTML, CSS and JavaScript (Biørn-Hansen et al., 2018). Web frameworks such as Ionic or Onsen UI offer progressive web app capabilities.

Although these approaches pursue the same idea, that is, to allow to deploy one code base to multiple mobile platforms, they differ with respect to aspects such as developer focus and end-product focus (Majchrzak et al., 2017). While some approaches allow developers more architectural choices, others ease the development of UI components. These internal differences entail performance differences that have already been investigated.

2.2 User-perceived Performance

Looking at performance perceived by users, previous studies provide varying results regarding the performance of apps developed using MCPD approaches. Xanthopoulos and Xinogalos (2013) rated different mobile cross-platform development approaches based on the user-perceived performance as low, medium or high. The classification was based on the authors experience and information published on the web. Andrade et al. (2015) conducted a real-world

experiment, with 60 employees of a company in the Brazilian public sector. A native and a hybrid app were developed and used by the employees for two weeks each. Only 13.33% of users noted a performance difference between the two approaches. Mercado et al. (2016) investigated user complaints from app markets of 50 selected apps created either natively or with MCPD approaches. Based on results of natural language processing the authors conclude that apps based on MCPD approaches are more prone to user complaints about performance issues.

2.3 Resource Usage Measurements

Previous research on the usage of resources (e.g., CPU, memory) of MCPD approaches show more consistent results compared to user-perceived performance. Dalmasso et al. (2013) created Android apps with two MCPD approaches, Titanium and PhoneGap. The authors measured memory, CPU and battery usage. Their findings indicate that the JavaScript framework used inside the WebView component has a high impact on memory and CPU consumption. However, their measurement approach could only provide CPU results for PhoneGap, not for Titanium. Willocx et al. (2015) measured several performance properties of two MCPD implementations and compared them respectively to a native implementation both on iOS and Android devices. Among measuring time intervals such as launch time, pause and resume time, time between page transitions, memory and CPU usage was measured. The authors repeated their measurements in Willocx et al. (2016) with 10 different MCPD approaches including measurements for Windows Phones. Overall, their findings show that the hybrid approaches are highest in resource consumption on all platforms. Non-hybrid based approaches have a significant lower resource consumption, however, still higher than native implementations. Ajayi et al. (2018) analyzed performance of algorithms, such as quicksort, implemented natively with Android and a hybrid approach. The results show that the native app outperforms a hybrid app in terms of CPU and memory usage.

The results of user-perceived performance studies provide a good indication that apps developed with MCPD approaches are inferior in terms of performance in comparison to native apps. The studies on resource usage validate these findings quantitatively. Although all measurements give good indications on differences of the approaches, it is unclear under which concrete UI interactions the results are created. Previous research mainly focused on performance differences of rather computing intensive mobile apps developed using MCPD approaches. Mobile apps rather focus on interacting with users through the UI and not on complex computations (Vallerio et al., 2006). Thus, when analyzing performance and resource consumption of mobile apps, UI interactions should not be left out. To close this research gap, this study pursues a different approach to performance analysis of mobile apps. In doing so, this study analyzes the performance of mobile apps developed using MCPD approaches when executing one typical UI interaction, that is, continuous swiping through a virtual scrollable list.

3 PROCEDURE

This section presents the procedure we applied to answer the research question. We start by describing the selection of MCPD approaches we used to develop mobile apps (Section 3.1). Then, we describe the mobile apps we developed (Section 3.2). We also present what UI components we used to implement the mobile apps using a native approach, and the two MCPD approaches. Afterwards, the test case we applied to test performance differences is presented (Section 3.3). Finally, we conclude this section by detailing the measurement tool and the mobile device we used for our test cases (Section 3.4).

3.1 Selection of Approaches

For analyzing the performance of MCPD approaches we selected two approaches. The selected approaches allow reusing web development skills for developing mobile apps. Additionally, they enable transferring existing code bases (e.g., web application developed in JavaScript) into mobile apps. We consider the reuse of existing code bases and of existing skills as a main advantage of MCPD approaches.

Ionic/Cordova and React Native are two approaches based on web technologies. Both approaches are supported by large developer communities (e.g., visible on GitHub and Stack Exchange). Many successful apps found in app markets are developed with one of both approaches.

3.2 Implementation Details

For the evaluation a basic contact app was envisioned. The app consists of a single screen with a scrollable list of contacts showing the contact's name and phone

number (see Figure 1). The app consists of the two main building blocks:

- A dataset of 1000 demo contact objects with attributes name and phone number, which is created at app start up and stored in memory for the lifetime of the app.
- A virtual scrollable list presenting the dataset.

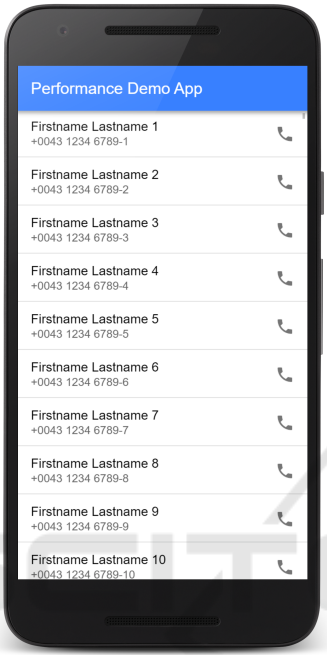

Figure 1: Performance demo app screenshot (Ionic/Cordova implementation).

Virtual scrollable lists are common UI components found in many mobile apps. Smart phones have only limited screen space available and thus only a small part of data presented inside a list is visible for a user. All hidden entries do not need to be processed and rendered before actually visible. The scrolling feature is simulated and new entries in the list are created at runtime while scrolling down or up a list. We selected this kind of app, and UI interaction, as it is frequently found in mobile apps, as there are efficient ways to implement it using various MCPD approaches and can be easily tested using an automated script. Additionally, this app is not computing intensive meaning that no complex computations need to be done. The focus of this app is on interacting with the user.

To answer our research question, we developed three instances of this app. One instance was developed using native Google Android development tools, two instances using MCPD approaches (React Native and Ionic/Cordova). All three apps were packaged and signed for release on Google Android. Thus, any performance degradation caused by debugging build features could be eliminated to minimize the creation of incorrect results. Table 1 provides an overview of the three apps showing the used version and the UI component for implementing a virtual scrollable list. These approaches are detailed in the following subsections.

3.2.1 Native Android App

For the native app the *RecyclerView* (Google, 2019b) component was used. This component is the default approach for realizing a virtual scrollable list on Android. An *adapter* component has to be implemented to provide an app specific binding to a data source. The adapter design pattern allows for dissolving any dependency between the data and the display logic. The demo data was generated on app start up and stored in memory. It was provided to the RecyclerView component through the adapter.

3.2.2 React Native App

React Native offers the *FlatList* (Facebook, 2019) component. This UI component is an abstraction over the native implementations of virtual scrollable lists, such as the RecyclerView on Android. Within a React component the demo data was generated at app start up and stored in memory. The data was directly referenced by the FlatList for actual display.

3.2.3 Ionic/Cordova App

Ionic is a JavaScript web-framework which aims to provide reusable UI components styled like Android or iOS native components. Ionic offers the *ion-virtual-scroll* (Ionic, 2019) component, which is a virtual scrollable list implementation for the web. The UI component emulates virtual scrolling within a WebView or web browser. The demo data for the list was generated at start up and stored in memory. The data was directly referenced by ion-virtual-scroll for actual display. The Ionic build was packaged inside a Cordova app for release.

3.3 Test Case

To test the three apps, we created a fully automated test case, which is independent of the specific apps. We could use the exact same test case for all three apps. This allowed a direct comparison of the test results.

Each step of the test case was exactly timed and the whole case could be executed repeatedly for one of the developed apps on a connected Android device.

Table 1: Overview of apps.

Approach	Version	UI component
Android native	Compiled Android API Level 28, Minimum Android API Level 23	RecyclerView
React Native	React Native 0.58	FlatList
Ionic/Cordova	Ionic 4.0.1, Cordova Android 8.0.0	Ionic Virtual Scroll

In the following a detailed step by step description of the test case is given. All of these steps were executed without human intervention.

- Install the app on the connected Android device.
- Start the measurement tool (Section 3.4) with a total recording time of 40 seconds and wait 2 seconds.
- Start the app by triggering an intent on the connected Android device and wait for 12 seconds. The different development approaches have different loading times, therefore a long waiting time is required. The apps were always loaded under 12 seconds of waiting time on the test device.
- Execute three bottom-to-top swipe gestures, centered on the device screen and wait three seconds between gestures.
- After the last swipe gesture wait for the end of the 40 second recording time of the measurement tool.
- Close the app and uninstall it from the device.

For all device management tasks the Android Debugging Bridge (adb) was used. The user interactions on the device could be simulated with the monkeyrunner library (Google, 2019a), which is part of the Android SDK. The test script with all device interactions was written as a python script. For each developed app, this test case was executed 50 times.

3.4 Measurement Tool and Device

As Android is a Linux-based operating system, it offers many of Linux' command-line utilities. vmstat (Henry and Fabian, 2009) is a Linux tool for executing continuous performance measurements. All performance measurement results produced by the proposed test case were recorded with vmstat.

vmstat was running for 40 seconds during the automated test cases and among other measurements, we recorded CPU usage and free memory of the mobile device. The recording interval was set to one second and each test run produced a time series of 40 measurement results.

The mobile device used for the measurement was the LG Nexus 5, which is a mid-range smart phone. The Android version of the device was updated to the maximum supported version of the manufacturer. In table 2 details of the device specification are listed.

Table 2: LG Nexus 5 specification.

CPU	Quad-core 2.3 GHz
Memory	2 GB RAM
Display resolution	1080 x 1920 pixels
Display size	4.95 inches
Android Version	Version 6.0.1, API Level 23

4 RESULTS

This section is devoted to the results of this study. We start by presenting our results with respect to CPU usage (Section 4.1). Afterwards, we describe the measured memory consumption of the developed apps (Section 4.2). The results were produced by executing the test case 50 times for each app in alternating order.

4.1 CPU Usage

Throughout the execution of the test case presented in Section 3.3 the CPU usage has been measured periodically every second. The topmost plot in Figure 2 shows a comparison of the CPU usage of the three apps. The vertical axis shows the averaged CPU usage in percent. That is, for each app, we averaged the CPU usage over the 50 test cases. On the horizontal axis the time in seconds is represented. The solid line shows the CPU usage of the Android native app, the dashed line shows the React Native app and the dotted line shows the Ionic/Cordova app.

Overall, we can see that the native development approach has the lowest CPU usage. Apps developed with MCPD approaches always come with a performance overhead compared to native development, as also others have found (Willocx et al., 2015; Dalmasso et al., 2013; Ajayi et al., 2018). React Native has the highest CPU usage peaks of more than sixty percent, followed by the Ionic/Cordova app with peaks of around fifty percent.

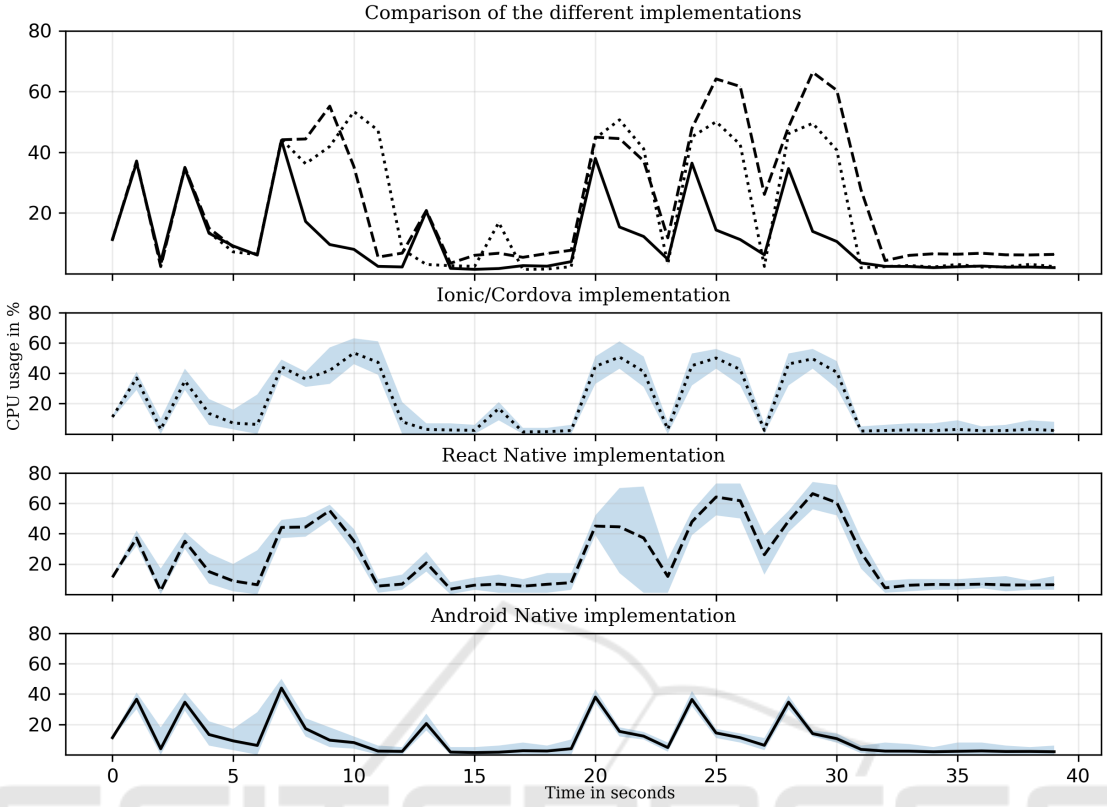

Figure 2: CPU usage comparison of different apps.

The app start up was happening roughly between second 5 and second 18. For the Android native app a single steep usage spike is visible. In comparison, the MCPD apps have broader plateau-like shaped usage peaks. Approximately between second 19 until second 34 the swipe gestures were executed. For all approaches three consecutive usage peaks are visible with a delay of about 3 seconds between each peak. The native app again produced steep usage spikes, which are contrasted by plateau-like shaped peaks for the MCPD approaches.

Table 3: Cumulative average of CPU usage.

Second	Native	Ionic/Cordova	React Native
5-18	10.0%	20.3%	18.8%
19-34	13.2%	26.6%	35.3%

For the average CPU usage it can be said that during app start up roughly twice as many resources are required by MCPD approaches. As this stays the same for Ionic/Cordova during the user interaction phase, React Native requires around 2.5 more CPU usage as the Android native app. In table 3 the cumulative results of the average CPU usage between the two time intervals (second 5 to 18 for app start up, and second 19 to 34 for user interaction) are summarized.

The additional three plots in Figure 2 show the variance of the 50 test cases. That is, they show a spread visualization between the 5%-quantil and the 95%-quantil around the average CPU usage of the different apps.

Table 4: Average variance in CPU usage.

Second	Native	Ionic/Cordova	React Native
5-18	3.7%	5.5%	4.0%
19-34	2.3%	6.2%	8.1%

During the start up phase the average CPU usage variance is almost similar for Android native and React Native. The average variance for Ionic/Cordova is slightly higher within the start up phase. Throughout the user interaction phase the Android Native implementation is almost negligible around 2%. However, React Native has a high spread mainly during the first swipe gesture. On average React Native has a variance of 8.1% and Ionic/Cordova 6.2% during the user interaction phase. Table 4 gives an overview of these results.

4.2 Memory Consumption

Similar to the CPU usage, the consumption of free memory on the device was measured periodically every second throughout the execution of the test cases. The topmost plot in Figure 3 shows a comparison of the different apps. The vertical axis shows the averaged memory consumption in percent. That is, for each app, we averaged the memory consumption over the 50 test cases. On the horizontal axis the time in seconds is represented. The solid line shows memory consumption of the Android Native app, the dashed line shows the React Native app and the dotted line shows the Ionic/Cordova app.

In comparison to the MCPD approaches the Android native app has an almost negligible memory footprint. The Ionic/Cordova app has a memory requirement of around 50% of the freely available memory throughout the test case. Although during the start up phase React Native has a requirement of around 15% of memory, the amount rises to above 30% during the user interaction phase.

Table 5: Cumulative average of memory consumption.

Second	Native	Ionic/Cordova	React Native
5-18	3.0%	32.7%	14.4%
19-34	1.8%	51.7%	26.5%

Cumulatively, for the average memory consumption it can be said that during application start up Ionic/Cordova requires around 10-times more memory and React Native 5-times more memory than the native counterpart. During the user interaction phase this rises to even 28-times more for Ionic/Cordova and 14-times more for React Native. Overall Ionic/Cordova consumes approximately twice as much memory than React Native. In table 5 the cumulative results of the average memory usage between the two time intervals are presented.

The additional three plots in Figure 3 show the variance of the 50 test cases. That is, they show a spread visualization between the 5%-quantil and the 95%-quantil around the average memory consumption of the different apps.

Table 6: Average variance in memory consumption.

Second	Native	Ionic/Cordova	React Native
5-18	7.8%	15.2%	8.6%
19-34	7.7%	20.2%	11.2%

Throughout the execution of the test cases, the average variance stays at around 7.7% to 7.8% for the native approach. For the MCPD approaches the average variance rises during the user interaction phase. In table 6 a summary of the different average memory consumption variances is given.

5 DISCUSSION

Prior work has documented that MCPD approaches consume more resources (e.g., CPU and memory) than native implementations (Dalmasso et al., 2013; Willocx et al., 2016). In this study a detailed view is put on the CPU usage and memory consumption during a typical UI interaction. The interaction with a common UI component, namely a virtual scrollable list was investigated more thoroughly. The results provide some guidance to mobile app developers in selecting a suitable MCPD approach. Even though developing mobile apps using such approaches has valid reasons, these advantages come at some costs, for instance, higher CPU load and memory consumption.

The selection of a mobile development approach highly influences mobile app projects. As prior research has shown, users perceive a difference between a native implementation and an implementation based on an MCPD approach (Andrade et al., 2015). Although this difference is acceptable if high-end devices are considered (Willocx et al., 2016), especially the market of Android smart phones is highly dispersed between low-end and high-end devices.

The reusability of existing code bases or the maintenance of a single code base are common arguments for using MCPD approaches. Both examined approaches are based on web-technologies. Thus, JavaScript code bases can be reused or shared. Additionally, both MCPD approaches leverage existing skills of web-developers, that is, HTML, CSS and JavaScript. Therefore, MCPD approaches are highly recommended for reducing development and maintenance costs for applications used in different settings (e.g., mobile and web version of an application).

However, the performance of a mobile app can be a competitive advantage in certain contexts. The increased resource usage of MCPD approaches has negative effects on the battery lifetime of smart phones (Ciman and Gaggi, 2017b). Also a lower user-perceived performance can lead to user complaints (Mercado et al., 2016). Thus, for apps which have a necessity for frequent user interactions, a native implementation should be considered. Users might switch to competing apps in favor of lower battery drainage or an increased user-perceived performance.

It should be noted, that there is a possibility to mix native development with MCPD approaches. Therefore, frequently used parts of an app can be im-

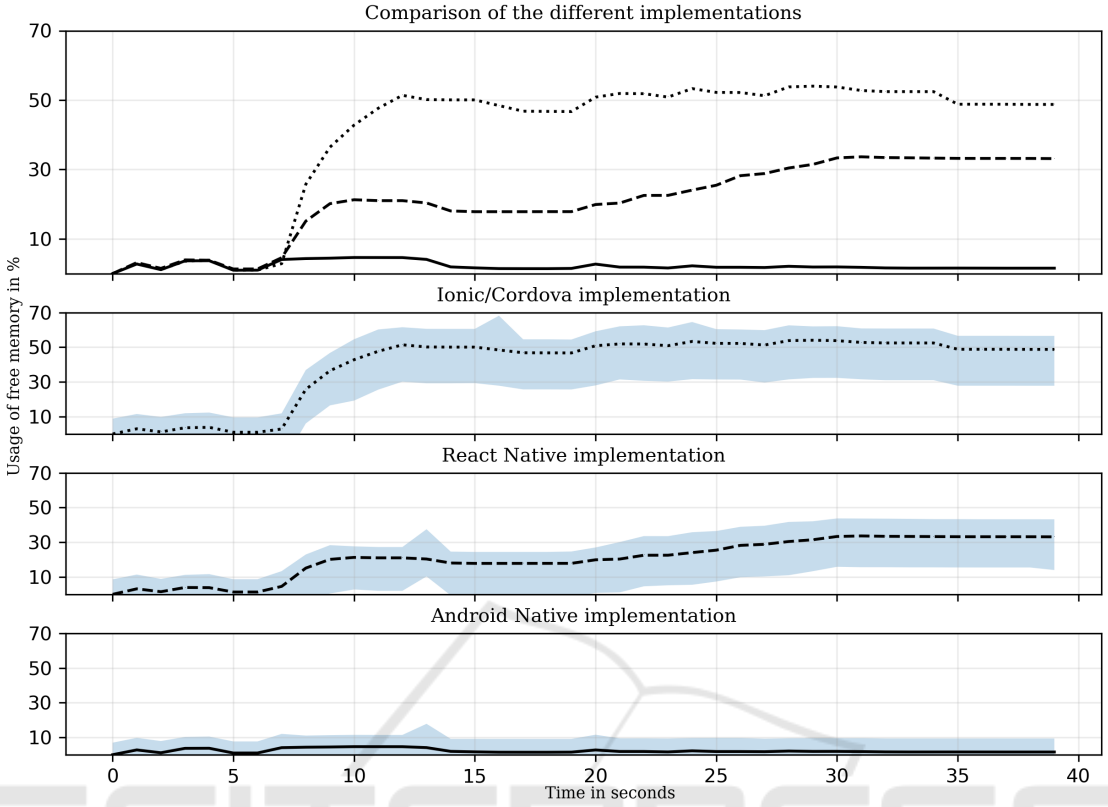

Figure 3: Memory consumption comparison of different apps.

plemented natively to reduce resource consumption. Other parts of an app with lesser importance (e.g., settings menu) can be implemented with MCPD approaches.

A number of caveats need to be noted regarding the present study. First, we only used an Android native implementation to compare the performance results of the two MCPD approaches. A native implementation for Apple iOS was not tested. Such a native implementation could shed more light on performance differences. Likewise, we only tested one typical UI interaction pattern, scrolling through a virtual list. Although this is a frequently found UI pattern, additional UI pattern need to be considered as well. Furthermore, the automated tests were conducted on only one device. Finally, we only considered two approaches for MCPD approaches, hybrid and interpreted. However, there are three other types of approaches Biørn-Hansen et al. (2018). To have an in depth analysis of MCPD approaches, all five categories of approaches should be included. Clearly, all these limitations reduce the generalizability of our results. Nevertheless, these results confirm results of previous studies. They also provide first insights into performance with respect to UI interactions and thus expand the current body of literature. Additionally, the automated test of implemented UI interaction show a novel approach of performance testing.

6 CONCLUSION

In this paper, a performance analysis of three implementations (2 MCPD approaches and 1 native approach) of one and the same UI interaction was conducted. We found that MCPD approaches use more than twice as much CPU than a native implementation when performing a typical UI interaction. Additionally, the memory consumption during the UI interaction was 28-times higher for Ionic/Cordova and around 14-times higher for React Native compared to the native app. These results are in line with results of previous studies. Nevertheless, in this study, we pursued a different approach to test the performance of mobile apps as we focus on interactions with the UI. Previous studies focused mainly on rather computing intensive apps.

The use of MCPD approaches for app development has substantial consequences for CPU usage and memory consumption. Mobile app developers face a difficult decision on the choice of the development ap-

proaches as performance can be a competitive advantage. We conclude that a mixture of a native development approach with a MCPD approach within the same app is plausible. For frequently used parts of an app a native implementation can increase the battery lifetime and the user-perceived performance.

As we only tested two MCPD approaches, future work should extend this research by increasing the number of MCPD approaches for the performance analysis of typical UI interactions. Also an iOS native implementation should be included as well as different UI interaction patterns. This would provide a broader picture of the performance of MCPD approaches.

REFERENCES

Ajayi, O. O., Omotayo, A. A., Orogun, A. O., Omomule, T. G., and Orimoloye, S. M. (2018). Performance evaluation of native and hybrid android applications. *Performance Evaluation*, 7(16).

Andrade, P. R., B.Albuquerque, A., Frota, O. F., Silveira, R. V., and da Silva, F. A. (2015). Cross platform app : A comparative study. *International Journal of Computer Science and Information Technology*, 7(1):33–40.

Biørn-Hansen, A., Grønli, T.-M., and Ghinea, G. (2018). A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. *ACM Computing Surveys (CSUR)*, 51(5):108.

Ciman, M. and Gaggi, O. (2017a). An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, 39:214 – 230.

Ciman, M. and Gaggi, O. (2017b). An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, 39:214–230.

Dalmasso, I., Datta, S. K., Bonnet, C., and Nikaein, N. (2013). Survey, comparison and evaluation of cross platform mobile application development tools. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE.

Dhillon, S. and Mahmoud, Q. H. (2015). An evaluation framework for cross-platform mobile application development tools. *Software: Practice and Experience*, 45(10):1331–1357.

El-Kassas, W. S., Abdullah, B. A., Yousef, A. H., and Wahba, A. M. (2017). Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal*, 8(2):163 – 190.

Facebook (2019). FlatList. https://facebook.github.io/react-native/docs/flatlist.html. [Online; accessed 11-February-2019].

Google (2019a). monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/. [Online; accessed 11-February-2019].

Google (2019b). RecyclerView. https://developer.android.com/reference/android/support/v7/widget/RecyclerView. [Online; accessed 11-February-2019].

Heitkötter, H. and Majchrzak, T. A. (2013). Cross-platform development of business apps with md2. In vom Brocke, J., Hekkala, R., Ram, S., and Rossi, M., editors, *Design Science at the Intersection of Physical and Virtual Design*, pages 405–411, Berlin, Heidelberg. Springer Berlin Heidelberg.

Henry, W. and Fabian, F. (2009). man page for vmstat. https://www.unix.com/man-page/linux/8/vmstat/. [Online; accessed 11-February-2019].

Ionic (2019). ion-virtual-list - Ionic documentation. https://ionicframework.com/docs/api/virtual-scroll. [Online; accessed 11-February-2019].

Latif, M., Lakhrissi, Y., Nfaoui, E. H., and Es-Sbai, N. (2016). Cross platform approach for mobile application development: A survey. In *2016 International Conference on Information Technology for Organizations Development (IT4OD)*. IEEE.

Majchrzak, T. A., Biørn-Hansen, A., and Grønli, T.-M. (2017). Comprehensive analysis of innovative cross-platform app development frameworks. In *Proceedings of the 50th Hawaii International Conference on System Sciences*. Hawaii International Conference on System Sciences.

Mercado, I. T., Munaiah, N., and Meneely, A. (2016). The impact of cross-platform development approaches for mobile applications from the user's perspective. In *Proceedings of the International Workshop on App Market Analytics*, pages 43–49. ACM.

Statista (2018). Global smartphone sales by operating system from 2009 to 2017 (in millions). https://www.statista.com/statistics/263445/global-smartphone-sales-by-operating-system-since-2009/. [Online; accessed 11-February-2019].

Statista (2019). Worldwide mobile app revenues in 2015, 2016 and 2020 (in billion U.S. dollars). https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/. [Online; accessed 11-February-2019].

Vallerio, K. S., Zhong, L., and Jha, N. K. (2006). Energy-efficient graphical user interface design. *IEEE Transactions on Mobile Computing*, 5(7):846–859.

Willocx, M., Vossaert, J., and Naessens, V. (2015). A quantitative assessment of performance in mobile app development tools. In *2015 IEEE International Conference on Mobile Services*, pages 454–461. IEEE.

Willocx, M., Vossaert, J., and Naessens, V. (2016). Comparing performance parameters of mobile app development strategies. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*, pages 38–47. IEEE.

Xanthopoulos, S. and Xinogalos, S. (2013). A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics on - BCI '13*. ACM Press.