Comparing performance parameters of mobile app development strategies

Michiel Willocx, Jan Vossaert, Vincent Naessens
MSEC, iMinds-DistriNet
KU Leuven, Technology Campus Ghent
Gebroeders Desmetstraat 1, 9000 Ghent, Belgium
firstname.lastname@cs.kuleuven.be

ABSTRACT

Mobile cross-platform tools (CPTs) provide an interesting alternative to native development. Cross-platform tools aim at sharing a significant portion of the application codebase between the implementations for the different platforms. This can drastically decrease the development costs of mobile applications. There is, however, some reluctance of mobile application developers to adopt these tools. One of the reasons is that the landscape of CPTs is so diverse that it is hard to select the most suitable CPT to implement a specific application. The contribution of this paper is twofold. First, it presents a performance analysis of a fully functional mobile application implemented with ten cross-platform tools and native for Android, iOS and Windows Phone. The performance tests are executed on a high- and low-end Android and iOS device, and a Windows Phone device. Second, based on the performance analysis, general conclusions of which application developers should be aware when selecting a specific (type of) cross-platform tool are drawn.

Keywords

mobile application development, cross-platform tools, performance analysis, Android, iOS, Windows Phone

1. INTRODUCTION

The smartphone opened up many opportunities to provide new types of services to users. Also, service providers are trying to attract new users and support existing users more efficiently by making their services available via the smartphone. To increase revenue, service providers want to reach as many users as possible with their mobile services. However, making services available on all mobile platforms is very costly due to the fragmentation of the smartphone and tablet market. Although Windows is extending their user base, iOS and Android still remain the biggest players on the market [23].

Developing native applications for each platform drastically increases the development costs. While native applications can fully exploit the features of a particular mobile platform, limited or no code can be shared between the different implementations. Each platform requires dedicated tools and different programming languages (e.g. Objective-C, C# and Java). Also, maintenance (e.g. updates or bug fixes) can be very costly. Hence, application developers are confronted with huge challenges. A promising alternative are mobile cross-platform tools (CPTs). A significant part of the code base is shared between the implementations for the multiple platforms. Moreover, many cross-platform tools use Web-based programming languages to implement the application logic. This facilitates programmers with a Web background to start developing mobile applications.

More than one hundred different CPTs [28] are currently available. Each cross-platform tool relies on specific technologies and programming languages. Selecting the most suitable tool is no sinecure. Even though a considerable amount of scepticism about CPTs exists and although surveys [28] have shown that the overall satisfaction concerning the development process has been low in the past, many tools have become more mature over the last few years. It is, therefore, interesting to see if these tools are able to overcome the scepticism and provide a viable alternative to native development.

Past studies mainly focused on a qualitative analysis and evaluation of CPTs. Amongst others, they give an insight in licensing costs, available support, programming languages and development environments. Although these parameters are certainly important when selecting a suitable CPT, they only address specific concerns during the selection process. The performance of the resulting app on the different platforms can also be a key factor. This is exactly the scope of our study.

Contribution. This paper presents the results of a quantitative performance analysis of cross-platform applications. A set of performance parameters related to mobile application behavior is defined and evaluated using a fully functional application that is implemented using the native development strategies for iOS, Android and Windows Phone and using ten commonly used/promising cross-platform tools. The performance parameters of the implementations are evaluated using a high- and low-end Android and iOS device an a Windows Phone device. Based on the results of the analysis, general conclusions of which application developers should be aware when selecting a specific (type of) cross-platform tool are drawn. The analysis is based on an existing third-party, mobile application, namely `PropertyCross`[1]. This application has been developed using both the native iOS,

MobileSoft'16, May 16-17, 2016, Austin, TX, USA
 ISBN 978-1-4503-4178-3/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897073.2897092

Android and Windows Phone development strategy and the cross-platform development strategy with several CPTs. The scope of this work is to analyse performance parameters related to general application behavior. This behavior does not depend on the specific type or functionality of the application. The analysis is done using a fully functional application implemented by experts in each tool. Hence, the results of the analyses should be representative for applications implemented by the selected tools. More information on the PropertyCross application is given in Section 4.

Initial performance analysis results were published as work in progress in [29]. Two cross-platform implementations were evaluated. This paper extends our previous work in multiple ways. First of all, more tools are tested. This allows us to cover many different cross platform development technologies and draw more representative and extensive conclusions. Secondly, Windows Phone is added to the comparison. Hence, the three most-used mobile platforms (95+% of the smartphone market) are covered by this research. Finally, in-depth guidelines are drawn that may steer the selection of a development strategy for mobile apps.

The rest of this paper is structured as follows. Section 2 points to related work. Section 3 discusses the different types of cross-platform tools. In section 4, the application used for the performance evaluation is presented along with the ten selected CPTs. Section 5 gives an overview of the five devices used for the performance tests. The evaluation criteria and the measurement tools are discussed in section 6. In the next section, the results of the performance tests are presented and discussed. In section 8, general conclusions of which application developers should be aware when selecting a specific (type of) cross-platform tool are drawn. The final section presents the conclusions and point to future work.

2. RELATED WORK

Many existing studies focus on the evaluation and comparison of cross-platform tools based on qualitative properties. Trajkovik et al. [16] give a global overview of the state-of-the-art regarding mobile application development and address major challenges and opportunities for developers. El-Kassas et al. [21] describe a wide array of technologies which can be used to achieve cross-platform applications for mobile devices. Other contributions give a more detailed overview of a small subset of cross-platform tools. For instance, Heitkötter et al. [22] present an in-depth comparison of some cross-platform tools based on several qualitative parameters such as licensing costs, supported platforms, look-and-feel, development environments, maintainability and scalability. They also focus on *user-perceived* application performance. Malavolta et al. [24] focus the end user's perception of hybrid applications by analysing hybrid app reviews in the Google Play Store.

Zibula et al. [31] give a detailed overview of the technologies that are used by different CPTs. Other qualitative analyses are presented in [25, 27] and [18]. Xanthopoulos [30] presents a demo application that is realized with multiple CPTs. Their analysis mainly focuses on the graphical user interface and the user experience. Rahul et al. [26] present a CPT selection strategy based on qualitative properties.

The amount of research providing a quantitative analysis of performance properties is rather limited. Dalmasso et al. [20] built a demo application for Android with a selection of four different cross-platform tools. The demo application was used to measure and evaluate the CPU usage, memory usage and battery consumption. They, however, did not use a native application as a baseline to evaluate the behavior of the CPTs, nor did they evaluate the behavior of the tools on the iOS platform. Ciman et al. [17] focus on the impact of using different cross-platform development approaches on energy consumption. Corral et al. [19] analyse response times when accessing different device features and sensors in a PhoneGap application and compare the results to a native approach.

3. CROSS-PLATFORM DEVELOPMENT STRATEGIES

Cross-platform tools are commonly put in five different categories [28]: JavaScript frameworks, Web-to-native wrappers, runtimes, source code translators and app factories. App factories allow people with no programming experience to make simple applications by using drag-and-drop mechanisms. The available features and customizability are rather limited when using an app factory and they are therefor not further discussed in this paper.

3.1 JavaScript frameworks and web-to-native wrappers

JavaScript frameworks use Web technologies for the development of mobile applications. The user interface of the application is developed with HTML and CSS, and the functionality is implemented using JavaScript. Compared to other JavaScript frameworks, the mobile JavaScript frameworks provide support for implementing user interfaces and navigation patterns specific to mobile applications. These mobile interfaces are designed for smaller screens compared to regular websites and, for instance, provide support for the touch UI of mobile devices. Some frameworks also provide native skins that tailor the UI of the application to the look and feel of the platform on which it is running. These skins, however, do not yet provide a fully native experience. Most JavaScript frameworks also support traditional design patterns such as MVC an MVVM to facilitate the development of well-structured and maintainable code. A major advantage of these tools is that they enable Web developers to participate in mobile application development.

Two distribution strategies can be applied. The most flexible solution is to host the Web app on a Web server. The user can access the Web app in a mobile browser by surfing to the URL of the Web app, regardless of the platform of the user. There are, however, also several disadvantages associated with this strategy. The availability and responsiveness of the application partly depends on the Internet access of the platform. Moreover, the functionality of the application is constrained by the JavaScript API of the browser (e.g. access to sensors such as accelerometer and GPS) and accessing the application is more cumbersome than starting an application installed on the device. A second strategy is to pack the Web app into a standalone application by using a Web-to-native wrapper. The packaging results in an application installer that can be submitted to the app stores of the different platforms. The Web app is no longer run in the browser, but in a chromeless webview which is packed together with the application. The Web-to-native wrapper also provides a JavaScript bridge that enables access to a broader range of platform APIs compared to the browser.

In this paper, all tests are run on packaged Web applications that are installed on the different devices.

3.2 Runtimes and source code translators

Runtimes are compatibility layers which shield applications from underlying platform differences. In most cases, a compilation step translates the source code to a binary or intermediary language that runs on the runtime. In a minority of tools, the source code will run straight on the runtime (e.g. Titanium). Some tools (e.g. NeoMAD) do not rely on a runtime and translate the source code to native source code for the different platforms. For each platform, the resulting source code is compiled using the development tools provided by the platform developer. In most cases, runtimes and source code translators make use of real native UI components, offering a real native experience.

4. TEST APPLICATION AND CPT SELECTION

The performance analysis is based on the PropertyCross application. PropertyCross is a community driven initiative. On the website, a concept application is described and implemented in over 20 cross-platform tools. The PropertyCross application enables users to search for properties that are for sale, based on a city name or on GPS location. Furthermore, the user is able to favorite properties, which are kept in local storage together with the search history. The PropertyCross application is a fully functional application with multiple screens, Web access, GPS location, and local storage.

The advantage of using an existing application instead of making a new one is twofold. First of all, learning tools and implementing applications is very time consuming. Moreover, each implementation available on the PropertyCross website is developed by programmers with experience with the used tool. This results in better code quality and, thus, ensures more representative measurements and conclusions.

For the selection of cross-platform tools for the performance analysis, several criteria were taken into account. First of all, there is the availability of a PropertyCross implementation using the tool. This does not restrict our research, as most well-known and often-used cross-platform tools are supported [28]. Second, there was sought to select a wide range of different technologies, programming languages and development strategies. In total, ten cross-platform tools are selected, containing commonly used cross-platform tools (e.g. Xamarin, Titanium (Appcellerator) and Sencha Touch 2) and several promising tools (e.g. Ionic and NeoMAD). The selection consists of four runtimes/source code translators and six JavaScript frameworks.

4.1 JavaScript frameworks and web-to-native wrappers

PhoneGap [10] is the most well-known and most used Web-to-native wrapper. Many JavaScript frameworks offer automated integration with PhoneGap to generate an installable application for the different platforms. This section further discusses the selected JavaScript frameworks that are all evaluated using PhoneGap as Web-to-native wrapper.

- **Ionic** [5] is an AngularJS-based [2] JavaScript framework. The main focus of Ionic is the look and feel and UI of the application. Furthermore, the use of AngularJS simplifies development by providing MVC support. Ionic also offers native skins, which enhance the user experience of the application.
- **Sencha Touch 2** [12] is one of the most popular JavaScript frameworks. The tool specializes in offering out-of-the-box native-looking UI components. It supports several mobile UI features (e.g. switching from landscape to portrait, typical mobile navigation patterns ...), which also enhances the user experience. Further, Sencha requires the use of the MVC design pattern.
- **jQuery Mobile** [6] is a jQuery-based JavaScript framework which allows developers to create responsive websites.
- **Intel App Framework (IAF)** [4] is also a mobile optimized version of jQuery. On top of the lightweight jQuery library, it also offers a MVC framework, additional UI components and a native look and feel.
- **MGWT** [7] is the mobile version of Google Web Toolkit. MGWT uses a unique approach to develop JavaScript apps. The application is written in Java, and is automatically translated to JavaScript at compile time (cf. GWT). The main difference between MGWT and GWT is the support for specific mobile application features (e.g. mobile widgets and touch navigation). One of the big advantages of this approach is that existing GWT applications can be very easily transformed to mobile applications using MGWT.
- **Famou.us** [3] is an open-source JavaScript framework which replaces the standard browser's DOM-based rendering mechanism by its own rendering engine. The main focus of this framework is to build high-performance web-apps by making use of GPU acceleration to allow 60 FPS.

4.2 Runtimes and source code translators

- **Adobe AIR** [1] stands for Adobe Integrated Runtime. Applications are programmed using HTML, JavaScript, Adobe Flash, ActionScript and optionally also Apache Flex. Adobe Air does not support Windows Phone.
- **NeoMAD** [9] uses source code translation and does not rely on a dedicated runtime environment. The application is implemented using Java. The Java code is then translated into native source code for each platform. The resulting source code is than compiled using the native development tools of each platform.
- **Titanium** (Appcelerator) [13] applications are completely written in JavaScript and make use of a runtime element. Although the strategy looks similar to the JavaScript frameworks combined with a Web-to-native wrapper, there is a clear difference between both technologies. Web-to-native wrappers use a webview element to display the UI. In Titanium, the JavaScript code is completely mapped to native components by the runtime. This means that the user interface actually consists of native UI components, resulting in a real native experience. The mapping, however, does not cover the complete native API. Currently Titanium does not yet support Windows Phone.

- **Xamarin** [15] uses C# as programming language and makes use of the open source Mono framework [8]. Xamarin uses different mechanisms on each of the supported platforms. In Android, the source code is translated to an intermediary language and bundled together with the Mono runtime. The Mono runtime uses JIT-compilation when running the application. In iOS, Xamarin uses AOT (ahead-of-time) compilation. The app is, hence, packaged as a fully compiled native ARM executable, together with the required .NET libraries. In Windows Phone, the source code is translated to an intermediary language that runs on the .NET runtime installed on the Windows Phone platform.

4.3 Native

The performance tests are also run on a native implementation for the different platforms. This provides a reference to which the CPT implementations can be compared. Native Android applications are developed in Java, Windows Phone applications in C# and iOS offers a choice between Objective-C and Swift. The native iOS implementation in this paper is implemented using Objective-C.

5. SELECTION OF DEVICES

In Table 1, an overview of the devices used for the performance analysis is given. For Android and iOS both a high-end and a low-end device was selected. Furthermore, no high-end Windows Phone 10 device was included because at the moment of writing this paper the Windows Phone Developer Power Tools had no Windows Phone 10 support. All devices were reset to the standard factory settings and updated to the most recent operating systems available for the device form the manufacturer. None of the devices were rooted nor jailbroken.

	Low-End	High-End
iOS		
Device	**iPhone 4**	**iPhone 6**
Operating System	iOS 7	iOS 9
RAM Memory	512 MB	1 GB
CPU	1 GHz	Dual-core 1.4 GHz
Android		
Device	**Sony Xperia E3**	**Motorola Nexus 6**
Operating System	Android 4.4.2	Android 6
RAM Memory	1GB	3 GB
CPU	Quad-core 1,2 GHz	Quad-core 2.7 GHz
Windows Phone		
Device	**Nokia Lumia 925**	
Operating System	Windows 8.1	
RAM Memory	1GB	
CPU	Dual-core 1,5 GHz	

Table 1: Devices used for the performance analysis.

6. EVALUATION CRITERIA AND MEASURING TOOLS

This section discusses the parameters measured to asses the performance of the different PropertyCross implementations. The different parameters are defined and the measuring methods are presented.

6.1 Evaluation criteria

This section lists and defines the measured parameters, together with the rationale behind the selection. More specifically, we argue on the relevance of each parameter with respect to the overall performance of an application.

6.1.1 Response times

Generally speaking, response times are an important factor regarding the user experience. This study measures the response times for three different actions: The *launch time* of the application, the *time to load a new page* of the application and the *time to return to the previous page*. The start time is the time it takes to completely start the application (i.e. from tapping the application icon to displaying the main screen of the application). Once the application is launched, it is important for the user to be able to navigate fluently between the different pages of the application. Slow response times will result in a negative user experience. For measuring the response times of page navigation, the *favorite* page of the application was selected. This was the only page, apart from the homepage, not requiring Internet access. Opening this page, hence, does not introduce additional communication overhead.

6.1.2 CPU usage

The CPU usage is the percentage of the total CPU capacity of the device used by the application in the measured time interval. CPU intensive applications may negatively impact other processes running on the device, decreasing user experience. This research considers the overall CPU usage of an application, as well as the CPU usage during two specific actions in the PropertyCross application: launching the application and using the GPS to search and list properties.

6.1.3 Memory usage

The memory usage parameter reflects the amount of RAM memory allocated by the application. Measuring the memory footprint of the application is especially important for low-end devices as the performance of the platform can be significantly degraded if a high percentage of the available RAM is allocated. In the evaluation three distinct measurements of this parameter are considered. The first measurement is the amount of RAM memory used when the app is fully launched. After launching the application, each page of the application was visited in a specified order. The second value is the maximal memory usage measured while going through the application. The final value is the memory usage measured after each page of the application was visited.

6.1.4 Disk Space

Regarding disk space, two different parameters are measured. The first parameter is the disk space taken by the installed application on the device. This is especially important for low- end devices that, typically, have limited persistent memory available. Second, the APK/IPA/XAP size is measured. These are the downloadable installers for the application, respectively for Android, iOS and Windows Phone. The compactness of this installer is important since applications can be installed over a mobile Internet connection.

6.1.5 Battery usage

Battery consumption is important on all mobile devices. Users do not want applications to excessively drain their batteries. Although cross-platform tools cause additional battery usage as a result of the processing overhead, the impact is negligible compared to the overhead introduced by the design/nature of the application. Applications that

Android	iOS	Windows Phone
	Response times	
DDMS	Instruments tool (Time Profiler)	Visual Studio Console
	CPU Usage	
ADB "top"	Instruments tool (CPU Activity)	Windows Phone Developer Power Tools
	Memory Usage	
ADB "dumpsys meminfo"	Instruments tool (Allocations)	Windows Phone Developer Power Tools
	Disk Space	
Visible on device	Visible on device	Visible on device

Table 2: The tools used to measure the performance parameters on each platform.

make intensive use of hardware resources such as wireless communication and sensors will have a higher energy consumption. Battery usage is, hence, not considered in this paper.

6.2 Measuring tools

Table 2 lists the tools used to measure the performance parameters on each platform. Android is a very open platform compared to iOS and Windows Phone. Hence, on Android, the measuring process can be more controlled/fine-grained compared to Windows Phone and iOS where the vendors provide analysis tools that visualize the measured parameters and there is no access to the raw data.

In Android and Windows Phone, the response times were measured by calculating the elapsed time between specific timestamps in the console log of a PC connected to the mobile device. Android and Windows Phone automatically log timestamps when an application is started and when it is fully running. However, for implementations running on PhoneGap a different timestamp was used. The *application running* timestamp for PhoneGap applications was not representative for the total launch time of the application because it is fired when the webview is running. However, after starting the webview, the JavaScript framework and application still need to be loaded. Hence, an additional log message was added to the JavaScript code so the entire launch time could be taken into account. Additional log messages were added to all implementations to measure the response times of page navigation.

In iOS, the Time Profiler feature of the Instruments tool is used for all response time measurements. This tool displays the total execution time of each component of the application. The startup times of the launch components were added to obtain the total launch time. A similar approach was taken for the page navigation response times.

For the memory and CPU usage measurements in Android, a custom Java program[2] was created which periodically executes the `top` and the `dumpsys meminfo` commands via the Android Debug Bridge (ADB) shell. For iOS, the Allocations and CPU Activity modules of the Instruments tool were used. For Windows Phone, the Performance Monitor of the Windows Phone Developer Power Tools was used.

[2]Java code can be found on github.com/MichielWillocx/cpt-performance-measurements.

7. RESULTS AND COMPARISON

This section lists and analyses the results of the performance measurements. All measurements are executed on a high-end (HE) and low-end (LE) device for Android and iOS. A confidence interval of 95% was used when processing the measurements. The tables in this section list the average values of the measurements. All measurements are publicly available[3]. As mentioned in section 4 Adobe Air and Titanium do not support the Windows Phone platform. Hence, no results are available for these tools on Windows Phone.

7.1 Response times

There are a few rules of thumb[4] concerning the response times of an application and the corresponding perceived user experience. Response times under 100 milliseconds will feel instantaneous to the user. Response times up to one (or a few) second(s) are acceptable to the user, if they rarely occur. Delays greater than a few seconds will significantly degrade the user experience.

7.1.1 Launch time

The launch times are listed in Table 3. The table clearly indicates that the overhead introduced by CPTs results in slower launch times compared to native development. For the JavaScript frameworks, the overhead is largely caused by the webview which needs to be started before the actual JavaScript application can be loaded. Between the different JavaScript frameworks, minor differences in the launch time of the application can be noticed. Overall, the implementation with Sencha Touch 2 is the slowest to start. The relative order of the other JavaScript frameworks is dependent on the platform on which the application is running. For instance, jQuery mobile has the best launch time on iOS while it performs much worse on Android and Windows Phone. Although less prominent compared to the webview for JavaScript framework implementations, the runtime is for the largest part responsible for the launch time overhead of the CPT implementations in the *runtime* category.

NeoMAD is a source code translator that doesn't use a runtime but translates the source code of the application to dedicated source code for the different platforms. Consequently, it is expected that the launch time of the NeoMAD implementation will closely resemble the native launch times. This is the case for the Android and iOS implementation but not for the Windows Phone implementation.

	Android		iOS		Windows
	HE	*LE*	*HE*	*LE*	*HE*
Native implementation for each platform					
Native	293	460	191	611	876
JavaScript Frameworks					
Famo.us	1282	1980	438	1495	2252
Intel App Framework	1009	1383	537	1806	1500
Ionic	1225	1810	731	2762	1750
jQuery Mobile	1790	2515	424	1223	2501
Mgwt	1186	1433	503	1789	2000
Sencha Touch 2	2434	2858	758	2967	2516
Source code translators and runtimes					
Adobe AIR	1364	2782	1191	5568	n/a
NeoMAD	392	500	285	805	3144
Titanium	820	1547	331	1152	n/a
Xamarin	890	1177	347	1383	1001

Table 3: Launch times in ms.

[3]The raw measurements can be found on github.com/MichielWillocx/cpt-performance-measurements.

[4]http://blog.teamtreehouse.com/perceived-performance

7.1.2 Time to navigate between pages of the application

This section analyzes the response times for navigating to a page (i.e. by tapping a button on the screen) and returning to a page (i.e. by using the *back* navigation item). The measurements are respectively contained in Table 4 and Table 5.

As illustrated in Table 4, the implementations using the JavaScript frameworks (e.g. Famo.us, Mgwt) generally achieve good response times of under 100ms. Notable exceptions are Sencha Touch 2 and jQuery mobile. This is caused by how the different JavaScript frameworks handle page switching. While most JavaScript frameworks use the traditional *window.push(page)* and *window.pop()* functions to navigate between pages, Sencha Touch 2 and jQuery provide custom ways to handle this, introducing additional overhead. However, once the page is created it is retained in memory and subsequent visits to the page have response times similar to the other JavaScript implementations. The implementations from the *Runtime and Source Code Translator* category also have response times similar to native applications. Noticeable is that the implementation using Xamarin performs worse on Android than on iOS compared to the native version.

Some JavaScript implementations have better response times compared to the native version. This can attributed to the fast rendering of a new HTML page in the webview compared to the lifecycle management that occurs when switching between views in native applications.

As illustrated in Table 5, response times when returning to the previous page are generally better than when opening a new page. Apart from the Sencha Touch 2 version on Windows Phone and the Xamarin version on the low-end Android device, all response times are below 100ms. Hence, users will experience these page transition as instantaneous.

	Android		iOS		Windows
	HE	*LE*	*HE*	*LE*	*HE*
Native implementation for each platform					
Native	91	109	28	43	129
JavaScript Frameworks					
Famo.us	26	20	4	12	31
Intel App Framework	49	50	32	50	75
Ionic	80	90	39	119	144
jQuery Mobile	100	113	71	295	296
Mgwt	38	40	15	26	53
Sencha Touch 2	267	425	135	492	821
Source code translators and runtimes					
Adobe AIR	4	5	10	8	n/a
NeoMAD	111	126	21	23	76
Titanium	113	198	45	163	n/a
Xamarin	207	216	36	54	75

Table 4: Time to open a new page (in ms).

7.2 Memory consumption

Memory consumption was measured while visiting each page of the application in a specified order (i.e. homepage → search on GPS location → get detail of location → add location to favorites → return to homepage → display favorites → remove property from favorites → return to homepage). Three values are considered in the evaluation: the memory consumption right after starting the application, the memory consumption after going trough all the functions of the application and the highest memory consumption measured while going through the application. The measured values are respectively shown in Table 6, Table 7 and Table 8.

	Android		iOS		Windows
	HE	*LE*	*HE*	*LE*	*HE*
Native implementation for each platform					
Native	12	20	2	8	12
JavaScript Frameworks					
Famo.us	12	20	19	26	16
Intel App Framework	29	30	16	31	52
Ionic	43	60	4	4	63
jQuery Mobile	1	3	1	3	77
Mgwt	21	20	19	25	27
Sencha Touch 2	1	1	1	1	337
Source code translators and runtimes					
Adobe AIR	1	1	8	7	n/a
NeoMAD	8	10	1	5	10
Titanium	1	3	2	7	n/a
Xamarin	40	244	3	10	12

Table 5: Time to return to previous page (homepage) in ms.

The tables clearly show differences in memory management strategies employed by the different platforms. Android and Windows Phone 8 allocate significantly more RAM for each implementation compared to iOS. Furthermore, Android application allocate significantly more memory on devices with more available RAM. In iOS, this is also the case for native applications and applications which use a runtime or source code translator, but not for the JavaScript frameworks. JavaScript frameworks use more memory on low-end devices in iOS. This is again the result of the difference in webview and JavaScript engine between the two iOS versions.

Overall the native implementations have a consistently low memory requirement. The implementations with runtime/source code translator tools have a little higher, but similar, memory usage to the native implementations, with the exception of the implementation with Adobe AIR that overall has a much higher memory footprint. Overall, NeoMAD has a near native memory requirement, followed by Xamarin and Titanium. On four of the five tested devices, the implementations using the JavaScript frameworks have, by far, the highest memory requirement. This is, however, not the case on the high-end iOS device. On that device, these implementations have the lowest memory requirement of all tested implementations. This can, most likely, be attributed to the different webview and JavaScipt engines used on the different platforms. iOS 7 uses UIWebView and JavaScriptCore as JavaScript engine, while iOS 9 makes use of the WKWebview and the more recent Nitro JavaScript engine. Hence, the performance of the different JavaScript implementations can differ based on the platform and even the platform version on which it is running. As can be derived from Table 7, the used memory increases proportionally to the memory allocated upon startup.

As can be derived from Table 8 and Table 7, the effect of garbage collection is generally much smaller for the implementations using CPTs compared to the native implementations. There are, however, some exceptions. The garbage collection behavior for the NeoMAD implementation is similar to the native implementation. Famou.us releases around 2MB memory once it reaches the threshold of 13MB. The other implementations generally only increased in memory usage, except for some rare momentary peaks in memory usage.

7.3 CPU usage

CPU usage represents the part of the total available CPU time spent by an application during a certain time interval.

	Android		iOS		Windows
	HE	*LE*	*HE*	*LE*	*HE*
Native implementation for each platform					
Native	72.62	9.80	11.62	2.88	19.20
JavaScript Frameworks					
Famo.us	113.65	33.68	7.65	10.54	43.84
Intel App Framework	114.61	28.33	7.23	8.48	39.25
Ionic	119.45	32.38	7.18	12.82	48.35
jQuery Mobile	141.11	34.74	7.24	13.28	44.13
Mgwt	117.12	32.15	6.61	10.70	46.44
Sencha Touch 2	161.61	42.73	7.20	17.32	52.14
Source code translators and runtimes					
Adobe AIR	57.55	43.87	199.49	68.60	n/a
NeoMAD	79.22	11.77	8.10	2.95	19.80
Titanium	89.58	26.66	12.92	6.45	n/a
Xamarin	81.99	17.77	9.29	4.16	19,84

Table 6: Memory consumption after launching the application (in MB).

	Android		iOS		Windows
	HE	*LE*	*HE*	*LE*	*HE*
Native implementation for each platform					
Native	80.11	15.63	12.83	4.31	33.03
JavaScript Frameworks					
Famo.us	137.84	46.11	9.30	13.58	65.20
Intel App Framework	123.73	32.61	10.09	10.82	42.86
Ionic	153.66	40.80	9.93	16.96	60.71
jQuery Mobile	161.24	41.99	10.91	17.24	65.27
Mgwt	137.95	40.81	11.11	15.18	55.08
Sencha Touch 2	178.60	56.99	10.19	22.51	67.11
Source code translators and runtimes					
Adobe AIR	98.82	71.85	242.04	86.83	n/a
NeoMAD	88.46	19.31	10.04	4.78	30.61
Titanium	102.41	35.47	16.56	12.01	n/a
Xamarin	96.51	29.92	15.04	9.78	34.60

Table 7: Memory consumption after using the application (in MB).

	Android		iOS		Windows
	HE	*LE*	*HE*	*LE*	*HE*
Native implementation for each platform					
Native	98.68	15.72	16.12	6.91	53.72
JavaScript Frameworks					
Famo.us	159.66	46.88	11.53	21.04	69.45
Intel App Framework	134.53	32.72	10.47	11.21	43.28
Ionic	160.58	40.82	9.97	18.81	61.52
jQuery Mobile	161.25	42.00	11.11	19.18	67.13
Mgwt	137.96	42.57	11.42	15.92	64.06
Sencha Touch 2	193.68	57.07	10.41	25.00	68.40
Source code translators and runtimes					
Adobe AIR	104.30	71.93	242.58	87.51	n/a
NeoMAD	126.80	19.32	15.45	9.06	42.93
Titanium	113.84	35.95	20.09	14.31	n/a
Xamarin	121.28	30.11	15.68	10.22	46,66

Table 8: Peak memory consumption while using the application (in MB).

Displaying CPU usage in tables as numerical values is not a trivial operation. In order to draw conclusions regarding the CPU use of specific actions in the mobile application, the sample time for the measurement intervals must be sufficiently small. This paper only analyzes the CPU behavior of the CPT implementations on the Android platform as the iOS and Windows Phone 8 measurement tools do not allow sufficiently small sample intervals. The results for Android can be found in Table 9. This table contains the CPU usage during two different actions in the life cycle of the application: during the launch of the application and while searching for properties with the GPS. It shows that native applications require the least CPU time and PhoneGap-based applications have the highest CPU requirement. Also, for every tested implementation, launching the application is more CPU intensive than searching and displaying properties. The online results also contain measurements for iOS and Windows Phone. Although these results are not as fine-grained as on Android, similar behavior can be observed on these platforms.

When an application is idle, most cross-platform tools will not use the CPU. There are, however, some exceptions. Famo.us is a cross-platform tool specialized in graphically demanding applications such as games. Games and other applications with advanced graphical components require an engine capable of generating fluently moving components. Therefore, Famo.us will aim to refresh the screen every 17 milliseconds in order to achieve 60 FPS. Hence, Famo.us applications will constantly require 5-15% CPU load. Our measurements show that on iOS, two other tools also required a constant CPU load, namely jQuery Mobile and Adobe AIR.

	On launch		GPS search	
	HE	*LE*	*HE*	*LE*
Native implementation for each platform				
Native	8	15.8	7.74	5.58
JavaScript Frameworks				
Famo.us	21.62	22.95	16.42	18.53
IAF	25.8	21.37	15.22	15.6
Ionic	23.81	24.94	20.75	16.83
jQuery Mobile	25.78	25.52	18.93	19.77
Mgwt	16.23	23.47	18.09	14.66
Sencha Touch 2	22.67	25.31	21.11	26.95
Source code translators and runtimes				
Adobe AIR	17.51	21.52	16.08	16.29
NeoMAD	11.8	10.5	9.13	12.77
Titanium	13.8	17.32	11.85	13.01
Xamarin	19.79	23.44	4.88	11.37

Table 9: CPU usage in Android during lauch and while searching for properties via GPS (in %).

7.4 Disk space

The sizes of the installers are displayed in Table 11, and the sizes of the installed applications on the different devices are displayed in Table 10. For both the installers and the installed applications, the native applications consume the least amount of persistent memory. This is mainly attributable to the extra components that need to be packed with the application (e.g. webview and runtime). Hence, the difference in size between a cross-platform and a native implementation will not increase linearly when more functionality is added to the application. This can be derived from Table 12 that lists the sizes of *empty applications* made in the different cross-platform tools. These *empty applications* are basic applications which only contain a blank page and have no functionality.

With Adobe AIR on Android, the programmer can choose whether or not to bundle the runtime with the application. If he chooses not to, a separate app containing the runtime must be downloaded from the app store. The advantage is that the APK size decreases and the runtime can be used by multiple applications. If the runtime is not bundled with the application, the size of the installer decreases drastically to 1,06MB. The separate runtime installer takes 18,9MB, the installed runtime takes 40,86MB.

With Xamarin, the difference in approach for Android and iOS is clearly visible. For Android, a runtime element is included in the application (JIT compilation), while in iOS the source code is directly compiled to an ARM executable (AOT compilation). Hence, the Android installer of the Xamarin implementation consumes considerable more disk space than the iOS version.

	Android		iOS		Windows
	HE	*LE*	*HE*	*LE*	*HE*
Native implementation for each platform					
Native	0,64	1,42	0,85	0,96	2,89
JavaScript Frameworks					
Famo.us	3,06	3,2	4,5	4,7	5,87
IAF	2,74	2,88	3	3,1	2,28
Ionic	4,3	4,56	8	8,2	7,29
jQuery Mobile	3,18	3,32	4,5	5	3,94
Mgwt	4,37	4,51	12,2	12,8	11,5
Sencha Touch 2	3,89	4,03	5,5	6	4,07
Source code translators and runtimes					
Adobe AIR	34,99	35,84	46	46,2	n/a
NeoMAD	2,76	8,18	9,8	10	5,73
Titanium	14,4	16,98	12,8	13,2	n/a
Xamarin	11,48	11,48	10,8	10,9	2,83

Table 10: Size of the application when installed on the device (in MB).

	Android	iOS	Windows Phone
Native implementation for each platform			
Native	0,64	0,73	0,49
JavaScript Frameworks			
Famo.us	3,05	2,62	1,44
IAF	2,73	2,24	1,13
Ionic	4,15	3,99	2,44
jQuery Mobile	3,17	3,06	1,56
Mgwt	4,36	4,25	2,77
Sencha Touch 2	3,88	3,78	2,07
Source code translators and runtimes			
Adobe AIR	13,00	—	n/a
NeoMAD	2,75	3,43	1,27
Titanium	8,75	6,06	n/a
Xamarin	8,61	3,83	0,47

Table 11: Size of the APK/IPA/XAP installer files (in MB).

	Android	iOS	Windows
Native	0,02	0,07	0,07
Phonegap (Webview)	1,7	1,17	0,14
Titanium (Runtime)	8,45	10,61	n/a
Adobe AIR (Runtime)	12,50	—	n/a
Xamarin (Runtime)	8,61	2,91	—

Table 12: Sizes of "empty" applications (in MB).

8. DISCUSSION

In this section general conclusions of which application developers should be aware when selecting a specific (type of) cross-platform tool are drawn. The first subsection discusses several observations based on the performance analyses conducted in this paper. However, the selection of a CPT also depends on several other non-performance related parameters. Several important non-performance related parameters are discussed in the second subsection.

8.1 CPT selection based on performance parameters

This section discusses several performance-related observations that developers should keep in mind when selecting a cross-platform tool and developing an application.

- Cross-platform tools of the same category show similar behaviour.
  - JavaScript frameworks are the most CPU intensive cross-platform technology and consume more memory than native applications, especially on Android and Windows Phone devices. They also show the slowest launch times. Nevertheless, once the application has launched, the response times while navigating in the application are generally similar to native response times.
  - Runtimes are characterized by their increased disk space and RAM usage compared to native implementations. This is caused by the runtime which is packed together with the application. The use of a runtime also increases the launch time of an application. Once the application has launched, the difference in response time when navigating trough pages of the application compared to native is less prominent.
  - Some cross-platform tools do not use a runtime or a webview and rely solely on source code translation. As can be derived from the results of the NeoMAD tool in the previous section, this approach can result in applications with overall near-native performance.
- The performance penalty resulting from the use of cross-platform tools is generally acceptable, especially on high-end devices.
- The performance of a cross-platform application strongly depends on the targeted platform (version). For instance, the version of the JavaScript engine/webview can have a significant impact on the performance of applications implemented with JavaScript frameworks. Some tools use a different strategy depending on which platform it is running. For instance Xamarin uses JIT for Android applications and AOT for iOS applications. Note that this can also apply to native applications. For instance earlier Android versions use Dalvik (JIT) while more recent versions use ART (AOT).
- The rendering of JavaScript framework applications results in fast response times. However, even though JavaScript frameworks offer native skins, the user interface still consists of HTML components that do not provide the same user experience compared to real native components. Most runtimes and source code translator allow the use of native UI components.
- Although JavaScript frameworks generally have fast response times, there are some exceptions such as Sencha Touch 2 and jQuery Mobile with respectively slow and moderate response times. After the application is launched, each first visit to the different pages of the application is slower than subsequent visits to the same page. For applications that consist of a limited number of pages, the impact on the user experience will be limited. However, applications consisting of many different pages might benefit from using CPTs with faster response times.

8.2 CPT selection based on non-performance parameters

In order to select a suitable cross-platform tool, several criteria other than performance have to be considered. Some of these parameters are often discussed (e.g. license cost, development environment/support, maturity and stability). Below, several other important parameters are discussed.

- **Existing infrastructure.** Developing new applications from scratch is time consuming. In many cases, existing code for other projects can be reused. For instance, existing GWT projects are easily converted

to MGWT projects. JavaScript frameworks allow the conversion of existing Web applications to mobile apps.

- **Skills of the developer.** Specific (types of) cross-platform tools may be selected based on the background of the developer. For instance, JavaScript frameworks and tools such as Titanium and ReactNative allow Web developers to use their extensive knowledge of Web technologies to develop mobile apps.

  Developers with experience in C# or Java can also select a cross-platform tool that matches with their programming background. For instance, NeoMAD and MGWT use Java, while Xamarin uses C#. Traditional programmers are often inclined to use runtimes and source code translators.

- **Type of application.** Some cross-platform tools specialize in the development a specific type of mobile application. For instance, Famo.us and Unity [14] provide specific support for custom graphical components and animated graphics, as is a key requirement in game development.

- **Platform-specific code.** Some cross-platform tools (e.g. JavaScript frameworks) use a 100% shared code base for all targeted platforms. Therefore, not all native features can be accessed. These tools are, hence, not suitable for the development of applications with very specific requirements regarding user interfaces and hardware access.

  Other tools (e.g. NeoMAD and Titanium) provide a uniform interface for the development of user interfaces and access to hardware resources such as sensors and persistent storage but also allow adding platform-specific code to access otherwise unreachable native APIs. This increases the engineering effort but allows realizing very specific requirements (e.g. applications developed for external clients) regarding user interfaces and access to hardware resources.

  Tools such as Xamarin[5] and ReactNative [11] focus on the shared implementation of the business logic. The hardware access and user interfaces are developed using platform-specific interfaces. The main advantage of this approach is that all native APIs can be accessed trough a single programming language, and the entire implementation for the different platforms is integrated in a single project.

9. CONCLUSION

This paper presented an in-depth analysis of the overall behavioural performance of a mobile application implemented with ten promising/commonly used cross-platform tools, and with native development tools of the three most prominent mobile platforms (i.e. Android, iOS and Windows Phone). The performance was evaluated on five different devices, two Android and iOS devices and a Windows Phone device. Based on the performance analysis, general conclusions of which application developers should be aware when selecting a specific (type of) cross-platform tool are drawn.

[5]Xamarin is also working on libraries that provide uniform interfaces for UI (Xamarin.Forms) and hardware access (Xamarin.Mobile).

This paper focused on the overall behavioural performance of applications implemented with different cross-platform tools. However, the use of cross-platform tools can have an impact on specific functional aspects of mobile applications. Hence, future work will consist of investigating these aspects by measuring the overhead related to, for instance, executing complex algorithms, accessing device resources such as sensors or persistent memory and communication with other devices.

10. ACKNOWLEDGMENT

We would like to show our gratitude to Nicolas Quartier who contributed many valuable measurements and results to this paper in the context of his master thesis.

11. REFERENCES

[1] Adobe air website. http://www.adobe.com/nl/products/air.html.
[2] Angularjs website. https://angularjs.org/.
[3] Famo.us website. http://famous.org/.
[4] Intel app framework website. https://app-framework-software.intel.com/.
[5] Ionic website. https://ionicframework.com/.
[6] jquery mobile website. https://jquerymobile.com/.
[7] mgwt website. http://www.m-gwt.com/.
[8] Mono framework website. http://www.mono-project.com/.
[9] Neomad website. http://www.neomades.com/en/.
[10] Phonegap website. https://phonegap.com/.
[11] Reactnative website. https://facebook.github.io/react-native/.
[12] Sencha touch 2 website. https://www.sencha.com/products/touch/#overview.
[13] Titanium website. http://www.appcelerator.org/.
[14] Unity website. https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/mobile-development.
[15] Xamarin website. https://xamarin.com/.
[16] S. Amatya and A. Kurti. Cross-platform mobile development: Challenges and opportunities. In V. Trajkovik and M. Anastas, editors, *ICT Innovations 2013*, volume 231 of *Advances in Intelligent Systems and Computing*, pages 219–229. Springer International Publishing, 2014.
[17] M. Ciman and O. Gaggi. Evaluating impact of cross-platform frameworks in energy consumption of mobile applications. In V. Monfort and K. Krempels, editors, *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies, Volume 1, Barcelona, Spain, 3-5 April, 2014*, pages 423–431. SciTePress, 2014.
[18] M. Ciman, O. Gaggi, and N. Gonzo. Cross-platform mobile development: A study on apps with animations. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 757–759, New York, NY, USA, 2014. ACM.
[19] L. Corral, A. Sillitti, G. Succi, A. Garibbo, and P. Ramella. Evolution of mobile software development from platform-specific to web-based multiplatform paradigm. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and*

Reflections on Programming and Software, Onward! 2011, pages 181–183, New York, NY, USA, 2011. ACM.

[20] I. Dalmasso, S. Datta, C. Bonnet, and N. Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 323–328, July 2013.

[21] W. S. El-Kassas, B. A. Abdullah, A. H. Yousef, and A. M. Wahba. Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal*, pages –, 2015.

[22] H. Heitkötter, S. Hanschke, and T. Majchrzak. Evaluating cross-platform development approaches for mobile applications. In J. Cordeiro and K. Krempels, editors, *Web Information Systems and Technologies. 8^{th} International Conference, WEBIST 2012, Porto, Portugal, April 18-21, 2012, Revised Selected Papers*, volume 140 of *Lecture Notes in Business Information Processing (LNBIP)*, pages 120–138. Springer, Berlin Heidelberg, 2013.

[23] International Data Corporation (IDC). Smartphone os market share, q4 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015.

[24] I. Malavolta, S. Ruberto, T. Soru, and V. Terragni. End users' perception of hybrid mobile apps in the google play store. In *IEEE MS*, pages 25–32. IEEE, 2015.

[25] M. Palmieri, I. Singh, and A. Cicchetti. Comparison of cross-platform mobile development tools. In *Intelligence in Next Generation Networks (ICIN), 2012 16th International Conference on*, pages 179–186, Oct 2012.

[26] R. Raj and S. Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *India Conference (INDICON), 2012 Annual IEEE*, pages 625–629, Dec 2012.

[27] A. Ribeiro and A. R. da Silva. Survey on cross-platforms and languages for mobile apps. In *Proceedings of the 2012 Eighth International Conference on the Quality of Information and Communications Technology*, QUATIC '12, pages 255–260, Washington, DC, USA, 2012. IEEE Computer Society.

[28] Vision Mobile. Cross-platform developer tools 2012: Bridging the worlds of mobile apps and the web. *February*, 2012.

[29] M. Willocx, J. Vossaert, and V. Naessens. A quantitative assessment of performance in mobile app development tools. In *2015 IEEE International Conference on Mobile Services, MS 2015, New York City, NY, USA, June 27 - July 2, 2015*, pages 454–461, 2015.

[30] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 213–220, New York, NY, USA, 2013. ACM.

[31] A. Zibula and T. A. Majchrzak. Developing a cross-platform mobile smart meter application using html5, jquery mobile and phonegap. In K.-H. Krempels and J. Cordeiro, editors, *WEBIST*, pages 13–23. SciTePress, 2012.