A Quantitative Assessment of Performance in Mobile App Development Tools

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*—The popularity and maturity of mobile cross-platform tools increased considerably during the last years. They can be used to increase the efficiency during the development cycle of mobile applications. Although it is common knowledge that cross-platform tools come with a performance penalty, the size of the overhead is unclear. Our study targets the assessment of two representative cross-platform tools. This paper presents a quantitative performance analysis that is applied to a non-trivial, multi-screen app. Relevant performance properties on both low-end and high-end devices are evaluated and compared to a native Android and iOS implementation. We further reflect about the impact of the results on the user acceptance and experience, and define guidelines for the selection of a particular tool.**

***Keywords*-mobile applications, cross-platform tools, performance analysis**

I. 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 [1]. 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 [2] 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 [2] 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 that is applied to an existing third-party, mobile application, namely `PropertyCross`[1]. This application has been developed with both native iOS and Android as well as with several CPTs, enabling a good basis for comparison between CPTs. For the evaluation, several performance parameters are defined and measured for both the native and two CPT versions of the application. The analysis shows significant performance differences between the different implementations. This may have an impact on the selection of a particular development strategy. The evaluation further discusses the factors that may cause those deviations. The performance analysis was performed on both high-end and low-end iOS and Android devices.

The rest of this paper is structured as follows. Section 2 points to related work. Section 3 classifies cross-platform technologies according to multiple categories after which two CPTs are selected, each belonging to a different category. This section also gives an overview of the devices used during the performance analysis. Section 4 presents the evaluation criteria and the methodology and tools used to measure the different criteria. In Section 5, the `PropertyCross` app is introduced. In the next section,

[1]http://www.propertycross.com

the test results are discussed in more detail, followed by an evaluation and reflection in section 7. The final section presents the conclusions and point to future work.

II. Related Work

Many existing studies focus on the evaluation and comparison of cross-platform tools based on qualitative properties. Trajkovik et al. [3] give a global overview of the state-of-the-art regarding mobile application development and address major challenges and opportunities for developers. Other contributions give a more detailed overview of a small subset of cross-platform tools. For instance, Heitkötter et al. [4] 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. Zibula et al. [5] give a detailed overview of the technologies that are used by different CPTs. Other qualitative analyses are presented in [6], [7] and [8]. Xanthopoulos [9] 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. [10] 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. [11] 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. [12] focus on the impact on energy consumption of using different cross-platform development approaches.

This paper focuses on an in-depth comparison of performance properties based on a quantitative analysis. The contribution compared to existing work is threefold. First, more parameters are measured. Apart from CPU, memory and battery usage, multiple response times are also measured as these have a significant impact on the user experience. Second, the CPT results are compared to native implementations, namely an Android and iOS version. To ensure a substantial part of the smartphone market is covered, two iOS and two Android devices of a different price range are selected as a representative for each platform.

III. Development Strategies and Devices

This section presents the scope of the experiments. First, CPTs are classified. Second, a representative CPT for two major categories is selected. Finally, the set of devices selected for the quantitative performance analysis is presented.

A. Cross-platform tool classification

CPTs are often classified according to five categories [2], namely Web app tools, Web-to-native wrappers, runtimes, source code translators and app factories.

Web apps are websites that are optimized to run on mobile platforms. JavaScript frameworks like Ionic, AngularJS and JQuery are used. This approach has multiple constraints. First, Web apps run in a standard Web browser. They are accessed via an URL just like a normal website and don't have a native look and feel. Moreover, a mobile Web application can only access a limited set of context information (e.g. geolocation). Context information that is available in some browser (e.g. accelerometer and camera) is not available in others. Second, an Internet connection is required to access the application. If the application consists of multiple Web pages, the responsiveness of the application greatly depends on the quality of the Internet connection.

Web-To-Native Wrappers are the most popular category. Similar to Web apps, Web languages like HTML5, JavaScript and CSS are used. Hence, programmers as well as Web developers can participate in mobile app development. Unlike Web apps, Web-to-native wrappers do not run in Web browsers but as a stand-alone app. This increases the user experience. When the application is launched, a chromeless webview starts in which the app runs. This strategy typically supports a wider range of native API calls compared to a normal Web browser. Additional components can be loaded to increase the amount of API calls. Cordova/PhoneGap is the most popular representative in this category.

A *runtime* is a cross-platform compatibility layer on top of a native operating system that shields the app from the underlying differences between platforms. Depending on the specific tool, the source code is either compiled or interpreted by the runtime during execution. Major representatives are Titanium Appcelerator, Adobe Air and Adobe Flex.

Source Code Translators cross-compile the source code to run on different platforms. Multiple translation strategies can be used. The source code can for instance be translated to the platform's native language or to executable byte code. Source code translators can be combined with a runtime element if the source code is translated to code that can be executed by the runtime environment. Popular examples of source code translators are Xamarin and Qt.

App Factories use visual drag-and-drop design. No code needs to be written by the developer. Hence, people without any IT background can create their own applications. Apps that are realized by app factories are, however, often small and have limited functionality. A prototypical tool in this category is AppMkr.

B. Cross-platform tool selection

For our study, a representative is selected in two major categories. The selection was inspired by the market share, developer reviews, and a poll that was sent out to 22 app developers in our industrial network. A majority of them were SMEs. Besides the cross-platform implementations, all tests are also executed on a native Android and a native iOS implementation of the same application. Hence, the performance penalty of each selected CPT can be compared to a native implementation.

PhoneGap is by far the most well known Web-to-native wrapper, and is often [2] stated as the most widely used CPT. The Web part of the application was developed using *Ionic* [3]. This is a JavaScript framework, built as an extension to the AngularJS framework, that is also often used to develop *Web apps*. It is still a very young framework. The alpha version was released in November 2013. Nevertheless, Ionic is often seen as one of the most promising JavaScript Frameworks. Its major strengths are the attractive look-and-feel and the user interaction which comes close to native applications. The Ionic code interfaces with the native features of the platform by running it in a PhoneGap webview.

Xamarin is a source code translator. Xamarin applications are developed in C#. Hence, the tool could attract many C# programmers who can immediately start developing mobile apps without te need to learn new programming languages and technologies. According to the Xamarin website, there are 900 000 active developers using their tool to develop cross-platform applications. Xamarin translates the C# source code to platform-specific native code. The translation is different for iOS and Android. In iOS, an *ahead-of-time* (AOT) compilation is applied. In this case, native assembly code is generated. In Android, the Xamarin compiler generates an *Intermediate Language*, which is compiled *just-in-time* (JIT) to native assembly (i.e. when the application launches). Xamarin maps most of the controls and layouts to their native counterpart which results in native user interfaces. Not only will this increase the performance, the app will also have a native feeling. An major disadvantage of Xamarin is the amount of platform specific code that must be developed. A big chunk of the code, including all interface code, is not shared between the different platforms.

C. Device selection

Table 1 gives an overview of the devices that are selected for the performance tests. Most applications should have acceptable performance on both high-end and low-end devices. Hence, for both Android and iOS, a high-end and low-end device was selected for the performance tests. Before running the tests, the devices were reset to their standard factory configuration. No devices were jailbroken nor rooted.

[2]http://www.developereconomics.com/pros-cons-top-5-cross-platform-tools/
[3]http://www.ionicframework.com

	Low-End	**High-End**
iOS		
Device	**iPhone 4**	**iPhone 6**
Operating System	iOS 7	iOS 8
RAM Memory	512 MB	1 GB
CPU	1 GHz	Dual-core 1.4 GHz
Android		
Device	**Acer Liquid E330**	**Motorola Nexus 6**
Operating System	Android 4.0.4	Android 5.0.1
RAM Memory	512 MB	3 GB
CPU	1 GHz	Quad-core 2.7 GHz

Table I
DEVICES THAT ARE SELECTED FOR THE PERFORMANCE ASSESSMENT.

IV. EVALUATION CRITERIA AND MEASURING TOOLS

This section first gives an overview of the parameters that are measured. Thereafter, an overview of the tools and methods that were used to perform the measurements is given.

A. Evaluation criteria

The analysis is based on a range of different performance parameters that is measured and evaluated. This section lists and defines the measured parameters, together with a rationale behind the selection. More specifically, we argue on the relevance of each parameter with respect to the overall performance of an application. Note that all tests are executed on release builds of the application in order to avoid any unnecessary overhead introduced by debug builds.

Response times are an important factor regarding the user experience. This study measures the response times for four different actions, namely starting the application, pausing the application, resuming the application, and navigating to another page of the application.

The start time is the time it takes to completely start the application (i.e. from tapping the application icon to displaying the first screen of the application). The resume time is the time it takes to move the app from the background to the foreground, and vice versa for the pause times.

The *memory usage* indicates the amount of RAM memory allocated by the application. Measuring the app memory footprint is especially important for low-end devices. This parameter is measured at different points in the lifecycle of the application. It is first measured when the app is fully launched. A second measurement is performed after moving the application to the background. This distinction is made since the memory consumption in both states tends to differ. An application in the background generally uses less RAM memory. The memory usage is also monitored after a specified set of features of the application is used. This gives an indication of the amount of memory allocated by the application when in use.

The *CPU usage* is the percentage of the total CPU capacity of the device used by the application in a specified time interval. CPU intensive applications may negatively impact other processes running on the device, decreasing user experience. For our evaluation, the CPU usage during the start of the application is measured. This provides an interesting benchmark to compare the different cross-platform tools. Cross-platform tools introduce additional overhead during the start of the application (e.g. launching a runtime and loading a webview).

Regarding *disk space*, two different parameters are measured. First, the space taken by the installed application on the device is measured. This is especially important for low-end devices, with limited resources. Second, the `apk/ipa` size is measured. These are the downloadable installers for the application, for Android and iOS respectively. The compactness of this installer is relevant to users who download applications over a mobile Internet connection.

Battery consumption is important on all mobile devices. Users do not want applications to drain their batteries. Therefore, it is essential to reason about the battery usage of the different cross-platform implementations under study.

B. Measuring tools and methods

Table 2 gives an overview of the tools that are used to measure the different performance parameters.

	Android	**iOS**
CPU	TOP-command	Instruments tool (Activity Monitor)
Memory Usage	Little Eye Tool	Instruments tool (Activity Monitor)
Disk Space	Visible on device	Visible on device
Response Times	DDMS	Instruments tool (Time Profiler)

Table II
MEASURING TOOLS.

For memory measurements on Android, Little Eye[4] was selected. This tool is able to monitor a set of performance related parameters (e.g. memory/CPU usage) for any process running on the device, plot the results at runtime and export the results to a CSV file. Although Little Eye can measure CPU usage, the TOP command was used for our measurements because it allows shorter measurement intervals (i.e. 0,1s), improving the accuracy. In iOS, the Instruments Tool from the Xcode development environment allows the monitoring of a wide range of parameters of iOS applications. The Activity Monitor feature allows the measurement of CPU and memory usage of iOS applications. To measure the response times in Android, the Dalvik Debug Monitor Server (DDMS) is used. In iOS, The Time Profiler feature of the Instruments tool is used. This feature displays the total execution time of each part of the application.

[4]http://www.littleeye.co

V. OVERVIEW OF THE APPLICATION

The PropertyCross application is used to perform the performance analysis. The app is implemented in over twenty different cross-platform technologies, and natively for iOS, Android and Windows Phone.

PropertyCross is a community driven initiative. The website enumerates the application specifications and requirements. Amongst others, the user interface and available functions are well defined. Programmers can contribute to PropertyCross by adding their implementation of the application in a tool that is not yet available on the website.

The PropertyCross application enables the user to search for properties that are for sale or rent. It returns a subset of houses in a specific area. The location is either entered by the user or obtained using the GPS sensor. The app queries a Web service and returns the query results to the user. The search history and favourites are kept on local storage. The main goal of the PropertyCross initiative is to evaluate how much platform-specific code must be written when developing an application in a CPT.

Our study performs a set of additional measurements on the existing implementations, and hence, uses the potential of the PropertyCross initiative to evaluate CPTs. Reusing the PropertyCross code is preferable over building an app from scratch in multiple tools for two reasons. First, the app size is substantial and contains a wide range of features. Building an app of this size from scratch in multiple tools would be very time-consuming. Second, and more importantly, the apps are built by experienced developers which results in high-quality code. The latter undoubtedly leads to more accurate conclusions.

VI. RESULTS AND COMPARISON

All measurements in this study have been executed multiple times and are publicly available[5]. The deviation between individual measurements was generally less than 5%. For sake of clarity, only the averages are included in the tables below.

A. Launch time

The launch time was measured in both Android and iOS. For measurements in Android, debug messages from the ActivityManager were used. The ActivityManager logs timestamps when applications are launched and when they have finished launching (i.e. when they are fully started).

[5]https://www.msec.be/crossmos/OverviewPerformanceAnalysis.xlsx

These messages can be read via the DDMS console. For the PhoneGap implementation, using the ActivityManager does not produce accurate results. The ActivityManager logs when the webview is started. This does not reflect the real launch time of the application as the Web app still has to be loaded in the webview. Once the Web application is fully started, PhoneGap logs a timestamp. This timestamp was used to measure the launch time of PhoneGap on Android.

In iOS, the *Time Profiler* feature of the Instruments tool of Xcode was used. The measurements reflect the sum of the execution times of the parts of the application active during launch. The results are displayed in Table III.

	Android		iOS	
	Nexus 6	*Acer Liquid*	*iPhone 6*	*iPhone 4*
Native	414	766	163	963
PhoneGap	1183	2778	656	2857
Xamarin	937	2018	281	1268

Table III
LAUNCH TIMES (IN MILLISECONDS).

The PhoneGap implementation has the highest launch time in both Android and iOS. Moreover, the launch time of the PhoneGap webview is higher than the launch time of the complete native implementation. Note that, after the webview is launched, the main HTML file of the application and initial scripts still need to be loaded. The user experience remains acceptable on high-end devices, as the launch time is less than one second. The situation is different on low-end devices. These are confronted with a start time of nearly three seconds, introducing a significant undesirable wait for users.

The implementation developed with Xamarin also suffers from longer start times compared to the native application. The additional delay is due to the runtime that has to be launched before the actual application can be started. This also explains why it takes more time to launch a Xamarin application in Android as opposed to its equivalent in iOS. The JIT compilation which is applied on Android typically results in slower application compared to AOT compilation which is used on iOS.

B. Pause and resume time

To measure the pause and resume times in Android, the ActivityManager debug messages were used in combination with a few lines of additional code that print a timestamp after the `onResume()` and `onPause()` methods have finished executing. Results of these measurements are shown in Table IV.

The difference in pause and resume times between the CPT implementations and the native implementation is not significant. Good development practises recommend that only a very limited amount of app-specific code should be added to these lifecycle methods. This is also valid for iOS development.

	Resume Times		Pause Times	
	Nexus 6	*Acer Liquid*	*Nexus 6*	*Acer Liquid*
Native	56	52	31	30
PhoneGap	44	54	35	24
Xamarin	39	68	32	28

Table IV
PAUSE AND RESUME TIMES IN ANDROID (IN MILLISECONDS).

C. Time to open another page of the application

For this part of the study, the time to open the *favorites* page of the application from the homepage was measured. This was the only page, apart from the homepage, not requiring Internet access. Opening this page, hence, does not introduce additional communication overhead, resulting in more accurate results. In Android, for the native and Xamarin implementation, the ActivityManager messages were used. For the PhoneGap implementation on Android, JavaScript code was added to print timestamps when the user pushes the *favorites* button and when the favorites page is fully loaded. In iOS, the *Time Profiler* feature of the Instruments tool was used. Here, we subtract the total execution time after loading the favorites page from the total executing time before pushing the favorites button. The results of the measurements are illustrated in Table V.

	Android		iOS	
	Nexus 6	*Acer Liquid*	*iPhone 6*	*iPhone 4*
Native	83	97	63	186
PhoneGap	35	188	217	618
Xamarin	98	105	63	111

Table V
FAVORITE PAGE LOAD TIMES (IN MILLISECONDS).

Two important conclusions can be drawn from this table. First, after the application is launched, the response times for the Xamarin and the native implementation are comparable. Once the Xamarin runtime has started, both the native implementation and the Xamarin implementation are running binary code. Second, PhoneGap achieves much better in-app response times in Android compared to iOS. Another remarkable result is that the PhoneGap implementation on the Nexus 6 is faster than the Xamarin and native implementation. Hence, not only the platform, but also the platform version can have a significant impact on the relative performance of the implementations.

D. Memory consumption

Memory usage was measured in both Android and iOS. For Android, the *Little Eye* tool was used. In iOS, the *Activity Monitor* from the Instruments tool was used. As mentioned earlier, the memory usage is measured in four different situations, giving a good overview of the overall behavior of the application. The memory measurements are displayed in Table VI.

	Android		iOS	
	Nexus 6	*Acer Liquid*	*iPhone 6*	*iPhone 4*
Memory consumption in foreground immediately after launch				
Native	125,10	25,65	10,6	7,38
PhoneGap	196,68	39,16	26,69	27,43
Xamarin	127,92	28,50	12,54	12,05
Memory consumption in background immediately after launch				
Native	63,77	22,50	10,63	7,35
PhoneGap	136,28	39,17	25,28	26,13
Xamarin	66,59	25,30	12,61	12,02
Memory consumption in foreground after usage				
Native	141,77	31,45	13,51	10
PhoneGap	421,22	60,60	39,84	33,57
Xamarin	150,07	40,72	25,07	22,77
Memory consumption in background after usage				
Native	74,72	27,77	13,25	9,84
PhoneGap	358,32	46,53	37,1	31,08
Xamarin	84,64	37,56	24,56	22,67

Table VI
TABLE OF MEMORY MEASUREMENTS (IN MB).

First of all, note that at the operating system level, differences can be observed between the iOS and Android memory management strategy. Android allocates more memory for an application compared to iOS. However, Android releases significantly more memory when moving an application to the background compared to iOS. Contrary to the results on Android devices, the native and the cross-platform implementations have similar memory consumption behavior on high- and low-end iOS devices.

In absolute values, PhoneGap allocates a lot more memory on Android than on iOS. The difference increases on high-end devices. However, the percentual increase compared to a native implementation is lower on Android than on iOS. For instance, the PhoneGap implementation on the Nexus 6 uses 50% more memory compared to the native implementation, while on the iPhone 6 the increase is over 150%.

Although the Xamarin implementations also use more memory compared to the native implementations, the difference with the native implementation is much smaller than with PhoneGap.

For all tested platforms and implementations, the amount of memory allocated for an application increases during the use of the application. For most implementations this increase is limited. However if a lot of memory is available, which is the case on the Nexus 6, the increase in memory is substantially higher for the PhoneGap implementation.

E. CPU usage

In our analysis, the average CPU usage during application launch was measured. In Android, the CPU usage was measured by periodically executing the TOP command via the Android Debug Bridge (ADB) shell. For iOS measurements, the *Activity Monitor* was used. This tool allows the automatic calculation of the average CPU usage during a specific time interval. Table VII lists the measurement results.

	Android		iOS	
	Nexus 6	*Acer Liquid*	*iPhone 6*	*iPhone 4*
Native	17	26	8	21
PhoneGap	29	43	27	81
Xamarin	24	40	24	69

Table VII
AVERAGE CPU USAGE DURING APP LAUNCH (IN %).

The table shows that low-end Android and iOS devices have a higher CPU load during the launch than high-end devices. This reflects the expected impact of the higher processing power of the high-end devices. Furthermore, the native apps use less CPU compared to their CPT counterparts. Overall, the CPU usage of CPTs on iOS devices compared to their native implementations is worse than on Android. Finally, the PhoneGap application is more CPU intensive than the Xamarin application.

F. Disk space

The installer sizes are given in Table VIII and the sizes of the installed applications on the different devices are shown in Table IX.

	Android	*iOS*
Native	0,86	0,62
PhoneGap	4,15	7,4
Xamarin	9,69	2,7

Table VIII
APK/IPA SIZES OF THE APPLICATION (IN MB).

	Android		iOS	
	Nexus 6	*Acer Liquid*	*iPhone 6*	*iPhone 4*
Native	3,66	2,29	0,65	0,62
PhoneGap	5,11	4,58	7,5	7,5
Xamarin	14,74	13,75	8,2	8,2

Table IX
SIZES OF THE INSTALLED APPLICATIONS ON THE DEVICE (IN MB).

For both the sizes of the installers and the installed applications, native applications consume less persistent memory. On the Android platform, the Xamarin implementation consumes most disk space. While on iOS, the PhoneGap implementation takes most persistent storage. Note that the difference in app size between the native and the CPT implementations will not increase linear to the lines of code added to the application. The majority of the overhead in Xamarin and PhoneGap is introduced by the runtime and the PhoneGap library respectively.

VII. Evaluation and Reflection

The experiments show that CPTs always come with a performance penalty compared to native implementations. However, the performance overhead is often acceptable for many applications. CPTs generally do not impose a barrier on the user experience, especially when high-end devices are used. However, the performance penalty is not linear to the complexity of the app. For instance, loading a runtime is app independent. This means that the CPT disadvantages decrease if the complexity of the app increases, and vice versa. Hence, adopting the native development strategy might be more beneficial for apps with limited functionality. The CPT selection strategy can depend on other parameters of the application. Xamarin is preferable if a lot of code is executed – as code is executed more efficiently – whereas PhoneGap can be selected if advanced user interface design is required. In contrast to Xamarin, PhoneGap does not require a lot of platform specific code for the user interface. The developer's knowledge of different programming languages can also have an impact on the CPT selection strategy.

Another major remark is the fact that the app performance does not solely depend on the CPT characteristics but also on underlying OS support. For instance, JIT compilation in older Android versions leads to slower launch times compared to AOT compilation. Although this paper does not explicitly focus on OS features, certain OS components will certainly have an impact on the performance of applications developed with CPTs. For instance, many CPTs heavily rely on the Web engine supported by the OS. The performance of the PhoneGap implementation compared to the native implementation is generally more efficient on Android than on iOS. This is illustrated by the response times, the CPU usage, the memory usage and the required disk space discussed in the previous section.

CPTs can also increase code maintainability for two reasons. First, a software update does not need to be implemented for multiple native implementations. Second, platform API updates are often transparent to CPT applications as these rely on the API provided by the CPT. Updates in the platform's API are tackled by the CPTs. Hence, the application source code does not necessarily have to be modified to benefit from new API features.

Increased user experience is often an important argument for building native apps. Although native apps can exploit new sensor technology at an earlier stage, many apps often need to behave well on older devices too. Hence, CPTs can offer sufficient support for many apps.

The battery consumption was not explicitly measured. Battery usage of an application is the sum of the battery usages consumed by the device's resources. Preliminary tests and literature [12] have shown that under normal circumstances, the display, the use of wireless communication technologies and the interaction with sensors (e.g. camera and accelerometer) have the highest impact on the energy consumption. Energy consumption by the display and wireless communication technologies is independent of the used development strategy. Although, native application can access sensors more efficiently compared to CPT applications, the difference in energy consumption is relatively small [12]. Other differences in battery consumption between different development tools can be found in longer execution times and more usage of the CPU. Nevertheless, the additional energy consumption introduced by cross-platform tools can be described as marginal compared to the total battery consumption.

VIII. conclusion

This paper presented an in-depth analysis of multiple performance parameters of a mobile app that was developed with two cross-platform tools, namely Xamarin and PhoneGap. Each cross-platform tools represents a different category of CPTs. The performance results are compared with native implementations of the same app for Android and iOS. The assessment shows that cross-platform tools introduce a performance penalty. This is valid for both Xamarin and PhoneGap, and can probably be mapped to many other development tools within the same category. However, the additional overhead is in many cases acceptable from the user's perspective, especially when high-end devices are adopted. Moreover, the amount of overhead compared to native apps is not linear to the complexity of the app.

The study further shows that the performance also depends on architectural decisions and the implementation of frequently used components of the underlying operating system. Finally, the selection of a specific tool can be driven by behavioural aspects. For instance, source code translators might be preferred for CPU intensive apps, whereas Web-to-native wrappers could be selected if advanced graphical user interface design is required. Future work will extend the experiments to other tools in order to draw more thorough conclusions about the different categories, and include the Windows Phone platform. Moreover, we target the in-depth inspection of the impact of OS components on the performance of multiple tools in order to return guidelines to OS developers.

References

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

[2] Vision Mobile, “Cross-platform developer tools 2012: Bridging the worlds of mobile apps and the web,” *February*, 2012. [Online]. Available: http://www.visionmobile.com/product/cross-platform-developer-tools-2012/

[3] S. Amatya and A. Kurti, “Cross-platform mobile development: Challenges and opportunities,” in *ICT Innovations 2013*, ser. Advances in Intelligent Systems and Computing, V. Trajkovik and M. Anastas, Eds. Springer International Publishing, 2014, vol. 231, pp. 219–229.

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

[5] A. Zibula and T. A. Majchrzak, “Developing a cross-platform mobile smart meter application using html5, jquery mobile and phonegap.” in *WEBIST*, K.-H. Krempels and J. Cordeiro, Eds. SciTePress, 2012, pp. 13–23.

[6] 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*, Oct 2012, pp. 179–186.

[7] 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*, ser. QUATIC ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 255–260.

[8] 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*, ser. SAC ’14. New York, NY, USA: ACM, 2014, pp. 757–759.

[9] 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*, ser. BCI ’13. New York, NY, USA: ACM, 2013, pp. 213–220.

[10] 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*, Dec 2012, pp. 625–629.

[11] 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*, July 2013, pp. 323–328.

[12] M. Ciman and O. Gaggi, “Evaluating impact of cross-platform frameworks in energy consumption of mobile applications,” in *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies, Volume 1, Barcelona, Spain, 3-5 April, 2014*, V. Monfort and K. Krempels, Eds. SciTePress, 2014, pp. 423–431.