Mobile Cross-platform development versus native development

A look at Xamarin Platform

Marc Armgren

Marc Armgren
VT 2015
Examensarbete, 15 hp
Examiner: Jerry Eriksson
Kandidatprogrammet i datavetenskap, 180 hp

Abstract

The world of mobile phones is currently inhabited by three large rivals. We have Apple with their iOS devices, Microsoft with the Windows Phone line, and various devices with Googles Android operating system. Developing for all platforms individually is labour intensive and time consuming. Therefore cross-platform development is of interest for companies large as small. The technique which gives performance closest to native development is called cross compiling, which compiles native applications from a common codebase. This thesis takes a closer look at the performance of the tool Xamarin Platform, in which applications for iOS, Windows Phone and Android can be created. The performance of Xamarin Platform applications on iOS and Android is native for most user interface components. Network performance is native or better. However computational tasks are very slow compared to natively developed applications. Xamarin Platform Windows Phone applications are not covered by this thesis.

Acknowledgements

I would like to thank my mentor Pedher Johansson for his guidence. I would also like to extend my gratitude to Dohi Sweden for all the help, all the coffee, and overall making me feel like a employee.

Contents

1 Introduction

Cross-platform development lowers the total development time of applications, since developing once is faster than developing multiple times. However what is the relative performance of the currently available tools?

According to Garret[12], the most important factor in a products success is its usability, not its functionality. Customers might abandon a product as soon as they suffer from bad user experiences.

This means developing cross-platform might not be a good idea, even though development time is lower, since usability may suffer from low performance. The purpose of this thesis is measuring the difference in performance between an application made in a cross-platform development tool, and a natively developed application.

1.1 Background

What is a bad user experience? According to the thesis ”Utvärdering av prestandaoptimeringsverktyg för Android”[11], application response time is the most important performance when considering the usability. The response time of an application is how fast it reacts to the users interactions, which simply means **UI (User Interface) performance** is the most important.

The next most important performance is **computational performance**, since low performance of this kind can be solved in many ways which will not impact the usability in the same way slow response times will. It is however, still important, since low computational performance makes a tool inappropriate for applications which are based on heavy calculations, for example an application which handles automated scheduling.

A lot of modern applications make use of **network** connectivity is some way shape or form. According to NNGROUP[4], users will often abandon websites if load times exceed 10 seconds, and load times of just a few seconds creates an unpleasant user experience. Since the world becomes more and more interconnected through internet communication, this type of performance is an important variable when deciding approach.

1.2 Cross-platform techniques

There are three main techniques that can be employed to develop cross-platform applications. They are as follows.

A **cross compiler** compiles code to a platform which is not the platform where the application is being developed[3]. Native development for Android, iOS and Windows Phone uses a cross-compiler[5], since it is compiled on a PC and executed on a phone. In the context of this cross-platform development, a cross compiling solution uses common

code in an arbitrary language, and compiles it to native applications for each platform.

When using an **abstraction approach** to cross-platform development, the code is not recompiled for each platform. Instead, each platform has its own version of a interpreter, which enables the programs to be executed. An example of this is the language Java, which has the "Java virtual machine" written for each platform[14], executing the java applications. Web/html5 applications are also an example of this type of solution, since it uses each platforms browser as interpreter.

Using a **hybrid solution** to cross-platform development entails a mixture of native and web technologies[13]. A native application which contains a web view is created, in which the platforms browser engine is used[13]. This approach enables native API calls, while using the strengths of a browser. The performance of the application becomes highly dependant on the platforms browser engine, which takes some control away from the developer.

To measure the best case scenario, the cross-platform development tool has to be a cross compiling one. Out of the available tools, Xamarin Platform was chosen too represent the cross compiling tools, since it is one of the older and more established tools.

1.3 Xamarin Platform

This section covers a general and technical description of the development platform Xamarin Platform.

1.3.1 General description

Xamarin Platform is a cross-platform tool which uses a common C# codebase to develop applications for Android, Windows Phone and iOS. It achieves cross-platform development through cross compiling, which means Xamarin Platform outputs native applications for all three platforms.

Xamarin Platform has the functionality of building native user interfaces and native calls for Android, iOS and Windows Phone. According to Xamarin[8] and "Comparison between Native and Cross-Platform Apps"[15], about 75% of the code can be shared between the applications when using this functionality.

The main focus of this thesis is full cross-platform development, which means close to 100% shared code. It can never be 100% using all platform functionality, since some are platforms exclusive. Xamarin platform has the tool Xamarin.Forms to accomplish full cross-platform development.

Xamarin.Forms is a UI toolkit in which a developer can create a user interface, which can be shared across Windows Phone, Android and iOS.[8] The Forms framework translates its components to native equivalents, which means the applications retains the native "look" of the platform. The tool supports a mix of this framework and the native UI toolkit, which means the applications can, if necessary[8], be tailored to each platform. Forms supports UI creation in both code, through the Xamarin.Forms API, and through XAML, a markup language from microsoft.

Only basic functionality and UI can be implemented using Forms, all other native functionality has to be written platform specific. Including but not exclusively:

- Networking

- Audio
- Video
- Geolocation
- Accelerometers

These are written using interfaces, which the developer has to implement for each platform.

1.3.2 Technical description

Xamarin Platform translates the shared C# code into native applications in three different ways. They are as follows:

For the **iOS** platform, the C# code is compiled to ARM assembly language, which can be run natively on iOS units[10].

Xaramin uses the mTouch linker to reduce the filesize of release builds of applications, which results in smaller filesizes. It manages this by using static analysis to determine what features is used by the application, and removing those which are not[9]. The linking has no effect on the performance of an application, only filesize.

For the **Android** platform, the C# code is compiled into CIL[1], which is executed by the Mono execution environment[6]. This execution environment is run side by side with the Java Virtual Machine[10], which means the applications are not executed the same as native application, however they are executed in a equivalent way. This means that the performance is tied to MonoVMs implementation.

MonoVM is packaged with the application[10], which means that Xamarin Platform applications can be installed in the same way as a native application, without any additional installs.

Xamarin Platform uses a linker to reduces the filesize of release builds of applications. It manages this by using static analasys to determine what features is used by the application, and removing those which are not[7]. The linking has no effect on the performance of an application, only filesize.

Windows Phone applications are compiled to CIL, which can be executed directly by the Windows Phone platform[10].

[1] Common Intermediate Language, the lowest form of human readable code defined in CLI (Common Language Infrastructure), which Xamarin Platform implements.

2 Method

This chapter covers the techniques used to evaluate Xamarin Platform performance, and the design/implementation of the applications used for evaluation. To accurately judge performance between the applications, equivalent implementations generated through the different development strategies was compared, with the help of an array of tools.

2.1 Application designs

To evaluate the performance of Xamarin Platform, an application had to be written three times. One application implemented in Xamarin Platform, one native implementation for iOS and one native implementation for Android. To ensure valid results, all benchmark applications were designed in such a way that they could be easily implemented in equivalent ways in both C# (Xamarin Platform, Windows Phone), Objective-C (iOS) and Java (Android).

2.1.1 UI benchmark design

To test the UI performance, the benchmark application had to be designed to encompass the mutual design and interaction paradigms of iOS, Android and Windows Phone. This information had to be taken directly form the manufacturers own documentation, since they are the ones who decide the design and interaction paradigms. Out of these mutual concepts, I chose the ones that most, in my experience, lend themselves to a modern UI design, resulting in the following design list.

One of the design concepts that works best cross-platform is navigation through multiple different tabs. It functions by allowing the user to navigate through the applications different screens, without forcing the user through a view hierarchy, in which you can move forward and backwards. Instead everything is accessible through interacting with the a static bar of tabs, which look similar on all three platforms.

There are also multiple mutual UI components, including but not limited to:

- Button - Simple implementation which executes code on touch.
- Picker - Simple scroll wheel in which a user can choose a specific piece of data from a list.
- List - Component displaying data in a list.
- Dialog - Small windows which appear above the normal UI and displays data.

From these design principles, interaction paradigms, and UI components, a simple design using three of the available components was created. The design simply contained a tab navigating structure, in which a screen containing a list, a screen containing a high

resolution image, a screen containing a Button which starts a Dialog window and an empty screen is accessible.

Measuring the performance of the UI of an application is not as easy as it may sound. Each platform handles UI components and interactions at a operating system level, abstracting developers from the process. Therefore the easiest way of measuring UI performance is using a external screen recording program, which record the entire process. Stepping through the video, counting how many frames there are between an interaction and the completion of the intended task, you get a measurement of the actual performance.

2.1.2 Computational benchmark design

To measure computational performance, my algorithm of choice had to be implemented. I chose tree traversal, since one of the most used data types in software development is the tree. Rarely can you find a program which is not in some way, shape or form using a tree. Because of this i chose to measure how fast the applications written can traverse a tree, in an recursive implementation. The following pseudocode was implemented on each platform and in Xamarin Platform. It is important to note that the algorithm is not constructed to be a reasonable tree traversal, it is designed to escape compiler optimizations and give reliable results.

```
input: Integer depth representing at what depth the search is currently at
Integer depthLimit representing the maximum depth the traversal is allowed to go

Set Integer branchingMultiple to 7;
if depth is smaller than depthLimit then
    Set integer q a random value, either 0 or 1;
    Set integer i to 1;
    if q is equal to 1 then
        repeat
            run TreeTraversal with depth + 1;
            set i to i + 1;
        until Integer i is equal to branchingMultiple;
    else
        repeat
            run TreeTraversal with depth + 1;
            set i to i + 1;
        until Integer i is equal to branchingMultiple;
    end
end
```

Algorithm 1: TreeTraversal

Quantifying the performance is done by using each platforms implementation of system time, comparing the time before executing the algorithm, versus after the algorithm finished.

The random number in the algorithm is used to make sure the compiler cant optimize the code by calculating the answer at compile time. Since it performs the same task whichever the number becomes, the results are comparable.

2.1.3 Network benchmark design

Most modern applications use one of two protocols for transferring data, either the **TCP** or the **UDP** protocol.

The **TCP** protocol is used when each packet is important, for example when transferring a file. If a packet is lost, the sender will resend the package. The **UDP** protocol is mostly used when each packet is not important, for example when streaming audio. It does not matter to the receiver if a packet is lost, it will not be audible for the user streaming, so the packages are never resent. I chose to work with the **TCP** protocol, since it is reliable, ordered and error-checked. The absence of these properties could affect the results.

I opted for the lowest level of internet communication which is available on all three platforms, which is stream sockets. The benchmark consists of two parts, the server and the client. The client is the application running on the mobile platform, while the server is running on the host machine. Each application uses its own implementation of stream sockets, while talking to the same server. All communication goes through the hostmachines localhost, which means that the networks architecture and performance is not a factor.

By measuring the time it takes to send and receive a number of packages to and from the server, the socket performance is quantified.

2.2 Benchmarking

Benchmarking sets its own array of challenges, since it has to provide valid and statistically significant data. This section covers the tools that were used to garner results, and the methods used to ensure the quality and correctness of the data.

All applications were run on emulators on a MacBook Pro purchased late 2014. To ensure this had no impact on the results, the computational benchmarking application was executed on both the emulator and a physical device. The results obtained by the emulator mirrored those obtained by the physical device. I chose to use emulators and not physical devices since the UI benchmarking approach i used can only be executed on emulators. This is since there is no reliable way of recording the screen of a physical device.

For the native Android applications, the lowest SDK allowed is set to API 15 (Ice Cream Sandwich). This was chosen since it covers 94%[1] of current Android devices, without loosing any critical functionality.

2.2.1 Method

To ensure the quality and validity of the data, the benchmarks and handling of data were performed with several different methods.

To get valid results where differences in device workload, available memory, etc has little to no impact, each benchmark was performed multiple times, and the mean calculated. The network and computational benchmark were executed 15 times per test, and the UI benchmark was executed 5 times per test. The number of executions is a product of the test error rate; the higher the error rate, the more executions were necessary.

To ensure the results were statistically valid, the results were adjusted for the error margin of the measuring tools, and the standard deviation of the benchmark result.

Compilers tend to compile nondeterministicly, depending on the specific implementa-

tion. The GCC compiler, for example, by default uses a randomized model to guess branch probabilities[2]. This can cause performance deviations which are dependent on compiling differences. To ensure nondeterministic compiling played no role in the results, applications and programs were recompiled before each execution.

2.2.2 Tools

To benchmark the UI performance, the program OBS (Open Broadcaster Software) was used. OBS captures a 120 frames per second videostream of the host screen. It was chosen because it is free and open source.

To step through the video OBS outputs, the media player Quicktime was used. It is one of the only media players in which you can step both forwards and backwards, frame by frame. This makes frame counting easier and faster.

To run native Android and Xamarin Platform Android applications, the emulator Genymotion was used. The emulator was a Samsung Galaxy s4 running API 19.

To run native iOS and Xamarin Platform iOS applications, the Xcode simulator was used. The simulator was a iPhone 4s running iOS 8.2.

3 Results of benchmarks

3.1 iOS

This section contains the results from the benchmark applications executed on the iOS platform.

3.1.1 UI benchmark

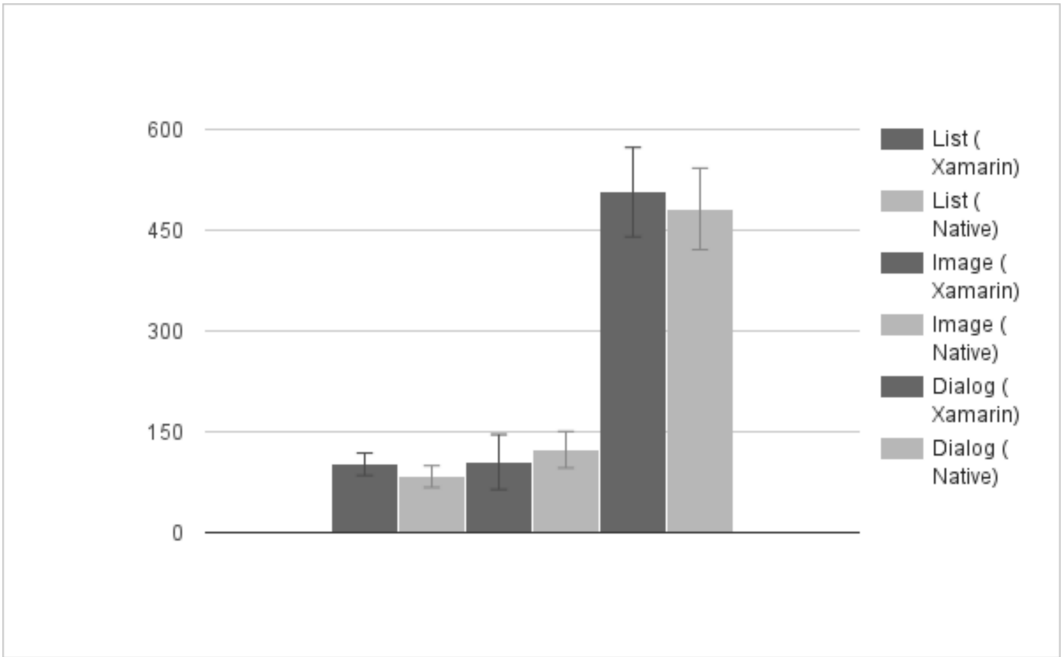

Figure 1: Result of the UI benchmarking application executed on iOS. Results are in milliseconds. The lines on the bars represent the margin of error. The results displayed are the mean of 5 seperate executions.

The results displayed in Figure 1 shows slight differences. However, as we can see on the error bars in Figure 1, the performance difference is less than the margin of error. Therefore the performance is equivalent. This means that the tested Xamarin Platforms UI components are as good, or the same, as the UI components available with native development, for the iOS platform.

3.1.2 Computational benchmark

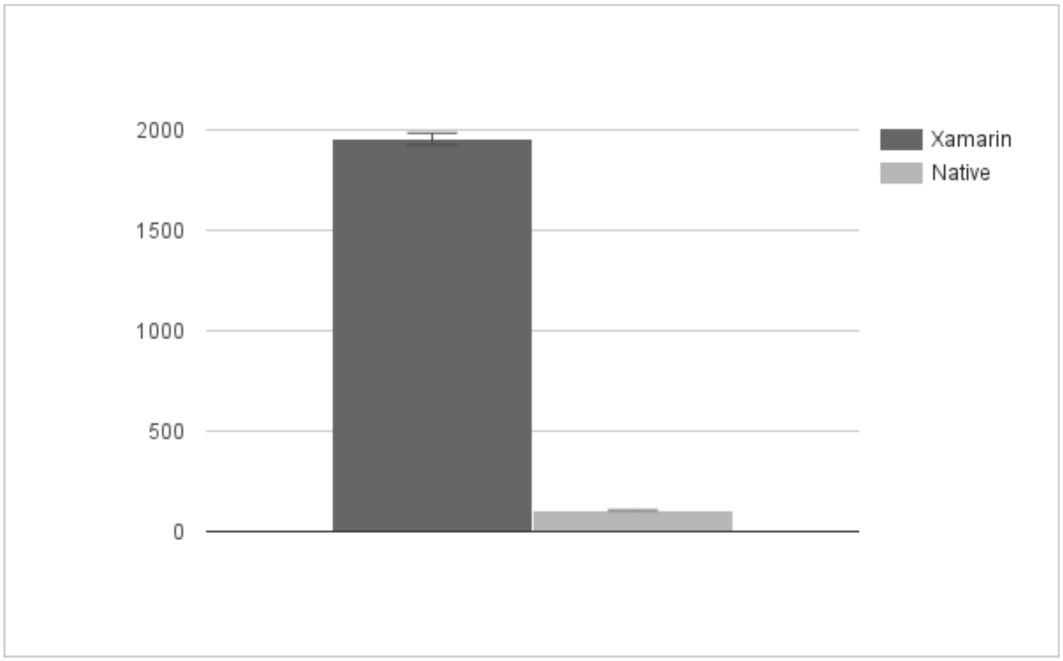

Figure 2: Result of the computational benchmarking application executed on iOS. Results are in milliseconds. The results displayed are the mean of 15 seperate executions. The lines in the bars represent the margin of error.

A significant difference in performance can be observed in Figure 2, which shows results differing about 4000 percent. As we can see in on the small error bars in Figure 2, the performance difference far exceeds the error margin of the raw data. This result is somewhat staggering, since a cross-compiling solution should, in theory, give native performance. This means that the Xamarin Platform compiler does not compile and optimize code as well as the iOS native compiler.

3.1.3 Network benchmark

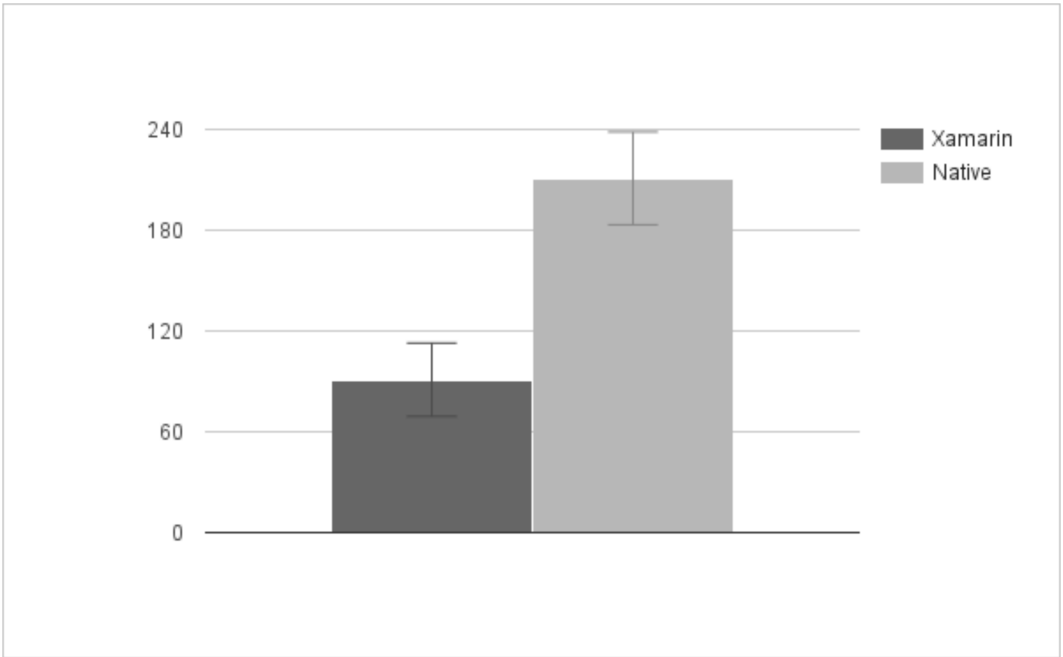

Figure 3: Result of the network benchmarking application executed on iOS. Results are in milliseconds. The results displayed are the mean of 15 seperate executions. The lines in the bars represent the margin of error.

Results displayed in Figure 3 shows a performance difference of about 230 percent. As demonstrated in Figure 3, the performance disparity is well outside the bounds of the error margin of the raw data. Therefore the network performance difference is statistically significant. The benchmark was implemented using the available Xamarin Platform and native network components, which means the performance difference is due to the components. As can be read in Subsection 2.1.3, other variables that may affect performance has been controlled.

3.2 Android

This section contains the results from the benchmark applications executed on the Android platform.

3.2.1 UI benchmark

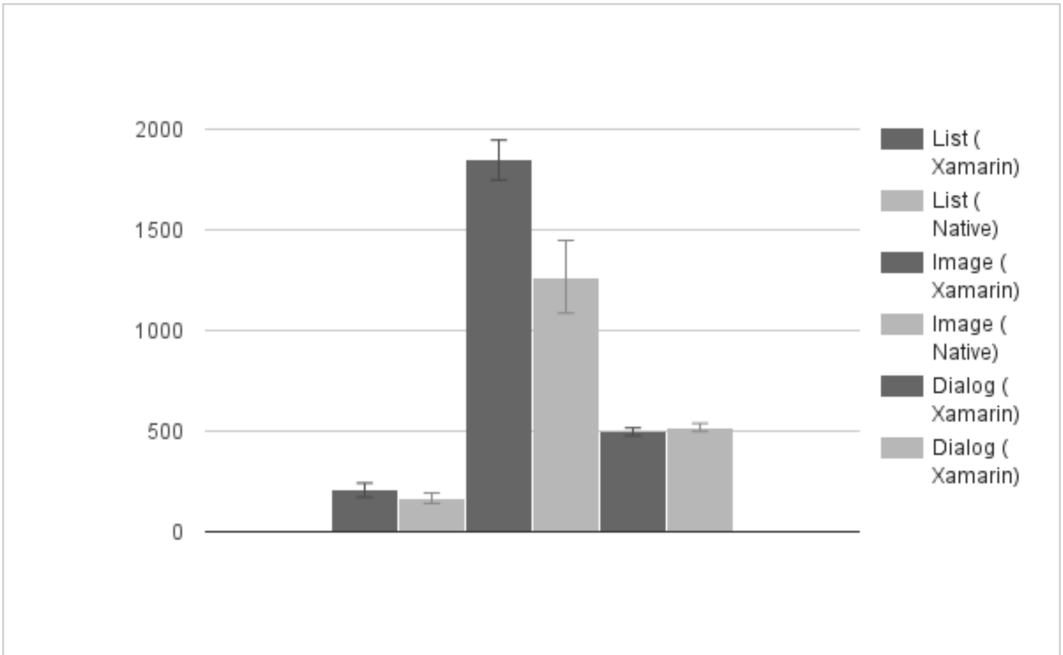

Figure 4: Result of the UI benchmarking application executed on Android. Results are in milliseconds. The results displayed are the mean of 15 separate executions. The lines in the bars represent the margin of error.

The performance difference in moving from the empty screen to the List screen, and moving from the empty screen to the Dialog screen, is well within the bounds of the standard deviation of the raw data. However, as visible in Figure 4, displaying an Image component is significantly slower in the application written with Xamarin Platform than the natively written application. This disparity is more than the error margin of the raw data, as we can see in Figure 4. Therefore the difference is statistically significant.

3.2.2 Computational benchmark

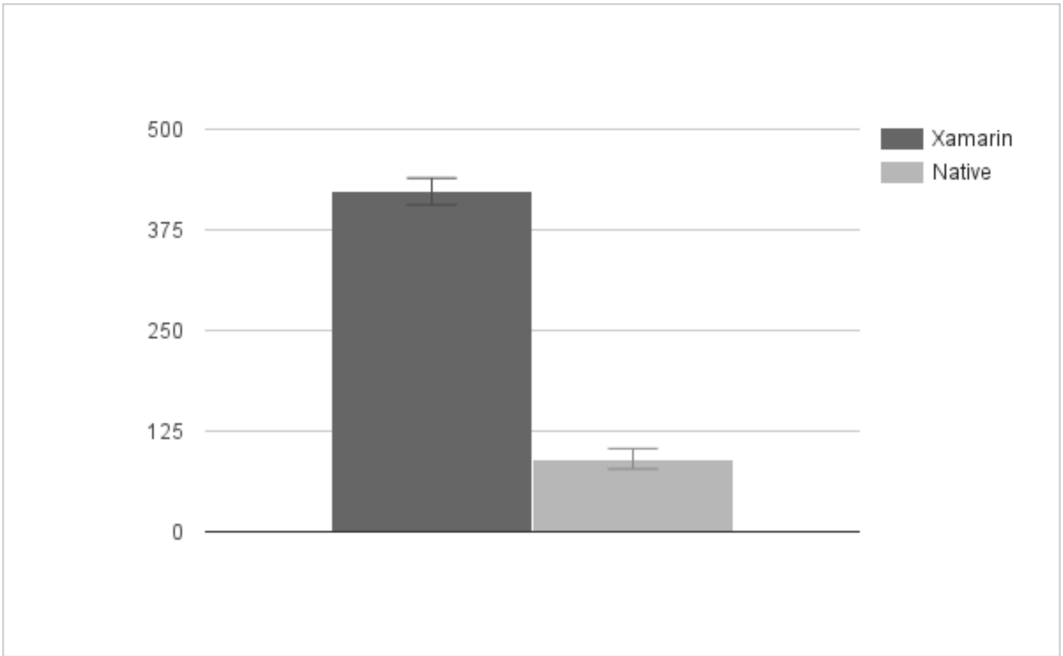

Figure 5: Result of the computational benchmarking application executed on Android. Results are in milliseconds. The results displayed are the mean of 15 seperate executions. The lines in the bars represent the margin of error.

The performance difference observable in Figure 5, is more than the error margin visible on the error bars in Figure 5. The performance difference is therefore statistically significant. This result is somewhat staggering, since a cross-compiling solution should, in theory, give native performance. This means that the Xamarin Platform compiler doesnt compile and optimize code as well as the Android native compiler.

3.2.3 Network benchmark

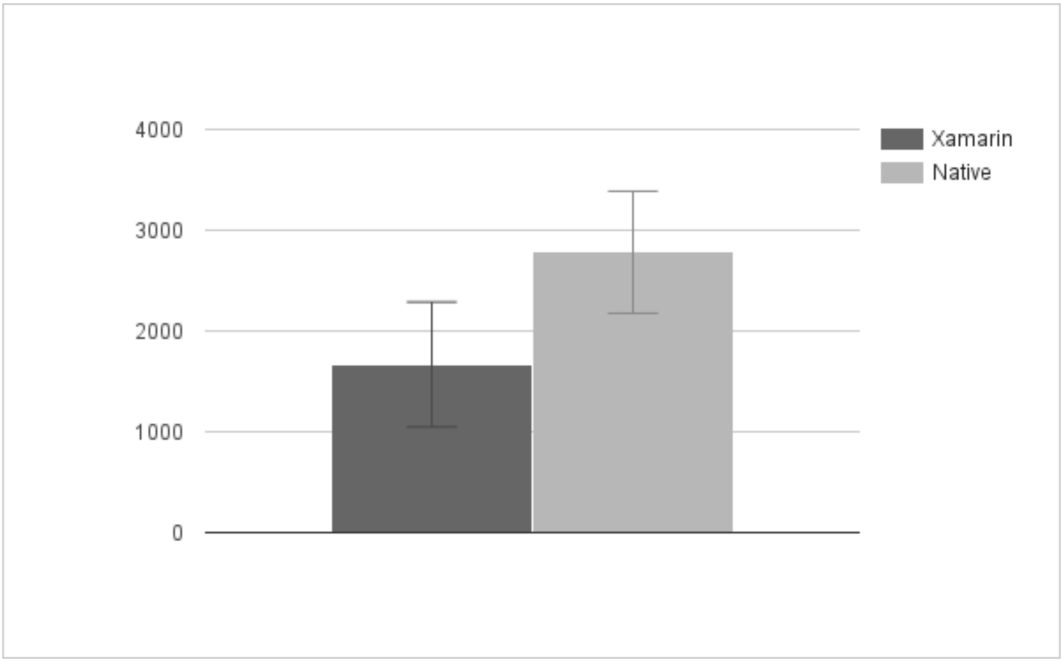

Figure 6: Result of the network benchmarking application executed on Android. Results are in milliseconds. The results displayed are the mean of 15 seperate executions. The lines in the bars represent the margin of error.

The results displayed in Figure 6 shows a performance difference of about 166 percent. However as we can see on the error bars in Figure 6, the performance disparity is less than the margin of error. Therefore the difference is not statistically significant, and the performance is equivalent. The benchmark was implemented using the available Xamarin Platform and native network components, which means the performance difference is due to the components. As can be read in Subsection 2.1.3, other variables that may affect performance has been controlled.

4 Discussion

As can be concluded from the benchmark results, applications developed with Xamarin Platform is a good idea for certain types of applications. It lends itself perfectly to applications where the UI contains most of the functionality, since the performance is equal to native implementations. However applications that require heavy calculations are not a good idea to implement with Xamarin Platform, since the calculation benchmark showed performance far below native implementations. However, as the benchmarks showed that the network performance was better or equal to native implementations, all applications containing server-side functionality can be implemented without usability issues.

Since cross-compiling development solutions in theory produces performance equivalent of native development, all current differences are a function of Xamarin Platforms implementation. For example the computational benchmark being 4000% slower than the native solution, points to Xamarin Platforms compiler not optimizing code adequately. The network performance were in some test better on Xamarin Platform applications than natively developed applications. Since I controlled for all variables in the design and implementation parts, the performance difference is probably due to the stream socket implementations available when developing. Since the computational results pointed towards Xamarin Platform applications being computationally slower than native applications, equivalent socket implementations should favor native development.

Developing with the platform works perfectly for iOS and Android, in my experience. The process of setting up project, compiling and executing was streamlined and fast. However, the Windows Phone part of the platform was not as well implemented. I could not get it to work within the time I had to my disposal, which is why Windows Phone has no benchmark results.

To build on this work, all benchmarks could be implemented on Windows Phone. Testing more UI components is also of interest, since this thesis could have stumbled through a minefield, only finding the well implemented components. Thread performance has not been touched by this thesis, and would also be of interest, since many mobile and computing platforms have processors with multiple cores.

16(17)

Bibliography

[1] Android apis. http://developer.android.com/about/versions/android-4.0.3.html/. Accessed: 2015-05-24.

[2] GCC options that control optimization. https://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Optimize-Options.html. Accessed: 2015-05-21.

[3] GNU cross-compilation. https://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Cross_002dCompilation.html. Accessed: 2015-04-23.

[4] NNGROUP website response times. http://www.nngroup.com/articles/website-response-times/. Accessed: 2015-05-22.

[5] Wikipedia cross-compilation. http://en.wikipedia.org/wiki/Cross_compiler. Accessed: 2015-04-23.

[6] Xamarin android linker. http://developer.xamarin.com/guides/android/under_the_hood/architecture/. Accessed: 2015-04-23.

[7] Xamarin android linker. http://developer.xamarin.com/guides/android/advanced_topics/linking/. Accessed: 2015-04-23.

[8] Xamarin introduction. http://developer.xamarin.com/guides/cross-platform/xamarin-forms/introduction-to-xamarin-forms/. Accessed: 2015-04-23.

[9] Xamarin ios linker. http://developer.xamarin.com/guides/ios/advanced_topics/linker/. Accessed: 2015-04-23.

[10] Xamarin under the hood. http://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_1_-_understanding_the_xamarin_mobile_platform/. Accessed: 2015-04-23.

[11] Mattias Cederlund. Utvärdering av prestandaoptimeringsverktyg för android. 2014.

[12] Jesse James Garrett. *Elements of User Experience, The: User-Centered Design for the Web and Beyond*. Pearson Education, 2010.

[13] Yuesong Li and Mark Powell. Html5, a serious contender to native app development or not? 2013.

[14] Tim Lindholm and Frank Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[15] Carlos Sirvent Mazarico and Marc Campillo Carrera. Comparison between native and cross-platform apps. 2015.