Comparing Android Runtime with native: Fast Fourier Transform on Android

André Danielsson

February 9, 2017

KTH – Royal Institute of Technology

Master's Thesis in Computer Science

KTH Supervisor: Erik Isaksson

Bontouch Supervisor: Erik Westenius

Examiner: Olle Bälter

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Sammanfattning

Svenska Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Preface

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

André Danielsson

Contents

Glossary

Android Mobile operating system. 1

FFT *Fast Fourier Transform* – Algorithm that implements the Discrete Fourier Transform. 2

JNI *Java Native Interface* – Framework that helps Java interact with native code. 3

CHAPTER 1

Introduction

This thesis explores differences in performance between bytecode and native libraries. The Fast Fourier Transform algorithm will be

1.1 Background

Android is an operating system for smartphones and as of November 2016 it is the most used (CHECK MORE SOURCES?)[1]. One reason for this is because it was designed to be run on multiple different architectures [2]. Google states that they want to ensure that manufacturers and developers have an open platform to use and therefore releases Android as Open Source software [3]. Android uses the Android kernel is based on the Linux kernel with some alterations to support the hardware on mobile devices.

Android applications are mainly written in Java to ensure portability in form of architecture independence. By using a virtual machine to run a Java app, you can use the same bytecode on multiple platforms. To ensure efficiency on low resources devices, a virtual machine called Dalvik was developed. Applications on Android have been using the Dalvik virtual machine until Android version 5 [4] in November of 2014 [5]. Since then, Dalvik has been replaced by Android Runtime. Android Runtime, ART for short, differs from Dalvik in that it uses Ahead-Of-Time (AOT) compilation. This means that it compiles during the installation of the app. Dalvik, however, exclusively uses a concept called Just-In-Time (JIT) compilation, meaning that code is compiled during runtime when needed.

To allow developers to reuse libraries written in C or C++ or just to write low level

code, a tool called Native Development Kit (NDK) was released. It was first released in June 2009 [6] and has since then gotten improvements such as new build tools, compiler versions and support for additional Application Binary Interfaces (ABI). With the NDK, the developers can choose to write parts of an app in so called *native code*. This is used when wanting to do compression, graphics and other performance heavy tasks.

1.2 Problem

Nowadays, mobile phones are fast enough to handle heavy calculations on the device itself. To ensure that resources are spent in an efficient manner, this study has investigated whether the performance boost from having the Fast Fourier Transform (FFT) compiled by the NDK instead of by ART is significant. Multiple different implementations of FFTs will be evaluated as well as the effects of the Java Native Interface (JNI), a framework for communicating between Java code and native shared libraries. The following research questions were formed on the basis of the requirements:

Is there a significant performance difference between implementations of a Fast Fourier Transform (FFT) in native code, compiled by Clang, and Dalvik bytecode, compiled by Android Runtime, on Android?

1.3 Purpose

This thesis is a study that evaluates when and where there is a gain in writing a part of an Android application in C++. One purpose of this study is to educate the reader about the process of porting parts of an app to native code using the Native Development Kit (NDK). Another is to explore the topic of performance differences between Android Runtime (ART) and native code compiled with Clang/LLVM. Because ART is relatively new (Nov 2014) [5], this study would contribute with more information related to the performance of ART and how it performs compared to native code compiled by the NDK.

The result of the study can be used to value the decision of implementing a given algorithm or other solutions in native code instead of Java. The FFT is frequently used for signal processing when you want to analyse a signal in the frequency domain. It is therefore valuable to know how efficient an implementation in native code is, depending on the size of the data.

(NDK OR JNI) AND

Android AND

(benchmark* OR efficien*) AND

(Java OR C OR C++) AND

(Dalvik OR Runtime OR ART)

Figure 1.1: Expression used to filter out relevant articles

1.4 Goal

The goal of this project is to examine the efficiency of ART and how it compares to natively written code using the NDK in combination with the Java Native Interface (JNI). This report presents a study that investigates the relevance of using the NDK to produce efficient code. Further, the cost to pass through the JNI will also be a factor when analysing the code. A discussion about to what extent the simplicity of the code outweighs the performance of the code is also present. For people who are interested to know about the impacts of implementing algorithms in C++ for Android, this study might be of some use.

1.5 Method

The method used to find the relevant literature and previous studies was to search through databases using boolean expressions. By specifying synonyms and required keywords, more literature could be found. Figure 1.1 contains the expression that was used to filter out relevant articles. For each article found, the liability was assessed by looking at the amount of times it has been referenced (for articles) and if it has been through a peer-review.

1.6 Delimitations

This thesis does only cover a performance evaluation of the FFT algorithm and does not go into detail on other related algorithms. The decision of choosing the FFT was due to it being a common algorithm to use for signal analysis. This thesis will not investigate the performance differences for FFT in parallel due to the complexity of the Linux kernel used on Android. This would require more knowledge outside the scope of this project and would result in a more broad subject.

1.7 Ethics and Sustainability

An ethical aspect of this thesis is ...

Environmental sustainability is fulfilled in this investigation because there is an aspect of battery usage in different implementations of algorithms. The less number of instructions an algorithm require, the faster will the CPU lower its frequency, saving power. This will also have an influence on the user experience and can therefore have an impact on the society aspect of sustainability. If this study is used as a basis on a decision that have an economical impact, this thesis would fulfil the economical sustainability goal.

1.8 Outline

- ***Chapter 1 - Introduction*** – Introduces the reader to the project. This chapter describes why this investigation is beneficial in its field and for whom it is useful.
- ***Chapter 2 - Background*** – Provides the reader with the necessary information to understand the content of the investigation.
- ***Chapter 3 - Methodology*** – Discusses the hardware, software and methods that are the basis of the experiment. Here, the different methods of measurement are compared and the most appropriate are chosen.
- ***Chapter 4 - Experiments*** – The result of the experiments are presented here.
- ***Chapter 5 - Discussion*** – Discussion regarding the results as well as the chosen methodology.

- ***Chapter 6 - Conclusion*** – Presents what the experiments showed and future work.

CHAPTER 2

Background

The process of developing, how Android installs the app and how it runs it is explained in this chapter. Additionally, some basic knowledge of the Discrete Fourier Transform is required when discussing differences in FFT implementations.

2.1 Android SDK

To allow developers to build Android apps, Google developed a Standard Development Kit (SDK) to facilitate the process of writing Android applications. The Android SDK software stack is described in Figure 2.1. The Linux kernel is at the base of the stack, handling the core functionality of the device. Detecting hardware interaction, process scheduling and memory allocation are examples of services provided by the kernel. The Hardware Abstraction Layer (HAL) is an abstraction layer above the device drivers. This allows the developer to interact with hardware independent on the type of device [7].

The native libraries are low level libraries, written in C or C++, that handle functionality such as the Secure Sockets Layer (SSL) and Open GL [8]. Android Runtime (ART) features Ahead-Of-Time (AOT) compilation and Just-In-Time (JIT) compilation, garbage collection and debugging support [9]. This is where the Java code is being run and because of the debugging and garbage collection support, it is also beneficial for the developer to write applications against this layer.

The Java API Framework is the Java library you use when controlling the Android UI. It is the reusable code for managing activities, implementing data structures and designing the application. The System Application layer represents the functionality that allows

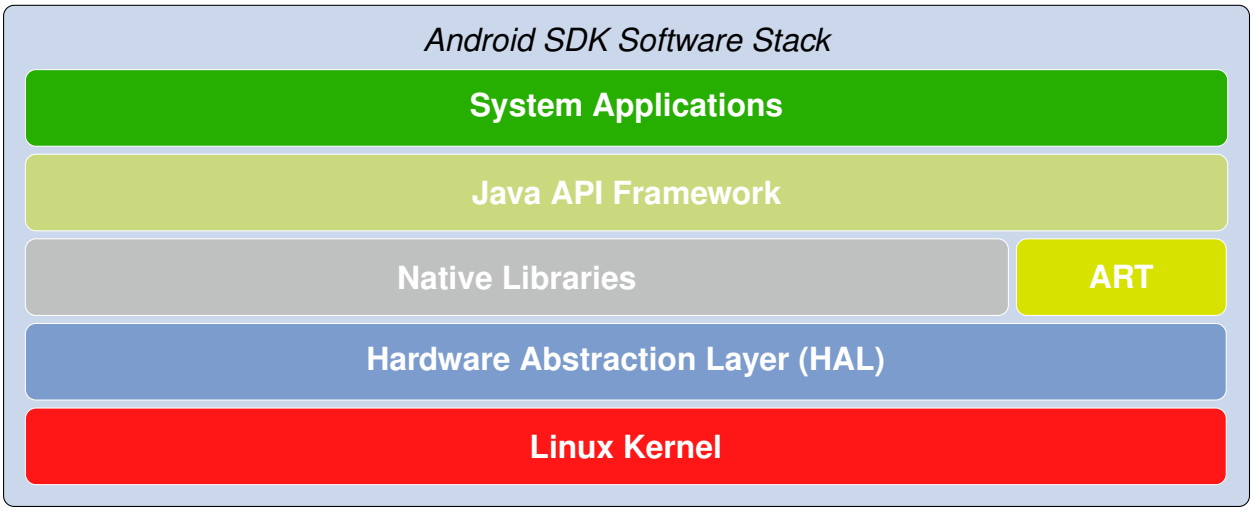

Figure 2.1: Android SDK Software Stack [9]

a third-party app to communicate with other apps. Example of usable applications are email, calendar and contacts [9].

All applications for Android are packaged in so called Android Packages (APK). These APKs are zipped archives that contain all the necessary resources required to run the app. Examples of such resources are the AndroidManifest.xml file, Dalvik executables, native libraries and other files the application depends on.

2.2 Dalvik Virtual Machine

Compiled Java code is executed on a virtual machine called the Java Virtual Machine (JVM). The reason for this is to allow compiled code to become portable. This way, every device, independent on architecture, with a JVM installed will be able to run the same code. The Android operating system is designed to be installed on many different devices [2]. Because of the many different devices, user applications would have to be compiled for all possible platforms it should work on. For this reason, Java bytecode is a choice when wanting to distribute compiled applications.

The Dalvik Virtual Machine (DVM) is the VM initially used on Android. One difference between DVM and JVM is that the DVM uses a register-based architecture while the JVM uses a stack-based architecture. The most common virtual machine architecture is the stack-based [10, p. 158]. A stack-based architecture evaluates each expression directly on the stack and always have the last evaluated value on top of the stack. Thus, only a

stack pointer is needed to find the next instruction on the stack.

Contrary to this, a register based virtual machine works more like a CPU. It uses a set of registers where it will place operands by fetching them from memory. One advantage of using register-based architecture is that fetching data between registers is faster than fetching or storing data onto the hardware stack. The biggest disadvantage of using register-based architecture is that the compilers must be more complex than for stack-based architecture. This is because the code generators must take register management into consideration [10, p. 159-160].

The DVM is a virtual machine optimized for devices where resources are limited [11]. The main focus of the DVM is to lower memory consumption and lower the number of instructions needed to fulfil a task. Using register-based architecture, it is possible to execute more virtual machine instructions compared to a stack-based architecture [12].

2.2.1 Dalvik Executables

Dalvik executables, or dex files, are the files that Dalvik bytecode is stored. They are created by converting a Java class file to the dex format. They are of a different structure than Java class files. Some differences are the header types that describes the data. One example of the differences is the string constant fields that are present in the dex-file.

2.3 Android Runtime

Android Runtime is the new default runtime for Android as of version 5.0 [4]. The big improvement over Dalvik is the fact that applications compiled to binary when they are installed on the device, rather than during runtime of the app. This results in faster start-up [13] lets the compiler use more heavy optimization that is not otherwise possible during runtime. However, if the whole application is compiled ahead of time it is no longer possible to do any runtime optimizations. An example of a runtime optimization is to inline methods or functions that are called frequently.

When an app is installed on the device, a program called **dex2oat** converts a dex-file to an executable file called an oat-file [14]. This oat-file is in the Executable and Linkable Format (ELF) and can be seen as a wrapper of multiple dex-files [15].

2.4 Native Development Kit

Native development kit (NDK) is a set of tools to help writing native apps for Android. It contains the necessary libraries, compiler, build tools and debugger for developing low level libraries. Google recommends using the NDK for two reasons: run computationally intensive tasks and usage of already written libraries [16]. Because Java is the supported language on Android, due to security and stability, native development is not recommended to use to build full apps, with exception of for example games.

Historically, native libraries have been built using makefile. Makefile is a tool used to coordinate compilation of source files. Android makefiles, `Android.mk` and `Application.mk`, are used to set compiler flags, choose which architectures that should be compiled for, location of source files and more. With Android Studio 2.2 CMake was introduced as the default build tool [17]. CMake is a more advanced tool for generating and running build scripts.

2.4.1 Java Native Interface

To be able to call native libraries from Java code, a framework named Java Native Interface (JNI) is used. Using this interface, C/C++ functions are mapped as methods and primitive data types are converted between Java and C/C++. For this to work, special syntax is needed for JNI to recognize which method in which class a native function should correspond to.

To mark a function as native in Java, a special keyword called `native` is used to define a method. The library which implements this method must also be included in the same class. By using the `System.loadLibrary("mylib")` call, we can specify the name of the shared object that should be loaded. Inside the native library we must follow a function naming convention to map a method to a function. The rules are that you must start the function with `Java` followed by the package, class and method name. Figure 2.2 demonstrates how to map a method to a native function.

The JNI also provides a library for C and C++ for handling the special JNI data types. They can be used to determine the size of a Java array, get position of elements of an array and handling Java objects. In C and C++ you are given a pointer to a struct with all the required functions to convert data back and forth between Java data and C/C++ data.

```
private native int myFun();
        ⇕
JNIEXPORT jint JNICALL
Java_com_example_MainActivity_myFun (JNIEnv *env, jobject thisObj)
```

Figure 2.2: Native method declaration to implementation.

2.4.2 LLVM and Clang

LLVM (Low Level Virtual Machine) is a suite that contains a set of compiler optimizers and backends. It is used as a foundation for compiler frontends and supports many architectures. An example of a frontend tool that uses LLVM is Clang. Clang is used compile C, C++, Objective-C [18].

Clang is as of March 2016 (NDK version 11) [19], the only supported compiler in the NDK. Google has chosen to focus on a supporting the Clang compiler instead of the GNU GCC compiler. This means that there is a bigger chance that a specific architecture is supported in the NDK.

2.5 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is a method of converting a sampled signal from the time domain to the frequency domain. In other words, the DFT takes an observed signal and dissects each component that would form the observed signal. Every component of a signal can each be described as a sinusoidal wave with a frequency, amplitude and phase.

If we observe Figure 2.3, we can see how a signal in time domain looks like in frequency domain. Function displayed in the time domain consists of three sin components, each with its own amplitude and frequency. What the graph of the frequency domain shows, is the amplitude of each frequency. This can then be used to analyze the input signal.

One important thing to note is that you must sample at twice the frequency you want to analyze. The Nyquist sampling theorem states that [20]:

The sampling frequency should be at least twice the highest frequency contained in the signal.

In other words, you have to be able to reconstruct the signal given the samples [21, Ch 3]. If you are given a signal that is constructed of signals that are at most 500 Hz, your sample frequency must be at least 1000 samples per second.

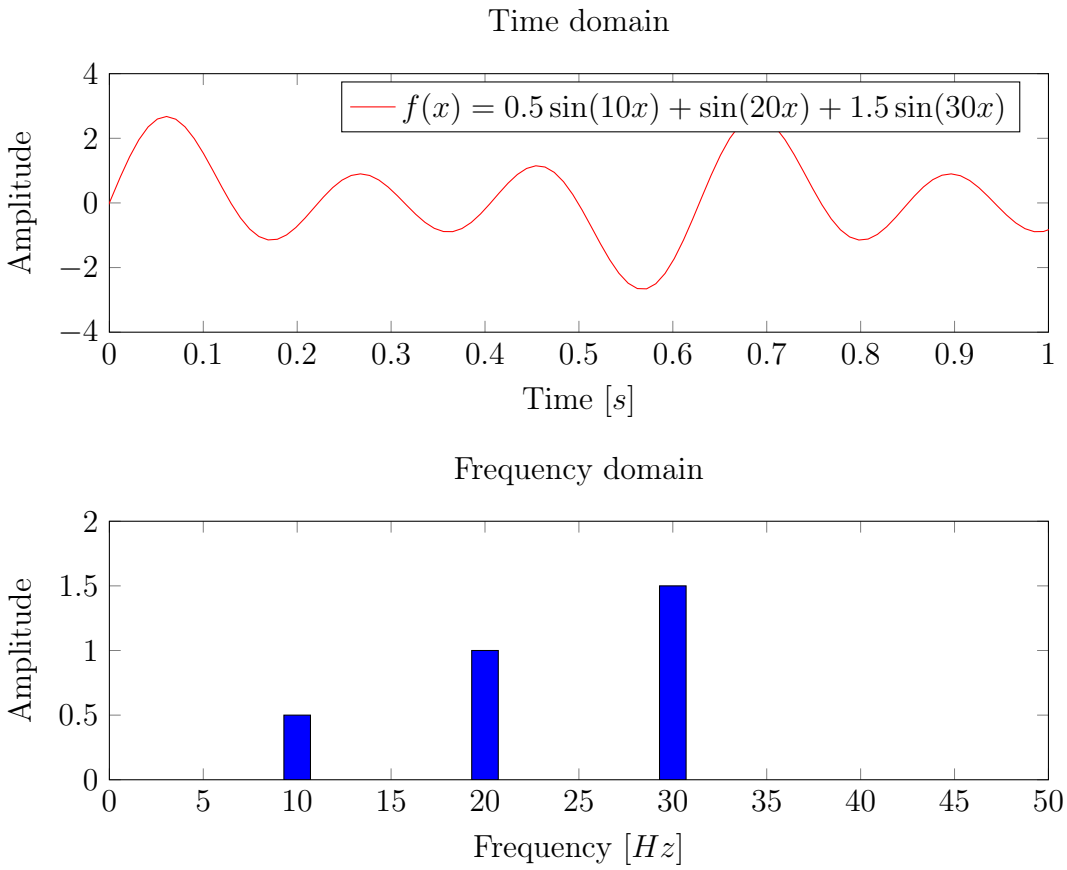

Figure 2.3: Time domain ⇒ Frequency domain

Equation 2.1 [22, p. 92] describes the mathematical process of converting a signal x to a spectrum X of x where N is the number of samples, n is the time step and k is the frequency sample. When calculating $X(k) \forall\, k \in \{x \in \mathbb{R} \mid 0 \leq x \leq N-1\}$ we clearly see that it will take N^2 multiplications. In 1965, Cooley and Tukey published a paper on an algorithm that could calculate the DFT in less than $2N\log(N)$ multiplications [23] called the Fast Fourier Transform (FFT).

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j2\pi kn/N}, \quad k = 0, 1, 2, \ldots, N-1 \tag{2.1}$$

2.6 Code Optimization

There are many ways you can optimize your code before it is being compiled to binary or byte code. This chapter first present some general optimization measures and will then describe some language specific methods for optimization.

Loop unrolling

Loop unrolling is a technique used to optimize loops. By explicitly coding multiple iterations in the body of the loop, it is possible to lower the amount of jump instructions in the produced code. Figure 2.4 following example demonstrates how unrolling works. The loop unroll executes two iterations of the first code per iteration. It is therefore necessary to update the `i` variable accordingly. Figure 2.5 describes how the change could be represented in assembly language.

The gain in using loop unrolling is that you "save" the same amount of jump instructions as the amount of "hard coded" iterations you add. In theory, it is also possible to optimize even more by changing the offset of `LOAD WORD` instructions as shown in Figure 2.6. Then you would not need to update the iterator as often.

```
for (int i = 0; i < 6; ++i) {
    a[i] = a[i] + b[i];
}
```

(a) Normal

```
for (int i = 0; i < 6; i+=2) {
    a[i] = a[i] + b[i];
    a[i+1] = a[i+1] + b[i+1];
}
```

(b) One unroll

Figure 2.4: Loop unrolling in C

```
loop: lw   $s4, 0($s1)    ; Load a[i]
      lw   $s5, 0($s2)    ; Load b[i]
      add  $s4, $s4, $s5  ; a[i] + b[i]
      sw   $s4, 0($s2)
      addi $s1, $s1, #4   ; next element
      addi $s2, $s2, #4   ; next element
      addi $s3, $s3, #1   ; i++
```

```
                    $s1 - a[] address  $s4 - value of a[x]
                    $s2 - b[] address  $s5 - value of b[x]
                    $s3 - i            $s6 - value 6

loop: lw   $s4, 0($s1)   ; Load a[i]       loop: lw   $s4, 0($s1)   ; Load a[i]
      lw   $s5, 0($s2)   ; Load b[i]             lw   $s5, 0($s2)   ; Load b[i]
      add  $s4, $s4, $s5 ; a[i] + b[i]           add  $s4, $s4, $s5 ; a[i] + b[i]
      sw   $s4, 0($s2)                           sw   $s4, 0($s2)
      addi $s1, $s1, #4  ; next element         addi $s1, $s1, #4  ; next element
      addi $s2, $s2, #4  ; next element         addi $s2, $s2, #4  ; next element
      addi $s3, $s3, #1  ; i++                  addi $s3, $s3, #1  ; i++
      bge  $s3, $s6, loop                       bge  $s3, $s6, loop
```

(a) Normal (b) One unroll

Figure 2.5: Loop unrolling in assembly

```
                  R1 - a[] address  R4 - value of a[x]
                  R2 - b[] address  R5 - value of b[x]
                  R3 - i            R6 - condition i < 6

OPTIMIZED ASSEMBLY                  TODO
```

(a) Normal (b) One unroll

Figure 2.6: Loop unrolling in assembly

```
bge  $s3, $s6, loop
```

2.6.1 Java

2.6.2 C++

Chapter 3

Methodology

To ensure that the experiment is carried out correctly, many different tools for measurements was evaluated. Different implementations of the FFT are also compared to choose the ones that would typically be used in an Android project.

3.1 Experiment model

3.1.1 Hardware

3.1.2 Benchmark Environment

3.1.3 Time measurement

3.1.4 Memory measurement

3.2 Evaluation

3.2.1 Data representation

3.2.2 Data interpretation

3.2.3 Statistical significance

3.3 Fast Fourier Transform Algorithms

3.3.1 Java libraries

JTransforms[24]

3.3.2 C++ libraries

[25]

CHAPTER 4

Experiments

Summarizing the chapter

CHAPTER 5

Discussion

CHAPTER 6

Conclusion

APPENDIX A

Example appendix

Bibliography

[1] IDC, "IDC: Smartphone OS Market Share 2016, 2015." http://www.idc.com/promo/smartphone-market-share/os. [Accessed: 2 February 2017].

[2] Android, "The Android Source Code." https://source.android.com/source/index.html. [Accessed: 1 February 2017].

[3] Android, "Why did we open the Android source code?." https://source.android.com/source/faqs.html. [Accessed: 2 February 2017].

[4] Google, "Android 5.0 Behavior Changes – Android Runtime (ART)." https://developer.android.com/about/versions/android-5.0-changes.html. [Accessed: 24 January 2017].

[5] Google, "android-5.0.0_r1 - platform/build - Git at Google." https://android.googlesource.com/platform/build/+/android-5.0.0_r2. [Accessed: 24 January 2017].

[6] C. M. Lin, J. H. Lin, C. R. Dow, and C. M. Wen, "Benchmark Dalvik and native code for Android system," *Proceedings - 2011 2nd International Conference on Innovations in Bio-Inspired Computing and Applications, IBICA 2011*, pp. 320–323, 2011.

[7] Google, "Android Interfaces and Architecture - Hardware Abstraction Layer (HAL)." https://source.android.com/devices/index.html. [Accessed: 30 January 2017].

[8] S. Komatineni and D. MacLean, *Pro Android 4*. Apressus Series, Apress, 2012.

[9] Google, "Platform Architecture." https://developer.android.com/guide/platform/index.html. [Accessed: 30 January 2017].

[10] I. Craig, *Virtual Machines*. Springer London, 2010.

[11] D. Bornstein, "Dalvik VM internals." 2008.

[12] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 4, p. 2, 2008.

[13] X. Li, *Advanced Design and Implementation of Virtual Machines*. CRC Press, 2016.

[14] Android, "ART and Dalvik." `http://source.android.com/devices/tech/dalvik/index.html`. [Accessed: 3 February 2017].

[15] L. Dresel, M. Protsenko, and T. Muller, "ARTIST: The Android Runtime Instrumentation Toolkit," *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pp. 107–116, 2016.

[16] Android Developers, "Getting Started with the NDK." `https://developer.android.com/ndk/guides/index.html`. [Accessed: 6 February 2017].

[17] Android Developers, "CMake." `https://developer.android.com/ndk/guides/cmake.html#variables`. [Accessed: 6 February 2017].

[18] UIUC, "Language Compatibility." `https://clang.llvm.org/compatibility.html`. [Accessed: 8 February 2017].

[19] Android Developers, "NDK Revision History." `https://developer.android.com/ndk/downloads/revision_history.html`. [Accessed: 6 February 2017].

[20] Bruno A. Olshausen, "Aliasing." `http://redwood.berkeley.edu/bruno/npb261/aliasing.pdf`. [Accessed: 9 February 2017].

[21] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub., 1997.

[22] L. Tan and J. Jiang, *Digital Signal Processing: Fundamentals and Applications*. Elsevier Science, 2013.

[23] B. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation Complex Fourier Series," pp. 297–301, 1964.

[24] P. Wendykier, "JTransforms - Benchmark." `https://sites.google.com/site/piotrwendykier/software/jtransforms`. [Accessed: 30 January 2017].

[25] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".