NATIVELY VS. NON-NATIVELY COMPILED THREADED ANDROID APPLICATIONS

A comparative study on time-efficiency

Bachelor Degree Project in Informatics

G2E, 22.5 credits, ECTS
Spring term 2014

Emil Andersson

Supervisor: Daniel Sjölie
Examiner: Joe Steinhauer

Abstract

The aim of this work is to investigate whether threaded Android-applications written in C or C++ are more time-efficient than applications written in Java.

The first part of the work was to perform a literature analysis in order to find out which types of algorithms were used in previous studies comparing the performance between non-threaded Android-applications written in Java and C/C++. Another literature analysis was performed where the outcome was to find suitable threaded versions of algorithms that could be used to compare the difference in time-consumption between algorithms implemented in Java and C/C++.

Results have shown that for simple arithmetic operations and sorting functions, it is still possible to gain performance in terms of time-efficiency by implementing applications in C/C++. However, there are clear indications that these gains are smaller than they are in the non-threaded case. In algorithms dealing with string operations, the Java-version was significantly more time-efficient than the C++-version.

Keywords: Android, multi-threading, concurrency, benchmarking, Java, C, C++

Table of Contents

1 Introduction

By the third quarter of 2013, the mobile operating system Android had a market share of 81 percent (IDC, 2014). Being the current leader of the smartphone market also makes it a widely discussed research area. A simple search in the digital library IEEE Xplore shows that 772 articles that somehow relate to Android were written between January 2013 and January 2014. In relation to this, a handful of studies have been performed during the past few years with focus on benchmarking Android-applications from different aspects.

For example, Lee and Lee (2011) performed a study in which they compared the time-consumption of Android-applications written in Java and C. They basically used four different types of algorithms to make their comparisons (integer, floating-point, memory access and string processing). Their results showed that in three out of four cases, the native application implemented in C would have a lower time-consumption compared to the equal version implemented in Java.

There are however reasons to question whether these results are applicable to threaded situations. Threading has been widely accepted as one out of many techniques that can be used to achieve more efficient usage of our resources, and a common advantage of threading is the ability to distribute the workload among several concurrently executing threads. Because of this distribution, it is possible that the relation between threaded applications implemented in Java and C/C++ is smaller compared to the non-threaded applications. This has been defined as the following hypothesis:

"Since the workload is distributed over several concurrently running threads, the differences in time-consumption between implementations of threaded Android-applications in Java and C/C++ will no longer be as remarkable as they are in the non-threaded case."

In order to investigate this hypothesis, the three most commonly studied types of algorithms were prioritized as targets for investigation (arithmetic operations, sorting functions and string operations). The results showed clear indications that in the cases where one could previously see big improvements by implementing a performance-critical part in C or C++, these cases show smaller gains with respect to time-consumption in the threaded case compared to the non-threaded case. Depending on the number of threads used, the improvements in time-consumption can differ as much as 61 percent between the threaded and non-threaded case.

The report is divided into seven main chapters where the first chapter, Introduction is supposed to introduce the reader to the problem area and give a brief summary on how the work was performed and what the results were. Chapter two, Background, introduces the reader to the concept of threading, the Android platform and benchmarking applications, basic knowledge necessary for the reader to understand the content of the report. Chapter 3, Problem, defines the question and hypothesis being investigated in this work and it also defines the objectives that are followed in order to answer the question.

In the next chapter, Method, the objectives are mapped to proper methods that are used to answer the main question and alternative methods are also discussed. As defined in chapter 4, Method, one of the methods is to perform a literature analysis. Chapter 5 Implementation shows the results of the literature analyses and introduces the reader to the algorithms that have been implemented. Furthermore, there is also a presentation of the experimental environment that was used when performing the experiments.

Chapter 6 shows the results that were achieved when running the implemented algorithms and it also consists of a section discussing the results and a conclusion-section that makes a brief conclusion of the results. The last chapter presents a summary of this work and presents ideas about future work.

2 Background

2.1 Threading

The concept of threading is a vital part of this report and necessary to understand in order to grasp the entire content of this report. The easiest way of explaining threading is by beginning at the absolute core concept: processes. A process represents a program being executed. It is within this process that all calculations are done and the program itself is run. But instead of doing only one thing at the time each process can be divided into several threads, each executing in parallel. Thereby the process can switch from doing only one thing into doing several things at the same time. Figure 1 illustrates the concept of threading.

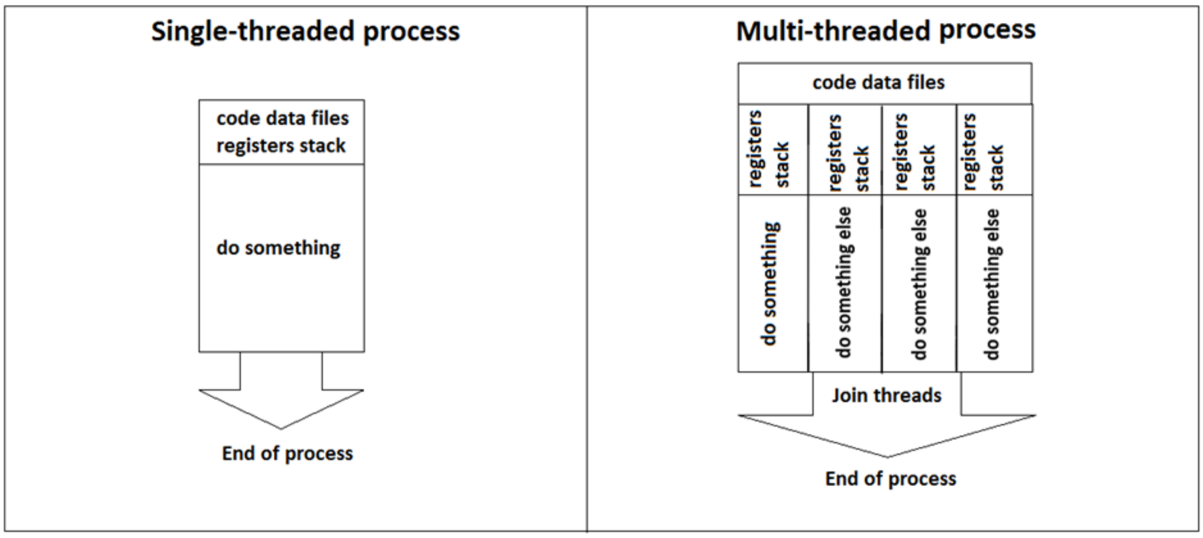


Figure 1 Multi-threaded processes can do several things at the same time

Threading is commonly known as one of the most useful techniques for improving the performance of an application. However, this hasn´t always been the case. Threading has mainly been enabled by the evolution of multi-core architectures. Without this evolution, applications would not have been able to take advantage of threading. Back in the days when multi-core was not available, threading your applications would not result in a bigger improvement since only one thread would occupy the processor at each time and the processor therefore would have to perform lots of switching between threads and processes.

Developers can benefit in several ways by threading their applications. Gagne, Galvin and Silberschatz (2010) summarized these benefits into the following categories:

- Responsiveness: Interactive applications can gain responsiveness by becoming threaded. This can be done by having one thread handle the interaction with the user, while another thread is performing heavy calculations. Without the threads the heavy calculations would have stopped the application from interacting with the user until the calculations were finished, but by threading these things can be performed simultaneously.
- Resource sharing: As depicted in figure 1, threads share resources (code, data and files). This is one of the major benefits compared to processes since processes need

some kind of external mechanism in order to share their resources (usually implemented by either shared memory or message passing).

- Economy: Creating a new thread is less costly than creating an entire process. When creating a process memory and other resources must be allocated, but when creating a thread most of these resources are already there which thereby will decrease the costs.
- Scalability: When executed on a multi-core processor, a threaded application will experience a higher level of parallelism and ability to scale better.

2.1.1 Threading in Java

When writing threaded applications in Java, a class named **Thread** is commonly used (Oracle docs, 2013). This class resides in the java.lang-package and consists of basic operations such as join, synchronize, run etc. There are mainly two ways of implementing a threaded Java-applications, where the first way is to implement your new class as a subclass of Thread. However, this way comes with certain drawbacks. The Java-language does not support multi-inheritance, which means that your new class would not be able to inherit functionality from any other class than Thread. Along with the Thread-class comes a method called run(), and it is within this method that one must put the actual code that is to be run by the thread.

```
public class myClass extends Thread {
          public void run() {
               //Do something
          }
     }
```

The other way of implementing a threaded applications in Java is to make your class implement an interface called **runnable**. Just like the first way, we must implement the run()-method. By implementing the runnable-interface, we can get the thread-functionality along with the ability to inherit functionality from other classes.

```
public class myClass implements Runnable {
          public Thread activity = new Thread(this);
          public void run() {
                //Do something
          }
}
```

2.1.2 Threading in C and C++

When writing threaded applications in C and C++ one must use the POSIX-library (Lawrence Livermore National Laboratory, 2014). It employs a concept that is slightly different from Java. In Java, the functionality of a thread is implemented as a specific object where this object implements the runnable-interface or extends the Thread-library. Arguments to this new thread can be passed by the constructor of this object. But in C/C++, the code being run by each individual thread is put in a specific function where this function returns a void-pointer. Arguments passed to a thread must also be of void-pointer type. If several arguments are to be sent, then these must first be encapsulated in a struct, and then a reference to this struct is sent.

```
struct threadArgs(
    int arg1;
    int arg2;
);
void * threadFunction(void *args){
    //Do something
    pthread_exit((void*)returnValue);
}
int main(){
    pthread_t myThread;
    struct threadArgs arguments;
    arguments.arg1 = 5;
    arguments.arg2 = 10;
    pthread_create(&myThread,NULL,functionality,(void*)&arguments;
}
```

2.2 Android

Android is as of Nov. 12, 2013 one of the most popular operating systems for mobile devices (IDC, 2014). An important feature of Android is that it is open-source. A true example of this openness is that basically anyone can download the source-code of Android (Android source code, 2014) and start building their own version. In order to modify the source code, build new versions and make your own applications the developer must be provided with some sort of developer tools, and this is where Android shows another great example of its openness. Android allows anyone to download developer tools for free and start making applications by themselves. Other manufacturers for mobile operating systems such Apple with their iOS (Apple, 2014b) charge an annual fee of 99 dollars for access to their iOS developer tools (Apple, 2014a).

Android was originally developed by Android Inc, but in 2005 this company was purchased by Google Inc. They continued the development of Android, and in 2007 another milestone in the history of Android was reached. By then, an alliance of software companies called the Open Handset Alliance was introduced. The aim of this alliance was to gather a more widespread area of expertise and thereby be able to offer a more competitive solution. The alliance currently consists of 84 companies where each company belongs to one of the following categories: *"Mobile Operators, Handset Manufacturers, Semiconductor Companies, Software Companies, Commercialization Companies"* (Open Handset Alliance, 2014a, June 17).

Android-applications are usually developed in the SDK (Google Inc., 2014a). SDK stands for Software Development Kit and it is used to build, test and debug Android applications. These applications are written by the developer in Java (note that parts of the application can also be developed in the Native Development Kit (NDK), see section 2.2.2.) As seen in figure 2, the Java source-code is initially compiled with the javac-compiler. The compilation of the Java-code produces Java bytecode, code that can be interpreted by the Java virtual machine (JVM). But instead of running the code in the JVM, the bytecode is translated to Dalvik Executable Format (Google Inc., 2014e). This code along with all other resources of the application are then put in an application package, the .apk-file and transferred to the target device.

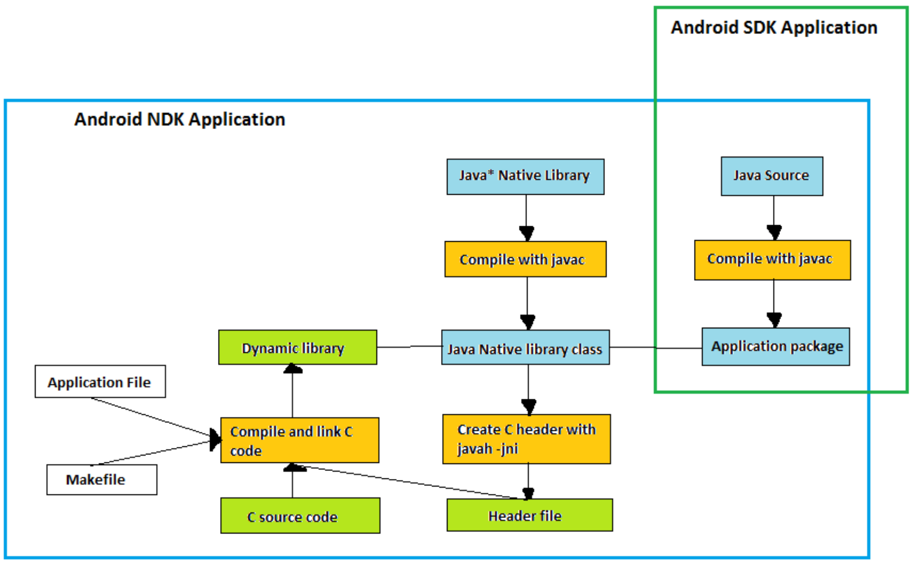


Figure 2 Structure of NDK (see 2.2.2) and SDK applications, based on Intel Corporation (2012)

2.2.1 The Android architecture

The Android platform is divided in several layers. On the bottom we have the Linux kernel. It includes the absolute basic features necessary for the device to work such as keypad-driver, Wi-Fi-driver, power management etc. On top of this kernel we have libraries that support the application framework (described below). There are also middleware-components such as the Dalvik Virtual machine. This is one of the key components of the Android architecture. It is within this virtual machine that the applications are actually run.

Above the middleware-components and libraries comes the application framework. This is the framework used by developers when developing their applications. It consists of several

Java-compatible libraries such as activity manager, resource manager, location manager and lots of other features.

Last but not least, there are the actual applications. These are seen as the top-layer in the Android architecture and it is here that the interaction with users takes place. Information about the architecture of Android was mostly gathered from Meier (2010). Please refer to figure 3 for an overview of the architecture.

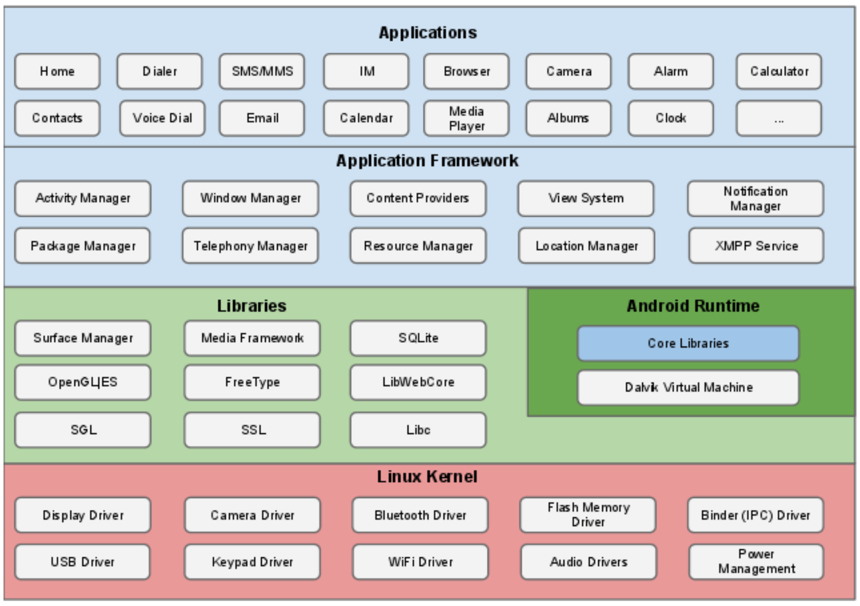


Figure 3 The Android architecture. Picture is released under creative commons license 2.5, see Creative Commons (2014b) and Google Inc (2014c, June 17)

2.2.2 Native Development Kit (NDK)

NDK stands for Native Development Kit and is basically a set of tools for developing applications in the native language of Android (native language is C, applications can also be developed in C++ even if this is not the native language of Android). The idea behind the NDK was to enable developers to write the performance-critical parts of their code in C or C++ and thereby gain performance in terms of memory usage and time consumption. It would also increase the level of reusability since developers could reuse code that they had written previously in C or C++. Note that NDK is not a replacement but only a complement to using SDK.

The process of creating an application using NDK is slightly more complicated than creating an application in the Software Development Kit (see section 2.2). Figure 2 illustrates the components necessary to create an NDK-application and how they interact.

The development usually begins by creating a Java Native Library. This library is written in Java and it declares all native functions that will be implemented in C or C++ and called upon from Java. The library is compiled into a class-file from which we extract a header-file written in C. The extraction of the header file is performed by an intermediary layer called JNI which basically handles all communication between the Java and C/C++ code.

As soon as we have our header-file written in C, we implement the functions declared in this header-file in C or C++. After having implemented the functions, a makefile (.mk) is created.

The makefile is used when we build a dynamic library that will be called upon from the Java-code. The makefile tells the compiler about files, variables and paths that are used to build the dynamic library. Then, by help of this makefile, the C/C++ code and header-file is compiled into a dynamic library.

There are pros and cons with using the NDK. In the non-threaded case, developers can gain performance in their applications by terms of time-consumption. Lin, Lin, Dow and Wen (2011) showed that by implementing performance-critical parts of the code in C/C++ developers can gain an average speedup of 34 percent. As mentioned previously, another useful feature of the NDK is that it can increase the level of reusability. Thanks to NDK, developers can take advantage of legacy code that otherwise would need to be translated into Java (which would require a lot of extra time).

2.3 Benchmarking applications

Benchmarking basically means that we measure the performance of a system. The term performance is however quite ambiguous since there are different kinds of performance that can be measured. It could be number of statements executed per second, lines of codes written per day, number of seconds taken by a function to execute, etc.

Lilja (2004) recognized that there are several common goals for analyzing the performance of a system:

- Compare alternatives
- Determine the impact of a feature
- System tuning
- Identify relative performance
- Performance debugging
- Set expectations

According to Georges, Buytaert and Eeckhout (2007), benchmarking on the Java-platform introduces some non-deterministic factors that might affect the results. For example, Java has Just-In-Time-compilation which means that the applications are compiled during runtime. If the measurements begin too early, while the compilation is being done, this compilation might cause an extra delay that will affect the results negatively.

As an example on how to treat the problem with JIT-compilation, Georges et al. (2007) made a distinction between startup-performance and steady-state performance. Startup-performance is the performance of an application that is affected by the JIT-compilation. Once the compilation has been finished, the steady-state is reached. For example, if we are only interested in steady-state performance we usually perform a number of iterations with the same input, but we do not record the first iterations (number of skipped iterations depends on how long it will take to finish the startup-state). Then, when we have skipped the first iterations, we start our measurements since these will not be affected by the JIT-compilation.

As mentioned by Georges et al. (2007), another step that can be taken in order to achieve a more statistically rigorous benchmark is to compute confidence intervals. These intervals tell us that there is a certain probability that the actual values will fall within these intervals. For example, let´s say we have measured the average amount of milk in a bottle that comes out of a factory to 1.45 litres. This average value is based on a number of samples that we have taken. When we calculate the 95-percent confidence intervals, we find that this interval is between 1.42 to 1.48 litres. This tells us that in 95 out of 100 cases, the amount of milk will be between 1.42 and 1.48 litres.

3 Problem

3.1 Problem description

The aim of this work is to answer the following question:

"Are natively compiled threaded Android-applications more time-efficient than non-natively compiled threaded applications?"

When developing Android applications, one can either choose to write the entire application in Java (these are called non-natively compiled applications) or to write the application in both Java and C/C++ where the performance-critical parts of the code are written in C/C++. The latter case is also referred to as being natively compiled.

As mentioned in the background section, previous studies regarding the difference in performance between Java-written and C/C++ -written Android-applications were targeted at non-threaded applications. In these studies it was possible to see that in most cases, writing your application in C or C++ instead of Java would result in great improvements. However, the following hypothesis provides us with a reason to question whether this phenomenon still remains in the case with threaded applications.

Hypothesis: *"Since the workload is distributed over several concurrently running threads, the differences in time-consumption between implementations of threaded Android-applications in Java and C/C++ will no longer be as remarkable as they are in the non-threaded case."*

If the results of this study can provide support for the hypothesis, this can be of value both from a business perspective and a scientific perspective. Seen from the business perspective, writing native applications requires more resources in terms of time and knowledge. Developers implementing a native application in C or C++ must not only reconsider the structure of their application, they must also consider things related to the Java Native Interface and C/C++ such as intermediary layers, conversion of datatypes, dynamic libraries etc. If the results were to find out that the gains in time-consumption in the threaded case are no longer as significant as they are in the non-threaded case, then this might serve as an argument for these developers to only develop their threaded applications in Java and thereby avoid the extra work required to create a native application.

Seen from the scientific perspective, if the results are able to show indications that the differences in time-consumption between applications implemented in different languages are lowered when applications become threaded, this can serve as an opening to new research possibilities. To the author´s knowledge little research has been done previously on how the difference in time-consumption between programming languages are affected by threading, so if the results indicate that the differences are lowered in the threaded case then this might lead to new studies about this phenomenon and whether it can be observed on other platforms than Android and with other types of algorithms than the ones investigated in this work.

Apart from the main question defined above, there are also two minor questions that will need to be answered. As defined in section 3.1.1., objective 3, the time taken by each implementation will be measured. This introduces the first minor question – *how will the* *time be measured?* Knowing how to measure the time is vital knowledge in this work and emphasis should therefore be put to ensure that the time is measured correctly.

Secondly, when implementing algorithms it is utterly important to address the problem of equivalence between threading mechanisms in Java and C/C++. Each programming language uses different mechanisms for handling thread functionality, and for the experiment to be valid the implementations should be as equal as possible to ensure that an implementation made in a specific programming language does not gain performance just because it has a more efficient implementation. This gives us the second minor question – *how do we ensure that the implementations are as equal as possible?*

3.1.1 Objectives

In order to answer the main question of this paper, the following objectives have been defined:

Objective 1 – Identify what types of algorithms were used in previous research when comparing the performance between Android-applications written in Java and C/C++

Objective 2 – Find parallelizable algorithms based on the result from objective (1)

Objective 3 – Measure and compare the time-consumption between threaded implementations of chosen algorithms

> Objective 3.1 – Study how time is measured in Java, C and C++
>
> Objective 3.2 – Identify how threading mechanisms in Java, C and C++ can be mapped between each other in order to ensure equivalence between implementations

4 Method

A proper way of addressing objective (1), (2), (3.1) and (3.2) is to perform a literature analysis. Berndtsson, Hansson, Olsson and Lundell (2008, page 58) describe a literature analysis as *"… a systematic examination of a problem, by means of analysis of published sources, undertaken with a specific purpose in mind"*. So by doing such an analysis we can in a systematic way search for information about algorithms that have been used in previous research when benchmarking Android-applications and find information about such benchmarking algorithms that can be parallelized. However, when performing a literature analysis it is vital to include material that originate from relevant sources. Berndtsson et al. (2008, page 59) describe the following techniques for identifying relevant sources to include in literature analyses:

> *"Bibliographic databases – by searching or browsing their content using a sensible strategy, e.g. by having a set of keywords which your background reading and problem statement have identified as potentially relevant, by using a reference to a given source and locating additional sources.*
>
> *Journals and conference proceedings – e.g. by browsing the table of contents in conference proceedings, or by searching for relevant articles using the search function at a journal´s web site (most scientific journals are published simultaneously in printed and electronic format, and libraries that subscribe to the journal also usually have free access to the electronic version)."*

Another approach to fulfil the first two objectives would be to interview people with experience in programming and ask them about algorithms that can be used. However, this method seems less reasonable since it probably would result in less coverage of all possible algorithms that can be used compared to the literature analysis. It could perhaps be seen as a complement to the literature analysis.

The implemented algorithms must be executed in an experimental environment. There are certain demands on this environment – it must be able to deal with confounds (described in next paragraph) and it should rely on a multi-core architecture for the applications to take advantage of the threading-concept. The first choice would be to run the applications on a virtual Android-device. However, this option comes with uncertainty. It is unclear whether the virtual device can take advantage of an underlying multi-core architecture. If the virtual device only has one dedicated core during execution, this device will most likely be seen as a single-core device. It would in this case make no sense to run threaded applications on this device, even if the underlying architecture is multi-cored.

The second and more realistic choice is to run the applications on a "real" device. This is beneficial since we get closer to the real-usage scenario and are able to utilize the underlying multi-core architecture to its full extent. However, the issue of confounds will prevail. Confounds are things that might impact the result e.g. a process running in the background.

Carlsson (2011) investigated how Android-applications could become more efficient seen from an energy-perspective. In this study Carlsson (2011) faced the problem of confounds, and he dealt with this by performing three main steps:

- Restart the device
- Deactivate background-processes e.g. applications that synchronize with internet
- Deactivate unused hardware such as 3G, GPS, Wi-Fi and Bluetooth

Carlsson (2011) also mentioned another measure that can be taken to handle confounds, which is to run multiple tests with the same input. If we do not do this, our tests will have a low statistical power. Wohlin (2012) describes low statistical power as one of the main threats against the conclusion validity. By running our tests multiple times we can see the standard deviation, minimal and maximal value and thereby increase the statistical power of the tests. This will also give us further confidence regarding the conclusions we draw from our results. Otherwise our tests might be negatively affected by some background-process that interrupt the current process and by this makes it seem like the process being run is slower than it actually is.

5 Implementation

5.1 Literature analysis

When performing the literature analysis, the following databases were used. These databases are commonly used and referred to, containing journal papers, conference material, magazines, eBooks etc.

1. DiVA (2013). A search-service and open archive for research publications and student theses.
2. IEEE Xplore (2014). A continually updated database with every scientific and technical publication ever produced by IEEE and its publishing partners. Contains more than 3 million articles with content dating back to 1872.
3. Springer Link (2014). Over 8.5 million scientific documents with new books and journals every day.
4. ACM (2014). A database containing every article ever produced by ACM in *The ACM Digital Library*. Also contains bibliographic references to literature in informatics.

When searching these databases for information that can fulfil objective (1) - *find out what types of algorithms that were used in previous research when comparing the performance between Android-applications written in Java and C/C++*, the following keywords were used both in singular form and in combination with one another.

- Android
- NDK
- Benchmarking
- Java
- C
- Dalvik

The initial literature analysis showed that a great diversity exists when it comes to what categories of algorithms were used. However, some categories overlap which made it difficult to make distinctions between them. Lee and Lee (2011) used five different categories of algorithms when they benchmarked their algorithms: JNI delay, integer, floating-point, memory access algorithm and a string processing algorithm.

Lin, Lin, Dow and Wen (2011) used algorithms that are somewhat different from Lee and Lee (2011). They used the following categories of algorithms: Numeric calculation with recursion, library facilities, user-made data structures, polymorphism, nested loops, random number generation, sieve of Eratosthenes and string operations.

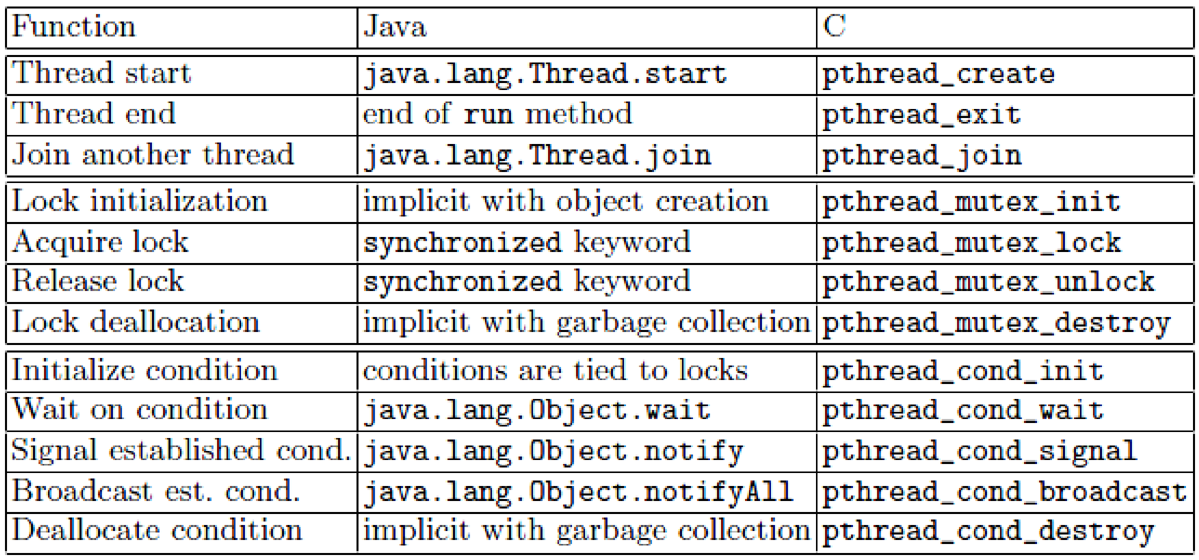
Kim, Cho, Kim, Hwang, Yoon and Jeon (2012) used a different approach to obtain their algorithms. They obtained their algorithms from an embedded benchmark suite called Mibench (see Guthaus, Ringenberg, Ernst, Austin, Mudge & Brown, 2001). This benchmark suite consists of six main categories from which the following algorithms were used: Basicmath & quicksort, Dijkstra & Patricia, FFT and stringsearch.

Given this outcome of the first literature analysis, several conclusions could be made regarding what algorithms were used in previous research when benchmarking Android-applications written in Java and C/C++. The JNI delay algorithm used by Lee and Lee (2011) was used in order to measure the delay caused by using JNI. However, using time.h will not suffer from this delay which is why the JNI delay algorithm has been left out in the table below. Furthermore, all three papers implemented algorithms with basic math and integer calculations (integer, numeric calculation with recursion and basicmath). One could also see that they all had an algorithm dealing with strings (string processing algorithm, string operations and stringsearch). The third most commonly type of algorithm in these papers were sorting-algorithms.

Since arithmetic operations, string operations and sorting algorithms were the most commonly used types of algorithms when comparing the performance between Android-applications written in Java and C/C++, the following keywords were used in the second literature analysis with aim to fulfil objective (2). Performing a literature analysis with these keywords gave rise to several sources of information about parallelizable algorithms as seen in section 5.2 below.

- Thread
- Parallel
- Parallelizable
- Algorithm
- String
- Operations
- Numerical
- Arithmetic
- Sorting

In section 3.1 the problem of equivalence between the Java and C/C++ code was introduced, and it was defined as an objective (3.2) to "*identify how threading mechanisms in Java, C and C++ can be mapped between each other in order to ensure equivalence between implementations*". Artho, Hagiya, Leungwattanakit, Tanabe and Yamamoto (2010) discussed the threading-models in Java and C and presented a mapping between the most central threading mechanisms such as creation of threads, joining threads, lock, wait-conditions and more. This mapping is presented in table 1 below. According to them, it is for the sake of simplicity recommended to begin by implementing a concurrent algorithm in Java and then translate this to C. This is mainly due to the fact that not all thread primitives in Java and C can be mapped directly between each other. Some of these primitives such as starting and joining threads can be mapped directly, but when it comes to locking and shared conditions there are certain differences. In Java, the locking of objects happens internally in the Java virtual machine while in C this has to be coded manually. Please note that the threading primitives in C are also valid in C++ since they both employ the POSIX thread library, see Lawrence Livermore National Laboratory (2014).



Function	Java	C
Thread start	`java.lang.Thread.start`	`pthread_create`
Thread end	end of `run` method	`pthread_exit`
Join another thread	`java.lang.Thread.join`	`pthread_join`
Lock initialization	implicit with object creation	`pthread_mutex_init`
Acquire lock	`synchronized` keyword	`pthread_mutex_lock`
Release lock	`synchronized` keyword	`pthread_mutex_unlock`
Lock deallocation	implicit with garbage collection	`pthread_mutex_destroy`
Initialize condition	conditions are tied to locks	`pthread_cond_init`
Wait on condition	`java.lang.Object.wait`	`pthread_cond_wait`
Signal established cond.	`java.lang.Object.notify`	`pthread_cond_signal`
Broadcast est. cond.	`java.lang.Object.notifyAll`	`pthread_cond_broadcast`
Deallocate condition	implicit with garbage collection	`pthread_cond_destroy`

Table 1 Comparison of thread primitives by Artho, Hagiya, Leungwattanakit, Tanabe and Yamamoto (2010)

Section 3.1. also mentioned the problem of measuring the time and it was defined as objective (3.1) to "*study how time is measured in Java, C and C++*". As mentioned previously Lee and Lee (2011) benchmarked non-threaded Android-applications written in Java and C and compared them against each other. They measured the performance in nano-seconds using the built-in function nanoTime() in Java (Oracle docs, 2010). It is, however, doubtful whether this method of measuring the time "manually" by using a built-in function in Java is a valid approach since it might affect the results negatively (at least for applications written in C/C++). The results might be negatively affected since there will always be a delay when calling the native method in C from the Java-code using JNI. One way to treat this obstacle is to measure the JNI-delay and then subtract this delay from the time taken by C/C++ algorithms. A more simple and reasonable approach is to use a standard C-library like time.h to measure the time for the applications written in C/C++.

5.2 Algorithm implementation

For what concerns the algorithms implemented, the first one implemented was a threaded merge-sort algorithm with basic ideas gathered from Rolfe (2010). His idea of implementing a merge-sort algorithm is basically to define the number of threads that will be used, and then split the original array to be sorted into equally sized partitions where each partition is sorted sequentially by an individual thread. Instead of having the parent-thread in idle, waiting for the result of its child-thread, the parent-thread calls the merge-sort function in itself, with only half of its previous dataset. This creates a basic structure as seen in figure 4 below.

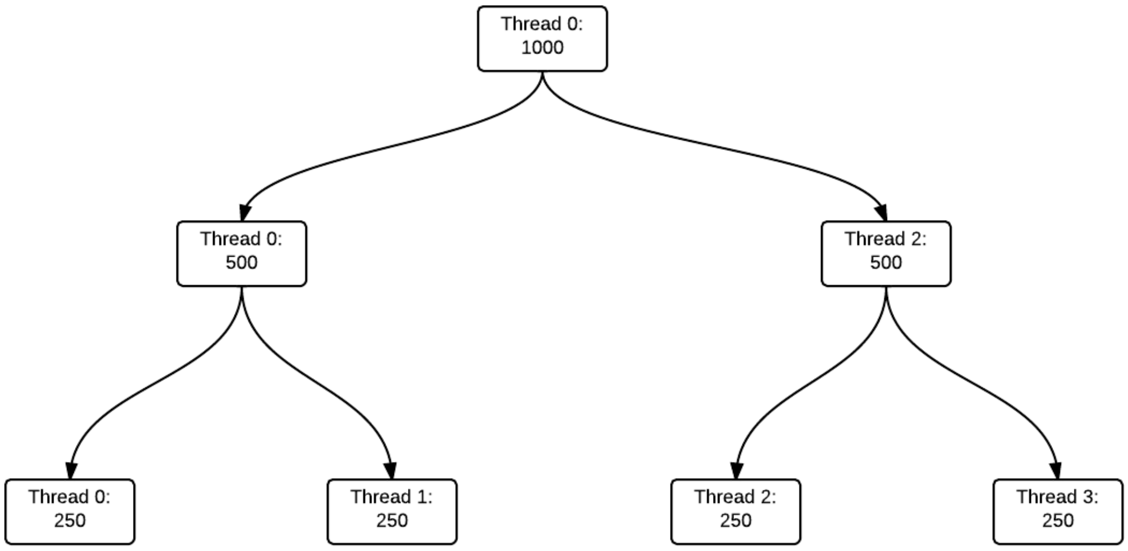


Figure 4 Basic structure of threaded merge-sort algorithm with original array-size of 1000 elements

The second algorithm implemented was a threaded algorithm calculating the Fibonacci sequence. The Fibonacci sequence is as sequence of integers, where the first two numbers are defined as 0 and 1, and the following integers are defined as the sum of its two predecessors. For example:

fib(0) = 0

fib(1) = 1

fib(2) = fib(1) + fib(0) = 1

fib(3) = fib(2) + fib(1) = 2

In the simplest case, calculating the Fibonacci sequence is done recursively by defining the first two numbers as 0 and 1, and then calculate the following integers according to the following rule:

fib(n) = fib(n-1) + fib(n-2)

In the recursive solution, the Fibonacci-function calls itself twice where one call is made with the current number minus one, and the second call is made with the current number minus two as input. In the threaded algorithm, one call is still made recursively (so the parent-thread will not have to idle) and the second call is replaced by creating a new thread in which we make a new function call to the Fibonacci-function with the current number minus two. All of this goes on until we reach the case where we have created the right number of threads (which is defined in the beginning of the function). By then, the number is calculated in a usual recursive manner and returned to the parent thread.

Dividing the workload into two new parts where one is passed on recursively and the other is sent to a new thread creates a binary tree structure that is very similar to the structure of the merge-sort algorithm as seen in figure 4 and 5.

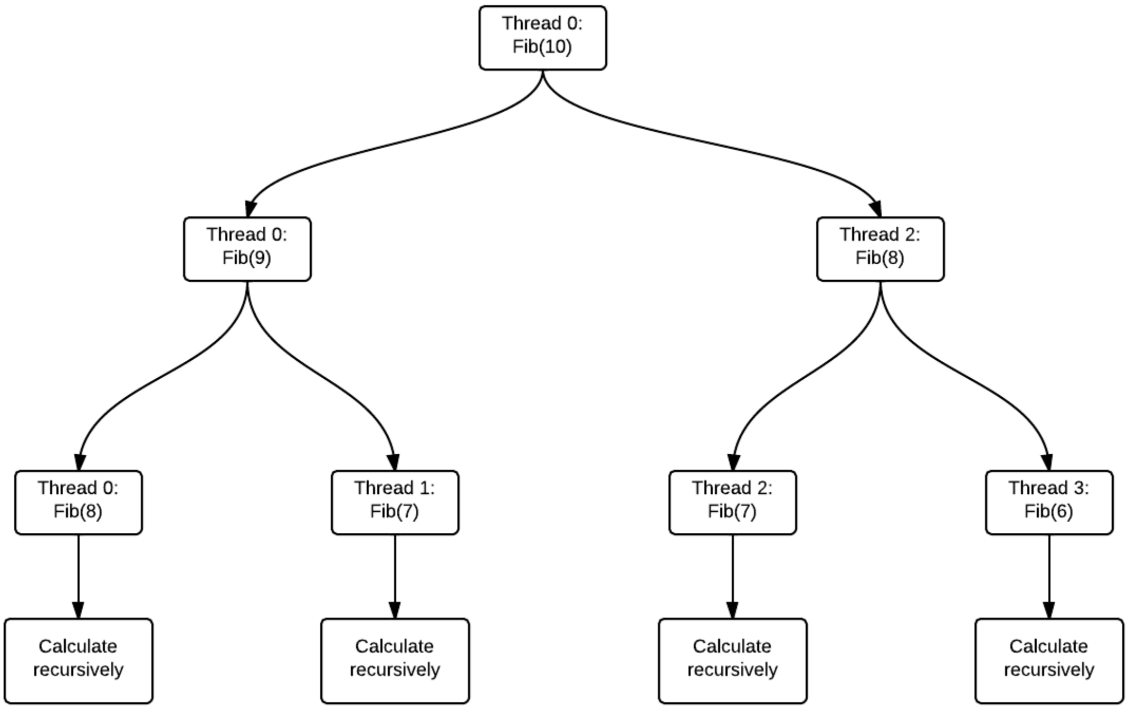


Figure 5 Structure of threaded algorithm calculating the Fibonacci-sequence

The third algorithm implemented was a string-search algorithm with the basic idea gathered from Lin, Tsai, Liu and Chang (2010). This is basically an algorithm that searches after a certain pattern within a text and returns the number of occurences of the pattern within the text. The algorithm begins by dividing the text into segments, where each segment is searched by an individual thread. According to Lin et al. this algorithm suffers from the "boundary-section" problem, a problem that has a potential risk of decreasing the performance. When the experiment was performed, much effort was used to ensure that this would never occur by putting the searched patterns within the span of each thread.

In order to describe the boundary-section problem, please refer to figure 6. The text is initially divided into four different segments where each segment is to be searched for the pattern "ABBA" by a thread. Thread 0 is supposed to search between index 0 and 6. But at index 4, it recognizes the beginning of the pattern. Therefore it is allowed to cross its boundary, searching after the pattern beyond its defined scope. Same thing occurs in thread 1. At the last position of the segment, it detects a character that fits the beginning character of the pattern. It is therefore allowed to continue and search within the segment of thread 2, extending the search-time required by the application.

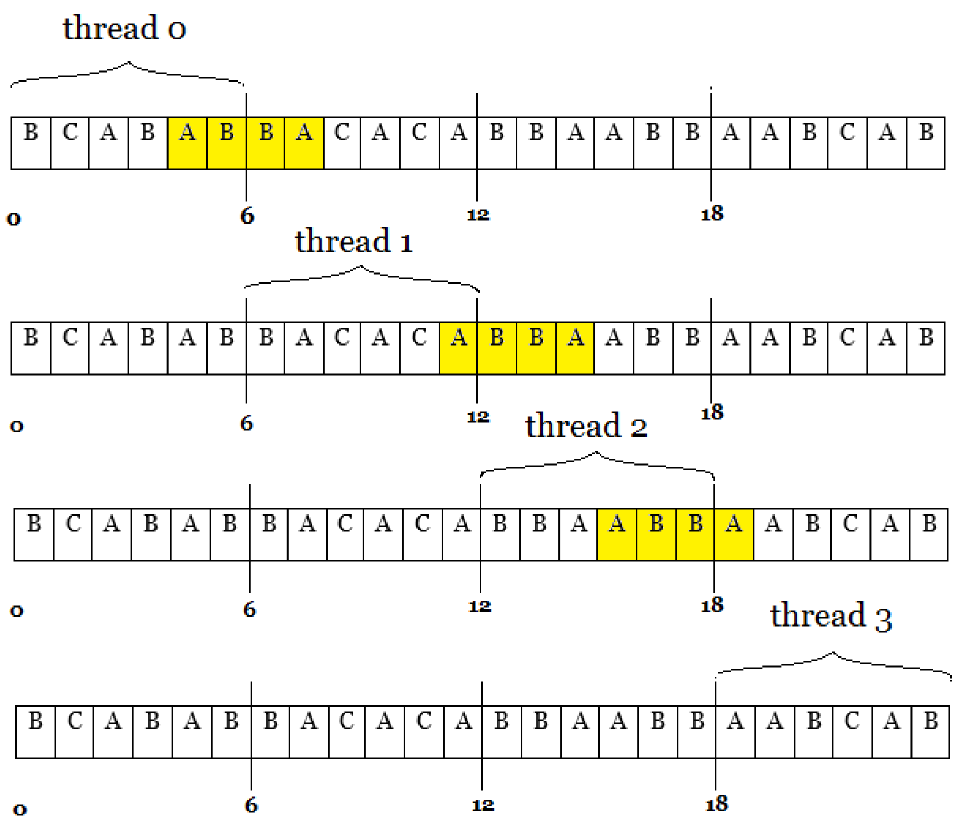


Figure 6 Boundary-section problem experienced in the string-search algorithm

5.3 Experimental environment

When it comes to the experimental environment, the applications will be run on a real Android-device called Sony Xperia Z1 compact. The main reason for choosing this device as experimental environment is that it was the closest multi-cored Android-device available. The main features of this device is described in table 2 below.

Figure 7 Sony Xperia Z1 compact. Picture is released under creative commons license 3.0, see Creative Commons (2014a)

Operating system	Google™ Android™ 4.3 (Jelly Bean MR2)
Processor	2.2 GHz Qualcomm MSM8974 Quad Core
GPU	Adreno 330
RAM	2 GB
Flash memory	Up to 16 GB
Expansion slot	microSD™ card, up to 64 GB (SDXC supported)
Battery (Embedded)	2300 mAh minimum
Heap size	512 MB

Table 2 Main features of Sony Xperia Z1 Compact, information gathered from Sony mobile (2014)

6 Evaluation

6.1 Results

In the beginning of each run, all unused hardware such as 3G, Bluetooth and synchronization processes were deactivated. All native versions were written in C++, mostly due to the author´s previous experience in this language. Each algorithm was run with a full heap size of 512 megabytes. **Please observe** that all black and blue lines in the figures below represent the confidence intervals.

As stated in the previous section, Implementation, the first algorithm implemented was a threaded merge-sort algorithm. This algorithm was run with combinations of threads ranging from 4 to 128, and number of elements ranging from 25 000 to 225 000. Combining these factors formed individual setups.

For example, the 4-threaded Java-version with 50 000 elements formed a unique setup, and the 8-threaded Java-version with 50 000 elements formed another unique setup. Each of these setups were iterated 35 times, and among these iterations the first 5 were not recorded to ensure that these are not affected by the JIT-compilation. Georges, Buytaert and Eeckhout (2007) recommended that in order to find the numbers of iterations to be skipped, one should begin by recording the time of all iterations and analyse these times to see how many iterations it takes to pass the JIT-compilation. This was done, but the results did not reveal a specific number of iterations where the application would no longer be affected by the JIT-compilation. Therefore the conclusion was drawn that results would most likely not be affected by the JIT-compilation very much, but as a safety measure five iterations were set just as an extra precautious step to ensure that the applications would not be affected by the JIT-compilation.

Figure 8 below shows the mean time consumption in nanoseconds for each combination of number of threads and version of merge-sort (Java or Native), and figure 9 shows how each version of merge-sort scales as the number of elements are increased. Note that the curves in figure 9 are not thread-specific, they only show the averaged mean time-consumption for all numbers of threads (this also applies to figure 11 and 13).

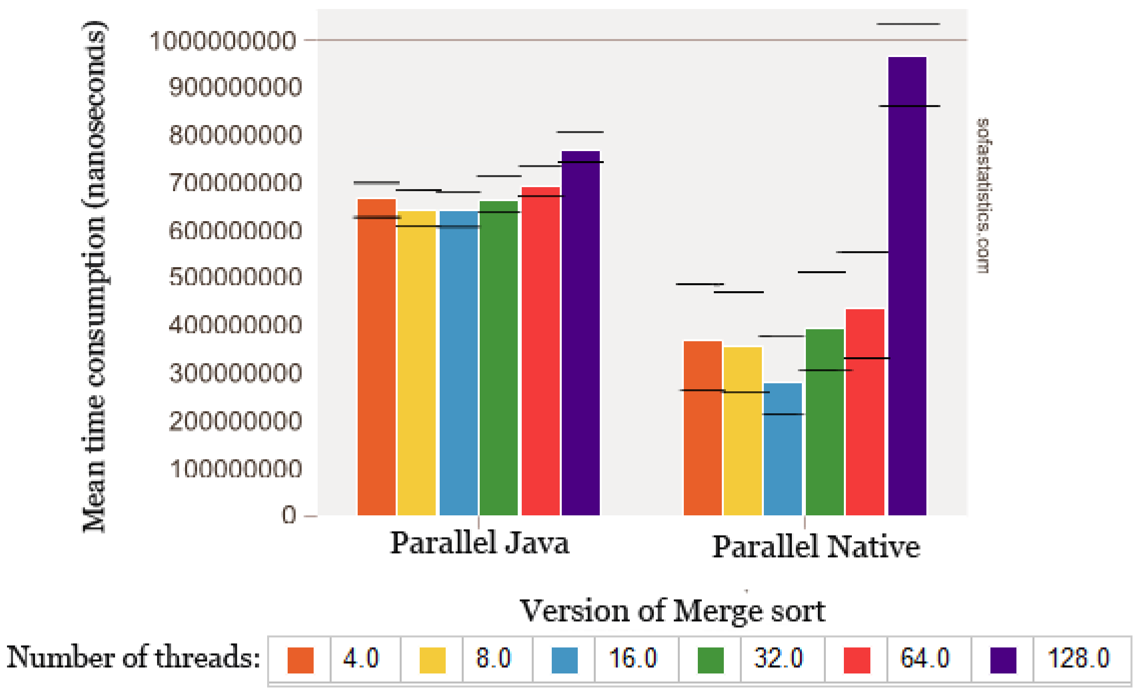


Figure 8 Mean time consumptions of threaded merge sort algorithm with variation in number of threads and 95% confidence intervals

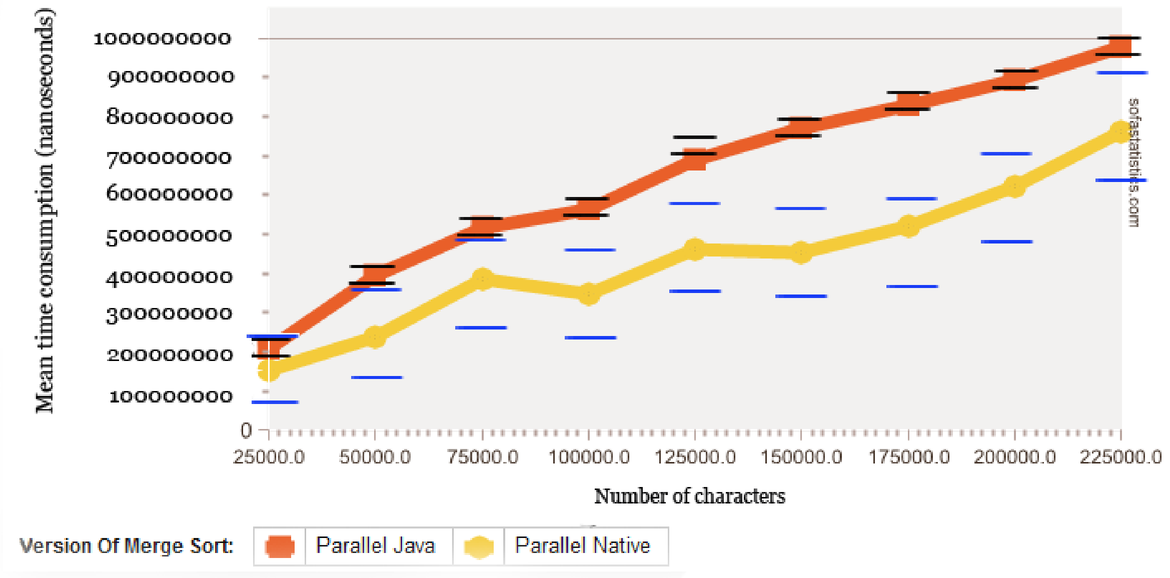


Figure 9 Mean time consumption of threaded merge sort algorithms with variation in number of elements and 95% confidence intervals

As to what concerns the threaded algorithm calculating the Fibonacci-sequence, the algorithm was run with Fibonacci-numbers ranging from 20 to 40, and with numbers of threads ranging from 4 to 128. By combining these factors in a manner similar to the one used in the previous algorithm, it gave rise to the results as seen in figure 10 and 11.

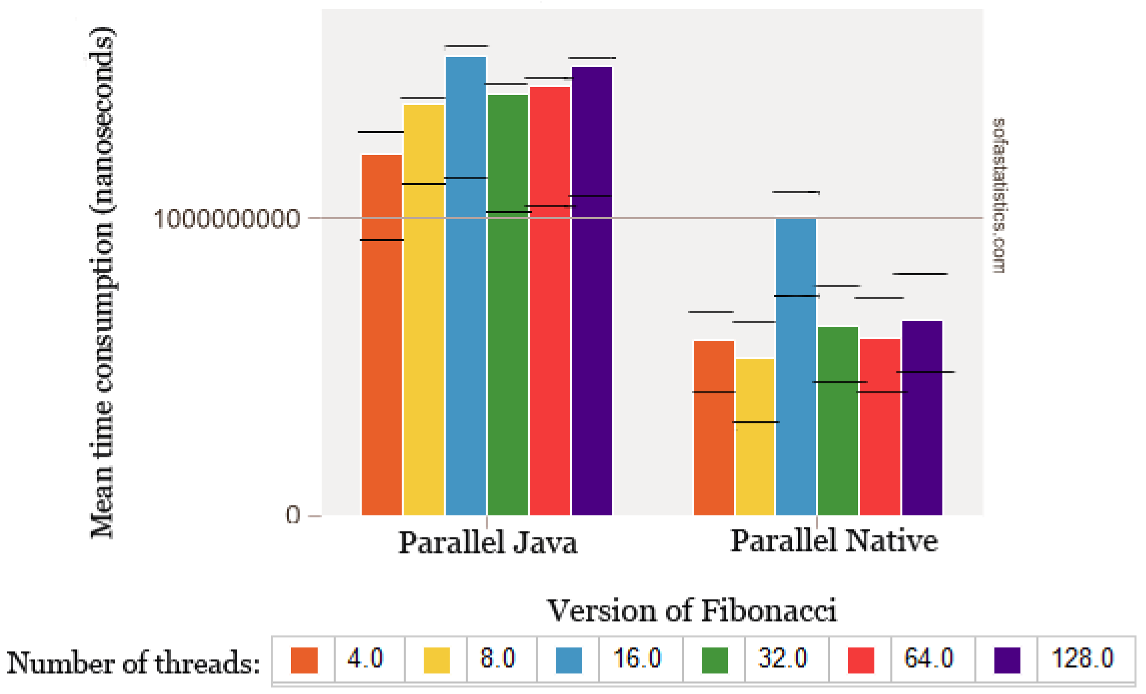


Figure 10 Mean time consumption of threaded algorithm calculating the Fibonacci sequence with variation in number of threads and with 95% confidence intervals

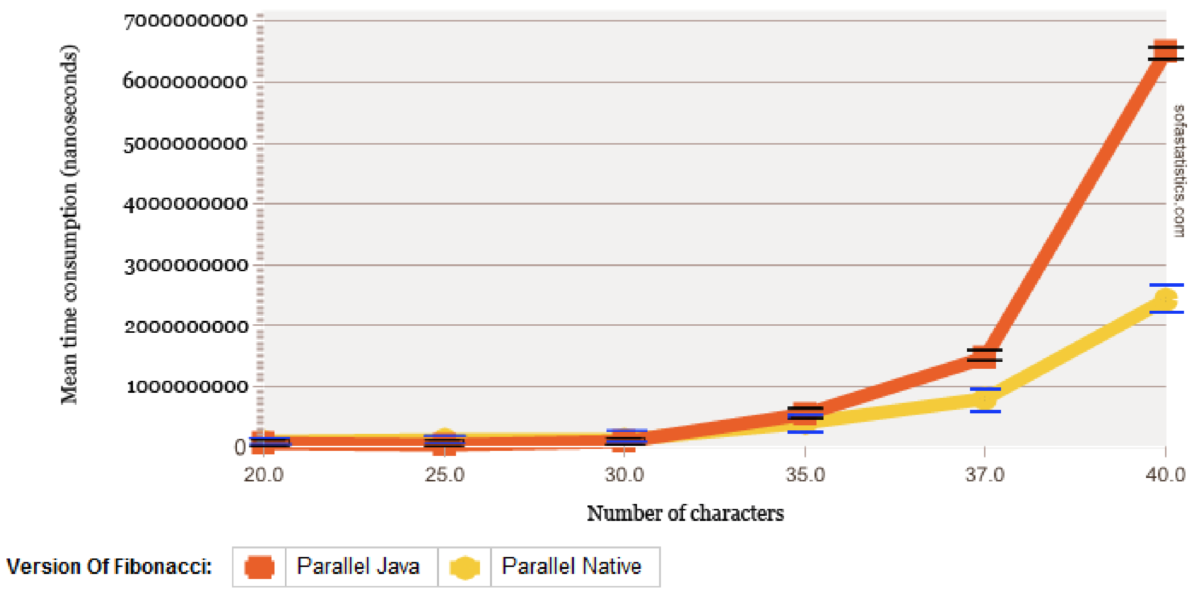


Figure 11 Mean time consumption of threaded algorithm calculating the Fibonacci sequence with variation in Fibonacci number and with 95% confidence intervals

Results for the last algorithm, the string-search algorithm, can be seen in figures 12 and 13. The algorithm was run with an auto-generated lorem-ipsum textfile with size of this text ranging from 5000 characters to 75 000 characters. In this text the word “lorem” was used as a keyword to search for. To minimize the risks of the boundary-section problem, each boundary was checked to ensure that it did not contain the word “lorem”.

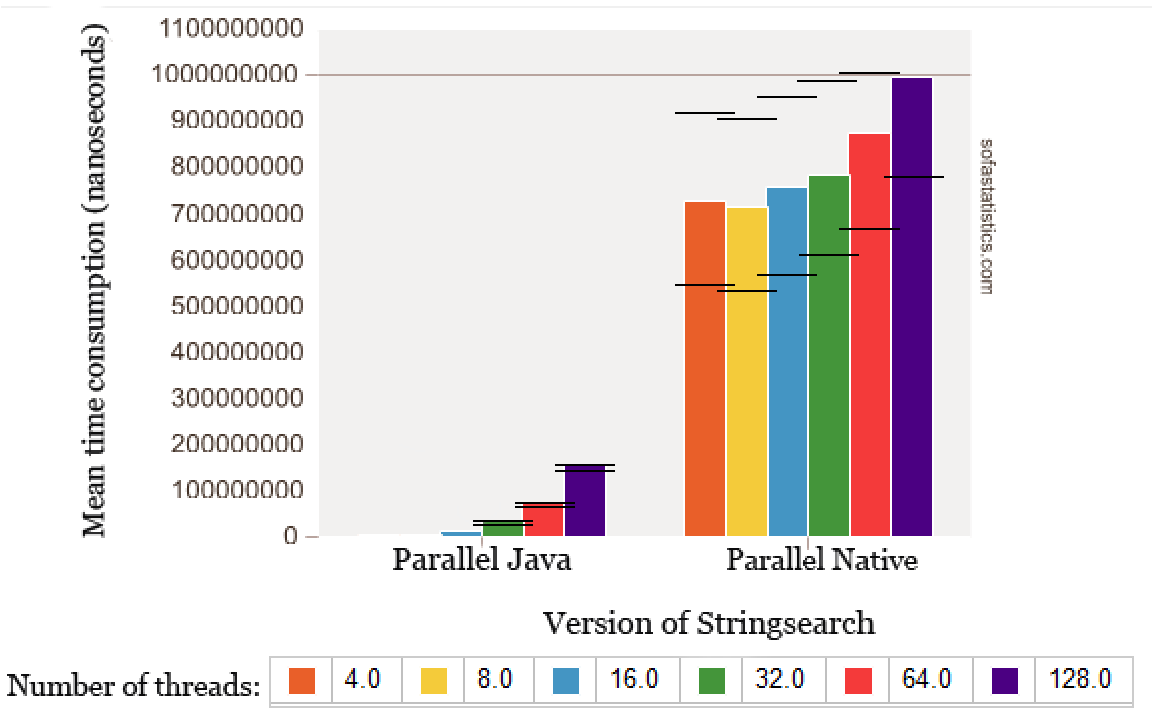


Figure 12 Mean time consumption of threaded string search algorithm with variation in number of threads, 95% confidence intervals

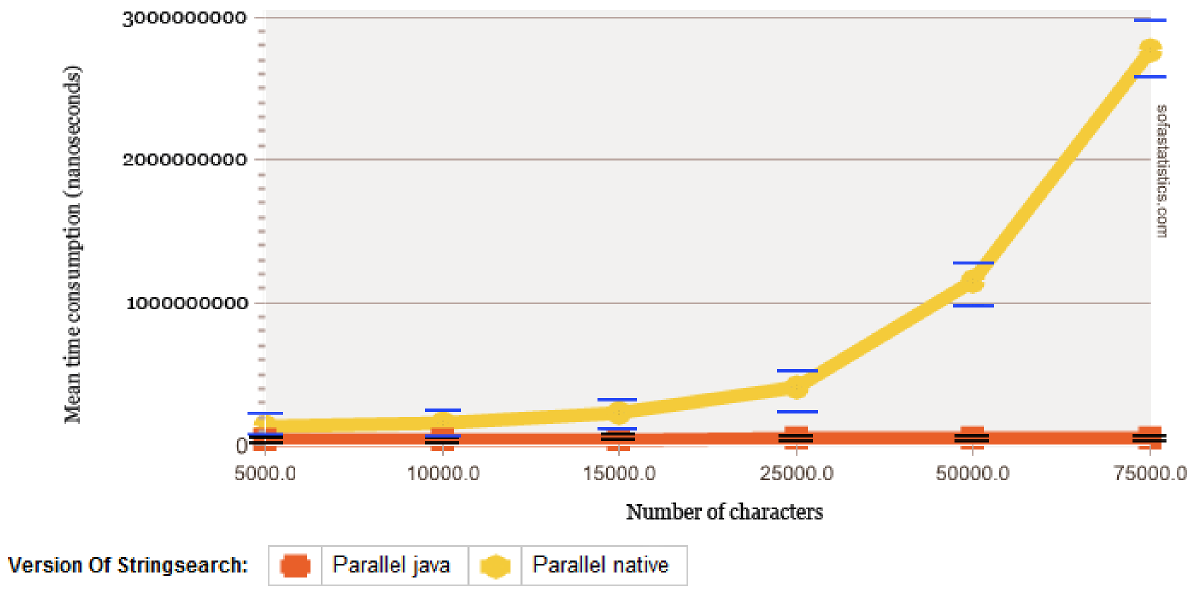


Figure 13 Mean time consumption of threaded string search algorithm with variation in number of characters and 95% confidence intervals

6.2 Discussion

Based on the results from the previous section, the following observations can be made.

Before we reach 128 threads in the merge-sort algorithm, the native version has a lower mean time-consumption compared to the equivalent Java-written version. It seems unreasonable to go beyond 16 threads since this appears as a turning-point where an increased number of threads will only increase the time-consumption. It is however interesting to note that when we reach 128 threads, the Java version actually has a lower time-consumption than the native version. What causes this is unknown, but it is probably related to the nature of the algorithm and not by temporary circumstances caused by external processes affecting the execution of the application since plentiful measures were taken to ensure that the results would not be interfered by such causes.

While looking at figure 9 one can see that when the number of elements increase, the times taken by the Java and the native version both grow in a linear manner. In other cases such as the one with the Fibonacci algorithm, it becomes evident that when the input is increased, the average time required by the Java version escalates faster than the native version.

For the sake of comparison, a non-threaded version of merge-sort was also implemented. Running this algorithm with the same input as in the threaded versions, results showed that in this non-threaded case the native version implemented in C++ was roughly 79 percent faster than the equivalent version implemented in Java. This can be put in relation to the threaded version of merge-sort in which the biggest speedup was observed in the 16-threaded case where the native version written in C++ was 53 percent faster than the Java-written version.

For what concerns the threaded algorithm calculating the Fibonacci sequence, the native version written in C++ always had a lower mean time-consumption compared to the non-native version written in Java no matter how many threads were used in the experiments. But despite this, there were cases where the differences were not that big. For example, the case with 16 threads clearly stands out from the rest. It is not only the case with the highest average mean time-consumption in each version, but it was also the case where the difference between the Java and native version was the smallest. It is unknown why the 16-threaded case stands out from the rest, but it probably has to do with the nature of the algorithm since the increased time-consumption in the 16-threaded case is visible both in the Java and C++ version. If the increased time would have been caused by an external process occupying the processor then the sudden increase of time would probably not have been visible in both the Java and the C++ version.

The Fibonacci-version seems quite naïve since we are performing the same calculations over and over again. One could easily construct a more effective solution by saving the results from each calculation in a table and then have this table checked before another value is calculated. But if this was done there would probably not be much of a point to write a threaded version. The focus has not been to write algorithms that are as effective as possible, the focus has been to write algorithms that are as equal as possible.

Lin, Lin, Dow and Wen (2011) used an algorithm calculating the Fibonacci-sequence recursively as one out of many algorithms to compare the performance between Android-applications written in Java and C/C++. They concluded that the algorithm implemented in the native language was almost 86 percent faster than the equivalent version implemented in Java. Their algorithm did calculate the Fibonacci sequence in a recursive way which is similar to how the calculations were done in this work, but they only tested their algorithm with Fibonacci of 35 as input. To get results that could be put in contrast to the threaded version, a non-threaded algorithm equal to the one used by Lin et al. (2011) was also implemented. By running this algorithm with numbers ranging from 20 to 40 as in the threaded version, results showed that from an average perspective the native version was as much as 96 percent faster than the equivalent version implemented in Java.

In section 3.1, the following hypothesis was defined:

"Since the workload is distributed over several concurrently running threads, the differences in time-consumption between implementations of threaded Android-applications in Java and C/C++ will no longer be as remarkable as they are in the non-threaded case."

It appears that the results of the threaded Fibonacci-algorithm aligns pretty well with the hypothesis since no matter how many threads were used in these experiments, it never came close to being as much as 96 percent faster (the biggest gain in performance was seen in the 8-threaded case with an average speedup of 62 percent, whereas the smallest gain in performance was seen in the 16-threaded case where the average speedup was 35 percent).

When it comes to the last algorithm, the stringsearch-algorithm, the results were drastically different from the other cases. While looking at figure 13, one can tell that an increase in number of characters barely affects the performance of the algorithm written in Java. Before they reach 10 000 characters in input-size they perform quite equally. But as the number of characters exceed 10 000, the native version starts to increase in an exponential pace, while the mean time consumption of the java version is almost constant or possibly slightly increasing in a linear manner.

During these experiments the number of occurrences of the pattern "lorem" within the auto-generated lorem-ipsum-text was kept constant. It might be interesting to vary the number of occurrences of the pattern within the text since an increased number of occurrences will cause the algorithm to perform more iterations and thereby affect the performance negatively.

As mentioned in section 5.1, Lee and Lee (2011) used a string processing algorithm when comparing the performance between Android-applications written in Java and C/C++. They also found that the Java version was far more efficient than the native version (but they did not provide any comparable numbers). They did however claim that the difference in performance is related to the encoding of characters.

6.3 Conclusions

The first conclusion that can be drawn from the observations in section 6.2 is that in the case with threaded sorting-algorithms and algorithms involving simple arithmetic operations, writing your Android-applications in C/C++ can result in improvements. It seems like these improvements are smaller compared to the non-threaded case, but it is probably too soon to tell whether it holds as an "universal truth" that the improvements in time-consumption seen on the Android-platform in the non-threaded case are always lowered in the threaded case. The results in this work do however show clear indications that this might be the actual case. As observed in the non-threaded case, the threaded string search algorithm implemented in Java has a significantly lower time-consumption in the case with the string search algorithm compared to the equivalent version implemented in C++. It is likely that this phenomenon holds for other types of threaded string processing algorithms as well, but further research is needed to validate this.

7 Concluding remarks

7.1 Summary

The aim of this work was to answer the following question:

"Are natively compiled threaded Android-applications more time-efficient than non-natively compiled threaded applications?"

In order to answer this question two literature analyses were performed where the outcome was threaded algorithms that could be used to compare the performance between threaded Android-applications written in Java (non-natively compiled) and C/C++ (natively compiled). As the chosen algorithms were implemented and evaluated, results showed that in the case with threaded sorting-algorithms and algorithms involving simple arithmetic operations, writing your Android-applications in C/C++ can result in improvements. Furthermore, the results show clear indications that in these cases where one could previously make big improvements by implementing a performance-critical part in C or C++, as these cases become threaded the gains are not as big as they are in the non-threaded case. Results also showed that in the case with threaded string algorithms, writing your threaded string algorithm in C or C++ instead of Java will most likely not result in improvements.

7.2 Future Work

While looking back at how this study was performed, several things come to mind. First of all it would have been interesting to investigate more cases with other types of algorithms, but due to restrictions in time algorithms with simple arithmetic operations, sorting functions and string processing were prioritized since they were the most commonly used algorithms in previous research.

A main difficulty with the results in this work is the ability to tell whether the results apply to all types of algorithms within the same area. For example, the threaded algorithm calculating the Fibonacci sequence belongs to the category of threaded algorithms with simple arithmetic operations. The question is whether the results of this algorithm are representative for all threaded algorithms with simple arithmetic operations. Because of this, it would be interesting to study the generalizability of the results in this work (perhaps by implementing more threaded algorithms within the same category of algorithms).

Not only would it be interesting to study more threaded algorithms within the same categories as used in this work, it would also be interesting to study other types of threaded algorithms such as float-calculations, user-made datstructures, and other types of sorting-algorithms. One could then put them in contrast to equivalent non-threaded algorithms and thereby determine if the hypothesis holds for these algorithms just as well.

Secondly, the results were run on a quad-core (4 cores) device. It would be preferable to perform the exact same tests on another hardware device with a dual-core processor (2 cores) since a change in hardware can affect the performance both negatively and positively. This would probably make the results more generalizable and applicable to other platforms.

8 References

ACM. (2014). *ACM Digital Library*. Retrieved June 17, 2014, from http://dl.acm.org/

Apple. (2014a). *Apple Developers Program*. Retrieved June 17, 2014, from Apple Developer: https://developer.apple.com/programs/

Apple. (2014b). *Apple - iOS7 - What is iOS*. Retrieved June 17, 2014, from Apple: https://www.apple.com/ios/what-is/

Artho, C., Hagiya, M., Leungwattanakit, W., Tanabe, Y., & Yamamoto, M. (2010). Model Checking Of Concurrent Algorithms: From Java to C. *International Federation for Information Processing 2010*, 90-101.

Berndtsson, M., Hansson, J., Olsson, B., & Lundell, B. (2008). *Thesis Projects - A Guide for Students in Computer Science and Information Systems*. Springer.

Carlsson, J. (2011). *Androidapplikationer - effektivisering ur ett energiperspektiv*. University of Skövde.

Creative Commons. (2014a). *Creative Commons - Attribution - NonCommercial-ShareAlike 3.0 Unported - CC BY-NC-SA 3.0*. Retrieved June 17, 2014, from Creative Commons: http://creativecommons.org/licenses/by-nc-sa/3.0/

Creative Commons. (2014b). *Creative Commons - Attribution 2.5 Generic - CC BY 2.5*. Retrieved June 17, 2014, from Creative Commons: http://creativecommons.org/license/by/2.5/

DiVA. (2013). *Digitala Vetenskapliga arkivet*. Retrieved June 18, 2014, from DiVA Portal: http://www.diva-portal.org/smash/search.jsf

Gagne, G., Galvin, P. B., & Silberschatz, A. (2010). *Operating system concepts*. John Wiley & Sons.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices - Proceedings of the 2007 OOPSLA conference*, 57-76.

Google Inc. (2014a). *Android SDK*. Retrieved June 17, 2014, from Android Developers: http://developer.android.com

Google Inc. (2014b). *Android NDK*. Retrieved June 17, 2014, from Android Developers: http://developer.android.com/tools/sdk/ndk

Google Inc. (2014c). *Android Security Overview*. Retrieved June 17, 2014, from Android Source Code: http://source.android.com/devices/tech/security/

Google Inc. (2014d). *Android Source Code - Downloading and Building*. Retrieved June 17, 2014, from Android Source Code: https://source.android.com/source/building.html

Google Inc. (2014e). *Dalvik Technical Information*. Retrieved June 17, 2014, from Android Source Code: http://source.android.com/devices/tech/dalvik/index.html

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., & Brown, R. B. (2001). MiBench: A free, commercially representative embedded benchmark suite. *IEEE International Workshop on Workload Characterization*, 3-14.

IDC. (2014). *IDC Worldwide Mobile Phone Tracker*. Retrieved June 16, 2014, from http://www.idc.com/getdoc.jsp?containerId=prUS24442013

IEEE Xplore. (2014). *IEEE Xplore digital library*. Retrieved June 18, 2014, from http://ieeexplore.ieee.org/Xplore/home.jsp

Intel Corporation. (2012). *Android* Application Development and Optimization on the Intel® Atom Platform*. Retrieved June 17, 2014, from https://software.intel.com/en-us/articles/android-application-development-and-optimization-on-the-intel-atom-platform

Kim, Y.-J., Cho, S.-J., Kim, K.-J., Hwang, E.-H., Yoon, S.-H., & Jeon, J.-W. (2012). Benchmarking Java application using JNI and native C application on Android. *2012 12th International Conference on Control, Automation and Systems*, 284-288.

Lawrence Livermore National Laboratory. (2014). Retrieved June 17, 2014, from POSIX Threads Programming: https://computing.llnl.gov/tutorials/pthreads/#AppendixA

Lee, J. K., & Lee, J. Y. (2011). Android programming techniques for improving performance. *2011 3rd International Conference on Awareness Science and Technology*, 386-389.

Lilja, D. J. (2004). *Measuring computer performance - A practitioner´s guide*. Cambridge University Press.

Lin, C. H., Tsai, S. Y., Liu, C. H., & Chang, S. C. (2010). Accelerating string matching using multi-threaded algorithm on GPU. *Global Telecommunications Conference 2010 IEEE*, 1-5.

Lin, C.-M., Lin, J.-H., Dow, C.-R., & Wen, C.-M. (2011). Benchmark dalvik and native code for Android system. *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, 320-323.

Mahafzah, B. (2013). Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. *Springer Science*.

Meier, R. (2010). *Professional Android™ 2 Application Development*. Wiley Publishing.

Open Handset Alliance. (2014a). *Alliance Members*. Retrieved June 17, 2014, from http://www.openhandsetalliance.com/oha_members.html

Open Handset Alliance. (2014b). *Android overview*. Retrieved June 17, 2014, from http://www.openhandsetalliance.com/android_overview.html

Oracle docs. (2010). *System (Java 2 Platform SE 5.0)*. Retrieved June 17, 2014, from http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html

Oracle docs. (2013). *Thread (Java Platform SE 7)*. Retrieved June 16, 2014, from http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Rolfe, T. (2010). Programming: Parallel merge sort implementation. *ACM Inroads Vol. 1 Issue 4*, 72-29.

Sony mobile. (2014). *Xperia Z1 Compacy | Camera phone - Sony Smartphones (Global UK English)*. Retrieved June 18, 2014, from http://www.sonymobile.com/global-en/products/phones/xperia-z1-compact/

Springer Link. (2014). *Home - Springer*. Retrieved June 16, 2014, from http://link.springer.com

Wohlin, C. R. (2012). *Experimentation in Software Engineering*. Springer.