**KTH Computer Science
and Communication**

Native code on Android

A performance comparison of Java and native C on Android

ANDREAS ULVESAND, DANIEL ERIKSSON

Abstract

This report evaluates possible performance differences between Java and native C on the operating system Android, by developing tests and analyzing the execution. The ambition is that each test should evaluate the performance of a certain task, such as memory access or arithmetic operations of different data types. The results were in some cases unexpected and show that the executed implementations were faster on C compared to Java on one of the test devices, but not the other. The conclusion partly opposes earlier research and this is probably partly due to the fact that the Java Virtual Machine has been improved vastly in the latest versions of Android.

Keywords: Android, smartphone, Java, C, native, NDK, performance evaluation

iv

Sammanfattning

Den här uppsatsen jämför eventuella prestandaskillnader mellan Java och native C på operativsystemet Android genom att utföra olika tester och analysera dessa. Testerna baseras på några utvalda algoritmer och körs på två olika fysiska Android-enheter. Ambitionen är att varje test ska undersöka prestandan av en viss typ av operation, till exempel minnesaccess eller aritmetiska operationer på olika datatyper. Resultaten var i några fall oväntade och visar att de exekverade implementationerna var snabbare i native C jämfört med Java på den ena testenheten, men att prestandan var likvärdig på den andra testenheten. Resultaten går delvis emot tidigare forskning och kan troligen förklaras med att Javas virtuella maskin förbättrats avsevärt i senare versioner av Android.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

The Android platform is relatively new yet one of the leading platforms for mobile devices. The popularity is increasing rapidly and by the end of 2010 Android led the market sales in favor of its competitors [1]. Esteemed advisory firm Gartner forecasts Android will be the by far biggest mobile operating system by 2012 [2].

The standardized way of developing applications for the Android platform is using Java and XML [3], but there is also a tool for Android development using native programming languages such as C and C++ called Native Development Kit (NDK) [4]. A compiled Java program can be executed on any platform that runs a JVM, but unfortunately Java has a reputation of showing poor performance [5]. Native C/C++ code however, which does not need a virtual machine since it is compiled into machine code to run on a specific processor, is in theory faster [5].

Support for native development on the Android platform improves constantly, and this creates possibilities to easier implement performance-critical parts of an application (or even the entire application) in native C/C++ code [6]. This is usually accomplished using Java Native Interface (JNI), but since programmers need to learn another interface in order to utilize native code the complexity factor inevitably increases [7].

The above facts constitute a need for investigation in which cases it is beneficial to replace certain parts of an application with native C/C++ code and compare it to the Java equivalent. This way one will optimally be able to decide when a certain developing technique is preferred and thus guide fellow Android developers.

1.2   Background

The Native Development Kit (NDK) is relatively new and few related studies have been made, some indicating performance benefits by using native code [7]. The conclusions made by Lee et al. [7] and Schmidt et al. [8] are however disputable since their experiments have been done using an Android emulator and not an actual Android device. Also, JIT compilation which has now been introduced to new Android versions is not tested nor mentioned in their reports.

Other more complex studies concerning this specific subject have not been found. Experiments using JNI in regular Java applications is no news and have proven to be an escape path for many Java users [9]. This report however will deal with Android applications exclusively.

1.3   Problem statement

Developing using Java is the standardized and "easy" way of writing applications for the Android platform, but reports [7] [8] show the possibility of certain algorithms running faster utilizing native C/C++ code, compared to Java.

By implementing native C code on the Android platform, this report will investigate the possible performance difference (with respect to time) compared to the standardized approach of using Java. More specifically; the intention is to investigate in which situations there is a significant performance benefit using C, and in which situations the increased difficulty and complexity level simply does not pay off.

1.4   Purpose

The purpose of this report is to benchmark different algorithms comparing native C performance with Java performance and optimally guide developers concerning native C development on Android. More specifically in which situations a native C implementation could be useful and motivated, taking performance (with respect to time) and the added complexity factor into account.

1.5   Statement of collaboration

The production of this report has involved several tasks such as acquire knowledge by reading educational literature and research articles, developing and implementing the tests, formation of hypothesis for each of the tests as well as extensive writing on the report itself. The two authors of this report have collaborated together and both have been involved and have contributed in each of the tasks describe above.

Chapter 2

Method

2.1 Approach

This report will present implementations of different fundamental algorithms in Java and C and the corresponding execution results in order to analyze the possible performance benefits of using native C code. The intention is to develop fundamental algorithms performing sets of computational operations, such as arithmetic operations and fundamental data processing, in order to reach a conclusion from the results. Rather than measuring the overall system performance or entire programs including UI, this report intend to measure sets of these operations.

The implementations will test different tasks such as memory access and arithmetic calculation with different data types, and they will optimally form a software micro-benchmarking suite.

2.2 Motivation

Several approaches can be used to reach a conclusion whether native C code is faster than Java on Android. In this case a practical approach is more desired, since a theoretical approach (analyzing and comparing Java with native C code) would be hard to verify and not yield any real measurable output. This report will benchmark fundamental algorithms and not overall system performance of whole programs including UI, since that would probably be irrelevant and very complex with too many variables taken into account. Benchmarking will yield a set of output and thus enabling the possibility of testing the hypothesis with real data, and from that hopefully being able to draw a conclusion.

Experiments show that in order to reach a conclusion concerning performance, implementation of different algorithms, is necessary [7][8].

2.3   Testing platform

This section presents the two devices that were used as testing platforms. It also presents the tools that were used in order to develop the tests. The section does not aim to thoroughly present each device concerning hardware and operating system, but rather the parts that is interesting with respect to the tests.

2.3.1   Devices

The implementations will be executed on two different Android devices, namely HTC Hero and HTC Desire HD. The specification of the two devices is presented below: [10][11]

	HTC Hero	**HTC Desire HD A9191**
Release date	July 2009	October 2010
Android Version	2.1 Eclair (update 1)	2.2 Froyo
CPU	528 MHz	1000 MHz
	Qualcomm MSM7200A	Qualcomm Snapdragon MSM8255
RAM / ROM	288 MB / 512 MB	768 MB / 1536 MB

Hero and Desire HD represent a medium priced and a high priced top of the line smartphone respectively, with the former one being around one and a half years old and the latter one released half a year ago, as of today.

2.3.2   Operating system

The Android versions that run on the two test devices and thus will be tested are Android 2.1 Eclair and Android 2.2 Froyo. There are significant differences between the two versions, a few of them effecting performance.

The most noticeable improvement to Froyo is that Dalvik introduced a performance boost, made possible by utilizing JIT compilation. This means that certain chunks of Java bytecode will be translated and optimized into native code. Depending on the complexity of a JIT implementation this can be a huge performance benefit. The Froyo JIT implementation is designed specifically for the environment in which handheld devices operate, that is minimal memory usage in order to prevent power drain and quick delivery of performance boost. The Froyo JIT implementation is initially designed with regards to the Trace Granularity JIT pattern, which is detecting "hot" chunks of code (for example finding loops that can be optimized) and leaving out "cold" chunks of code. This way only the most necessary chunks of code will be optimized leaving the rest. This is in contrast to the Method Granularity JIT pattern which focuses on optimizing whole methods and thus happens to compile cold code within hot methods. [12]

Another improvement to Froyo is regarding kernel memory management. Benchmark results has shown around 20 times better performance measuring memory reclaim. [13]

2.3.3 CPU

Hero has a Qualcomm MSM7200A processor and Desire HD has a Qualcomm MSM8255 Snapdragon processor. MSM7200A was released 2007 and is thus relatively old compared to MSM8255 which was released 2010. They are based on different CPU cores and different instruction sets, and have maximum recommended clock frequency at 528 MHz and 1000 MHz respectively. [14] [15]

A Vector Floating-Point coprocessor (VFP) is used by the processor exclusively to calculate floating points, making this type of calculation faster by utilizing dedicated hardware. The MSM7200A in Hero utilizes the CPU core ARM1136EJ-S which does not have a Vector Floating-Point coprocessor (VFP), and thus have no hardware for calculating floating points. [16] In Desire HD however, the MSM8255 processor based on the Cortex A8 CPU core is used, which have a VFP coprocessor implementing the VFPv3 architecture. [17]

2.3.4 Developer tools

The Java implementations were compiled using standard Java (Sun 1.6.0_24) compiler and the native C implementations were compiled using the standard NDK (Revision 5b) compiler (ARM GCC 4.4.3). The following settings were exclusively enabled for Desire HD:

```
LOCAL_ARM_MODE := arm
APP_ABI := armeabi-v7a
TARGET_CPU_ABI := armeabi-v7a
```

These settings, written in the make file, are used to build for the ARM v7 instruction set, in order for Desire HD to utilize improved instruction set (compared to standard ARM v6 that Hero uses) [18].

The C compiler is using standard soft-fp, which is an implementation for calculation of floating points. This is one of the fastest implementation for software calculation of floating points for GCC [19].

Each test was executed multiple times and the statistics presented in the Result section is thus the arithmetic average. As for the Java implementations the algorithms were "warmed up" before timing, to give the JIT compiler a chance to possibly optimize them. Input for each test was determined as to match both devices considering time (the tests need to run for a while in order to notice possible differences) as well as memory space (especially Hero have a rather small memory and

can thus not run tests using too much memory). This is done all in accordance with the Benchmarking article written by Boyer. [20]

Chapter 3

Implementation

3.1 Prime Number calculation

The purpose of the test Prime Number Calculation is to implement an algorithm that perform calculations with integers, thereby exposing the possible performance difference between a native and a standard Java implementation of basic operations on integers.

Method calls to the algorithm include a parameter of how many prime numbers to calculate. The calculations of these numbers begin at zero and ends when the maximum numbers have been reached.

Pseudo code:

```
primeNumberSearch(max : # primes to calculate) {
    for(i=2; nr < max; i++) {
        for(j=2; j <= i/2; j++) {
            if(( i % j ) == 0) break;
        }
        if(j > i/2) nr++;
    }
    return timeElapsed;
}
```

3.2 Floating Point calculation

A floating point is a standard primitive type typically used for representation of non-integer numbers. Floating point calculation is usually performed by a hardware unit called the Floating Point Unit (FPU), but some processors' lack this unit and is therefore software calculated. Software based calculation is usually much slower than calculation utilizing dedicated hardware. [17]

The purpose of the implementation is to perform basic operations (such as addition, subtraction, multiplication and division) on floating points and measure the possible performance difference.

Pseudo code:

```
floatCalculation(max : # iterations, float : a float used for calculation) {
    for(i = 0, i < max; float++, i++) {
        temp = temp * float;
    }
    for(i = 0; i < max; float++, i++) {
        temp = temp / float;
    }
    for(i = 0; i < max; float++, i++) {
        temp = temp + float;
    }
    for(i = 0; i < max; j++) {
        temp = temp - float;
    }
    return timeElapsed;
}
```

3.3   Recursion

Recursion is when the solution of a problem depends on dividing the problem into smaller instances of the same problem. Recursion result in a lot of method calls, which is interesting to benchmark. It is also interesting since recursion is frequently used in computer science generally.

Sorting a list is a common problem, where Quicksort is a fast recursive algorithm. Quicksort therefor makes an excellent algorithm to test. Since our experiments show that the JIT communication delay is irrelevant (a simple measurement confirmed this), time can be measured outside the method. Measuring time inside the method is more complex when using recursion.

Quicksort chooses a pivot element and reorder its list so all elements to the left of the pivot element are lower than those to the right. Then, the algorithm recursively calls itself with both the list of the smaller elements and the list of the greater elements until the list contains no more than one element. When all base cases have been reached, the algorithm is finished.

Pseudo code:

```
function quicksort(array){
    var list less, greater
    if length(array) <= 1
      return array  // an array of zero or one elements is already sorted
    select and remove a pivot value pivot from array
    for each x in array {
        if x <= pivot then append x to less
        else append x to greater
    }
    return concatenate(quicksort(less), pivot, quicksort(greater))
}
```

3.4 Memory Access

The Memory access test is an implementation of an algorithm that performs a lot of memory accesses, namely Bubblesort. Memory access is an important benchmark due to the fact that most applications read from the memory frequently.

Bubblesort is an algorithm known for its frequent use of memory. The algorithm will compare all pair of elements from left to right and switch elements that are not sorted. When the end of the list is reached, the algorithm will start all over until the last iteration contains no new changes. This simulation will benchmark worst case performance which is $O(n^2)$. Worst case scenario occurs when the last element needs to be sorted to the first position. Each iteration throughout the list will then access and compare every element twice and only move the lowest element one step in the correct direction. With a list of n elements, this will result in 2*n*n memory accesses.

This is a basic example of a worst case scenario, memory accesses are marked with bold type:

123450 -> 123450 -> 123450 -> 123450 -> 123405
123405 -> 123405 -> 123405 -> 123045 -> 123045
123045 -> 123045 -> 120345 -> 120345 -> 120345
120345 -> 102345 -> 102345 -> 102345 -> 102345
102345 -> 012345 -> 012345 -> 012345 -> 012345
012345 -> 012345 -> 012345 -> 012345 -> 012345 -> list sorted

Figure 3.1. A diagram showing how bubblesort sorts a list.

Pseudo code:

```
function bubbleSort( A : list of sortable items ){
 do
   swapped = false
   for each i in 1 to length(A) - 1 inclusive do:
     if A[i-1] > A[i] then
       swap( A[i-1], A[i] )
       swapped = true
     end if
   end for
 while swapped
end procedure
}
```

Chapter 4

Evaluation

4.1 Prime Number calculation

4.1.1 Hypothesis

Similar prime number calculation tests performed on a computer running Microsoft Windows Server 2003 with modern compilers [21] show that both C and C# are significantly faster than the Java implementation, where C is about 5 times faster and C# about 3 times faster. There are no strong reasons to believe that a similar Android implementation would perform considerably different and therefor, this particular test will probably show to be faster in C also on Android. However, since the JVM of Hero is not utilizing JIT compilation the native C implementation on Hero may be even more superior to the Java implementation, while it will probably be a slightly closer call on Desire HD.

4.1.2   Result

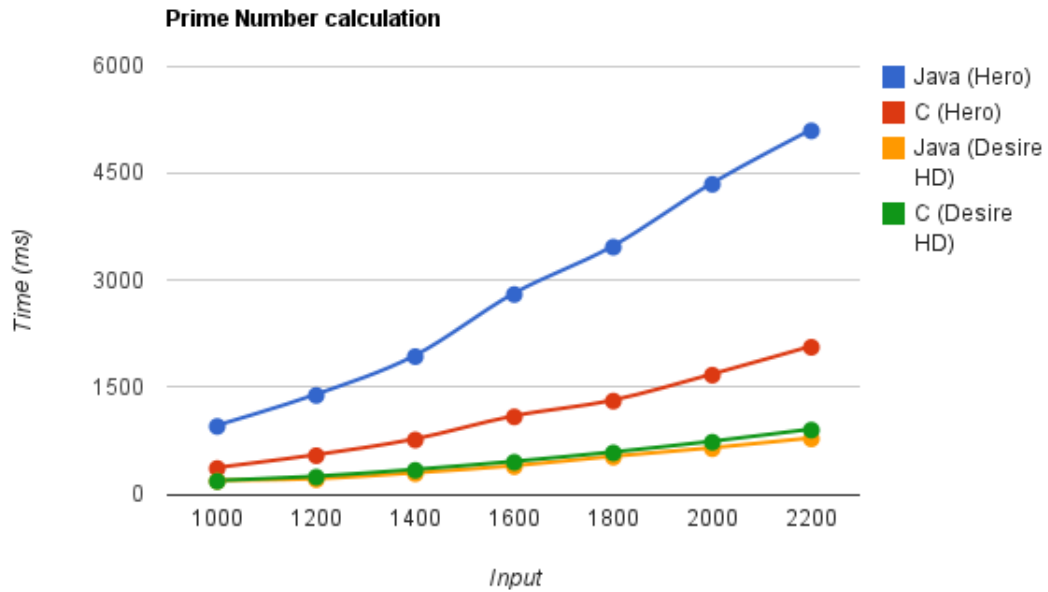The result is presented in the figure below.

Figure 4.1. A graph showing timed performance in milliseconds for each device and implementation of the Prime Number calculation test. The plots are estimated from 7 measured data of different input parameters.

4.2   Floating Point calculation

4.2.1   Hypothesis

A floating point calculation test performed on Microsoft Windows Server 2003 [21] shows that the C implementation is about 6 times faster than the Java implementation, which could work as an indication for this particular test, namely that C will be faster also on Android. However, since one of the test devices (Desire HD) has a VFP coprocessor and the other one does not, it is likely to believe that there are significant performance differences between the two test devices, with Desire HD being much faster, since it uses dedicated hardware to calculate floating points instead of software. The C implementation on Desire HD though is not utilizing VFP; see more details in 2.3.4 Developer tools. This means that Java will probably be faster than C, at least on Desire HD.

4.2.2 Result

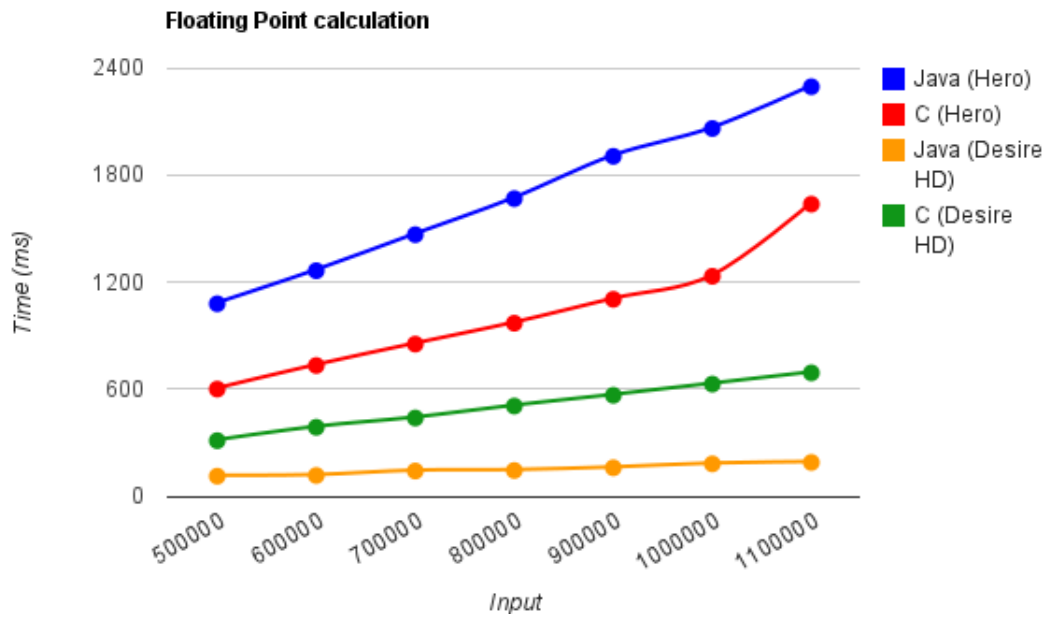The result is presented in the figure below.

Figure 4.2. A graph showing timed performance in milliseconds for each device and implementation of the Floating Point calculation test. The plots are estimated from 7 measured data of different input parameters.

4.3 Recursion

4.3.1 Hypothesis

This test will produce lots of method calls. Informal sources state that method calls in Java are basically free, where one of the arguments consist of a smart compiler that will inline automatically [22]. Inline boosts performance by inlining the code instead of making a method call. In recursion however, this is not a possibility due to the constant method calls that results in an immense amount of code to inline. Java is still expected to hold strong against C.

4.3.2   Result

Two Recursion tests were performed since Hero could not handle any greater input numbers. The bigger test was performed in order to notice any difference between the two implementations. The results are presented in the figures below.

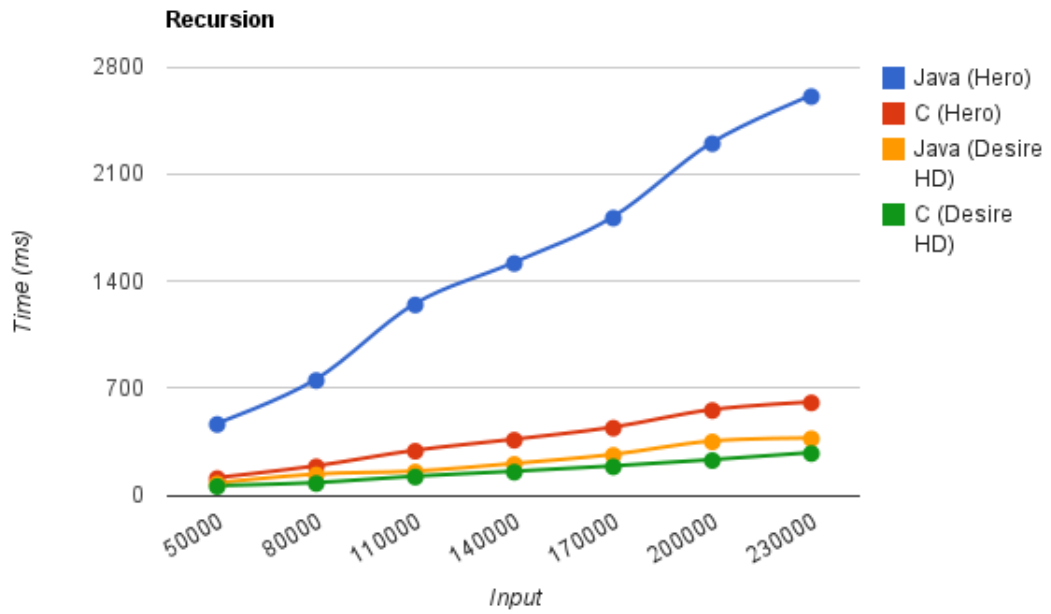**Figure 4.3.** A graph showing timed performance in milliseconds for each device and implementation of the Recursion test. The plots are estimated from 7 measured data of different input parameters.
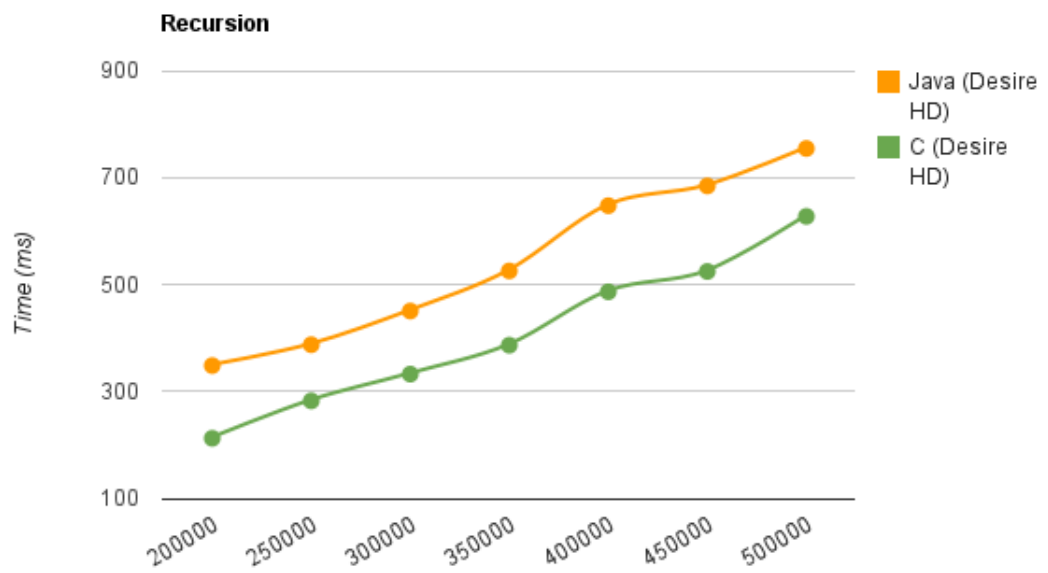
Figure 4.4. A graph showing timed performance in milliseconds for Desire HD and each implementation of the Recursion test. The plots are estimated from 7 measured data of different input parameters.

4.4   Memory Access

4.4.1   Hypothesis

Memory Access is a benchmark where the difference between Java and native C code is expected to be the most. Sources indicate that memory access in Java is one of its bottlenecks partly due to bound checks [23]. Benchmarks show results of huge difference in performance [6]. Performance expectations of the native C implementation is therefor high compared to the Java implementation.

4.4.2   Result

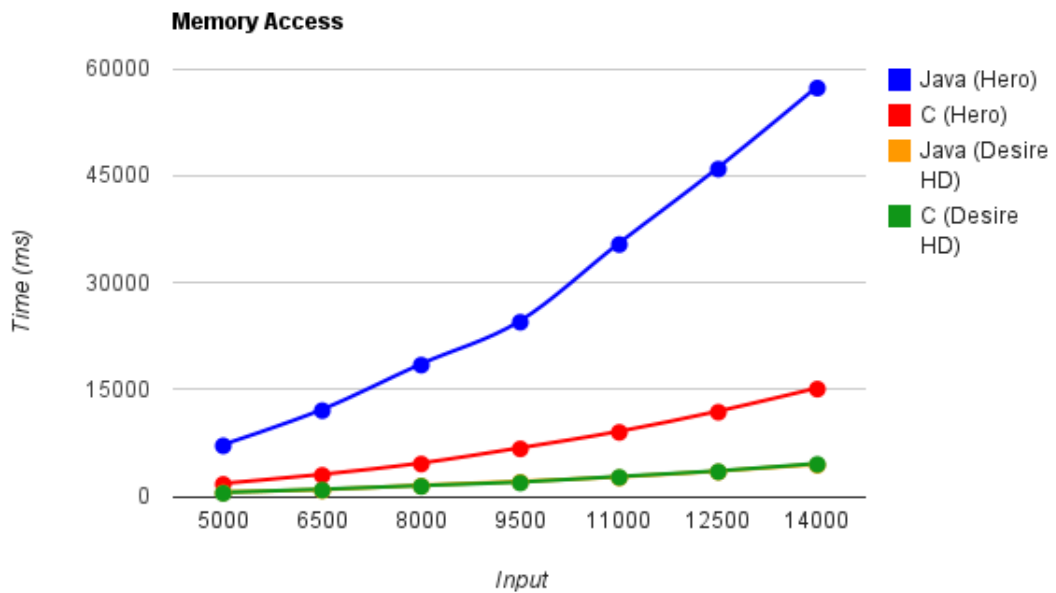The result is presented in the figure below.

Figure 4.5. A graph showing timed performance in milliseconds for each device and implementation of the Memory Access test. The plots are estimated from 7 measured data of different input parameters. The green curve coincides with the yellow one.

Chapter 5

Discussion

This section presents discussions and conclusions of the results presented above. The tests performed yielded interesting results, which in some cases were unexpected. Java's performance was most surprising, which was not expected to be comparable against the C implementations in any of the cases. Java performed well in many cases, and possible reasons for this will be debated below.

5.1   Java & Native C performance

The native C implementations were faster than the Java implementations on Hero. The ratio between native C and Java grows as input increases in every single one of the tests, leaving native C rather preeminent to Java. This result is probably due to the fact that the Dalvik JVM is not utilizing JIT compilation and not performing any advanced optimization and thus the native C code runs much faster since it is not bound by any JVM, as described earlier in the report. Java is also classically said to be slower because of its automatic garbage collection, however the tests are neither confirming nor contradicting to this statement.

However, on Desire HD the performance of the native C implementations were similar with the Java implementations in three out of four tests, even for greater inputs yielding longer executions (the tests were actually executed with much bigger input than the figures show in order to confirm the equal values). This shows that the improved Dalvik, with the new JIT compiler, running on Desire HD has been optimized better compared to the version utilized by Hero, as the difference between Java and native C were almost absent.

The Floating point calculation test was the only test yielding remarkable difference compared to the other tests. Hero, which does not have the VFP coprocessor, had as expected much longer execution time in Java. The native C performance on Hero was also according to the hypothesis. The C implementation on Desire HD was also as the statement of the hypothesis, with longer execution time than

17

the Java equivalent. Java performed approximately three times faster than the C
implementation for this device. This is most likely due to the fact that the Java
implementation uses the VFP but the C implementation does not.

5.2   Device performance

Common for all tests are the performance differences between the devices. Desire
HD had significantly better performance than Hero in each one of the tests. This
is mostly due to the fact that Desire HD has a faster processor than Hero. Desire
HD's Java implementation was approximately ten times faster than the same im-
plementation on Hero. This was all according to the hypothesis, however, the C
implementations were not which is described in section 5.1.

Another notice regarding the recursion test specifically is that Hero returned a
"StackOverflow Exception" for very high input, due to the fact that each method
call ends up on the stack area of the memory. Desire HD was able to cope with
this since it has more memory, and thereby larger stack area, and didn't yield any
exceptions.

5.3   Emulator performance

The standard SDK Emulator, used to simulate an Android device, showed similar
results as Hero, where the Java implementations was much slower. The emulator
showed the same pattern as Hero in all of the test cases. Results of the emulator
performance are however not presented in this report due to the lack of importance.
Presenting these results would not contribute to any relevant information and this
will be described below.

Using the SDK emulator to compare performance is not the correct way of evalu-
ation for many reasons. The emulator depends heavily on the computers perform-
ance and can easily be tricked. If a background process requires more attention
during the test, results will be affected. The emulator does not take advantage
of specific platform hardware such as the VFP. Any results using the emulator to
present conclusions concerning performance on Android is therefor not acceptable.

5.4   Sources of errors

When obtaining measurable output it is hard to eliminate all possible sources of
errors. Below are some of these possible sources listed, that could effect the results
and corresponding conclusions.

The execution time of each implementation is measured. Different timing functions

are used, one for the C implementations and another one for the Java implementations. This means that, if one or both of the functions return a big enough source of error for each timekeeping, this could spoil the results obtained. However the obtained difference in time between the C and the Java implementations differ much more for each test than the possible source of error for these timing functions are likely to produce, and so this should not be a problem for the results [20].

The vision of this report is to guide Android developers in their work of applications for Android smartphones. Results presented in this report are based upon the two devices Desire HD and Hero from the same manufacturer HTC. Due to the fact that both devices are developed by the same manufacturer, results could be misguiding. Hardware from other Android smartphones could be different and thereby present other conclusions. Most of the Android smartphones are using similar hardware such as the Snapdragon processor [25], which indicates that the conclusions in this report are likely general.

The tests implemented in this report are developed in the aim of creating an overall benchmark of common operations in Android applications. This goal is difficult to reach and there will always be exceptions which will not benefit from these conclusions, applications where similar operations won't be used. Creating an overall benchmark directed to all type of applications is an impossible task. Results presented in this report are however expected to give fair guidelines to most developers.

5.5   Miscellaneous

Java Native Interface (JNI) is used to communicate between Java and Native C/C++ code. This interface results in a slight communication delay [23] which was measured during the first test. Time was first measured within the native implementation, as appendix B.1 shows, and then compared with a measurement including the JNI calls. Difference was a few milliseconds and it was easy to confirm that this delay had no important effect on the result. Since all the tests presented in this essay depend on the same type of JNI calls, conclusion was that this did not affect any of the tests.

There are also other reasons of using native code such as C and C++ for Android. These languages provide low level features which Java lacks. Pointers and references which is a part of these low level languages in some cases boost overall performance, if implemented in a wise way. Using references avoid copies of the same object and pointers can be used for direct access to the memory which Java is missing [24]. These low level features give extra functionality and can in more complex programs gain performance. Writing code for specific hardware is only possible in low level languages, and direct access from Java to the native operating system or hardware needs to be accessed via JNI, or some other interface such as

JNA [26]. Java has however pre-written libraries to access most hardware components. Converting old C and C++ code to Java can also be demanding and requires huge amount of work. Using JNI to access the already existing code can then be a huge asset.

Chapter 6

Conclusions

The evaluations show that Java implementations on Android 2.2 perform better than Android 2.1, where the performance of Java is approaching the performance of native C. This implies that the need for native code on Android probably decreases as Dalvik improves and devices utilizing newer versions of Android are shipped, paradoxically since the NDK is still in its cradle. The main need for the NDK in the future thus appears to be the need of running already written C/C++ code on Android, instead of having to port the existing code into Java. The JNI interface is complex and requires time to take advantage of. Android programmers are not recommended to use the NDK while developing simpler applications for newer Android smartphones, since the increased complexity by implementing native code, likely will not pay off.

Future work includes topics as whether or not Java applications run on Dalvik can outnumber native C implementations on future Android versions, as Dalvik improves.

Bibliography

[1] ArsTechnica, *Android tops everyone in 2010 market share; 2011 may be different*, http://arstechnica.com/gadgets/news/2011/01/android-beats-nokia-apple-rim-in-2010-but-firm-warns-about-2011.ars , [2011, January 31], [2011, February 26].

[2] Gartner, *Gartner Says Android to Command Nearly Half of Worldwide Smartphone Operating System Market by Year-End 2012*, http://www.gartner.com/it/page.jsp?id=1622614 , [2011, April 7] [2011, April 8].

[3] Android Developers, *Application Fundamentals*, http://developer.android.com/guide/topics/fundamentals.html , [2011, March 3], [2011, March 8].

[4] Android Developers, *What is the NDK?*, http://developer.android.com/sdk/ndk/overview.html, [2011, February 26].

[5] Wikipedia, *Java (programming language): Performance*, http://en.wikipedia.org/wiki/Java_%28programming_language%29#Performance, [2011 February 20], [2011, February 27].

[6] Android Developers, *Android NDK, Revision 5 (December 2010)*, http://developer.android.com/sdk/ndk/index.html , [2011, February 27].

[7] ICCAS 2010 - International Conference on Control, Automation and Systems; Lee S., Jeon J.W.; 2010; *Evaluating performance of android platform using native C for embedded systems*; page 1160-1163; 978-1-4244-7453-0.

[8] Mobile wireless middleware, operating systems, and applications; Schmidt A.D., Schmidt H.G., Camtepe A, Albayrak S; 2009; *Developing and Benchmarking Native Linux Applications on Android*; volume 7; page 381-392.

[9] Wikipedia, *Java Native Interface*, http://en.wikipedia.org/wiki/Java_Native_Interface, [2011, March 22].

[10] HTC, *HTC Desire HD Specification*,
     http://www.htc.com/europe/product/desirehd/specification.html   ,   [2011,
     March 8].

[11] HTC, *HTC Hero Specification*,
     http://www.htc.com/europe/product/hero/specification.html , [2011, March
     8].

[12] Google I/O; Ben Cheng, Bill Buzbe, 2010; *A JIT Compiler for An-
     droid's Dalvik VM*; http://dl.google.com/googleio/2010/android-jit-compiler-
     androids-dalvik-vm.pdf , [2011, March 22].

[13] Android   Developers,   *Android   2.2   Highlights*,
     http://developer.android.com/sdk/android-2.2-highlights.html , [2011, March
     22].

[14] PDAdb.net, *Qualcomm MSM7200A RISC Microprocessor with embedded DSP*,
     2008, http://pdadb.net/index.php?m=cpu&id=a7200a&c=qualcomm_msm7200a
     , [2011, March 31].

[15] PDAdb.net,   *Qualcomm   Snapdragon   MSM8255
     RISC   Microprocessor   with   embedded   DSP*,   2010,
     http://pdadb.net/index.php?m=cpu&id=a8255&c=qualcomm_snapdragon_msm8255
     , [2011, March 31].

[16] ARM, *ARM1136JF-S and ARM1136J-S Technical Reference Manual*,
     http://infocenter.arm.com/help/topic/com.arm.doc.ddi0211k/DDI0211K_arm1136_
     r1p5_trm.pdf , [2011, March 30].

[17] ARM, *Cortex A8 Revision: r3p2 Technical Reference Manual*,
     http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_
     a8_r3p2_trm.pdf , [2011, March 31].

[18] Android NDK Dev Guide, 2011, *Android Native CPU ABI Management*,
     http://developer.android.com/sdk/ndk/overview.html#docs , [2011, April 7].

[19] GNU   GCC,   *Software   floating   point   in   GCC*,
     http://gcc.gnu.org/wiki/Software_floating_point , [2008, January 8] [2011,
     April 12].

[20] IBM,   *Robust   Java   benchmarking,   Part   1:   Issues*,
     http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html
     , [2008, June 24] [2011, April 8].

[21] Cherrystone   Software   Labs,   *Algorithmic   Performance   Compar-
     ison   Between   C,   C++,Java   and   C#   Programming   Languages*,
     http://www.cherrystonesoftware.com/doc/AlgorithmicPerformance.pdf   ,
     [2010, August 28], [2011, April 6].

[22] Sun, Steven Meloan, *The Java HotSpot Performance Engine: An In-Depth Look*,
http://java.sun.com/developer/technicalArticles/Networking/HotSpot/index.html
, [1999, June 1] [2011, April 13].

[23] Addison-Wesley, Bloch J., 2008, *Effective Java*, Second Edition.

[24] Wikipedia, *Comparison of Java and C++*,
http://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B, [2011 April 9], [2011, April 12].

[25] Wikipedia, *Snapdragon_(system_on_chip)*,
http://en.wikipedia.org/wiki/Snapdragon_(system_on_chip), [2011 April 13], [2011, April 13].

[26] Java.net, *Java Native Access (JNA)*, http://java.net/projects/jna , [2011, April 13].

Appendix A

Glossary

Name	Description
Android	An operating system primarily used on handheld devices.
C	A programming language developed between 1969 and 1973 by Dennis Ritchie.
C++	A programming language developed by Bjarne Stroustrup 1979 as an enhancement to the C language.
Dalvik	Android's custom JVM.
FPU	Floating Point Unit, a coprocessor for calculating floating points.
Java	A programming language developed and released 1995 by Sun Microsystems.
JIT	Just In Time compilation, a method to improve runtime performance.
JNI	Java Native Interface, an interface for communication between Java and native C/C++ code.
JVM	Java Virtual Machine, a virtual machine where Java code is executed.
Native code	Machine code written in C/C++ for a certain CPU.
NDK	Native Development Kit, a bundle of software for native Android development.
SDK	Software Development Kit, a bundle of software for Android development.
UI	User Interface.
VFP	Vector Floating Point coprocessor, a special FPU.

Appendix B

Program Code

B.1 Prime Number calculation

Java implementation:

```
public long primeNumber(int max) {
    int i, j;
    int nr = 0;
    long starts = System.currentTimeMillis();
    for ( i = 2; nr<max; i++ ) {
        for ( j = 2; j <= i/2; j++ ) {
            if ( ( i % j ) == 0 ) break;
        }
        if ( j > i / 2 ) nr++;
    }
    long ends = System.currentTimeMillis();
    return (ends-starts);
}
```

Native C implementation:

```
jint Kandidatexamen_kex_PrimeTest_primeNumberJNI( JNIEnv* env,
                                        jobject thiz, jint max ) {
    jint i, j;
    jint nr = 0;
    clock_t uptime_before = clock() / (CLOCKS_PER_SEC / 1000);
    for ( i = 2; nr<max; i++ ) {
        for ( j = 2; j <= i/2; j++ ) {
            if ( ! ( i % j ) ) break;
        }
        if ( j > i / 2 ) nr++;
    }
    clock_t uptime_after = clock() / (CLOCKS_PER_SEC / 1000);
```

```
    return uptime_after-uptime_before;
}
```

B.2   Floating Point calculation

Java implementation:

```
private void floatCalculation(int max) {
    double temp = 1.5;
    double fl = 6.691842;
    for(int i = 0; i < max; fl++, i++) {
        temp = temp * fl;
    }
    for(int i = 0; i < max; fl++, i++) {
        temp = temp / fl;
    }
    for(int i = 0; i < max; fl++, i++) {
        temp = temp + fl;
    }
    for(int i = 0; i < max; fl++, i++) {
        temp = temp - fl;
    }
    return;
}
```

Native C implementation:

```
void Kandidatexamen_kex_FloatTest_floatCalcJNI(JNIEnv* env, jobject thiz,
                                               jint max) {
    jdouble temp = 1.5;
    jdouble fl = 6.691842;
    int i;
    for(i = 0; i < max; fl++, i++) {
        temp = temp * fl;
    }
    for(i = 0; i < max; fl++, i++) {
        temp = temp / fl;
    }
    for(i = 0; i < max; fl++, i++) {
        temp = temp + fl;
    }
    for(i = 0; i < max; fl++, i++) {
        temp = temp - fl;
    }
    return;
```

```
}
```

B.3 Recursion

Java implementation:

```
public class Quicksort  {
        private int[] numbers;

        public void sort(int[] values) {
            // Check for empty or null array
            if (values ==null || values.length==0) return;
            this.numbers = values;
            quicksort(0, values.length - 1);
        }

        private void quicksort(int low, int high) {
            int i = low, j = high;
            // Get the pivot element from the middle of the list
            int pivot = numbers[low + (high-low)/2];

            // Divide into two lists
            while (i <= j) {
             // If the current value from the left list is smaller then the pivot
               // element then get the next element from the left list
                while (numbers[i] < pivot) {
                    i++;
                }
             // If the current value from the right list is larger then the pivot
              // element then get the next element from the right list
                while (numbers[j] > pivot) {
                    j--;
                }
             // If left list is larger then the pivot element and if we have found
             // a value in the right list which is smaller then the pivot element
                // then we exchange the values.
                if (i <= j) {
                    exchange(i, j);
                    i++;
                    j--;
                }
            }
            // Recursion
            if (low < j) quicksort(low, j);
```

```
        if (i < high) quicksort(i, high);
    }

    private void exchange(int i, int j) {
        int temp = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = temp;
    }
}
```

Native C implementation:

```
void Kandidatexamen_kex_QSortTest_qsort(JNIEnv* env, jobject thiz,
                                        jintArray arr, jint arrayLength) {

    jint *carr;
    carr = env->GetIntArrayElements(arr,NULL);
    // Check for empty or null array
    if (carr == NULL){ //funkar inte|| carr->length==0) return;
    numbers = carr;
    quicksort(0, arrayLength -1);
}


void quicksort(jint low, jint high) {
    jint i = low, j = high;
    // Get the pivot element from the middle of the list
    jint pivot = numbers[low + (high-low)/2];

    // Divide into two lists
    while (i <= j) {
      // If the current value from the left list is smaller then the pivot
        // element then get the next element from the left list
        while (numbers[i] < pivot) {
            i++;
        }
      // If the current value from the right list is larger then the pivot
        // element then get the next element from the right list
        while (numbers[j] > pivot) {
            j--;
        }
      // If left list is larger then the pivot element and if we have found
      // a value in the right list which is smaller then the pivot element
        // then we exchange the values.
        if (i <= j) {
            exchange(i, j);
```

```
            i++;
            j--;
        }
    }
    // Recursion
    if (low < j) quicksort(low, j);
    if (i < high) quicksort(i, high);
}

void exchange(jint i, jint j) {
    jint temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}
```

B.4   Memory access

Java implementation:

```
private static void bubbleSort(int[] listOfElements) {
    int arrlength = listOfElements.length;

    //Loop once for each element in the array.
    for(int counter=0; counter<arrlength-1; counter++) {
        //Once for each element, minus i.
        for(int i=0; i<arrlength-1-counter; i++) {
          //Compare pairs of numbers to indicate which pairs to switch.
            if(listOfElements[i] > listOfElements[i+1]) {
                int temp = listOfElements[i];
                listOfElements[i] = listOfElements[i+1];//Swap first
                listOfElements[i+1] = temp; //Swap second
            }
        }
    }
}
```

Native C implementation:

```
void Kandidatexamen_kex_BubbTest_bubbSortJNI(JNIEnv* env, jobject thiz,
                                  jintArray arr, jint arrayLength) {
    int tmp, i, index;
    jint *carr;
    carr = (*env)->GetIntArrayElements(env,arr,NULL);

    //Loop once for each element in the java array.
```

```
for(i=0; i<arrayLength-1; i++) {
    //Once for each element, minus i.
    for(index=0; index<arrayLength-1-i; index++) {
        //Test if the pair needs a swap or not.
        if(carr[index] > carr[index+1]) {
            tmp = carr[index];
            carr[index] = carr[index+1]; //Swap first
            carr[index+1] = tmp; //Swap second
        }
    }
}
}
```