Bachelor Thesis in Software Engineering
June 2012

Performance and Energy Optimization for the Android Platform

Andreaz Lewerentz and Jonathan Lindvall

Contact Information:
Authors:
Andreaz Lewerentz andreaz.lewerentz@gmail.com
Jonathan Lindvall jonathan.lindwall@gmail.com

University advisor:
Nina Dzamashvili Fogelström

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Software developers are faced with several challenges when creating applications for the new generation of mobile devices. Smartphones and tablets have limited processing power and memory resources, and a small battery is the only thing that keeps the hardware components running. End users have little patience for slow applications that drain the batteries of their devices. To satisfy the needs of their customers, developers must take these hardware limitations into account; they must make an effort to optimize the performance and energy efficiency of their applications.

This thesis provides a general overview of performance and energy optimization in the mobile domain. A specific sub-area is explored in great detail: the use of native C code for performance and energy optimization of Android applications. An experiment was conducted to see how the performance of native code compares to that of Java. This is the first time that such measurements have been made on both emulators and physical devices. The devices were running recent versions of Android that have not been used for similar experiments before: 2.3.3, 3.2 and 4.0.3. It is also the first time that native code has been compared to Java in terms of energy consumption.

The results show that the latest updates to the Android platform have brought Java closer to native code in terms of performance, but native code is still the best choice for certain types of operations. It is also evident that there is a close correlation between performance and energy efficiency. Finally, the results show that Android emulators are unreliable for performance measurements. This could be a reason to question the validity of previous research.

Contents

1 Introduction

The use of sophisticated mobile devices such as smartphones and tablets has increased rapidly over the last few years. Analysts believe that this increase will continue [1] and that mobile devices will overtake the PC as the most common web access device by 2013 [2]. Software developers that are targeting mobile platforms are faced with several new challenges. Aspects such as performance and battery life must be taken into account for an application to be commercially successful. So, how do you develop high performance applications that are energy efficient?

The goal of this thesis is to provide valuable insight into the Android platform and to allow application developers to make qualified decisions for maximizing the performance and energy efficiency of their applications. The Android platform was chosen because it currently has the largest market share and is expected to continue to grow over the coming years. [1]

This thesis summarizes work that has been done in the area of performance and energy efficiency and examines a specific sub-area in detail, namely the use of native C code for performance and energy optimization of Android applications. The aim is to provide guidelines for when native code should and should not be used. The statements in the report are backed up by experiment results that allow for comparisons between different development methods. Important differences between the recent versions of the Android platform are also discussed.

In section 2, *Background*, the reader is introduced to one problem area within mobile application development, namely energy efficiency and performance. This section also provides some general information about the Android platform.

The research questions are stated in section 3, *Research questions and research design*, which explains the design of the literature survey and provides an overview of problems related to energy consumption and performance. In section 4, *Literature survey results*, previous research in the area is summarized and the main focus of this thesis is presented.

Section 5, *Experiment design*, explains how the measurements were prepared and executed. The results are presented and analyzed in section 6, *Experiment results and analysis*. Section 7, *Data synthesis*, compares the results to previous research and presents the main findings.

In the final section, *Conclusion*, the results are summarized and some topics that could be covered in future work are suggested.

2 Background

2.1 Problem area

A common problem among portable devices such as smartphones and tablets is their high energy consumption and limited battery capacity. Central Processing Units (CPUs), network adapters and high resolution displays are some of the most energy-consuming pieces of hardware found in mobile devices [3]. The software can have a major impact on the energy consumption as well. If an application processes large amounts of data, the CPU will be active for a long duration of time with high energy consumption as a consequence. If a program displays bright pictures and requests user input frequently, the screen will remain activated, and this will also have a negative impact on battery life.

Several platforms have been developed for mobile devices. The most popular ones are Apple iOS, Android, Blackberry and Windows Phone. Although released by different companies under different conditions, they all share many of the same problems and limitations. The work presented in this thesis focuses on the Android platform since it is one of the most widespread platforms and is expected to stay in the lead for the coming years [1].

2.2 The Android platform

Android is an operating system mainly used for mobile devices such as smartphones and tablets. It was commercially released in September 2008 by the Open Handset Alliance (OHA), a collection of companies with the common goal of establishing an open and standardized platform for mobile development. The Android Open Source Project (AOSP) is a group led by Google Inc. that handles the development and maintenance of the Android platform. The Android source code is licensed under Apache License version 2.0 [4] and the Linux kernel patches under GNU GPLv2 [5]. This makes Android an open source platform that is free to use, modify and distribute.

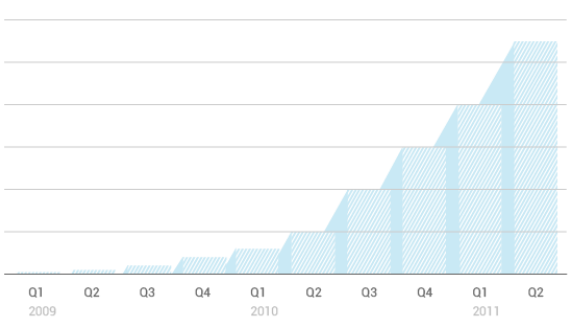
Since the premiere in September 2008, several new versions of Android have been released. Each new version has brought improvements to the code; software defects have been removed and new functionality has been added to the platform. The 3.0 Honeycomb release added special support for tablets and other large screen devices such as the next generation of Internet-connected televisions. Table 1 marks the most important updates of the Android platform.

Version	**Codename**	**Release date**
1.0	-	September 2008
1.5	Cupcake	April 2009
1.6	Donut	September 2009
2.0	Eclair	October 2009
2.2	Froyo	May 2010
3.0	Honeycomb	February 2011
4.0	Ice Cream Sandwich	May 2011

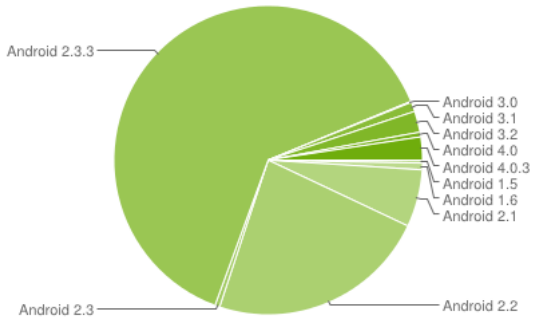
Table 1: Android releases schedule

Approximately 300 million Android devices have been sold since the platform's launch in 2008 [6]. In 2011, 350 000 new devices were activated every day [7] and this trend is expected to continue

over the coming years. Figure 1 (a) demonstrates the number of newly activated devices between 2009 and 2012. Figure 1 (b) displays the current distribution of Android versions among active devices. This distribution is of importance when developing applications because an app targeting a recent version is not compatible with a device running an older version of the operating system.



(a) Activated devices 2009 - 2012



(b) Platform version distribution among all devices

Figure 1: Android statistics, *Google Inc. May 2012* <*http://www.android.com/developers*>

The Android platform is based on a Linux kernel and a collection of C/C++ libraries that handle tasks such as graphics rendering, media playback and database connection management. The actual applications are executed in the Dalvik Virtual Machine (DVM), a custom Java Virtual Machine (JVM) tailored for mobile devices. Figure 2 demonstrates the different layers and components of the Android platform.

Figure 2: Platform overview, *Google Inc, May 2012* <*http://www.android.com/developers*>

Android applications are traditionally written in the Java programming language. However, there is also the option of executing native code written in C or C++. The main reason for using native code instead of Java code is, according to Google, to be able to reuse existing code and libraries [8].

However, under some circumstances this native code can be used for performance optimization as well. The Android platform supports the use of native code through the Java Native Interface (JNI) and a development framework known as the Native Development Kit (NDK).

3 Research questions and research design

3.1 Research questions

The research questions that are answered in this thesis are all concerned with the performance and energy consumption of mobile applications. Whereas the first question is a general question on this topic, the subsequent questions are increasingly specific. The second question is only concerned with performance and battery life optimization for a certain mobile platform: Android. Finally, the third question is related to the use of native code for performance and energy optimization of Android applications.

1. What aspects impact the performance and energy consumption of mobile applications?
2. Which approaches have been proposed to improve the performance and energy efficiency of Android applications?
3. How does the use of native code affect the performance and energy consumption of applications running on recent versions of Android?

Figure 3 illustrates the main topics of this thesis and how they are related to each other. The research questions are labelled *Q1*, *Q2* and *Q3* and they each correspond to a single topic. Note that the third topic is a subtopic of the second topic, and the second topic is a subtopic of the first topic. The shade of each ellipsis indicates the extent to which the topic will be covered - a light shade of gray indicates that the thesis will provide only a general overview of the topic (*Q1*) whereas a darker shade of gray indicates that the topic will be covered in greater detail (*Q3*).

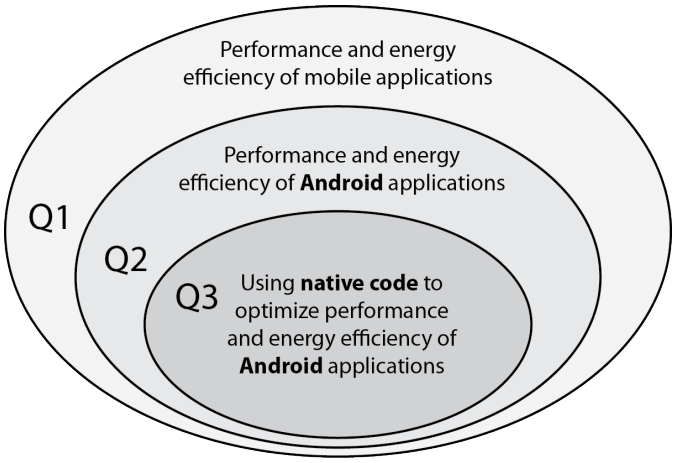


Figure 3: Thesis topics and research questions

3.2 Research design

To answer the first two questions listed above, a literature review was conducted. The answers to these questions are mainly based on the results of previous research within the field.

The literature review also provided some valuable information that relates to the third research question. However, the findings from the literature review are not sufficient to answer the question

in detail. An experiment will be conducted in order to give a detailed and current answer to this final question.

Most of the research papers were retrieved from two of the major research databases in the field of computer science: the ACM Digital Library and IEEE Xplore.

3.2.1 Literature survey design

Question 1: What aspects impact the performance and energy consumption of mobile applications? The following search string was used to retrieve literature related to this area:

((smartphone* OR android OR ios OR iphone OR (windows AND phone)) AND
(performance OR efficien* OR effective* OR battery* OR power OR energy))

This first question is quite general in nature, and the search specified above returned hundreds of results in the IEEE Xplore database. There appears to be a substantial amount of research in this area, and providing a complete coverage of that research is outside the scope of this thesis. Instead, this thesis will provide a general overview of the area and shed light on certain aspects that affect the performance and battery consumption of mobile applications.

Because of the sheer number of research papers that were retrieved, it was deemed impossible to review them all. Instead, the papers were sorted by relevance, and only the most relevant papers were reviewed. Since the intention was to provide a general overview of the area rather than to cover it completely, the strategy was to select at least one paper that related to each of the interesting sub-areas within the domain.

Question 2: Which approaches have been proposed to improve the performance and energy efficiency of Android applications?
A number of different search strings were used to find literature relating to this question. The following search string was used to retrieve papers related to the battery consumption of Android applications:

(android) AND
(power OR energy OR battery) AND
(efficien* OR perform* OR optimiz* OR consum* OR life OR sav*)

Wildcards were used in order to catch terms with similar meanings. A search string such as "optimiz*" can be used to catch words like optimiz*e*, optimiz*ed*, optimiz*ing*, optimiz*ation*, etc. Because of the wildcards and the boolean operators, the search string above will capture a variety of expressions, e.g., "power optimization", "energy efficiency" and "battery efficient".

When the search string above was used in a *metadata only* search in the IEEE Xplore database, 114 results were returned. Some of the research papers retrieved were unrelated to the topic of this thesis; this could be an indication that the search string needs to be refined. However, a decision was made not to make the search any more narrow; a relatively wide search was necessary to guarantee that relevant papers weren't overlooked.

Question 3: How does the use of native code affect the performance and energy consumption of applications running on recent versions of Android?
The following search string was used in order to retrieve papers on this subject:

(android) AND
(native* OR ndk OR jni)

This search string captures all papers that contain the word *Android* as well as one or more of the words *native*, *NDK* and *JNI*. Because both *Native Development Kit* and *Java Native Interface* contain the word *native*, there is no reason to explicitly include the full names in the search string.

Only research papers that were closely related to the use of native code on Android were considered interesting. Because of this, a *metadata only* search was performed; a vast majority of the relevant papers were expected to have the word *Android* as well as *native*, *NDK* or *JNI* in their metadata.

The search produced 22 results in the IEEE Xplore database. From these 22 results, 7 research papers were selected. Papers were chosen only if they looked at native code from a performance or energy perspective.

4 Literature survey results

4.1 Research question 1: What aspects impact the performance and energy consumption of mobile applications?

A survey exploring end users' smartphone usage [3] showed that there is no typical usage pattern common to all people. However, the results showed that an inactive phone consumes significantly less energy (approximately 70 mW) than a device that is activated by a phone call or by other types of user interaction (300 mW to 2000 mW). The survey also concluded that the workload and energy consumption varies greatly among users. Some users make a lot of phone calls, others browse the Internet and some use their smartphone as a portable music player. Despite this variation in usage there are some pieces of hardware that dominate the power consumption in a smartphone.

The hardware components that typically consume the most energy are:

- Screen: 35%
- CPU: 13%
- Networking: 11%

An LCD display creates images on a surface by emitting light through a combination of polarized glass and liquid crystals. The energy consumption is in direct correlation with the amount of light emitted. Most smartphones allow the user to choose the brightness level of the screen; a high brightness level makes the display more clear and improves readability, but the energy consumption is increased. Although screen brightness levels can be changed on most devices, the typical user does not actively do so [9]. According to Shye, Scholbrock and Gokhan [3], 70% of the average user's *screen intervals*, i.e., when the screen is activated and receives user input, are around 100 seconds. The remaining 30% of the screen intervals are less than 100 seconds.

A typical CPU has a maximum clock rate measured in Hz (typically MHz or GHz). A higher clock rate indicates that the CPU can perform calculations faster, but it also means that more energy is consumed. Most smartphones use DFS (Dynamic Frequency Scaling) to dynamically lower the CPU frequency when the application is not in need of full processing power. As an example, the HTC G1 smartphone uses a 528MHz CPU but scales down its frequencies to 384MHz and 246MHz [10]. The DFS algorithm typically used in devices is based on screen activity. If a process is running and the screen is on, the full capacity of the CPU is utilized, but if a process is running when the screen is turned off, the DFS sets the CPU to its lowest setting.

Network traffic, whether using 3G, GSM or WiFi networks, always induces a significant energy waste [11]. When a network transaction is finished the network activity does not stop instantly; instead, the device remains in an active, high-power state for a brief amount of time. If no further transactions occur the device then returns to an idle state. The purpose of this *tail-time* is to avoid the delay and resource overhead involved in going from an idle state to an active state (in case of further network usage). A typical tail-time is a couple of seconds during which the device consumes a substantial amount of energy.

Many of the most popular applications for smartphones use a considerable amount of networking. Services that interact with social media sites, email servers, news feeds, etc. continuously download data to keep the end user up to date with recent events. This is most commonly achieved

by techniques such as *polling* and *push notifications*. Polling is when the smartphone application actively sends a request to a server to fetch new data (if any). This is done at a preset interval such as once every ten minutes. A push notification is when the server holding the data sends a message to the client, telling it to perform a poll to retrieve the new piece of information. From an energy efficiency standpoint the push technique is preferred.

Since activation and usage of the network link is expensive from an energy consumption viewpoint, it is also important to keep data transaction time at a minimum. Different formats are used when data is being sent from one host to another over the Internet. Typical formats include XML, JSON and binary. The data can also be compressed so that the download time can be reduced.

4.2 Research question 2: Which approaches have been proposed to improve the performance and energy efficiency of Android applications?

Since the screen and the CPU are the most energy consuming pieces of hardware in smartphones, one way to minimize battery drain is to avoid unnecessary screen and CPU usage. Different schemes to incrementally reduce the brightness of the screen during user interaction have proven successful [3]. Combining such methods with an improved scheme for DFS that also incrementally lowers the CPU frequency has been shown to lower the total battery consumption of a typical Android application by around 10%. This scheme was applied to the devices of 20 different users and tested for a week. The majority of the end users, 75%, did not notice the drop in screen brightness and CPU performance during normal phone activities such as web browsing, writing emails, calling and listening to music. However, the scheme proved inefficient for games because they are often very demanding of the phone's resources.

Balasubramanian and Venkataramani [11] propose two schemes for common applications such as email and news feeds, that effectively reduce energy consumption. Their idea is to implement a scheduler, named the *Tail Ender*, that has the user specify an acceptable delay before having the phone request new emails and news feeds. To illustrate how the scheme works, assume that a user writes two emails and subscribes to one news feed. The *Tail Ender* then synchronizes the sending and retrieving actions for these different activities. This means that the network is activated once instead of three times. The *Tail Ender* scheme reduces energy consumption for email by 35% and RSS feeds by 52%. A similar batch-scheduling mechanism is introduced in [12] and, since Android API level 3, there is a native way to give scheduled tasks an *inexact repeating* which leads to the same positive batch effects.

The importance of applying compression to data sent to mobile devices is stressed in [13]. When an application retrieves large amounts of data from a server, a compression request should always be set in the HTTP request head to allow for compression. This can reduce network uptime and energy consumption while at the same time improving the performance of the download. [14] argues that between XML, JSON and the binary format protocol buffers, JSON is the preferred format for textual data interchange as long as compression is applied. The performance gain is especially noticeable when using slower networks such as 3G or GSM. For a large transaction in JSON format, the difference in time between sending compressed and uncompressed data can be more than two seconds on a 3G network. The difference would be much smaller on a fast WiFi network, approximately a hundred milliseconds. WiFi is the preferred way of accessing the network for large data transactions.

Several different resource handling schemes have been proposed to save energy on a mobile device. [15] proposes a context-aware control system that, with the help of sensors, finds out if the end user is indoors or outdoors and whether he is moving or not. It then uses a matrix with predefined values for different resources and their expected behavior in various contexts. While an end user is taking a walk to work, the GPS would, according to the matrix, be activated and receive the necessary amount of CPU power. When the user is indoors, the GPS would be turned off and the CPU would be set to a lower clock rate.

Limiting the use of hardware components such as network interfaces, GPS, Bluetooth etc. is a good start to writing energy efficient code. Google has provided more general guidelines on how to produce more efficient Java code for Android applications [16]. Examples of suggestions are to avoid internal getters and setters, to use the enhanced for-loop syntax and to avoid floating-point values. Since Android 2.3 there is also the possibility to write programs fully or partially in native C/C++ code. This native code is not executed in the DVM but in the underlying Linux system. This may result in a performance boost for certain types of code.

4.3 Research question 3: How does the use of native code affect the performance and energy consumption of applications running on recent versions of Android?

The official Android NDK documentation states that whereas the use of native code always increases the complexity of an application, it does not automatically result in a performance increase [17]. However, previous research shows that native code, if used correctly, *can* have a major impact on the performance of Android applications [18] [19] [20] [21] [22] [23].

Lee and Lee [18], Lee and Jeon [21] and Ulvesand and Eriksson [23] have compared the performance of Java and native code implementations of various algorithms. There are slight variations in the actual algorithms used, but all of the experiments include basic operations such as integer calculations, floating-point calculations and memory access.

4.3.1 Integer calculations

For integer calculations, the use of native code often results in a significant performance increase [18] [21]. Lee and Jeon [21] looked at the performance of an algorithm for prime number calculation, and the native C implementation of that algorithm was approximately three times faster than its Java counterpart. The experiments in [18] and [21] were, however, not performed on a physical device but using an Android emulator. The target versions of Android in [18] and [21] were 2.3 and 2.1, respectively.

Ulvesand and Eriksson [23] performed their tests on actual Android devices: an HTC Hero running Android 2.1 and an HTC Desire HD running Android 2.2. They criticize the use of emulators for performance measurements and point out that an emulator does not take advantage of specific platform hardware such as a Vector Floating-Point coprocessor (VFP).

When using the Android 2.1 device, Ulvesand and Eriksson [23] saw results that were very similar to those of [18] and [21]. Their integer calculation experiment showed that the native code implementation was 2-3 times faster than the Java implementation. Somewhat surprisingly, when the same tests were performed on the Android 2.2 device, the results were very different. The difference in performance between the Java and native C implementation was much less significant; in fact, the difference was barely noticeable. This may be due to the Just In Time (JIT) compilation

that was introduced to Android in version 2.2. The addition of the JIT compiler allowed for significant performance improvements, and it may have brought the performance of the Java code executing in the DVM environment closer to that of the native C code [23].

4.3.2 Memory access operations

The experiments that served to test the performance of memory access operations produced similar results. Lee and Lee [18], Lee and Jeon [21] and Ulvesand and Eriksson [23] all chose to test memory access operations by running a worst-case scenario of the bubblesort algorithm. The experiments conducted in [18] and [21] showed that the performance of the native C code was vastly superior to the performance of the Java implementation. Lee and Jeon [21] found that the bubblesort algorithm ran up to 30 times faster when using native code.

The authors of [23] also found that native C outperformed Java in the memory access department, but only on the Android 2.1 device. On the HTC Desire HD (Android 2.2), the results were identical whether native code was used or not.

4.3.3 Floating-point calculations

When it comes to floating-point calculations, previous research indicates that there is only a minor performance difference between native code and Java code. The results from [18] and [21] suggest that the Java code takes about 30 % more time to execute. Ulvesand and Eriksson [23] found that the performance of floating-point operations could be somewhat increased by using native code on Android 2.1. On Android 2.2, on the other hand, the native C implementation was noticeably slower than the Java implementation. This may be due to the fact that the Java implementation takes advantage of the dedicated floating-point coprocessor, whereas the C implementation does not [23].

4.3.4 String processing

In addition to memory access, integer calculations and floating-point operations, Lee and Lee [18] also looked at the performance of string processing in native C. Interestingly, they found that the performance dropped significantly when using native code. The reason for this is simple: Java, JNI and C all use different character encoding systems (Unicode, UTF-8 and KSC 5601, respectively). Processing strings in native code thus requires type conversion, and according to [18], this can have a very negative impact on the overall performance. These statements are backed up by Son and Lee [24].

Performance measurements of string processing were also performed by Lin et al. [19]. Their results, however, do not match those from [18]. In their string processing experiment, the native code implementation actually beat the Java implementation in terms of performance. Unfortunately, these discrepancies cannot easily be explained because the article by Lin et al. does not provide much information on how their experiment was conducted.

4.3.5 Complete applications

Most of the research mentioned above focuses on the performance of simple, isolated algorithms, but some researchers have studied how the use of native code can affect the performance of an actual Android application. Son and Lee [24] used native C code to improve the performance of an augmented reality engine, *NyARToolKit*. Method profiling was used to determine which Java classes were used the most frequently. The first step to improving the performance of an existing application, according to Son and Lee, is to find the parts of the code that are executed frequently.

Native code can then be used to optimize those specific parts of the software. In the case presented in [24], much of the code execution took place inside a single Java method. By optimizing this one method using the NDK, Son and Lee managed to speed up their application by 86.9 %.

Paik et al. [22] used native code to improve the performance of a commonly used block-cipher algorithm. They also found that their NDK implementation was remarkably faster than the traditional Java implementation.

4.3.6 Summary

Table 2 provides an overview of the previous research focusing on the performance of isolated algorithms:

Article	Environment	Android version	NDK version
Lee and Jeon [21]	Emulator	2.1	R3
Lee and Lee [18]	Emulator	2.3	R5b
Ulvesand and Eriksson [23]	Physical devices	2.1, 2.2	R5b
Lin et al. [19]	Physical device	2.2	R6

Table 2: Overview of previous research

The table shows that most of the previous experiments have targeted Android versions 2.1 and 2.2. Only Lee and Lee [18] have targeted version 2.3, and they executed their tests on an Android emulator.

In their *Conclusions* chapter, Ulvesand and Eriksson talk about the work that remains to be done in the area. They speculate that the improved DVM of future versions of Android may bring the performance of Java very close to the performance of native C, or even surpass it [23]. They also criticize the use of Android emulators for performance tests but they do not provide evidence to back up their statements. One of the goals of this thesis is to determine whether or not these theories and statements are correct. With this goal in mind, experiments will be performed on both physical devices and Android emulators. The tests will be executed on three of the most recent Android versions, namely 2.3.3, 3.2 and 4.0.3.

No previous research regarding how native code affects the energy consumption of an Android application was found during the literature survey. Thus, another goal of the experiment is to gather enough data to learn more about this relationship.

5 Experiment design

5.1 Overview

The main purpose of the experiment is to determine how the performance is affected when parts of an application are written in native C instead of Java. All tests will be executed on three different physical devices, each running its own version of Android. If there is a performance difference between native C and Java, the experiment will also show how this difference changes between older and more recent versions of Android. In addition to the physical devices, two Android emulators will be used. This is to determine whether or not an emulator is a worthy replacement of a physical device when it comes to performance tests. Ulvesand and Eriksson [23] state that emulators should never be used for performance measurements, but they do not provide enough evidence to support this claim.

To evaluate how the choice of programming language, Java or native C, impacts the energy consumption of an application, energy measurements will be made alongside the performance measurements.

5.2 Choice of algorithms

In order to make the comparison between Java and native C as fair as possible, a variety of different algorithms will be tested. Four different fundamental operation types are to be evaluated: integer calculations, floating-point calculations, memory access operations and recursion. For each of these operation types, two algorithms have been implemented in both Java and native C.

Though it may prove impossible to use the exact same code for the Java and C implementations, the goal will be to reach the highest possible level of similarity. This is to make sure that any performance differences are not the results of differences between the two implementations.

The algorithms that will be implemented for each of the four operations types are mentioned below:

Integer calculations

In previous research, prime number algorithms have been used to test the performance of integer calculations [18] [19] [21] [23]. Such an algorithm, the *sieve of Eratosthenes*, will be used in this experiment as well. In addition to this, an algorithm that performs basic arithmetic operations will be tested. This algorithm simply calculates the area of a large number of squares.

Floating-point calculations

Lee and Jeon [21] tested the performance of floating-point calculations by looking at trigonometric functions such as sine and cosine. In this experiment, one algorithm will use language-specific library routines to calculate the sine and the cosine of a large number of angles. The second floating-point algorithm simply uses floating-point arithmetic to calculate the volume of a large number of spheres.

Memory access operations

A worst-case scenario of the bubblesort algorithm has commonly been used to evaluate the performance of memory access operations [18] [21] [23]. Such an algorithm will be implemented in this experiment as well. In addition to this, an algorithm that copies a large number of primitive values from one array to another will be evaluated.

Recursive operations

To examine the performance of recursive operations, a previous experiment involving the

quicksort algorithm will be repeated [23]. A recursive implementation of the Fibonacci algorithm will also be tested.

These measurements will provide a good foundation for evaluations and comparisons. The results will also be compared to those from previous research.

5.3 Energy consumption

The experiment will also determine how the use of native code affects the energy consumption of an application. For each algorithm, the energy consumption of the Java and native C implementations will be measured. The measurements will be made using an application called PowerTutor [25]. PowerTutor is a piece of software that can measure the energy consumption of the hardware components of an Android device. It calculates the amount of energy (in joules) that has been consumed by each application. It also displays how much energy the CPU has consumed while executing code for a specific application. According to the PowerTutor developers, the energy consumption measurements are fairly accurate. Typically, they should not deviate by more than 5 % from the actual values. However, this accuracy is only guaranteed for certain phone models: HTC G1, HTC G2 and Nexus One. For other devices the measurements may not be as precise. Because of this, the results from PowerTutor should only be viewed as rough approximations. They will only be used to give an *indication* of the difference in energy consumption between the Java and native C implementations.

5.4 Validity threats

Android environment
Some parts of the Android environment are hard to control as an application developer. System processes may demand CPU time while the tests are running, causing inaccuracies in the test results.

Energy measurements may not be accurate enough
The devices used in the experiment are not officially supported by the PowerTutor application. Accurate results are not guaranteed for unsupported devices.

Hardware differences between devices
The physical devices used in the experiment all share the same CPU architecture. However, other hardware differences between the devices may affect the results in unpredictable ways.

Difficulties writing similar code in Java and C
It may prove impossible to write an algorithm equally in Java and C because of the structural differences between the two languages.

5.5 Experiment preparations

5.5.1 Physical devices

The physical Android devices used for the experiment include a slightly older phone, a tablet and a modern smartphone. These devices have different hardware components and processor speeds, but they all share the same CPU architecture, the ARMv7. They are also running different versions of the Android platform: The older smartphone uses version 2.3.3, the tablet uses version 3.2 (which has been optimized for large-screen devices), and the modern smartphone uses the recent 4.0.3 version. Thanks to this wide range of platforms, the experiment will show if the performance differences between Java and native C are consistent across Android versions.

The devices used for the experiments are:

Sony Ericsson Xperia X10
1.0 GHz Qualcomm single core Scorpion CPU
384 MB of dedicated RAM
Android version 2.3.3

Samsung Galaxy Tab 10.1
1 GHz dual core Nvidia Tegra 2 CPU
1 GB of dedicated RAM
Android version 3.2

Samsung Galaxy S II
1.2 GHz dual core ARM Cortex-A9 CPU
1 GB of dedicated RAM
Android version 4.0.3

Before each experiment the devices will be prepared to meet the following conditions:

1. The battery is fully charged
2. The SIM card has been removed
3. The device has been rebooted
4. All optional background processes have been terminated. Only the programs necessary for the operating system to be functional remain active
5. WiFi, Bluetooth and GPS have all been deactivated
6. Screen timeout has been deactivated
7. Synchronization has been turned off

This is all to minimize the risk for external interference that could decrease the accuracy of the test results.

5.5.2 Android emulators

The Android emulator used for the experiment is the virtual mobile device emulator that is included in the Android SDK (Software Development Kit). It emulates a physical Android device and lets the developer run an Android application without the need of deploying it to an actual smartphone or tablet. Two emulator instances were used for the experiment: one running Android version 2.3.3 and the other running version 4.0.3.

5.6 Experiment execution

5.6.1 Performance measurements

The algorithms typically manipulate values in primitive arrays or use those values to perform various calculations. The maximum number of elements in the array is different for each algorithm but, in most cases, the maximum value is slightly lower than the value that would cause the application to run out of memory and throw an OutOfMemoryException. High input values were chosen to increase the execution time. Longer execution times should increase the accuracy of the time measurements and reduce the impact of random external interference.

Time is measured in milliseconds between the start and the end of each algorithm execution. All time measurements are made in the Java code using the *System.nanoTime()* method. This library routine is called before and after the call to each method (Java) or function (C) that is to be evaluated. The time elapsed is calculated by subtracting the first value returned by *System.nanoTime()* from the second value.

To improve the accuracy of the results, each algorithm execution is iterated ten times. The time is measured for every iteration and, finally, the median value is chosen as the result. Using the median value will prevent extreme values, that may be the products of external interference, from corrupting the results. Such extreme values were sometimes encountered, and they were probably caused by interference from system processes and/or garbage collection.

The eight different algorithms were implemented in both Java and native C. These sixteen implementations were tested on three physical devices and two different versions of the Android emulator. For each implementation, ten different input values were used, and each measurement was iterated ten times. In total, 8000 time measurements were made.

A visual overview of the performance measurements is provided in appendix A.

5.6.2 Energy measurements

The PowerTutor application was used to make energy measurements for each of the algorithms. The algorithms were executed with their maximum input values. These values were chosen so that the measurements would be less sensitive to external interference. Five measurements were made for each algorithm, and the median value was chosen as the result. The PowerTutor experiments could not be automated; that is why only five measurements were made for each algorithm. The energy measurements were only performed on the physical Android devices.

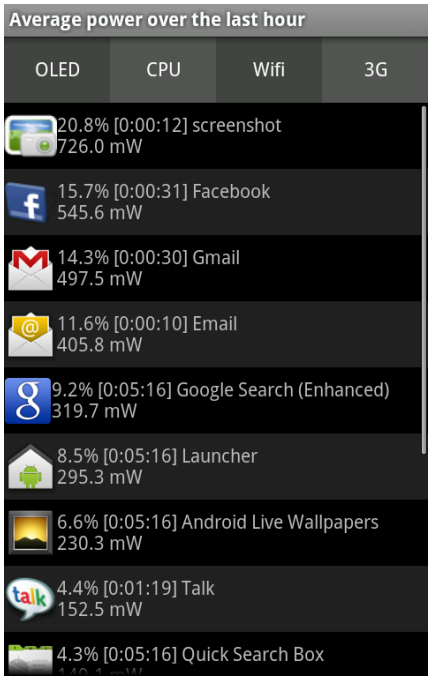
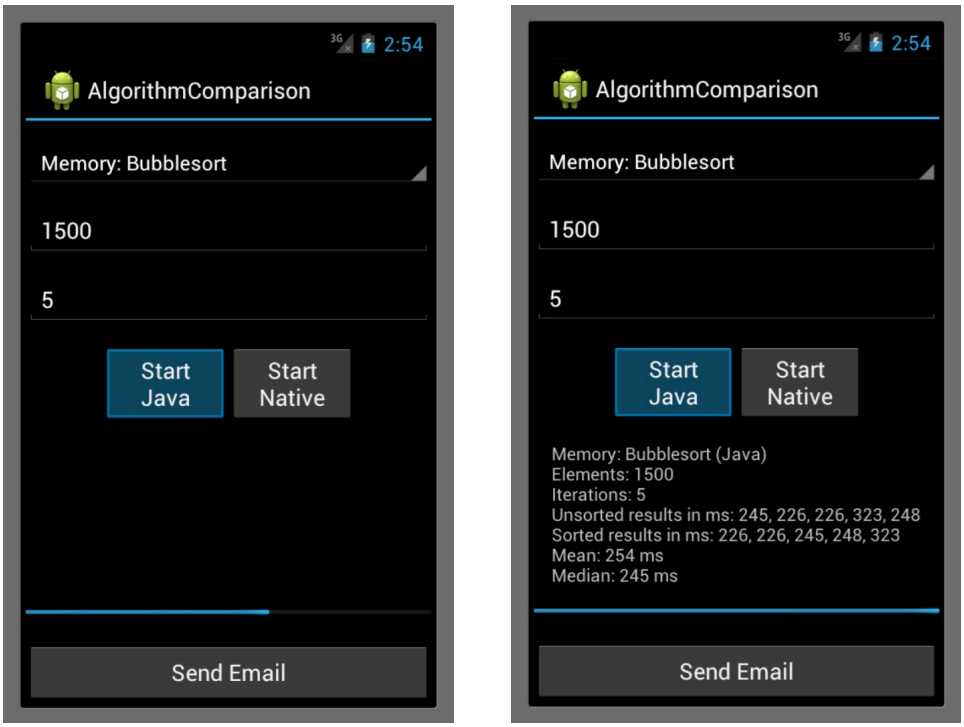


Figure 4: PowerTutor application, *May 2012* <*http://ziyang.eecs.umich.edu/projects/powertutor*>

5.7 AlgorithmComparison application

The user interface of the Android application that was created for the experiment is shown in figure 5. One of the eight algorithms is selected from a drop-down list; in this case, the bubblesort algorithm has been chosen. In the other input fields, the user specifies the input value (number of elements) and the number of iterations. In this case, the user has chosen to sort 1500 elements and to repeat the test 5 times. The results of these 5 measurements are displayed in figure 5 (b).

Three different versions of the application were created, targeting Android versions 2.3.3, 3.2 and 4.0.3. An automated version of the application was used for the final measurements. In this automated version, all of the implementations are executed, one at a time, with pre-defined input values. The results of the time measurements are stored in a buffer. When the execution is finished, the gathered test data is delivered by email.



(a) Algorithm execution (b) Execution results

Figure 5: Algorithm comparison application

6 Experiment results and analysis

6.1 Algorithm execution results

The results of all experiments are presented in appendix B. Each page of the appendix displays the results for a single algorithm. For each algorithm, the performance results are presented in five diagrams; there is a separate diagram for each of the five devices. These diagrams show how much time was required for the Java and native implementations for each of the 10 input values. Since the hardware differs between the different devices, the comparison between versions should not be based on the actual time of execution. What is interesting is the performance difference between the Java and the native implementations for each version. As such, the reader should focus on the relationship between the Java performance curve (red) and the native C performance curve (blue).

The last diagram on each page illustrates the differences in energy consumption between the different implementations and Android versions.

6.2 Analysis of performance results

6.2.1 Overview

In this section, the results of the performance measurements are summarized. The algorithms tested are related to four different operation types: integer calculations, floating-point calculations, memory access operations and recursion. The results for each operation type are discussed in separate subsections.

Each subsection highlights the performance differences between the Java and native implementations. The results for the different Android versions are also compared to each other, and the results from the emulators are compared to those from the physical devices.

When the performance differences between Java and native implementations are quantified, the numbers are always based on the results for the largest input value (the largest number of elements), i. e., the rightmost values in the diagrams. Large input values will increase the execution time and, statistically, longer execution times should decrease the impact of random external interference and provide more accurate results.

When reading about the results for the different operation types below, please refer to the corresponding diagrams in appendix B.

6.2.2 Integer calculations

The experiment results show that the performance of integer calculations can be improved by replacing Java code with native code. However, it is also evident that the latest updates to the Android platform have brought the performance of Java closer to that of native code.

On the Android 2.3.3 device, the Java implementation of the prime number algorithm (appendix B.1) runs significantly slower than the native implementation. The Java code requires 79.9 % more time to execute than the native code. The native implementation emerges victorious on the Android 3.2 and Android 4.0.3 devices as well, but the differences there are *much* less significant: 12.4 % and 11.6 %, respectively.

The results from the square area algorithm executions (appendix B.2) confirm that the performance of Java code has been improved in the latest updates to the Android platform. In this particular case, it would appear that the performance of Java code has been brought only *slightly* closer to that of native code in Android 3.2, as compared to version 2.3.3. The differences between

these two versions are negligible, but on the Android 4.0.3 device, the Java performance is unmistakably closer to that of the native code. Here, the Java implementation takes 59.3 % longer to execute than the native implementation. The corresponding numbers for Android versions 2.3.3 and 3.2 are 89.6 % and 88.0 %.

It is also worth noting that the results from the Android emulators differ somewhat from the results from the actual devices. For instance, the results from the Android 4.0.3 emulator suggest that the native implementation of the prime number algorithm (appendix B.1) performs much better than its Java counterpart. On the physical device running Android 4.0.3, on the other hand, the native implementation only has a slight advantage over the Java implementation. This is also true for the square area algorithm (appendix B.2), where the differences between the native and Java implementations appear greater on the emulators than they actually are on real devices.

6.2.3 Floating-point calculations

The results from the floating-point calculations prove that the use of native code does *not* necessarily improve the performance of an Android application. On the contrary, the use of native code for floating-point calculations can, in some cases, cause a major *decrease* in performance.

The first floating-point algorithm (appendix B.3) uses library routines to calculate the sine and the cosine of a large number of angles. It appears that there are no major differences between the performance of the Java implementation and the performance of the native implementation. Furthermore, these results are surprisingly consistent across Android versions; all devices and emulators produce similar results. This may be due to the fact that most of the execution takes place in well-implemented library routines; such routines may have already been optimized for performance in Android version 2.3.3. Because the results for Java and native C are so similar, it is also possible that the Java library methods used are actually implemented in native code. According to the Android Developers website, the system replaces some calls to library methods with hand-coded assembler [16].

The second floating-point algorithm (appendix B.4) does not make use of any library routines. It simply utilizes basic arithmetic operations such as multiplication and division to calculate the volume of a large number of spheres based on their radii. This explains why the results are completely different from the results of the previous algorithm. When it comes to this basic floating-point arithmetic, the Java implementation is vastly superior to the native implementation on all physical devices. The greatest differences are found on the devices running recent versions of Android. On the device running Android 3.2, the Java implementation is 478 % faster than the native implementation. This number is even greater on the Android 4.0.3 device: 768 %.

The performance of Java is superior to the performance of native code because the Java implementations take advantage of dedicated hardware for floating-point arithmetic. A floating-point unit, such as a VFP, can be used to drastically improve the performance of floating-point operations. Native code does not take advantage of such hardware [23].

All of the physical devices have VFPs, since they all use the ARMv7 processor architecture [26]. This explains why the performance of the Java implementation is greatly superior to that of the native implementation. It appears that the Android 2.3.3 emulator, in contrast to the 4.0.3 emulator, is not designed to take VFPs into account [27]; that would explain why the results from that emulator deviate from the other results. This proves, once again, that Android emulators are not always reliable for performance measurements.

6.2.4 Memory access operations

The performance of certain memory access operations can be greatly improved by using native code. The results from the worst-case bubblesort algorithm (appendix B.5) show that the native implementation outperformed the Java implementation on all devices and emulators. On all physical devices, the Java implementation takes more than four times longer to execute than the native implementation. There are some differences between the devices, but the differences are small and there is nothing to indicate that the performance of Java has been brought closer to that of native code in the recent updates to the Android platform.

As for the emulators, the native implementation once again appears to have more of an advantage than it actually has on real devices. On the Android 2.3.3 emulator, the Java implementation takes 588 % longer to execute than the native implementation, but on the actual Android 2.3.3 device, that number is significantly lower: 337 %. The corresponding numbers for the emulator and the physical device running Android 4.0.3 are 436 % and 326 %, respectively.

This difference between emulators and devices is even more palpable when looking at the results from the second memory access algorithm, *array copy* (appendix B.6). The gaps between the Java implementations and the native implementations are noticeably larger on the emulators than they are on the devices running the same versions of Android. In other words, using native code appears to be more advantageous on an emulator than it actually is on a physical device.

However, the results from this second algorithm differ in some interesting ways from the results of the bubblesort execution. Here, there is a relative improvement in the performance of Java as we move from older devices to devices running more recent versions of Android. On the Android 2.3.3 device, the native implementation is almost twice as fast as the Java implementation (99.6 % faster). That number is lower, 61.5 %, on the Android 3.2 device and even lower on the Android 4.0.3 device: 54.9 %.

6.2.5 Recursion

The results from the first recursive algorithm, *quicksort*, follow the same pattern as some of the previous results. In terms of performance, the native implementation is superior to the Java implementation on all devices and emulators. However, on newer devices running recent versions of Android (3.2 and 4.0.3), the difference in performance is noticeably smaller: 106 % and 97.9 %, as opposed to 155 % on the Android 2.3.3 device.

Like before, the results from the emulators suggest that the native implementation has a greater advantage over the Java implementation than it actually has on physical devices.

The results from the second algorithm, *Fibonacci*, show how the performance is affected when the number of recursive calls increases exponentially. It would appear that a native implementation is preferable for this kind of deep recursion; the native implementation can handle a few more elements before the execution time starts to increase very rapidly.

6.3 Analysis of energy consumption results

Most of the results from the energy measurements show that there is a close correlation between execution time and energy consumption. When one of the implementations takes longer to execute than the other, that implementation typically consumes more energy as well. In a majority of the cases, the energy consumed appears to be proportional to the execution time. This holds true for many of the algorithms (appendix B.1, B.3, B.5, B.6, B.7 and B.8).

Unfortunately, the precision of the tool used to measure energy consumption cannot be verified. When the same measurement was made several times, variations were sometimes observed. This

may indicate that the precision of the tool is somewhat lacking, which would explain why there are some deviations in the results: Sometimes, the energy consumption does *not* appear to be proportional to the execution time. One such example is the *square area* algorithm (appendix B.2). The results from that experiment show that, in terms of performance, there is a clear difference between the two different implementations (on all three devices). When it comes to the energy consumption results, however, the Java and native implementations appear to be more equal.

The results from one of the algorithms differ significantly from the others. The *volume calculation* algorithm (appendix B.4) ran much slower when implemented in native code instead of Java; on the Android 4.0.3 device, for instance, the native implementation required 768 % more time to execute than its Java counterpart. Somewhat surprisingly, the difference in energy consumption between the two implementations was much smaller: only 95 % on the Android 4.0.3 device. As mentioned before, in this particular case there are some significant differences between the Java and the native implementations. The Java implementation makes use of the Vector Floating-Point coprocessor whereas the native implementation does not. Because of this, one explanation could be that the VFP calculations simply require more energy per time unit than regular CPU calculations. Another explanation would be that PowerTutor, the application used to measure energy consumption, is not accurate for VFP calculations.

7 Data synthesis

7.1 Comparison to previous work

7.1.1 Integer calculations

Lee and Lee [18] and Lee and Jeon [21] found that the performance of integer calculations could be improved significantly by using native code rather than Java. Our measurements confirm that native code outperforms Java when it comes to integer calculations, but the differences were not as large as those observed in [18] and [21]. This could be due to the fact that their measurements were made on Android emulators. Our research shows that emulators have a tendency to give native code a greater advantage over Java than it actually has on physical devices (as discussed in section 6.2).

Ulvesand and Eriksson [23] found that there was a major difference between the performance of Java and native code on an Android 2.1 device. However, they found that the performance difference was almost negligible on an Android 2.2 device. Our results show a similar pattern: As we move to more recent versions of Android, the gap between Java and native code decreases. On Android versions 3.2 and 4.0.3, the gap is almost non-existing. It is somewhat surprising that Ulvesand and Eriksson found the performance differences to be negligible as early as Android 2.2; our research suggests that the performance gap is still present on Android 2.3.3. These results are backed up by the work of Lin et al. [19]. They also conducted their experiments on a physical device running Android 2.2, but, unlike Ulvesand and Eriksson, they concluded that native code still had a great advantage over Java.

7.1.2 Floating-point calculations

The research conducted by Lee and Lee [18] and Lee and Jeon [21] suggests that native implementations only have slight advantages over Java implementations for floating-point calculations. Their experiments were conducted on Android emulators and their results match our results from the Android version 2.3.3 emulator.

Ulvesand and Eriksson concluded that Java was vastly superior to native code for floating-point calculations on an Android 2.2 device. Our results show that this is also the case for Android versions 2.3.3, 3.2 and 4.0.3 (appendix B.4).

7.1.3 Memory access operations

When it comes to memory access operations, our results are equivalent to those from [18] and [21]. Interestingly, there is one set of results from previous research that differs from all others: the Android 2.2 results from Ulvesand and Eriksson [23]. They found that there was no performance difference between the two implementations of the bubblesort algorithm. Our results, on the other hand, show that native code is still vastly superior to Java on all recent versions of Android. These results are based on a large number of measurements and we have found no reason to question their validity.

7.1.4 Recursion

There are only two other papers that evaluate the performance of recursive algorithms. Ulvesand and Eriksson [23] looked at the performance of the quicksort algorithm. They came to the conclusion that the native implementation was much quicker than its Java counterpart on Android 2.1. The difference was much smaller on the Android 2.2 device. Our results also show that the native

implementation has an advantage over Java; in fact, we found the performance difference between Java and native code to be a bit larger. Lin et al. [19] tested a recursive implementation of the Fibonacci algorithm and found that the native code executed more than 7 times faster than the Java code when the number of elements was set to 35. We also used 35 as our highest input value, but we found the performance difference to be smaller: The native implementation was approximately twice as fast as the Java implementation.

7.2 Discoveries

7.2.1 Performance differences between Java and native code

Even on the most recent versions of Android, native implementations are superior to Java for certain types of operations: integer calculations, memory access operations and recursion. However, the results also show that native code should not be used for floating-point calculations; this is also stated by Ulvesand and Eriksson [23]. They conducted their experiment on a device running Android 2.2. On the more recent versions of Android, the performance differences are even more evident. Thus, Java implementations are preferable for floating-point calculations.

7.2.2 Differences between Android versions

The research shows that the latest updates to the Android platform have improved the performance of Java as compared to native code. Indications of this have been observed for all four operation types (integer calculations: appendix B.1 and B.2, floating-point calculations: appendix B.4, memory access operations: appendix B.6, recursion: appendix B.7). However, for certain algorithms, the performance differences are fairly consistent across all recent versions of Android (appendix B.3, B.5 and B.8).

Ulvesand and Eriksson [23] found that, in several cases, the performance of Java was equivalent to that of native code as early as Android 2.2. They anticipated that, in the future, Java implementations could actually surpass native code in terms of performance due to improvements in the DVM. Though the results in section 6.1 confirm that the performance gap has been reduced, the use of native code is still justified for performance optimization of certain algorithms. The operation types that still benefit from the use of native code include recursion and memory access. The performance of integer calculations can also be improved by using native code, but on the most recent Android versions (3.2 and 4.0.3), Java and native implementations sometimes show similar performance.

7.2.3 Differences between physical devices and emulators

The results from [18] and [21] are based on tests executed on Android emulators. This approach is criticized in [23], but Ulvesand and Eriksson do not show any measurements that back up their criticism. However, the results from section 6.1 do confirm that this criticism is justified. It appears that, in many cases, emulators tend to give native code a greater advantage over Java than it actually has on physical devices. The results also show that older emulators are *very* inaccurate for performance measurements of floating-point calculations. With these discoveries in mind, there are reasons to question the validity of the results from [18] and [21].

7.2.4 Energy consumption

No previous research related to how the use of native code affects energy consumption was found during the literature survey. The results from this experiment indicate that there is a close corre-

lation between execution time and energy consumption. When one of the implementations takes longer to execute than the other, that implementation typically consumes more energy as well.

8 Conclusion

The conclusions in this chapter are derived from the experiments that were conducted. They are all related to the main topic of this thesis: performance and energy efficiency comparisons between Java and native implementations on Android. This thesis also summarizes previous work that has been done in the area of performance and energy efficiency of mobile applications. However, these topics are not covered in this chapter. For a general overview of this problem area, please refer to section 4.1. The approaches that have been suggested to improve the performance and energy efficiency of Android applications are discussed in section 4.2.

8.1 Main findings

Whether or not to use native code for performance optimization of Android applications depends on the circumstances. The results of the measurements, in conjunction with the previous research, lead to the following conclusion: The use of native code is justified if an application involves large amounts of data processing and performs integer calculations, memory intense operations or deep recursion. This is particularly applicable for applications targeting earlier versions of the Android platform since the overall performance difference decreases with each subsequent version.

For floating-point calculations, the use of native C is not recommended. In many cases, such operations are executed faster when implemented in Java.

The use of Android emulators for performance measurements and similar evaluations is strongly discouraged. Results from emulators prove to be highly inaccurate when compared to those from physical devices. Though emulators targeting Android version 4.0 and up *can* emulate hardware support for floating-point operations, the differences between emulators and physical devices are still too substantial to be disregarded.

The results from the energy measurements demonstrate a close correlation between execution time and energy consumption. Under certain circumstances, native implementations execute faster than their Java counterparts. When the time of execution is decreased, the energy consumption will typically decrease as well.

8.2 Future work

Future work within this area could involve repeating similar measurements on devices running future versions of the Android platform. This would determine whether or not the performance of the Dalvik Virtual Machine continues to improve. Even if the performance of Java does not surpass that of native code, it may come close enough to make native code obsolete for performance optimization.

It would be interesting to see more examples of applications that are written mainly in native code and to see how those compare to their Java equivalents in terms of overall performance. Most of the current research is focused on isolated algorithms rather than complete applications.

It could also be interesting to investigate how the hardware components affect the relative performance of Java as compared to native code. The perfect scenario would be if several different versions of Android could be tested on the *same* physical device. Would this generate different results, or would the results stay the same?

Finally, it would be desirable to increase the precision of the energy measurements. This could involve using more sophisticated tools or conducting the experiments on one of the phones that are officially supported by the PowerTutor application.

9 Glossary

AOSP	-	Android Open Source Project
ARM	-	Advanced RISC Machine
CPU	-	Central Processing Unit
DVM	-	Dalvik Virtual Machine
DFS	-	Dynamic Frequency Scaling
FPU	-	Floating-Point Unit
JIT	-	Just In Time compiler
JNI	-	Java Native Interface
JVM	-	Java Virtual Machine
NDK	-	Native Development Kit
OHA	-	Open Handset Alliance
RISC	-	Reduced Instruction Set Computer
VPF	-	Vector Floating-Point coprocessor

References

[1] K. Nagamine. Worldwide smartphone market expected to grow 55% in 2011 and approach shipments of one billion in 2015. http://www.idc.com/getdoc.jsp?containerId=prUS22871611, 2012.

[2] Gartner. Gartner highlights key predictions for it organizations and users in 2010 and beyond. http://www.gartner.com/it/page.jsp?id=1278413, January 2010.

[3] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[4] The Apache Software Foundation. Apache software license version 2.0. http://www.apache.org/licenses/, April 2012.

[5] Free Software Foundation, Inc. GNU General Public License, version 2.0. http://www.gnu.org/licenses/gpl-2.0.html, April 2012.

[6] Andy Rubin, Google, Inc. Andy Rubin Google Plus. https://plus.google.com/u/0/112599748506977857728/posts/Btey7rJBaLF, 2012.

[7] CNN. Android passes BlackBerry as No. 1 on smartphones. http://money.cnn.com/2011/03/07/technology/android/, 2011.

[8] Google, Inc. What is the NDK? http://developer.android.com/sdk/ndk/overview.html, 2012.

[9] Alex Shye, Benjamin Scholbrock, Gokhan Memik, and Peter A. Dinda. Characterizing and modeling user activity on smartphones: summary. *SIGMETRICS Perform. Eval. Rev.*, 38(1):375–376, June 2010.

[10] Jia Hao, Seon Ho Kim, Sakire Arslan Ay, and Roger Zimmermann. Energy-efficient mobile video management using smartphones. In *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, pages 11–22, New York, NY, USA, 2011. ACM.

[11] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[12] M. Calder and M.K. Marina. Batch scheduling of recurrent applications for energy savings on mobile phones. In *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on*, pages 1 –3, june 2010.

[13] Chin-Liang Tsai, Hsiao-Wen Chen, Jiun-Long Huang, and Chih-Lin Hu. Transmission reduction between mobile phone applications and restful apis. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 445–450, New York, NY, USA, 2011. ACM.

[14] Bruno Gil and Paulo Trezentos. Impacts of data interchange formats on energy consumption and performance in smartphones. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication*, OSDOC '11, pages 1–6, New York, NY, USA, 2011. ACM.

[15] K. Nishihara, K. Ishizaka, and J. Sakai. Power saving in mobile devices using context-aware resource control. In *Networking and Computing (ICNC), 2010 First International Conference on*, pages 220 –226, nov. 2010.

[16] Google, Inc. Designing for performance. `http://developer.android.com/guide/practices/design/performance.html`, 2012.

[17] Google, Inc. Android NDK. `http://developer.android.com/sdk/ndk/index.html`, April 2012. Accessed April 23, 2012.

[18] Jae Kyu Lee and Jong Yeol Lee. Android programming techniques for improving performance. In *Awareness Science and Technology (iCAST), 2011 3rd International Conference on*, pages 386 –389, sept. 2011.

[19] Cheng-Min Lin, Jyh-Horng Lin, Chyi-Ren Dow, and Chang-Ming Wen. Benchmark dalvik and native code for android system. In *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*, pages 320 –323, dec. 2011.

[20] Yann-Hang Lee, P. Chandrian, and Bo Li. Efficient java native interface for android based mobile devices. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1202 –1209, nov. 2011.

[21] Sangchul Lee and Jae Wook Jeon. Evaluating performance of android platform using native c for embedded systems. In *Control Automation and Systems (ICCAS), 2010 International Conference on*, pages 1160 –1163, oct. 2010.

[22] Jung Ha Paik, Seog Chung Seo, Yungyu Kim, Hwan Jin Lee, Hyun-Chul Jung, and Dong Hoon Lee. An efficient implementation of block cipher in android platform. In *Multimedia and Ubiquitous Engineering (MUE), 2011 5th FTRA International Conference on*, pages 173 –176, june 2011.

[23] Andreas Ulvesand and Daniel Eriksson. Native code on Android: A performance comparison of Java and native C on Android. Bachelor's thesis at NADA, KTH Royal Institute of Technology, Stockholm. 2011.

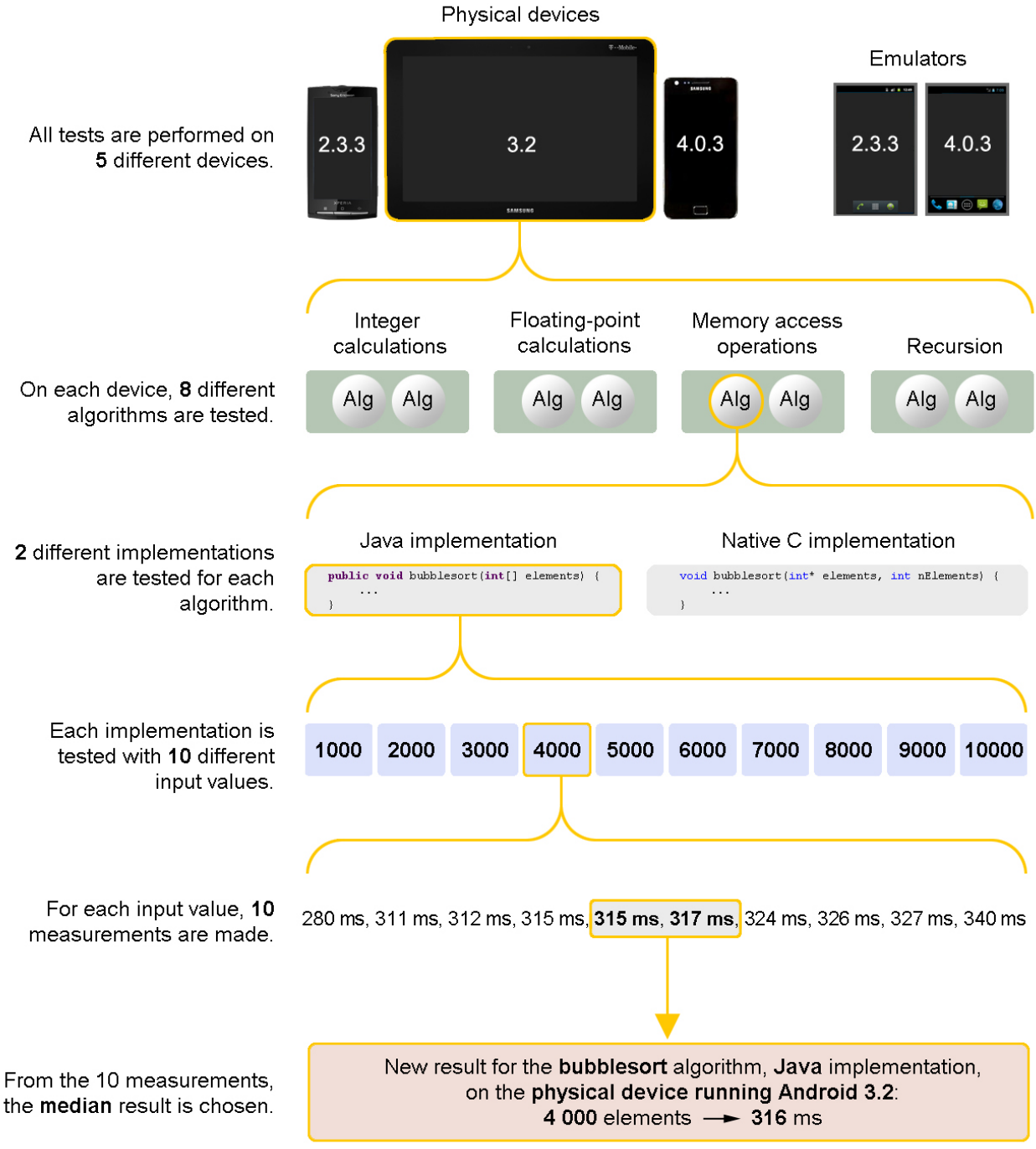
[24] Ki-Cheol Son and Jong-Yeol Lee. The method of android application speed up by using ndk. In *Awareness Science and Technology (iCAST), 2011 3rd International Conference on*, pages 382 –385, sept. 2011.

[25] Lide Zhang Mark Gordon and Birjodh Tiwana. A power monitor for android-based mobile platforms. `http://powertutor.org/`, 2012.

[26] ARM Ltd. The ARM processor architecture. `http://www.arm.com/products/processors/technologies/instruction-set-architectures.php`, 2012.

[27] Google Inc. A faster emulator with better hardware support. `http://android-developers.blogspot.se/2012/04/faster-emulator-with-better-hardware.html`, 2012.

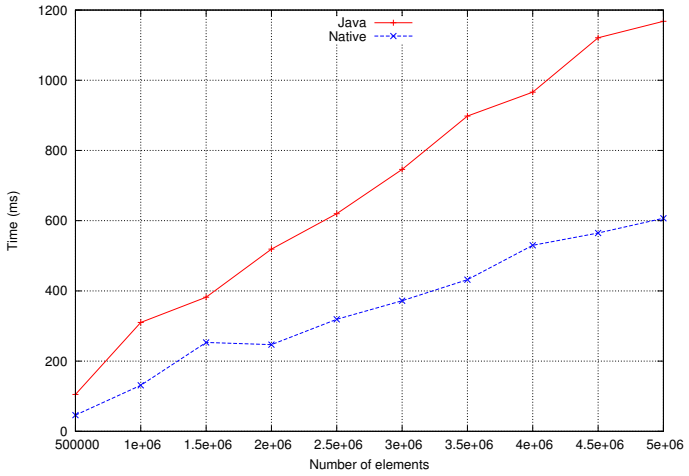
A Appendix: Overview of performance measurements



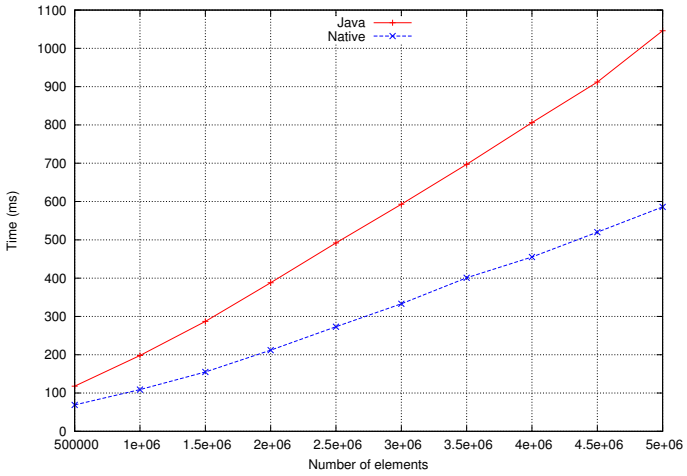
B Appendix: Algorithm execution results

B.1 Integer calculations: prime numbers

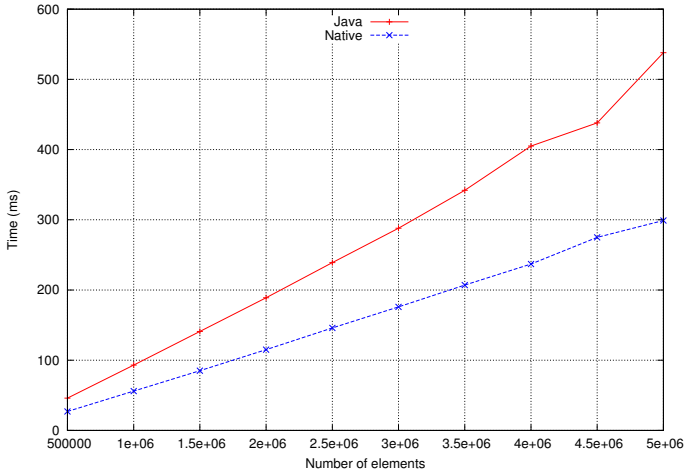
Performance, v2.3.3 emulator



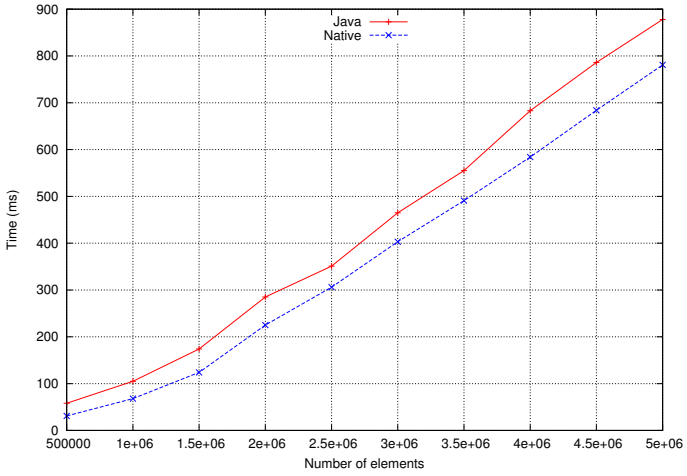
Performance, v4.0.3 emulator



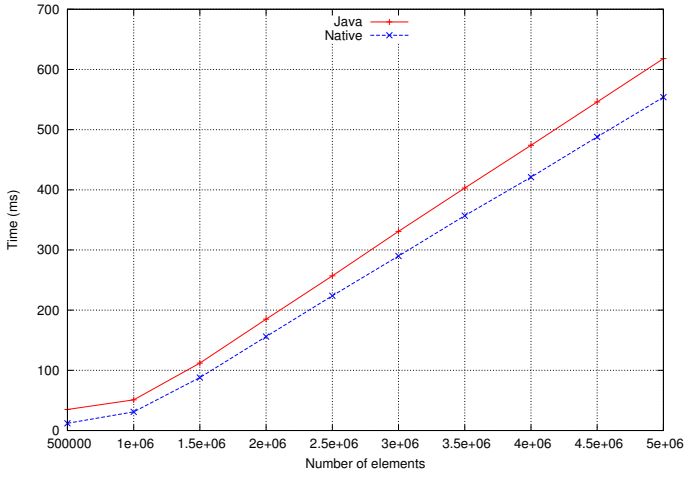
Performance, v2.3.3 device



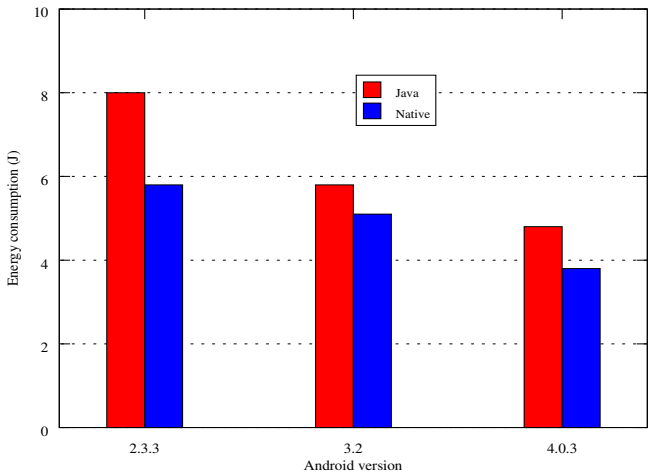
Performance, v3.2 device



Performance, v4.0.3 device

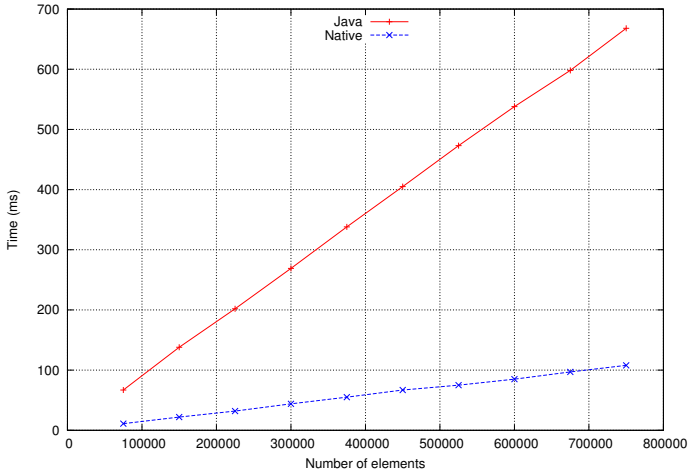


Energy consumption of devices

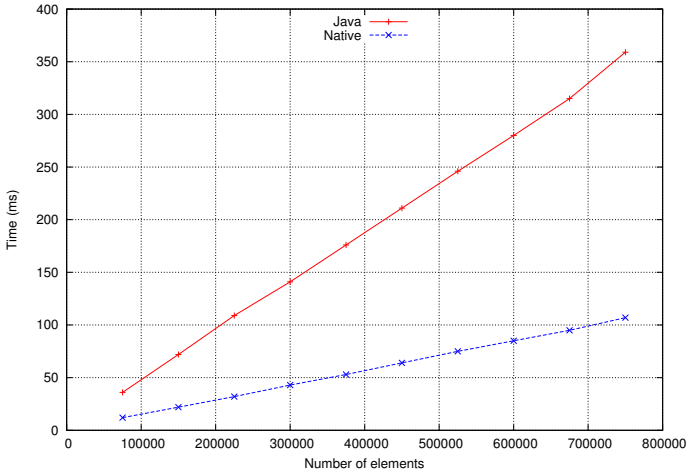


B.2 Integer calculations: square area

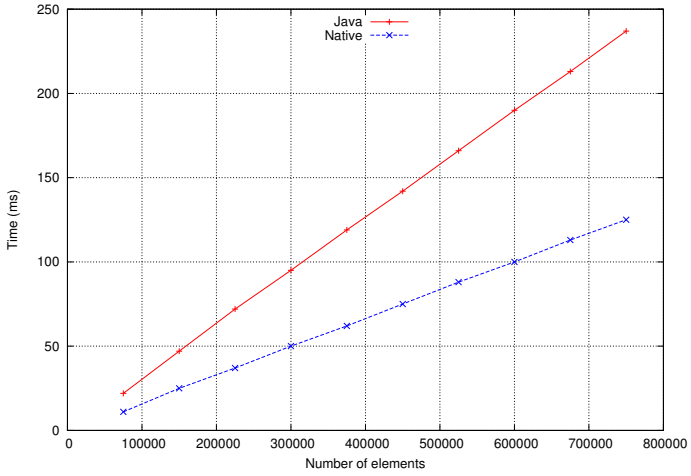
Performance, v2.3.3 emulator



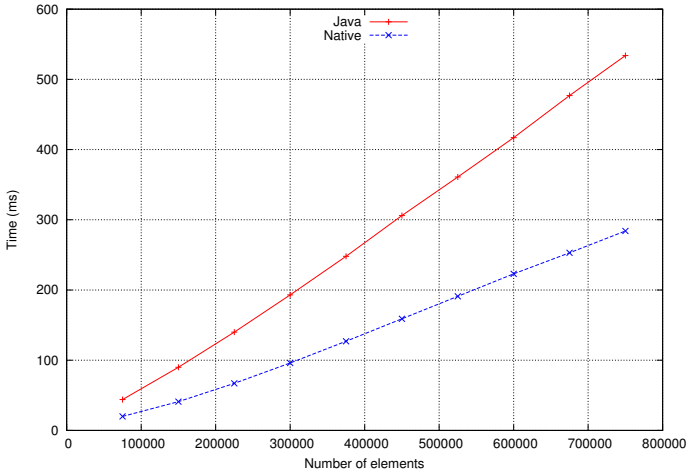
Performance, v4.0.3 emulator



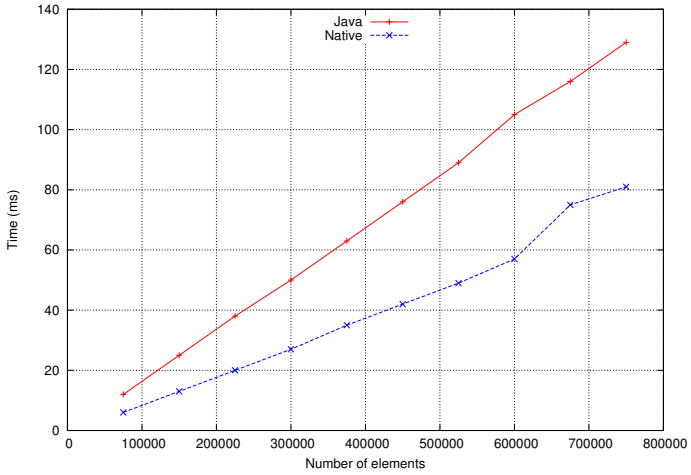
Performance, v2.3.3 device



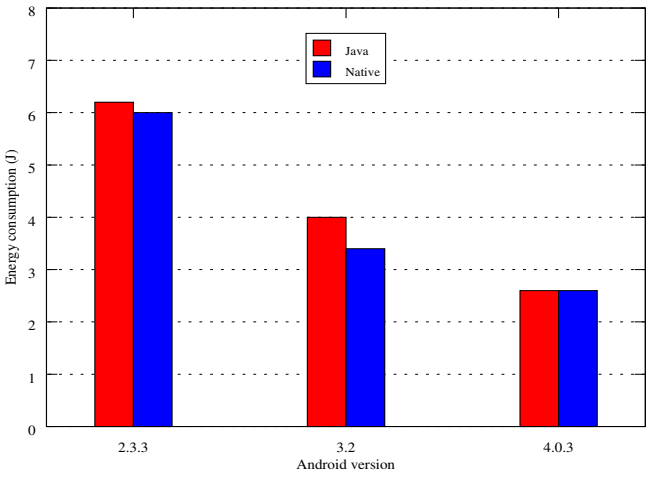
Performance, v3.2 device



Performance, v4.0.3 device

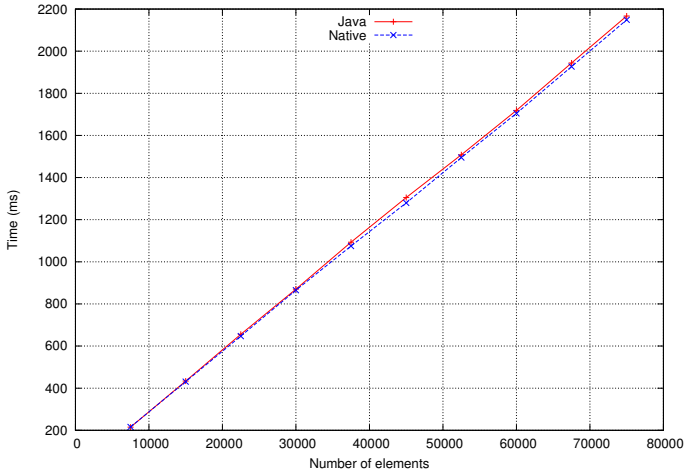


Energy consumption of devices

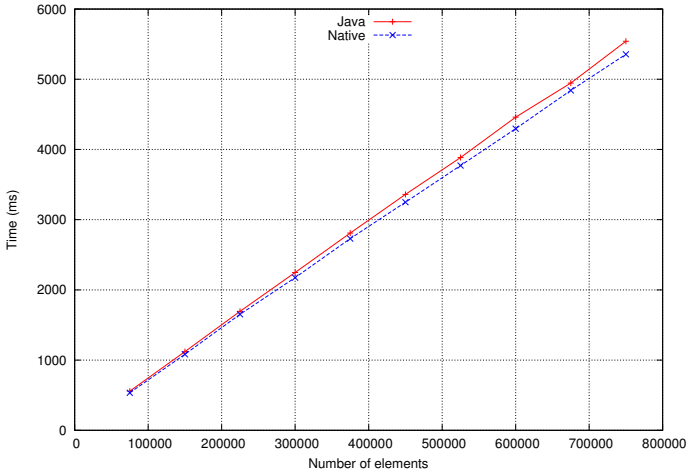


B.3 Floating-point calculations: sine and cosine

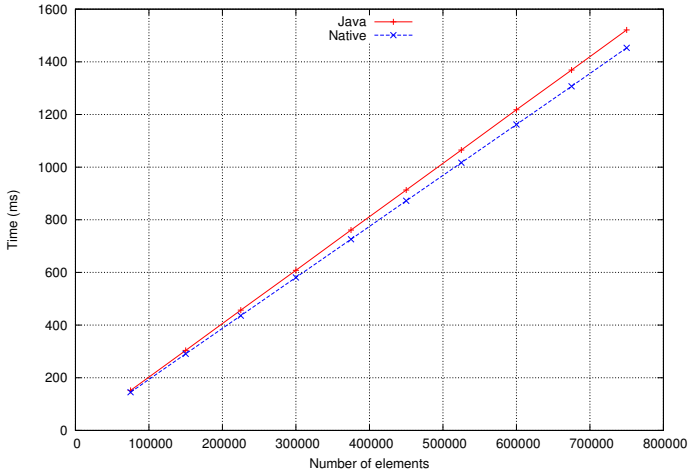
Performance, v2.3.3 emulator



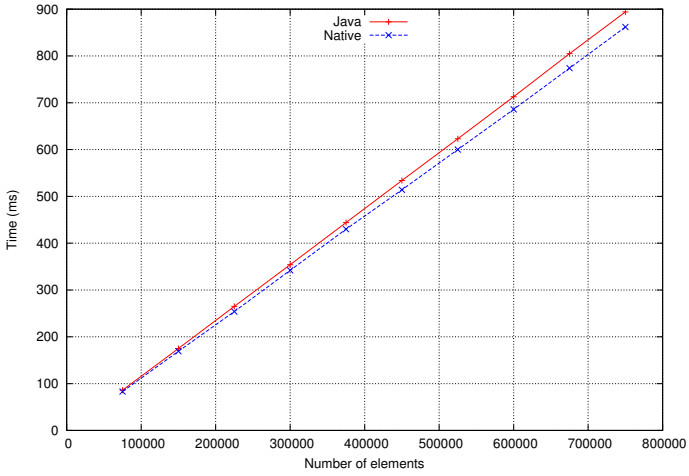
Performance, v4.0.03 emulator



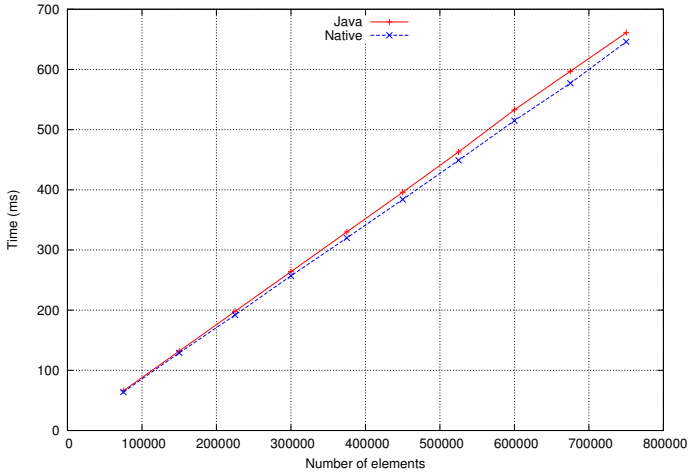
Performance, v2.3.3 device



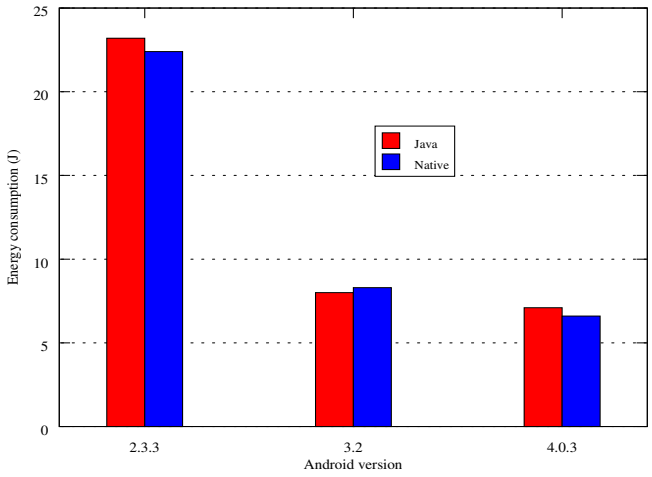
Performance, v3.2 device



Performance, v4.0.3 device

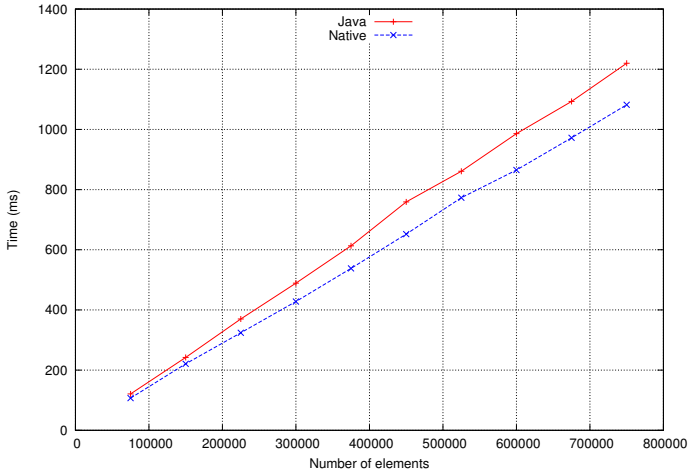


Energy consumption of devices

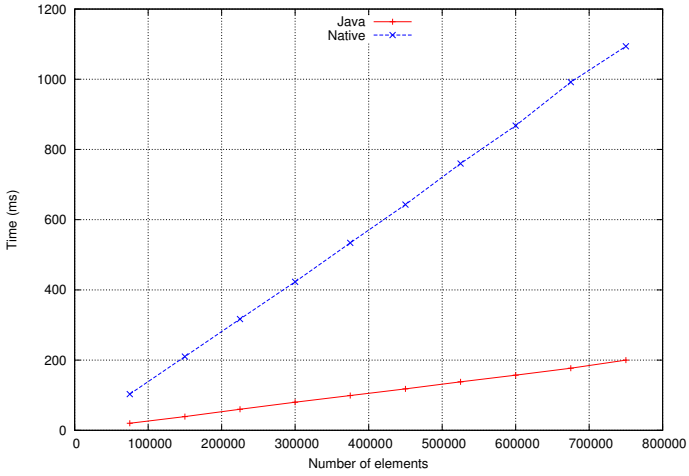


B.4 Floating-point calculations: volume

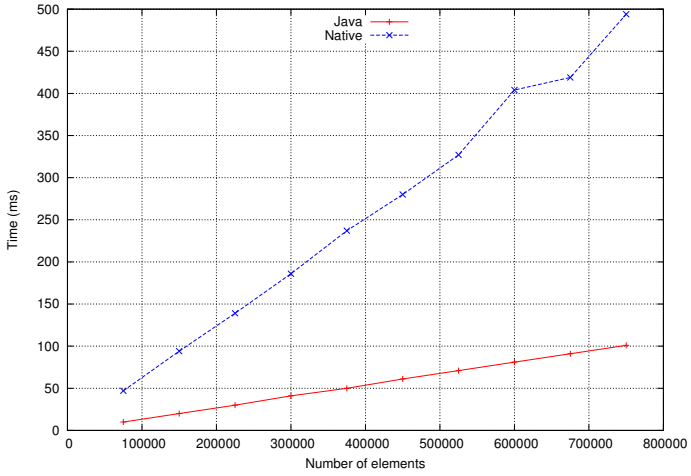
Performance, v2.3.3 emulator



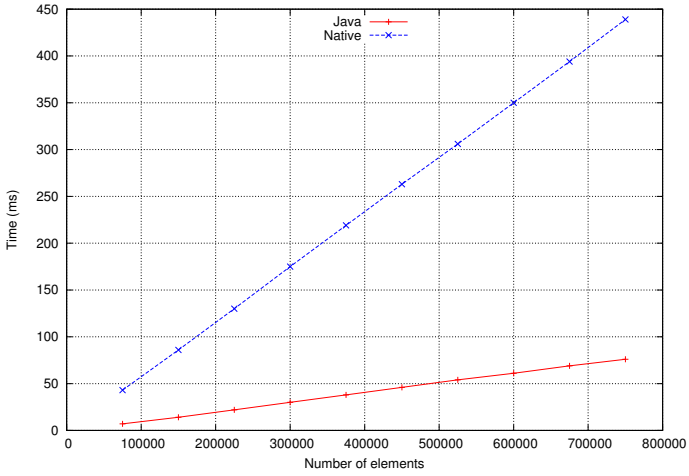
Performance, v4.0.3 emulator



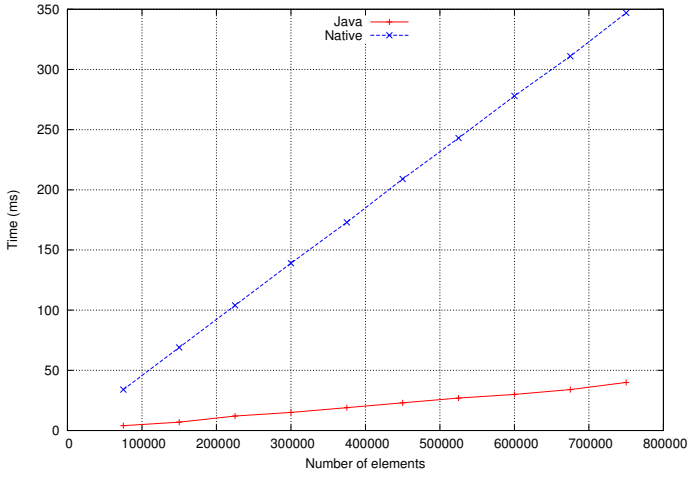
Performance, v2.3.3 device



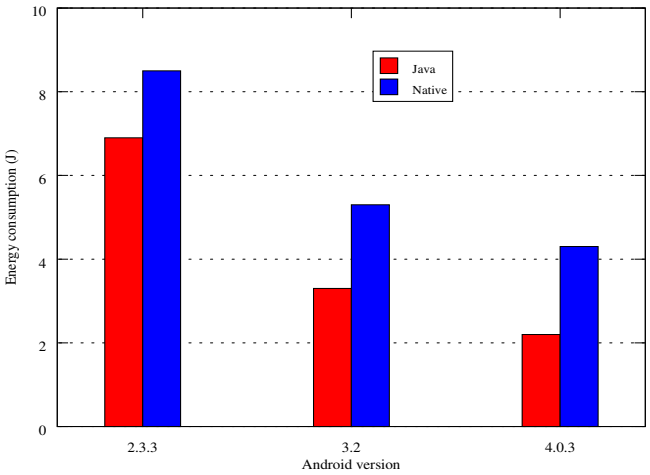
Performance, v3.2 device



Performance, v4.0.3 device

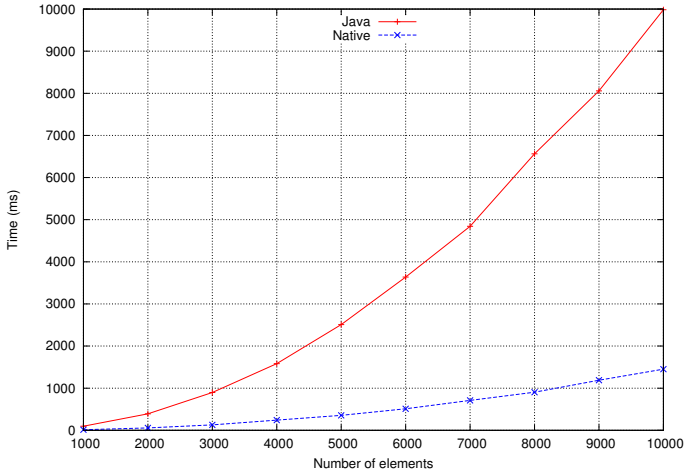


Energy consumption of devices

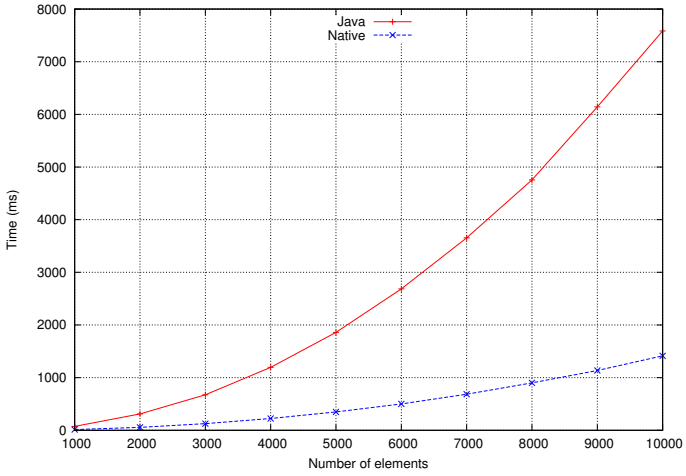


B.5 Memory access operations: bubblesort

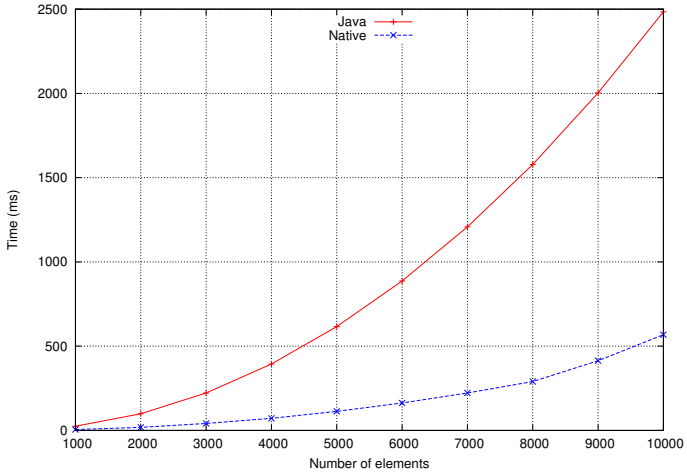
Performance, v2.3.3 emulator



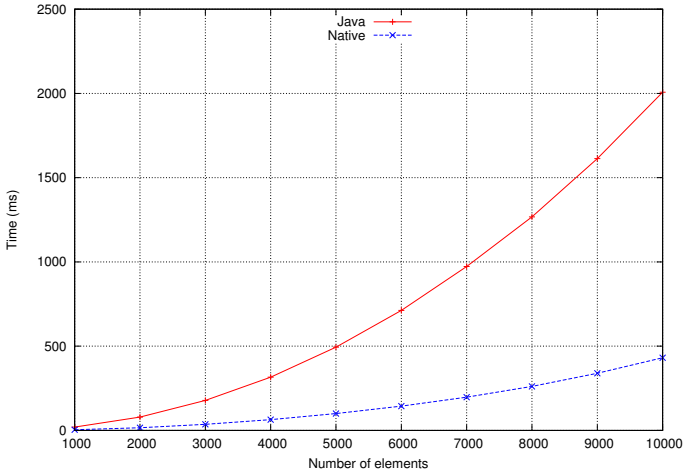
Performance, v4.0.3 emulator



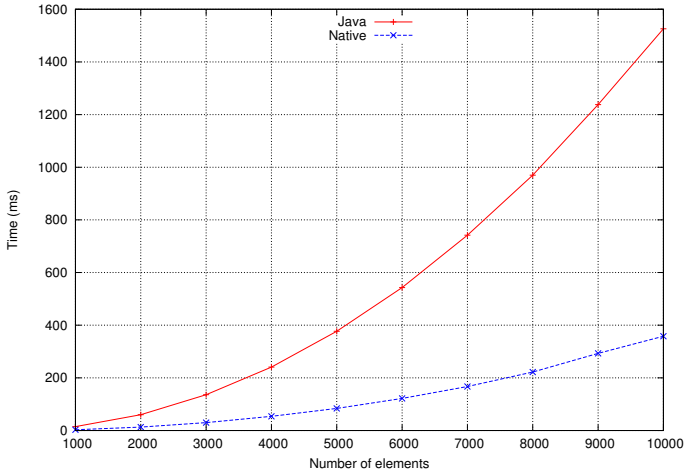
Performance, v2.3.3 device



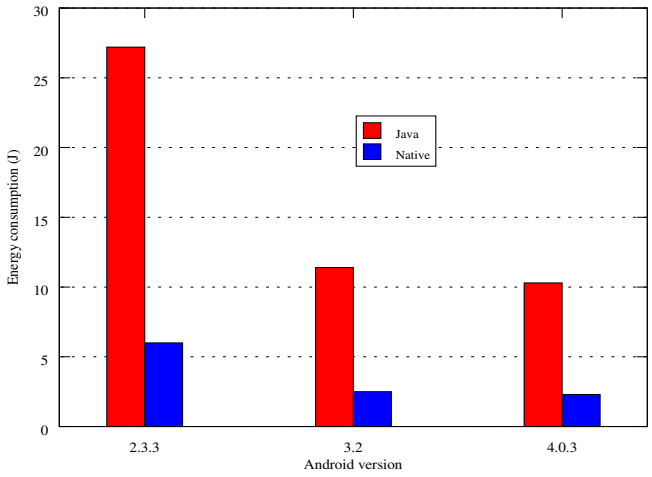
Performance, v3.2 device



Performance, v4.0.3 device

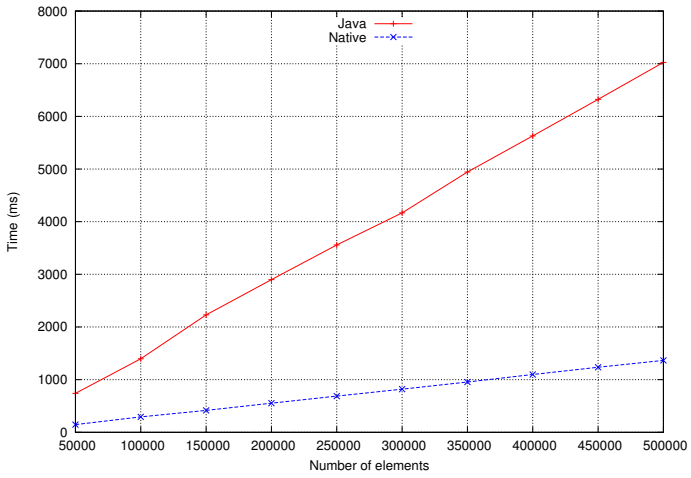


Energy consumption of devices

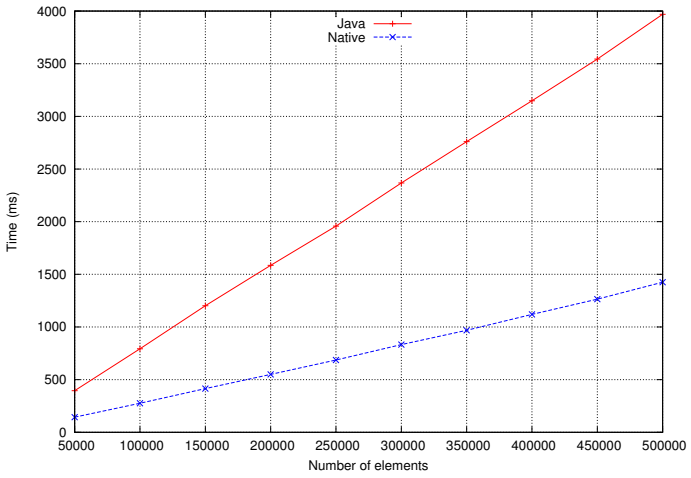


B.6 Memory access operations: array copy

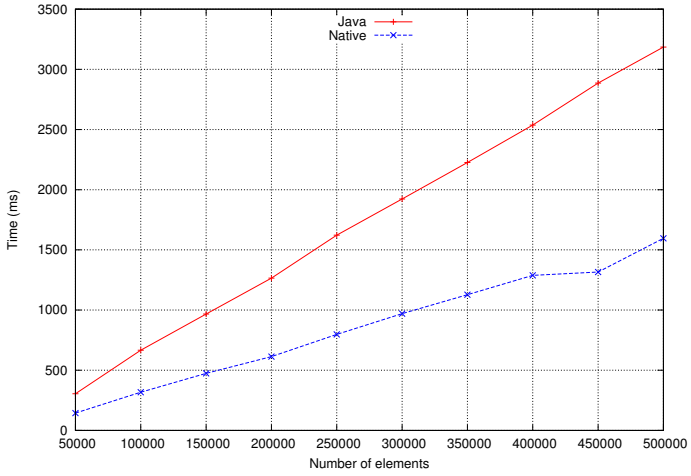
Performance, v2.3.3 emulator



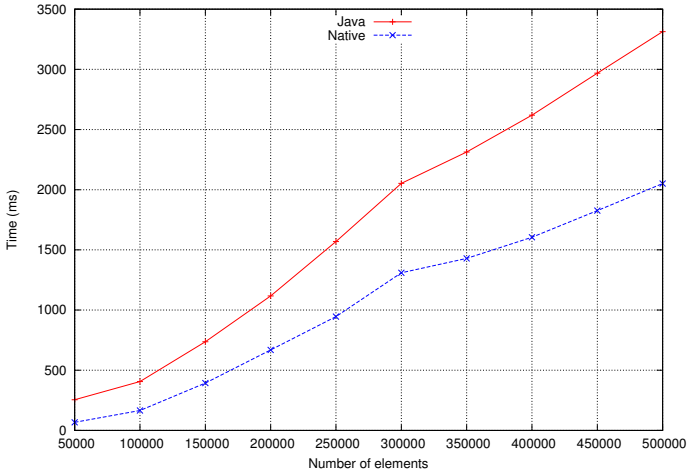
Performance, v4.0.3 emulator



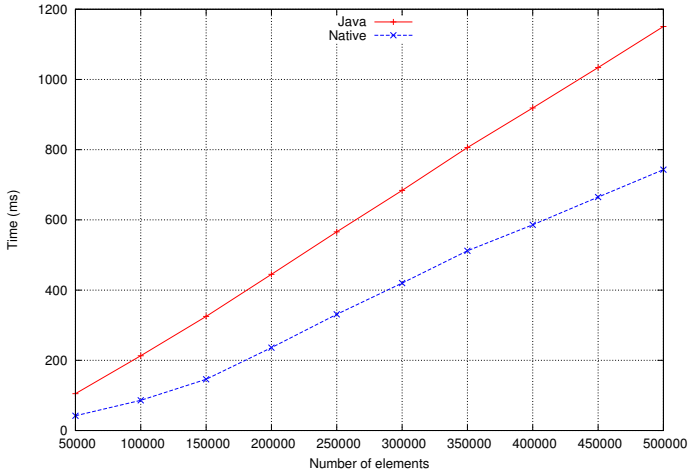
Performance, v2.3.3 device



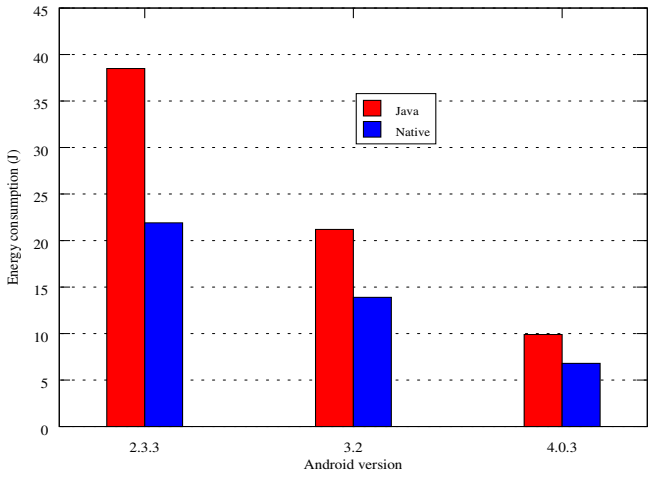
Performance, v3.2 device



Performance, v4.0.3 device

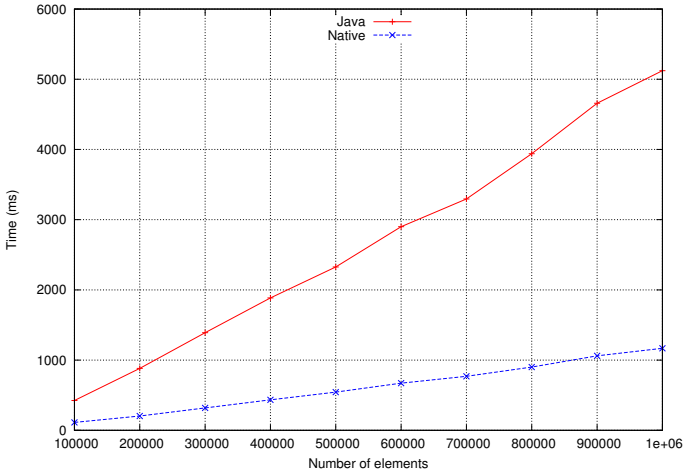


Energy consumption of devices

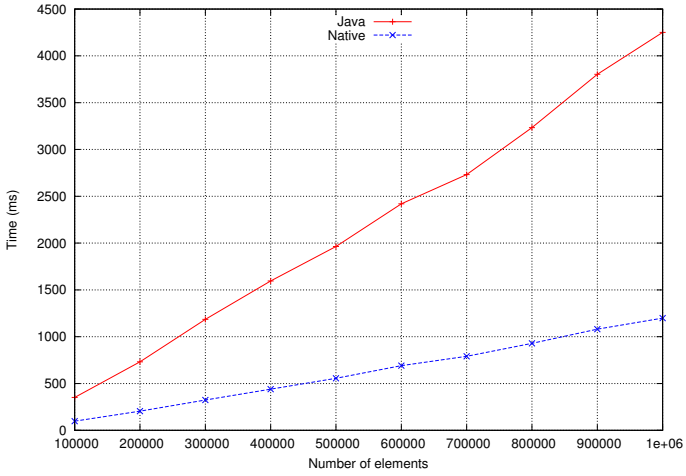


B.7 Recursion: quicksort

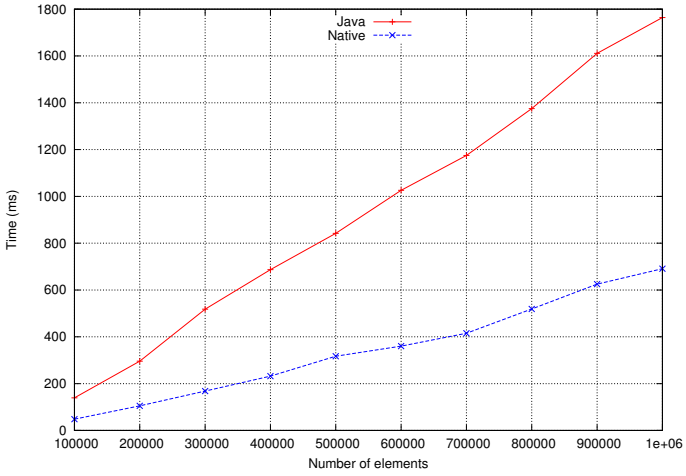
Performance, v2.3.3 emulator



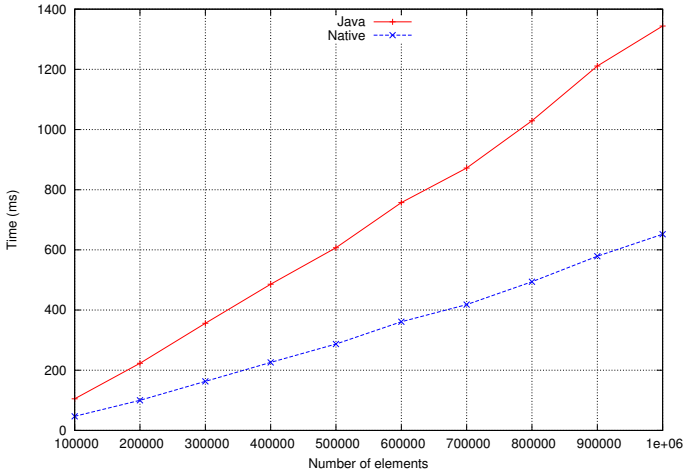
Performance, v4.0.3 emulator



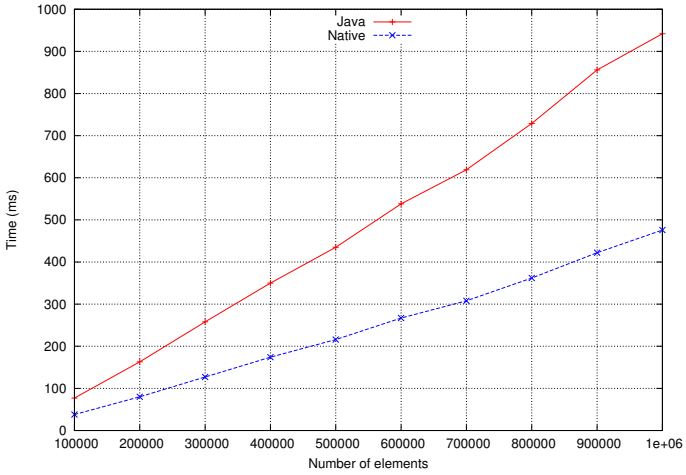
Performance, v2.3.3 device



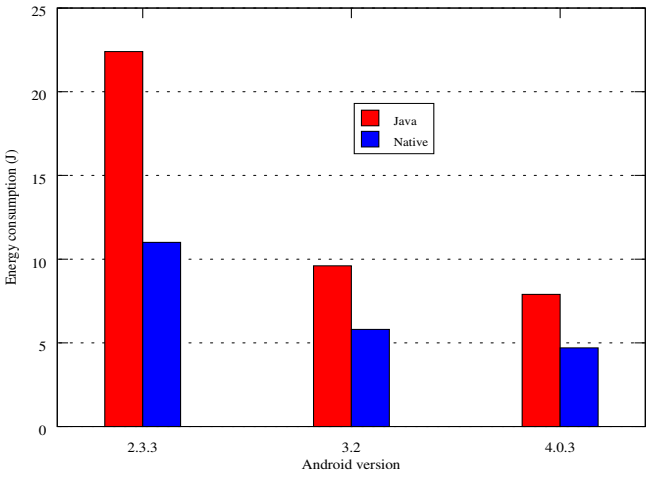
Performance, v3.2 device



Performance, v4.0.3 device

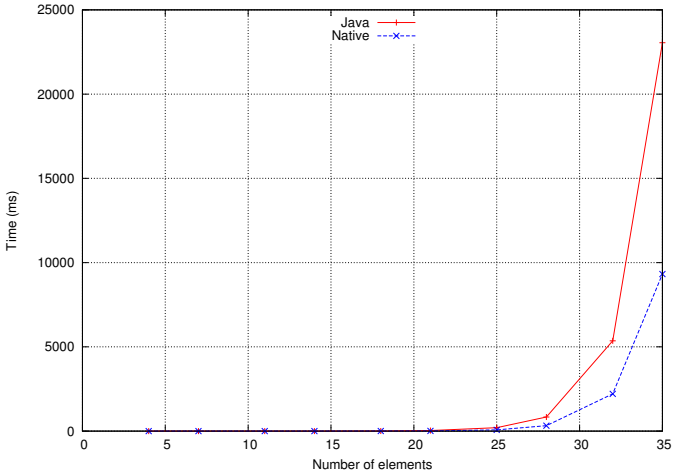


Energy consumption of devices

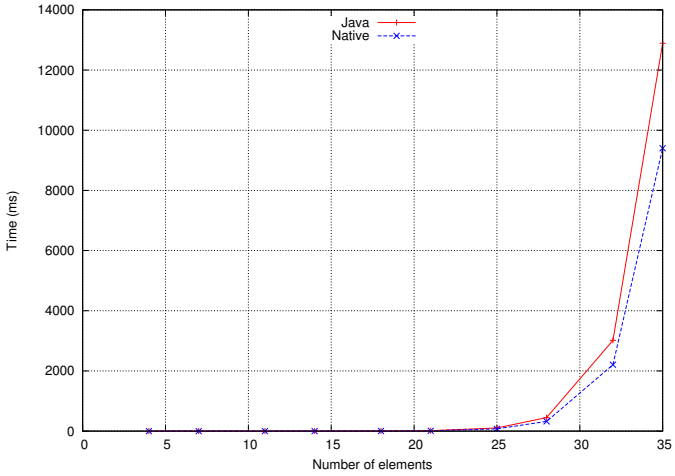


B.8 Recursion: fibonacci

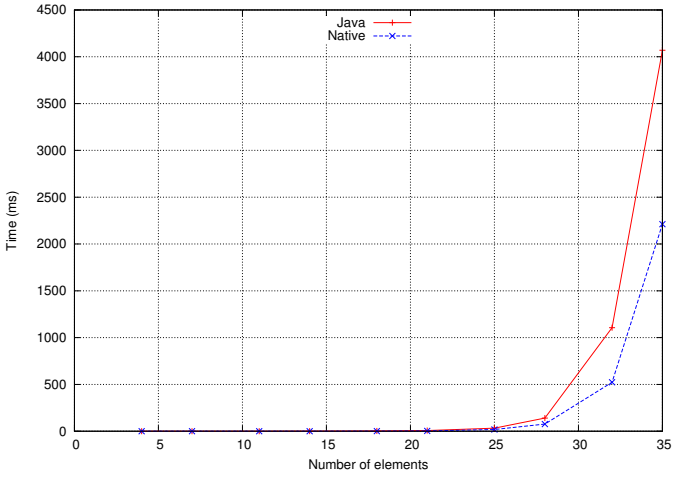
Performance, v2.3.3 emulator



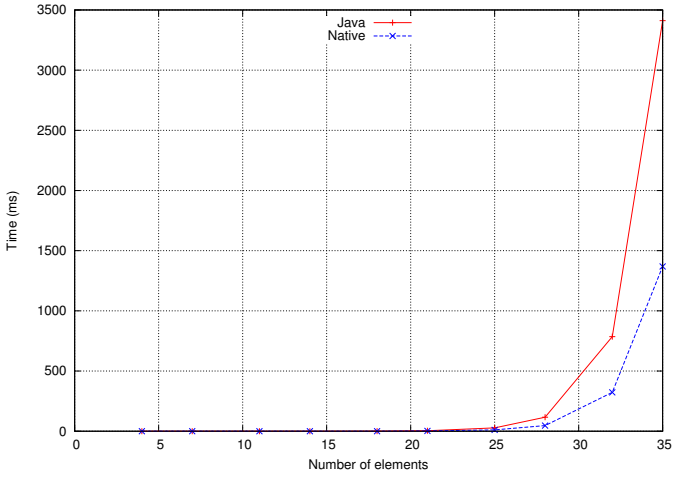
Performance, v4.0.3 emulator



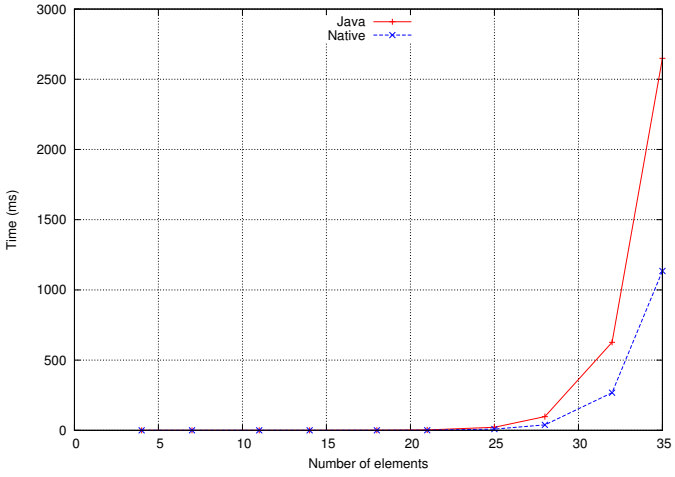
Performance, v2.3.3 device



Performance, v3.2 device



Performance, v4.0.3 device



Energy consumption of devices