Performance and energy-efficiency of Scala on mobile devices
Mattia Denti, Jukka K. Nurminen
Aalto University School of Science
mattia.denti@aalto.fi, jukka.k.nurminen@aalto.fi
May 19, 2013

Abstract
The objective of this study was to compare how new languages like Scala perform on mobile environments compared to the classical languages.

Since Scala also runs on the JVM, it is then possible to run the code on Android devices and to compare it with Java. After analysing the existing literature about Scala-Java comparisons, and after finding what constitutes a good benchmark for our subject of study, we found and adapted two benchmarks for both Scala and Java and ran them on an Android device.

We measured the power consumption and the execution time of the benchmarks, as well as the memory usage and the size of the application.

The results show that Scala performs faster or slower than Java, depending on the task, and on average it consumes more power than Java on the Android device, while using less memory. The size of the application written in Scala is also bigger in size due to the presence of the Scala libraries.

KEYWORDS: Android, Java, Scala, Benchmark, Performance, Comparison

1 Introduction
In the last years, several new languages have come out that use the Java Virtual Machine (JVM) as a runtime environment. Some examples of this trend are Scala and Clojure. In this paper we choose to study Scala because it’s very similar to Java in many aspects, unlike Clojure that is Lisp-based.

Scala is a multi-paradigm programming language that integrates both the functional and Java’s object-oriented programming paradigms. Like Java, Scala is statically typed, but, while staying fully interoperable with Java, it introduces many new functionalities from the functional programming which make it appealing to the more modern programmers.

With the ever increasing number of mobile devices used worldwide and their evolution, it’s only natural to want to use these modern languages when developing mobile applications.

Android is currently the most diffused mobile operating system in the world, reaching over 70% of the global market share [5], and its application distribution platform (Google Play) contains more than 600000 applications [6].

Compared to traditional computers, mobile devices have reduced processing power, smaller memory, and a limited energy supply. It’s very important to keep these limitations in mind when creating applications for these devices, but the upper-level code written by the developers is responsible only in part for the performance of their applications, other factors include the libraries used and the underlying machine or virtual machine.

Android, to deal with the constraints of the mobile devices, uses the Dalvik virtual machine, which is designed to be suitable for devices with limited memory and processor speed, and can execute JVM-compatible code after it has been converted into Dalvik executable files. This also gives the possibility to write applications for the Android devices using the new languages like Scala, but leaves uncertainty on their performance and energy-efficiency in this environment since it was designed to run Java applications.

This paper analyses the existing literature containing comparisons of Java and Scala, and continues by comparing the performance of Java and Scala on an Android device. The metrics for the comparison are five: the execution time, the startup time, the power consumption, the memory usage, and the application size.

2 Background
2.1 Scala
Scala is a relatively new programming language that was created as an alternative to commonly used statically typed languages like C# or Java, to remedy to the lack of support for component abstraction and composition, and the lack of scalability of the tools provided by the languages.

What it means is that it concentrates on creating deep abstraction for its components, and on giving them broader use in different scales.

It tries to do this by unifying the classical object-oriented paradigm of programming languages, with the more recent functional programming paradigm.

It was designed to integrate and work well with both Java and C#: it adopts data types, basic operators, and control structures; but it does not just add new functions to those languages: a lot of the mechanisms already present have been rewritten or removed completely, to give Scala bigger uniformity of concepts, therefore making it a very different language.[10]
2.2 Differences between Scala and Java

There are several differences between Scala and Java, most notably syntax, handling primitive types, iteration over data, and the functional programming component of Scala. One of the most evident differences is Scala’s different syntax and its syntactic flexibility. In fact, Scala doesn’t require the use of a semicolon at the end of a statement, or the use of () to use functions that take no argument, and any method can be used as an infix operator without the use of the dot to connect it.[11]

Unlike Java, Scala makes no distinction between primitive types (int, boolean) and reference types (classes, e.g. Integer, String): in Scala everything inherits from the class Any, be it primitive types (AnyVal) or reference types (AnyRef). Furthermore, Scala classes do not have static members, but singletons can be created to take that role. For example, every Java class is seen as composed by two entities: a class containing the dynamic members and a singleton containing the static ones.[11]

Another difference is the use of for-comprehensions instead of the classical for-loops. These allow for easy iteration over data sets and directly allow for the creation of a new collection of the same type of the one over which we are iterating.[11]

However, the biggest difference between Scala and Java is the functional programming component of the language. It is what makes Scala a more modern language. The functional programming style makes no distinction between statements and expressions: everything is an expression; this means that all the statements return a value and there is no need to use the "return" statement.

Scala infers the type of the variables without it being explicitly defined; and passing a function as a parameter can be done without declaring and naming said function, but directly in-line.

Moreover, Scala uses tail call optimization to avoid stack overflow problems while using recursion; it offers traits to extend classes as opposed to Java’s interfaces; and it comes with an expressive static type system.[11]

2.3 Existing comparisons between Scala and Java on traditional PCs

Comparing Scala to other programming languages hasn’t been subject of a big amount of study in recent years. Few of the papers available have successfully compared and analyzed some of the core features of Scala like effort, code structure and complexity, usage, and performance.

Based on the studies Scala code is less complex and more compact than Java, but it also requires more effort from the programmer.

In [12] the authors studied thirteen programmers working on three projects, in both Java and Scala. These programmers were master’s students close to graduation and had the basics of software engineering. The study produced 39 Java and 39 Scala programs and it also obtained data from an industry software engineer who worked on the same Scala project. Effort, code size, language, usage, performance and programmer satisfaction were analyzed and the study confirmed that the Scala code is more compact than the Java code. In fact, in the projects Scala had median of 533 lines of code and Java had median of 547. The claims of Scala having lower programming and lower debugging effort were refuted, though.

During the study the programmers received the same training with both Java and Scala. This training covered parallel and functional programming, the use of Eclipse and NetBeans development environments, debugging, and how to analyze the performance of parallel programs. Every programmer implemented both Dining philosophers problem and mergesort with both languages and, after the initial training, the programmers were able to create more complex and efficient parallel applications.

Based on the study, Scala requires more effort than Java: the median hours spent on making the same features with Scala was 56 hours and with Java 43 hours. In addition to the language used, the programming skills of each individual programmer influenced the effort needed to finish the projects.

In addition, the study showed that for median values, both for programs with parallelism and for sequential programs, Scala had better performance. However, for programs with multiple threads, the best times came from the Java programs. The study suggests though that the best performance in the projects was achieved combining functional and imperative programming style.

In [8], the authors run a comparison between several languages. These languages are C++, Java, Go, and Scala. This study compares several characteristics of the languages: the code complexity, the compilers, the compilation time, the size of the binaries, the run-times, and the memory footprint. The benchmarks show large differences in all the dimensions examined.

A well defined algorithm employing higher-level data structures, iterations over collection types, object oriented features and memory allocation patterns was implemented in the four languages. All implementations use default data structures, type systems, memory allocation schemes and iteration constructs in each language. No language-specific adaptations nor optimizations were implemented either. The aspects of multi-threading or higher level type mechanisms were not observed, since the variation in these is large between the languages.

The study showed that Scala’s features allows for the best optimization of code complexity but the effects of Java’s garbage collection are complicated and hard to tune, and since Scala runs on the JVM, it had the same issues.

In [13] the authors compare the memory behaviour of programs written in Java and Scala. They do so by running the DaCapo benchmark suite for Java, and a Scala benchmark suite based on the DaCapo benchmarks developed by some of the authors themselves. These benchmarks are mostly accepted as good benchmarks by the scientific community, in the sense that they successfully represent the typical programs for the languages.

By using profilers on the JVM they analysed and compared the workload on the garbage collector, the object churn, how much the size of the objects matters for the language, how much the immutability of fields impacts on the language’s performance, the usefulness of zeroing fields
during allocation, how much the use of Scala impacts on common implementations of synchronization primitives optimized for Java, and how much the identity hash-codes are used by the languages.

The results obtained in this study show that the memory behaviour of the Scala programs is in some way similar to the typical memory behaviour of programs written in other functional programming languages. To expand, the objects in the Scala program "are more likely to be small, immutable, and to die young"[13] than the objects of a typical Java program. This difference in memory behaviour makes it more difficult for Scala to perform as well as Java in the JVM, even with the optimizations brought by the Scala compiler.

In the Computer Language Benchmarks Game[2] we can find a collection of benchmarks and a comparison of several languages, including Java and Scala. The comparison is done by running the same algorithms on the same machine with the same inputs and measuring the execution time, the memory usage, the CPU load and the size of the code. The algorithms are written, optimized, and validated by the website's community so that the quality of the code is good and the results are more indicative of the performance of the language rather than the coding ability of the programmer.

These benchmarks, on the other hand, represent only a small part of the possible software that can be written in those languages, and don’t consider or test the languages for their purpose; for example, Scala is not tested for its scalability. From the results recorded on the website, we can see that on desktop computers Scala performs slightly worse than Java when it comes to execution time and memory usage, but results in a smaller size of the code.

### 2.4 Benchmarks and their characteristics

To analyse how Scala fares against Java we need to define which characteristics constitute a good benchmark. Some of the characteristics that we found relevant are fairness, verifiability, repeatability, and relevance.

The fairness: the benchmark can’t favour specific characteristics of the tested systems, it has to test aspects of the systems in a way that doesn’t favour any of them. In other words, it should not test the systems based on cutting edge features that are too new to be optimized in all systems.

The repeatability: the benchmarks should give similar results if run on same or similar setups. Because of this, when comparing Scala and Java, it is important to define the algorithms and benchmarking suite accordingly.

The verifiability: the benchmark has been studied in depth and the results are clearly displayed in a proper manner and certified by experts.

The relevance: this is the most important characteristic because it defines how useful the benchmark actually is. If a benchmark is not relevant, running it will not yield any useful results; on the other hand, a relevant benchmark can help improve a system drastically or, like in our case, help studying it more in depth.[1]

To be able to proceed and test Scala and Java on mobile platforms, we need to understand the problem we are trying to study and find benchmarks that are relevant to it.

The difference between mobile devices and desktop solutions are relevant. The size limitations of the mobile devices force the constructors to compromise between the size of the chips and their characteristics. The presence of a battery also influences how much power these chips can use and their performance.

These limitation make of optimization an important factor in the development of a mobile application, and define on which areas our benchmarks should be concentrating: the execution time, the power consumption, the memory usage, and the size of the application.

### 3 Experiments

#### 3.1 Experimental Setup

The code was installed and run on a HTC Desire HD. This device runs Android 2.3.5 and contains a CPU with a frequency of 1GHz.

To test the two languages we adapted two benchmarks from the ones available in the Computer Language Benchmark Game[2]. The first benchmark is an algorithm used to calculate the n-body problem; the n-body problem is the problem of simulating and predicting the movement of massive bodies that interact with each other through gravity; in our case we simulate the motion of the part of our Solar system. The code in the second benchmark allocates, traverses, and deallocates several binary trees and tries to test the memory behaviour of the languages.

These benchmarks were chosen for their different natures, the first one concentrates on calculations and processing power, the other on memory behaviour and garbage collection. Together they should give us a good overview of the languages in our metrics: the execution time, the power consumption, the memory usage, and the application size.

The application developed runs the benchmarks 10 times and counts measures the time used to execute each benchmark run. It then shows the results on the screen and stops.

Several tools were used to take the other measurements on the devices, including PowerTutor and Proguard.

PowerTutor is an application for Android that measures the power consumption on different components of the phone, as well as the total memory usage of the phone. It also allows to measure the power consumption of the CPU contributed by a single application and will be the main source of data for our measurements.

Proguard is a tool that "detects and removes unused classes, fields, methods, and attributes. It optimizes bytecode and removes unused instructions. It renames the remaining classes, fields, and methods using short meaningless names."[7]

It’s a necessary tool for the development with Scala on Android because including the whole Scala library would make the startup time of the application much longer and the size of the application much bigger.[9]
4 Results

4.1 Application size and startup time

<table>
<thead>
<tr>
<th>Language</th>
<th>App size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>180 KB</td>
</tr>
<tr>
<td>Java + Proguard</td>
<td>180 KB</td>
</tr>
<tr>
<td>Scala</td>
<td>4041 KB</td>
</tr>
<tr>
<td>Scala + Proguard</td>
<td>568 KB</td>
</tr>
</tbody>
</table>

Table 1: Application size

Table 1 shows that the use of Proguard does not affect the size of the Java application while it greatly decreases the size of the Scala application by removing the unused parts of the Scala library. On the other hand, compared to the Java application, the size of the Scala application is 3 times bigger when using Proguard and 22 times bigger without it. The code itself is shorter, but the addition of the Scala library is very significant for small and medium sized applications.

In our tests, the time needed to install the applications increased significantly for the Scala application without Proguard while being very quick for the other situations. On the other hand, the time required to start the application did not seem to be affected by the presence of the library, in any of the cases.

4.2 Execution time

The applications we developed keep track of the execution times of each benchmark run and show the results to the screen once the benchmark finishes executing.

The desktop versions of the algorithms showed that the execution times of the benchmarks in the two languages are very close. [4][3]

However, on the Android device the execution times of the N-Body benchmark are very different (Figure 1) and the Java version is significantly faster than the Scala version.

For the Binary Trees benchmark, instead, the measurements (Figure 2) show that, unlike with the N-Body benchmark, the execution time of the Scala algorithm was slightly faster than the Java counterpart.

We can also notice a great variance in the execution times, we suspect this is due to the behaviour of the garbage collector.

4.3 Power consumption

The power consumption of the benchmarks was measured using Power Tutor, and it excludes the consumption coming from the LCD screen and the network due to the fact that the algorithms don’t specifically use either.

In the power measurements of the Java version of the N-Body benchmark (Figure 3), we can see that the consumption has an average of 1017mW, and a total execution time for the 10 runs of 49s.

The power consumption of the Scala version of the benchmark (Figure 4) instead has an average power consumption of 1004mW, and a total execution time of 66.5s.

The comparison of the results for the N-Body benchmark shows that the Java and the Scala versions of the algorithm have a very similar behaviour for the power consumption.

Furthermore, the Scala benchmark seems to have a slightly lower power consumption, but due to the much longer execution time the total energy consumption for a single run becomes higher than the Java version (Figure 7).
For the Binary Trees benchmark we found that the Java version of the algorithm (Figure 5) had an average power consumption of 785mW. The total execution time for the runs of this benchmark is 118s. The Scala version of this benchmark (Figure 6) instead had an average consumption of 889mW, with a total execution time of 113.3s. Comparing the results for the two languages we can see that the Scala version of the Binary Trees benchmark has a higher power consumption.

However, even though the execution time of the Scala version of the benchmark is lower than the Java version, the total energy consumption (Figure 7) of each benchmark run is much bigger, due to the higher average power consumption.

### 4.4 Memory usage

Using PowerTutor we can also find what is the status of the memory of the system. Unfortunately, PowerTutor gives us memory readings only every 10 seconds, which is a very low frequency for our experiment. For our measurements we kept track of the amount used memory on the system during the execution of both the N-Body and the Binary Trees benchmark.

In the chart (Figure 8), the values between the 3rd and the 7th reading correspond to the executions of the N-Body benchmark, while the values between the 8th and the 19th reading correspond to the Binary Trees benchmark.

From our data we can see that the system has much more free memory while running the Scala benchmarks, compared to the execution of the Java ones. One reason for this could be the behaviour of the garbage collector with the different memory behaviours of the two languages: the objects in the Scala program are smaller and have shorter lives than the ones in the Java program[13], so they can be cleared by the garbage collector in a quicker way.

On the other hand, this result is not necessarily indicative of the characteristics of the languages due to the fact that the reading represents the memory status of the whole system, not just of our application, and it is affected by all the other applications running on the phone at the same time.

However, we can also notice a similarity between the behaviours of the Java and Scala applications during the execution of the benchmarks, with a low memory usage during the N-Body benchmark and a somehow constant usage during the Binary Trees benchmark.
5 Summary and Conclusions

We wanted to see how using a modern language based on the Java Virtual Machine on an Android device would affect the performance of the application and the device compared to Java. We picked Scala as a comparison because of its similarities and interoperability with Java.

We went through the literature and chose what would be interesting to measure with a benchmark for a mobile device. We then looked at the available benchmarks and found two good benchmarks already implemented in the two languages, with very similar performances on the desktop computers, and that cover well our metrics.

We adapted and integrated the benchmarks in an Android application and, with the help of PowerTutor and Proguard, measured their behaviour and performance.

Running the benchmarks on the devices we found that:

- The size of the Scala application is significantly greater than the Java application due to the presence of the Scala library, even when using Proguard correctly, but while slowing down the installation time, it does not affect the startup time of the application.
- Scala, for the same tasks, consumes more energy than Java. It may take a shorter or longer time to conclude the task, but the energy consumption seems to always favour Java.
- The Scala application uses less memory than the Java application during the execution of the benchmarks. These result may not be significant due to the imprecision of the tools used, but still showed a similarity in the memory behaviour between the two languages.

In conclusion, we feel that Scala will not play a major role in the mobile application development in the future, due to the importance of keeping a low energy consumption on the device. The strong point of the Scala language is the way its components scale, which is not of major importance in mobile devices where applications tend to not scale at all and stay small.

References


