UNIVERSITY OF TEXAS

ORC

Running ORC on Android

João Barbosa [jpbarb@student.dei.uc.pt]
Ricardo Bernardino [rjrocha@student.dei.uc.pt]
8/13/2012

This report describes an Orc version for the android mobile platforms. While in a desktop we don't need to be too much worried about memory and CPU constraints, in an android system we have very strict limits, otherwise the application cannot run properly or it may even crash. Since Orc is implemented in Scala, which is a very extensive language, we need to remove everything that is not essential, leaving only what is strictly necessary in order to leave the final application as light as possible. As it is described on this report, these limitations brought more responsibilities for the programmer since they will always have to declare explicitly which functions and libraries they need to use.

INDEX

INTRODUCTION

In 2011 the number of smartphones was already impressive, reaching some amazing 490 million units sold[1]. But the growth doesn't seem to be stabilizing. On the contrary, at the end of 2015, it is expected that this number will be increased to 1 billion [2]. These values are already extraordinary, but if we include the growth of tablets, we should expect 10 billion devices against 7.3 billion people in the world [1].

Due to gaming demands, multitasking experience and energy consumption concerns, the growth of multi-core architectures in mobile platforms has also been increasing a lot faster than the transition from single core to multicore in the desktop environments. Right now we have plenty of systems already with the new powerful Tegra 3 processor. [3][4].

Summing to the previous values, and the fact that at the end of 2012 almost 60% will be running Android OS[5], we have the perfect scenario to apply a programming language that runs on these systems and which can take advantage of the multi-core environment. However, the complexity of such a language may bring some performance challenges that we have to deal with, in order to overcome the tradeoff of memory restrictions.

The idea of this project came precisely by the team who created the language [6], Orc. This is a new language mainly oriented to distributed and concurrent programming, designed to deal with problems such as acquiring data from remote services, performing calculations with those data, and invoke perhaps other services with the results [7]. Those services are called sites and their integration orchestration. Behind Orc there is a theory which says that "smooth orchestration requires only four simple combinators: parallel computation, sequencing, selective pruning, and termination detection. Together, these combinators prove powerful enough to express typical distributed communication patterns"[8]. At the beginning, this project was not feasible since the mobile platforms couldn't handle parallel computation properly due to their hardware limitations. However, and as we said above, right now we have powerful and economic multi-core processors on mobile platforms allowing us to port the Orc language into to the Android systems.

This report will start with a quick overview over the all process so that we can easily understand where the main difficulties were and how they can be solved.

THE APP

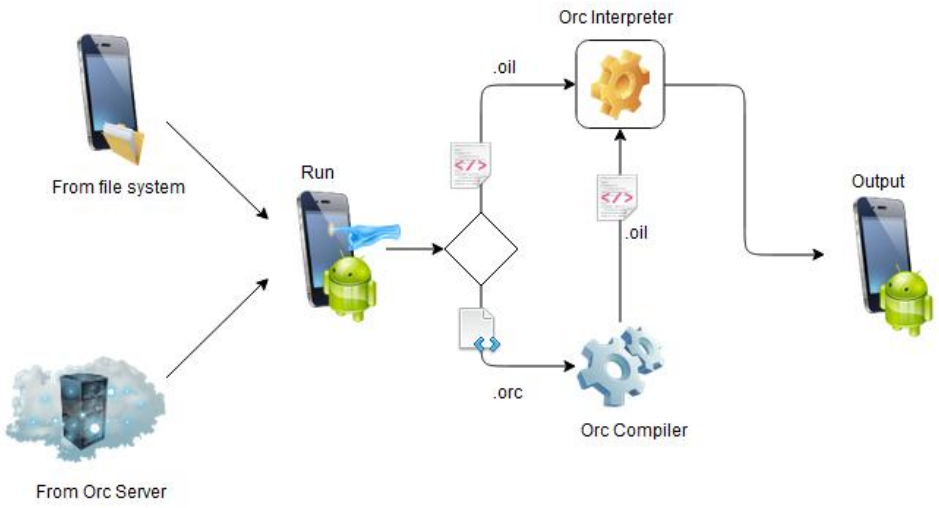


Figure 1- Project architecture overview

The Orc programming language is a little different from the common ones that we are used to. For example, in Java, the process to run is very simple. The .java files are compiled into bytecode, to run on the JVM. However, in the Orc language, its bytecode works like a simple XML file called .oil file which can be easily interpreted. This file can be obtained by compiling the .orc files.

This way, our Android application can get both file's types from the android system or just from examples hosted at the Orc servers. Depending on the file chosen, our application will call different components. In case of an .oil file we can just interpret it immediately. Though, if we selected an .orc file, before interpreting it, the application will need to translate the .orc file to .oil format with the Orc compiler [Figure 1].

After this process, the Android application will launch the Orc program, showing its results on the output console. Of course that if we have a prompt call we can simply make an input.

ORC INTO ANDROID

As we said before, Orc is mainly written in Scala language which means that first we needed to run Scala on Android.

RUN SCALA ON ANDROID

THE PROBLEM

When we program in Scala we can just use scalac to compile the .scala files to JVM .class files without any problems. Though, in the Android systems this procedure is quite different. In fact, Android doesn't run Java byte code, but Dalvik byte code – because of legal issues with patents [9]. So basically we need to convert the .class files into .dex so that they can run on the phone – this is called the dexing process.

A quick solution would be just trying to *dex* all the .class files with the scala-library.jar. However there are two big problems. Firstly, the Android applications are self-contained which means that they can't share libraries. Second, the scala library has a huge runtime, making a simple *HelloWorld* apk (Android application package file) wih 800Kb extremely slow to compile. Even if it succeeds in compiling, it will be extremely slow to run.

BUILD USING SCALA, ADT AND PROGUARD

To solve the problems mentioned above, we will have to use ProGuard [10]. This plug-in is very important since it will cut everything that we don't need in the Scala library, reducing the final size of our APK. Proguard is also used to optimize and obfuscate code, but in our case we have disable this option in the configuration file.

To enable ProGuard on Android, we must change the project.properties file on the android project directory by adding the following line:

```
proguard.config=proguard.cfg
```

The *proguard.cfg* will be the ProGuard configuration file. The Android SDK already has a few sample configuration files, including some for projects written in the Scala language. These files can be found in <sdk path>/tools/proguard/examples directory. The first configuration file we used was automatically generated by the AndroidProguardScala plug-in for Eclipse, but after spending some time we realized this plug-in was not including any Orc classes to the generated jar, only the essential Scala library classes. To solve this problem we disabled the previous plug-in, copied the generated configuration file and added some lines specifying which Orc classes we would like to keep in the generated jar. In order to run ProGuard, we have to click on the Android project folder, click Android Tools, and finally click "Export Unsigned Application Package". After some time, there might be an Exception but it has to do with the generation of the Application Package and not ProGuard. Now in the main project folder there will be two additional folders: proguard and proguard-cache. The former is not relevant for our project [11]. The latter is the output folder for the generated jar, but it can be altered in the ProGuard configuration file on the – *outjar* statement. Since we need the .jar for the Android program to operate correctly, we must copy or move the .jar file to the *libs* directory.

SETTING THE ENVIRONMENT

But, before start coding the application we must set up our environment. Below we can see the steps:

1. Install Eclipse 3.7.2
2. Install Android SDT plug-in by pointing Eclipse to https://dl-ssl.google.com/android/eclipse/ (API 16)
3. Install Scala IDE version 2.9.2 by pointing Eclipse to http://download.scala-ide.org/releases-29/stable/site

CREATING A PROJECT

Having our environment running properly we are now able to start a project. It's very simple, thought, we need to pay attention to some details.

1. First we create an Android project as usual.
2. Then, we right-click on the project, "Configure", "Add Scala nature"
3. Make sure that the name of the Scala file is the exactly the same as the one of the main activity contained in it (eg.: MainActivity.java will become MainActivity.scala)
4. Create a ProGuard configuration file, and alter the project.properties file as already stated above.

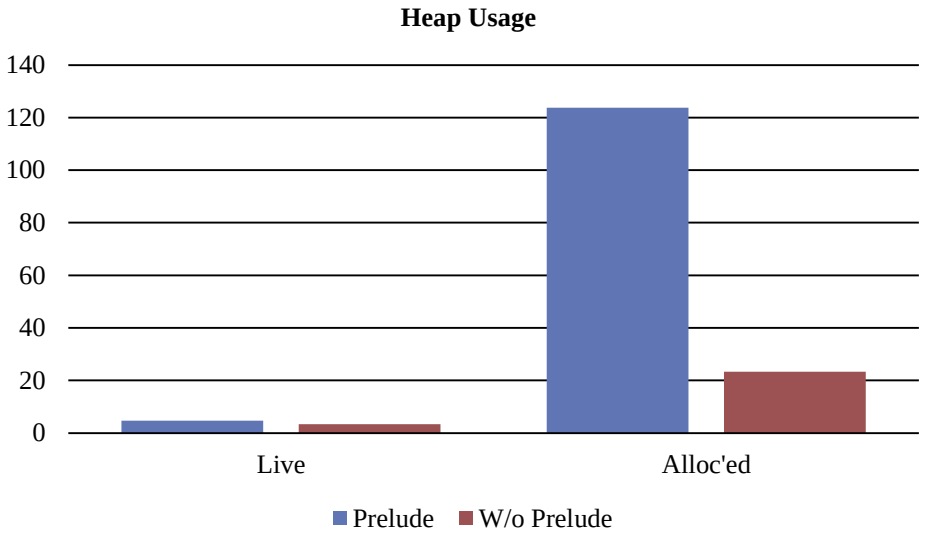
RUN ORC ON ANDROID

After following the steps above, we are now ready to run Orc on Android. In this project, we run Orc files on the RunActivity in Android, and the code is commented for easier understanding. There is also the possibility of running pre-compiled Orc files (*.oil*) in order to reduce the time taken to run the desired program. The files to be ran must either be available in the Try Orc website, or in the SD card of the device. In the code provided, there is also a commented line explaining how to run .orc or .oil files stored in the project folder, more specifically in the res/raw folder. This might be useful while developing since it is easier to alter the file right on Eclipse.

It is very important to note that we may have to add some Orc classes to the ProGuard configuration file, since it does not detect them by its own, as the Orc program is not part of the Android Project in Eclipse. Also, the code provided was developed for Android API level 16, so there may be some problems with the UI if the API level is changed.

PRELUDE PROBLEM

Having the application operational in theory, we tried to start it. Though, we noticed that it just could not run properly displaying constant messages related with the GC (Garbage Collector) about memory reaching its limit and being freed. After that, we made a quick debug and we found that the application was being stuck during the *prelude phase*. This way, we opt to benchmark it, using de java HPROF tool to analyze what was going on with the heap of our application [12]. So, we built a jar with a desktop version of the Orc Android application and ran the *java -agentlib:hprof=heap=sites –jar application.jar* getting the results into a graphic [Graphic 1]. The application consisted in compiling and running an *.orc* file of a simple sum, outputting the result into a variable x (eg.: 1+2 > x > x). This profile test was done on a machine with 4GB DDR3 RAM, Intel Core2 Duo T9400 @2.53Ghz running MS Windows 7 32-bit SP1.



Graphic 1 - Android application profiling results

As you may see, without the *prelude* our application consumes much less memory, proving our suspects with problems at the Prelude. This behavior is easily justified by the fact that during the prelude, the compiler adds all the libraries, increasing the memory heap usage. After that, it simply prunes those which are not useful. This way, we get a little live memory usage (it's a very short application) but a massive alloc'ed memory usage.

Besides creating a specific module for the Android compile process –which is extremely time consuming - we don't have many more solutions. So we came up with the solution to simply shut down the prelude. However, it brings some implications for the programmers. For instance, every simple operation just like a sum will need to be imported manually. In the sum case: *import site (+) = "orc.lib.math.add"*
To solve this problem we could take two ways: build a package with the most basic operations; or make a static code analysis and then import the operations found.

ANDROID

ANDROID COMPATIBILITY (UI FRAGMENTS)

Since Android 3.0 (API level 11), Google decided to introduce fragments, mainly to support more dynamic UI designs on bigger screens, like tablets. The reason for this new approach is due to tablet's screen size [13]. Since it is much larger than those from smartphones, usually the programmers want to combine and interchange more UI components. With Fragments we are now allowed to manage complex changes to the view hierarchy and change activity's appearance even during the runtime.

According to Android Developers, "you can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running" [13]. In fact, a fragment is like a sub-activity with its own life cycle, making it possible to combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. These are the main reasons why we choose to use fragments, despite the problems for older versions than

3.0. For any further work on this project, it will be easier to reuse code or just add new UI with new functionalities.

STRUCTURE

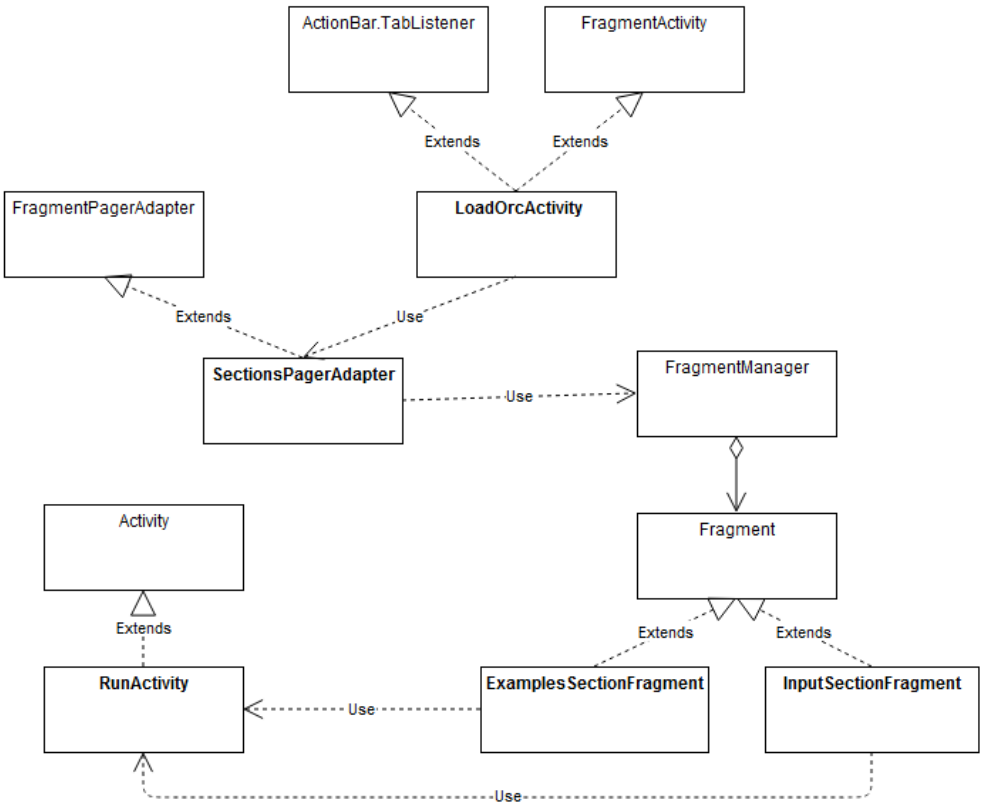


Figure 2 - Simple class diagram of the android application (bold classes - developed classes)

As we may see on Figure 2, this application has a very simple structure. First, the *LoadOrcActivity* is responsible for starting the whole process. It initializes both fragments through the *SctionsPagerAdapter* and consequently the *FragmentManager* [Figure 3]. These fragments are responsible for letting the user to browser the files to run on the application. In the *ExamplesSectionFragmen*t we will parse the HTML from the http://orc.csres.utexas.edu/tryorc/ on real time, allowing the user to navigate through the examples available on the Orc Server. The *InputSectionFragment* works in the same way as the previous fragment providing an Android file explorer and filtering only the *.orc* and *.oil* files.

After selecting the file, the fragment in use will send the path of the file to the *RunActivity* [Figure 4] where the Orc application will actually run. At this time, the user is allowed to answer *prompt* calls and check the output.

We advise you, that for more information about the implementation, you should check the source code where everything is commented in great detail.

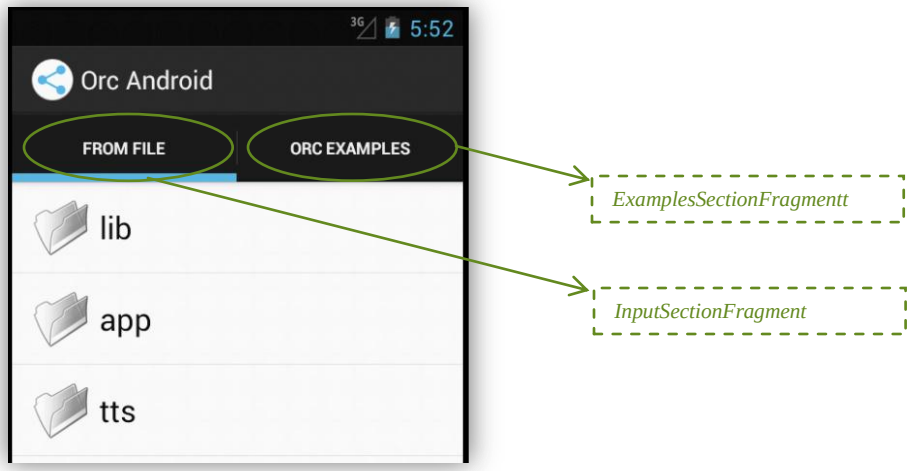


Figure 3 - LoadOrcActivity user interface.

Figure 4 - RunActivity user interface

FUTURE WORK

MORE SITES

As we saw on the beginning of this report, the smartphones world is growing at an impressive rate. It's important to develop interesting applications in order to make people use them. Therefore, there is a strong potential in developing Android specific sites to take advantage of smartphone's capabilities. A very simple example would be a GPS which collects its position at a certain rate (seconds, minutes,…) and possibility that would also collect data from the smartphones' sensory mechanisms (e.g.: accelerometer) in order to build a dead reckoning application.

MAKE IT EASIER TO PROGRAM

As we said before, one of the problems that we faced was due to the limitations of the smartphones' hardware, making it impossible to use the prelude. Consequently, this situation obligates the programmer to include every function manually. We came up with two possible ideas.

BASIC PACKAGE

The easiest would be creating a package which would include the most basic operations (like sum for example), freeing the programmer from the need to include every operation.

REBUILD PRELUDE PROCESS

This solution is much more complex and time consuming. Since the prelude includes every operation that Orc supports and then prunes what isn't necessary, it should be possible to rebuild this process in order to make it simpler and lighter, so that it could be easily handled by the smartphone's hardware. However, we are not sure if even with this solution, a simple smartphone would be able to run the prelude. This way, a deeper study has to be done before developing this solution.

ACKNOWLEDGMENTS

We would like to thank the members of the Orc research group - David Kitchin, Arthur Peters, Prof. Jayadev Misra, John Thywissen - for their crucial contributions and guidance throughout this project. We also would like to thank Prof. Keshav Pingali, Cayetana Garcia and Prof. Luís Silva for this internship program and for their support during our stay at the University of Texas at Austin.

CONCLUSION

Ever since the beginning of this project, we knew, due to time constraints, that we wouldn't have enough time to solve the faced problems properly. However, we think that our goals for this Orc side project were accomplished successfully.

Programming for the first time in Scala on an Android platform takes its time to get used to some constraints, just like the huge size of the Scala library. Such a small OS can't handle the complexity, and ultimately size, of such of libraries. Consequently, we took a lot of time to get into the ProGuard way of working. Since that, we only had to deal with performance issues regarding the compile process of *.orc* files into *.oil* files. The perfect solution would be to reformulate the compile process, in particular the prelude phase. However, as we said, it was not possible since it would be a very time consuming approach. We ended up overcoming this problem by disabling the prelude phase, obligating the programmer to import every operation needed.

To conclude and in view of the above, we built an application which is capable of running more than just a simple HelloWorld in the Orc language with a strong Android base, allowing others to improve this project faster and easily.

REFERENCES

1. The future of smartphone growth: By the numbers, (2012), Retrieved August 10, 2012, from http://theweek.com/article/index/224535/the-future-of-smartphone-growth-by-the-numbers
2. Booming Growth Of Smartphones And Apps, (2012), Retrieved August 10, 2012, from http://www.viralblog.com/mobile-and-apps/booming-growth-of-smartphones-and-apps/
3. Behind the Birth of M^3, (2012), Retrieved August 10, 2012, from http://www.fixstars.com/en/m-cubed/history/
4. NVIDIA - The Benefits of Quad Core CPUs in Mobile Devices, 2011, Retrieved August 10, 2012 from http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911a.pdf
5. Android growth to end in 2012 as Microsoft begins to steal Google and Apple's thunder, (2012), Retrieved August 10, 2012, from http://news.yahoo.com/android-growth-end-2012-microsoft-begins-steal-google-124524707.html
6. Orc Wiki - CS395T Project Ideas , 2009, Retrieved August 10, 2012 from http://orc.csres.utexas.edu/wiki/Wiki.jsp?page=CS395T%20Project%20Ideas
7. The University of Texas at Austin - Orc User Guide v2.0.2, 2011, Retrieved August 10, 2012 from http://orc.csres.utexas.edu/documentation/html/userguide/userguide.html
8. Orc Language Project – Why Orc? , 2012, Retrieved August 10, 2012 from http://orc.csres.utexas.edu/introduction.shtml
9. Retrieved August 10, 2012 from http://chneukirchen.org/blog/archive/2009/04/programming-for-android-with-scala.html
10. ProGuard version 4.8, (2012), Retrieved August 10, 2012 from http://proguard.sourceforge.net/
11. Android Developers - ProGuard, Retrieved August 10, 2012 from https://developer.android.com/tools/help/proguard.html
12. Oracle – HPROF: A Heap/CPU Profiling Tool in J2SE 5.0 , 2004, Retrieved August 10, 2012 from http://java.sun.com/developer/technicalArticles/Programming/HPROF.html
13. Android Developers - Fragments, Retrieved August 10, 2012 from http://developer.android.com/guide/components/fragments.html