Diving into Dalvik

80-minute tech talk

Andrea Janes, Dainius Jocas, Alberto Sillitti, Giancarlo Succi

Free University of Bozen/Bolzano, Center for Applied Software Engineering

{andrea.janes, alberto.sillitti, giancarlo.succi}@unibz.it, dainius.jocas@stud-inf.unibz.it

Abstract

"Dalvik" is the name of the virtual machine that runs programs on Android devices. The architecture of Dalvik is designed to match the energy consumption and performance requirements of devices with limited energy and hardware capabilities. Therefore, the study of the architectural decisions behind Dalvik is relevant for researchers and practitioners interested in the capabilities and limitations of the machine executing mobile programs and the specific programming styles required for battery-powered devices.

To understand how Dalvik works, we compare it with a Java Virtual Machine running on desktop computers using the following criteria: the objectives, the compilation process, and the format of executables of Dalvik.

This tutorial points out strengths of Dalvik, and how these strengths are achieved.

Researchers and practitioners involved in mobile programming or Green Computing will understand how programs are executed and which strategies Dalvik adopts to retain battery power. This knowledge can be used to optimize the execution of programs or to adopt similar strategies in similar projects.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks

General Terms Design, Performance

Keywords Android, Dalvik

1. Advertising biography

Andrea Janes is an Assistant Professor at the Free University of Bolzano-Bozen (Italy) where he teaches Architectures of Digital Systems. He has experience in mobile development in Java and C#.

Dainius Jocas is a master student in Computer Science at the Free University of Bolzano-Bozen (Italy) and passionate about Android development in Java.

Alberto Sillitti, Ph.D., is an Associate Professor at the Free University of Bolzano-Bozen (Italy) where he teaches Internet and Mobile Services. He has experience in Android development in Java and C/C++.

Giancarlo Succi, Ph.D., is Professor at the Free University of Bolzano-Bozen (Italy) where he teaches the Advanced Internet Technologies. He has experience in Android development in Java and C/C++.

SPLASH '13 October 26-31, Indianapolis.

2. Objectives of the tech talk

The talk has two objectives:

1. illustrate the architectural and implementational differences of the Dalvik Virtual Machine (DVM) compared to a standard Java Virtual Machine (JVM), and
2. inspire researchers and practitioners to consider these differences in their own projects and to conduct research in the area of mobile software development or Mobile Green Computing.

3. Topics

The talk will describe:

1. the objectives of the DVM,
2. how these objectives are reflected in the architecture of the DVM and the Android software stack,
3. the patterns used in the implementation of the DVM,
4. how a program is executed explaining hooks, extension points, inversion of control, and dependency injection, and
5. how a program can be written in languages different than Java such as C, C++, and Scala.

There are only few resources available about each single topic, and this talk will provide an in-depth view of the internals of Dalvik. We briefly describe the first two topics below.

3.1 The objectives of Dalvik

Smartphones are—compared to the first computer of one of the presenters, an IBM PS/2 with 1 MB of RAM and 20 GB of space on the hard disk drive—powerful computers build into the case of a mobile phone. Smartphones are one of the consequences of the progress in the miniaturization of computer technology.

The need to use a mobile energy source, i.e., a battery, causes a number of additional implications, e.g., the requirement to use passive cooling instead of active cooling to reduce the size of the device as well as to save battery energy.

A typical smartphone [2] has an ARM CPU running at 500 MHz clock speed, 128 MB of RAM, and external flash storage (usually NAND) that cannot be used as swap memory, and a 1500 mAh battery. If, to run a Java program on the smartphone, we would use the same Java Virtual Machine as on a desktop computer, the performance would be unacceptable and the battery would be empty in a matter of minutes.

The DVM (and the software stack of which Dalvik is part of, Android) are optimized to execute Java programs on such hard-

ware, with the given limitations of CPU speed, memory, and battery power.

3.2 The architecture of Dalvik

In this part we discuss the parts of a virtual machine and how the DVM differs from a typical JVM.

A virtual machine should emulate the operations carried out by a physical computer and thus ideally implement the following concepts [6, 8]:

- compile the source language into the language that the virtual machine can execute,
- support data structures to store instructions and operands,
- implement a call stack for function call operands,
- implement an instruction pointer, pointing to the next instruction to execute, and
- implement a virtual CPU, i.e., an instruction dispatcher that
  - fetches the next instruction (addressed by the instruction pointer),
  - decodes the operands, and
  - executes the instructions.

A stack-based virtual machine stores the operands using a stack data structure. Such a machine executes operations reading data from the stack using a "pop" instruction, and processing and storing them back to the stack using a "push" instruction.

In a stack based virtual machine, adding two numbers occurs as in fig. 1: the stack contains 4 operands: 20, 7, 13, and 45. To add the first two, we pop them from the stack, add them and push the result back to the stack.

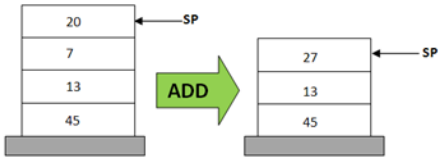

Figure 1. Adding two numbers in a stack based virtual machine [6]

The instructions that have to be carried out are shown in the listing below.

```
1 POP 20
2 POP 7
3 ADD 20, 7, result
4 PUSH result
```

A register-based virtual machine operates without using a stack but using a large number of (in this case virtual) registers. Depending on the instruction, 16, 256, or 64K registers can be accessed [4]. This reduces the number of instructions that have be fetched and dispatched. It permits to design instructions with a higher semantic density than in a stack-based virtual machine, and reduces memory access.

Since register-based virtual machines understand instructions with a higher semantic density, they require less instructions to be executed, and the executable file (i.e., the instructions to execute) is smaller than for stack-based virtual machines.

To execute code written for a stack-based machine such as the JVM on a register-based virtual machine, we need to translate the code into instructions supported by the register-based virtual machine. This additional compilation process is further explained in section 3.3 and transforms Java byte code into Dalvik byte code.

The DVM is part of the Android software stack (see fig. 2). The Linux kernel forms the heart of the Android software stack, on which the libraries and the Android runtime depend. The libraries (blue) are mostly written in C, the Android Runtime containing Dalvik is responsible to execute Dalvik byte code. The Application Framework provides system-wide services that applications can rely on. In top layer contains bundled applications as well as applications downloaded from the Google Play store[1].

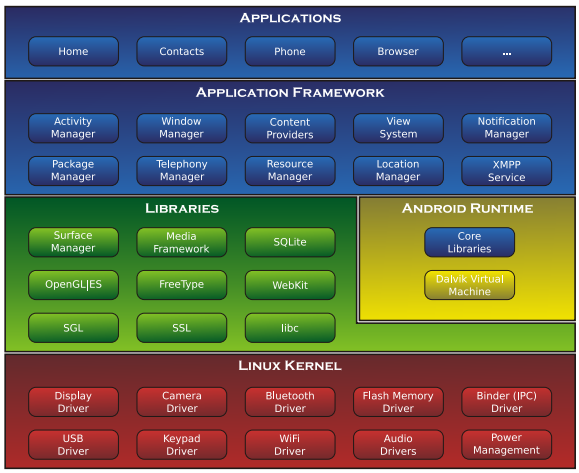

Figure 2. The Android software stack [1]

3.3 The compilation process in Dalvik

As we described in section 3.2, the execution of Java code on Android requires a double compilation process (see fig. 3).

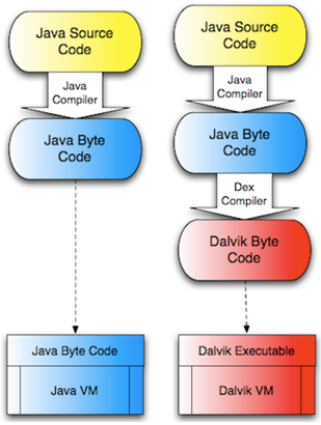

Figure 3. The compilation process in Dalvik [6]

Traditionally, Java source code is compiled by a Java compiler into Java byte code, which is then executed by the JVM. An application that has to run on Android requires to compile the Java byte code into Dalvik byte code by the Dex (Dalvik EXchange) compiler. The output of Dex, the Dalvik byte code is then executed by the DVM.

The most significant differences between the JVM and DVM bytecode are [7]:

- an opcode in JVM is one byte, in Dalvik 2 bytes;

[1] https://play.google.com/store

- the DVM has 226 opcodes while the JVM has 225 different opcodes;
- In JVM, the opcodes for integers and single float, and long integers and double float constants are different. Dalvik implements the same opcode for both, integers and float constants.
- Dalvik bytecode does not have a specific null type. Instead, Dalvik uses a 0 value constant. This requires to distinguish the ambiguous implication of constant 0.
- JVM bytecode uses different opcodes for an object reference comparison and a null type comparison, while Dalvik simplifies them into one opcode. Thus, the type info of the comparison object must be recovered during decompilation.

Android uses trace-based Just-In-Time compilation, which means that only the parts of the code that are executed frequently are compiled to native code at the runtime. Just-in-time compilation (JIT), also known as dynamic translation, translates the original bytecode to the native code just before it is needed.

Since JIT at runtime increases the system load and therefore battery usage, the DVM JIT compiler implements different optimizations such as: a part of the compilation occurs upfront to be fast at run time, another part occurs on the development PC, and some compilation is performed while the device is charging.

3.4 The format of executable files in Dalvik

In standard Java environments, Java source code is compiled into Java byte code, which is stored within .class files. The .class files are read by the JVM at runtime. Each class in the Java source code will result in one .class files. This means that if you have, say, one .java source file that contains a public class, a static inner class, and three anonymous classes, the compiler will output 5 .class files.

A Java .class file consists of the following sections [3]: a header containing a "magic number" and a version number, a constant pool, access rights of the class encoded by a bit mask, a list of interfaces implemented by the class, a list containing fields of the class, a list of methods of the class, and a list of the class attributes (e.g. exceptions).

Typically .class files are distributed in collections. Those collections are .jar files, which are nothing more than a zipped collection of the .class files.

A Dalvik .dex file contains the byte code for the DVM. A .dex file—as a .jar—file contains all byte code of all classes, but is not just a .zip file, it has its own structure (see fig. 4). A .jar file can be converted to .dex files automatically.

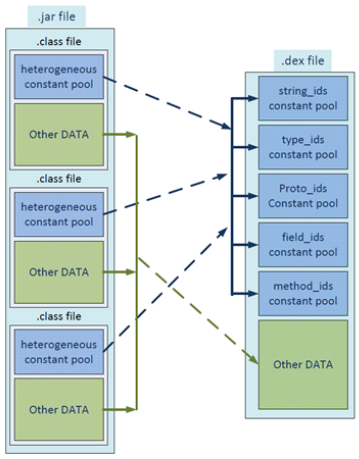

Figure 4. .jar vs .dex files [5]

A .dex file consists of the following sections [9]: a header containing a "magic number" and a version number, a constant pool for each family of elements, definition of classes, definition of methods, and reserved space for general data (see fig. 4).

A .dex file has a limitation of 64K method references. Usually, one Android application has one .dex file, containing all necessary code. Developers can partition parts of the program into multiple secondary dex files, and load them at runtime, if more than 64K method references are needed.

3.5 The support for Java libraries in Dalvik

Not all java libraries are supported in Dalvik. In fact, the libraries java.{applet, awt, beans, lang.management, rmi}, javax.{accessibility, activity, imageio, management, naming, print, rmi, security.auth.kerberos, security.auth.spi, security.sasl, swing, transaction, xml (except javax.xml.parsers)}, org.{ietf.*, omg.*, and w3c.dom.* (sub-packages)} are not supported.

This means, that e.g., applets or applications that rely on AWT or Swing have to be (partly) rewritten.

3.6 Conclusion

The conclusion will recap how the architectural decisions behind Dalvik reflect the goals behind its development: the architecture is designed to reduce the number of executed instructions, the compilation process supports this architecture and is optimized to consume resources only when needed (Just-In-Time compilation), the format of executable files is chosen to minimize space consumption and to increase the execution speed on the actual device.

4. Presentation approach

The concepts of this tech talk will be presented using slides and source code examples.

5. Target audience

The target audience are practitioners and researchers with some experience in Java programming, a basic understanding of the computer architecture, and interested in software development for the Android platform.

References

[1] Android developers, app framework. http://developer.android.com/about/versions/index.html, retrieved 10 April 2013.

[2] F. Maker. Android dalvik virtual machine. http://jaxenter.com/android-dalvik-virtual-machine-35498.html, retrieved 10 April 2013, April 2011.

[3] Oracle, Inc. The class file format. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html, retrieved 10 April 2013.

[4] G. Paller. Dalvik opcodes. http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html, retrieved 10 April 2013.

[5] Security Engineering Research Group. Analysis of dalvik virtual machine and class path library. Technical report, Institute of Management SciencesPeshawar, Pakistan, 2009.

[6] M. Sinnathamby. Stack based vs register based virtual machine architecture, and the dalvik vm. http://markfaction.wordpress.com/2012/07/15/stack-based-vs-register-based-virtual-machine-architecture-and-the-dalvik-vm, retrieved 10 April 2013.

[7] M. Spreitzenbarth. Comparison of dalvik and java bytecode. http://forensics.spreitzenbarth.de/2012/08/27/comparison-of-dalvik-and-java-bytecode, retrieved 10 April 2013, August 2012.

[8] A. S. Tanenbaum and T. Austin. *Structured Computer Organization*. Pearson Education Ltd., Essex, England, 6th edition, 2013.

[9] The Android Open Source Project. .dex dalvik executable format. http://source.android.com/tech/dalvik/dex-format.html, retrieved 10 April 2013, 2007.