David Ehringer
March, 2010

THE DALVIK VIRTUAL MACHINE ARCHITECTURE

Introduction

Java has always been marketed as "write once, run anywhere." The capability has largely been made possible by the Java Platform, the foundation of which is the Java Virtual Machine (JVM). Although this goal has largely been met for the Java platform on desktop (JSE) and server (JEE) environments, the mobile Java ecosystem (JME) is a bit more fragmented with various configurations, profiles, and packages causing significant modifications to applications in order to support different devices.

While Google has selected Java as the language for developing Android applications, it has chosen to abandon both JME and the JVM in favor of an alternative deployment target, the Dalvik virtual machine. Google has also chosen to use an alternative and limited implementation of the standard Java libraries. Both of these are non-standard Java, and effectively represent forking of the Java platform. This seems to break from the trend of people targeting the JVM as a the runtime platform for a very wide range of languages. Scala, Groovy, JRuby, Jython, and Clojure are only a few of the dozens of languages that run on the JVM. These languages can not only take advantage of the many features of the JVM but also seamlessly leverage all the standard Java libraries and the countless custom libraries created by individuals and organizations. Recently, Google has been subtlety bucking this trend by created non-standard Java technologies. This includes non only the Android platform but also Google Web Toolkit (GWT) and Google App Engine, the reasons of which they are non-standard is beyond the scope of this paper as they aren't mobile technologies. But the trend may be applicable to what has happened in the JME space. Non-standard Java implementations may further fragment the platform cause additional headaches for developers looking to support the wide array devices currently on the market.

This paper looks to understand the architecture of Dalvik virtual machine and understand the technical reasons for why Google chose to developer their own non-standard virtual machine. While there are most likely also business reasons behind creating Dalvik, this paper only addresses technical considerations.

Design Constraints Imposed by the Target Platforms

The Android platform was created for devices with constrained processing power, memory, and storage. The minimum device requirements for an Android device are the following:[1]

Feature	Minimum Requirement
Chipset	ARM-based
Memory	128 MB RAM; 256 MB Flash External

Feature	Minimum Requirement
Storage	Mini or Micro SD
Primary Display	QVGA TFT LCD or larger, 16-bit color or better
Navigation Keys	5-way navigation with 5 application keys, power, camera and volume controls
Camera	2MP CMOS
USB	Standard mini-B USB interface
Bluetooth	1.2 or 2.0

Most full featured smart phones will vastly exceed these minimum recommendations and devices will only continue to get more powerful, but these requirements clearly indicate that the platform is targeting a wide array of resource constrained devices. Additionally, a range of operating systems is intended to be supported with

> the baseline system is expected to be a variant of UNIX (Linux, BSD, Mac OS X) running the GNU C compiler. Little-endian CPUs have been exercised the most heavily, but big-endian systems are explicitly supported.[2]

Given the extremely wide range of target environments, it is critical for the application platform to be abstracted away from the underlying operating system and hardware device.

Only a core, although robust and powerful, set of applications will be provided with the platform. An ecosystem of third party developers will be creating thousands of independent, custom built applications that users can download and run on their devices. To maintain maximum stability and security of both the individual applications and platform itself, it is critical that applications are strictly and clearly segregated and cannot interfere in anyway with either the platform or other applications.

The above requirements can be summarized by saying that the Android runtime must support the following:

- limited processor speed
- limited RAM
- no swap space
- battery powered
- diverse set of devices
- sandboxed application runtime

Design Overview

Given that the Android application runtime must support a diverse set of devices and that applications must be sandboxed for security, performance, and reliability, a virtual machine seems like an obvious choice. But a virtual machine-based runtime doesn't necessarily help you balance those requirements with the the limited processor speed and limited RAM that characterizes most mobile devices. So how did Google address all of these somewhat conflicting requirements? To summarize, their approach for implementing an application runtime environment given these constraints is that

> every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool.
>
> The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.[3]

Given every application runs in its own process within its own virtual machine, not only must the running of multiple VMs be efficient but creation of new VMs must be fast as well.

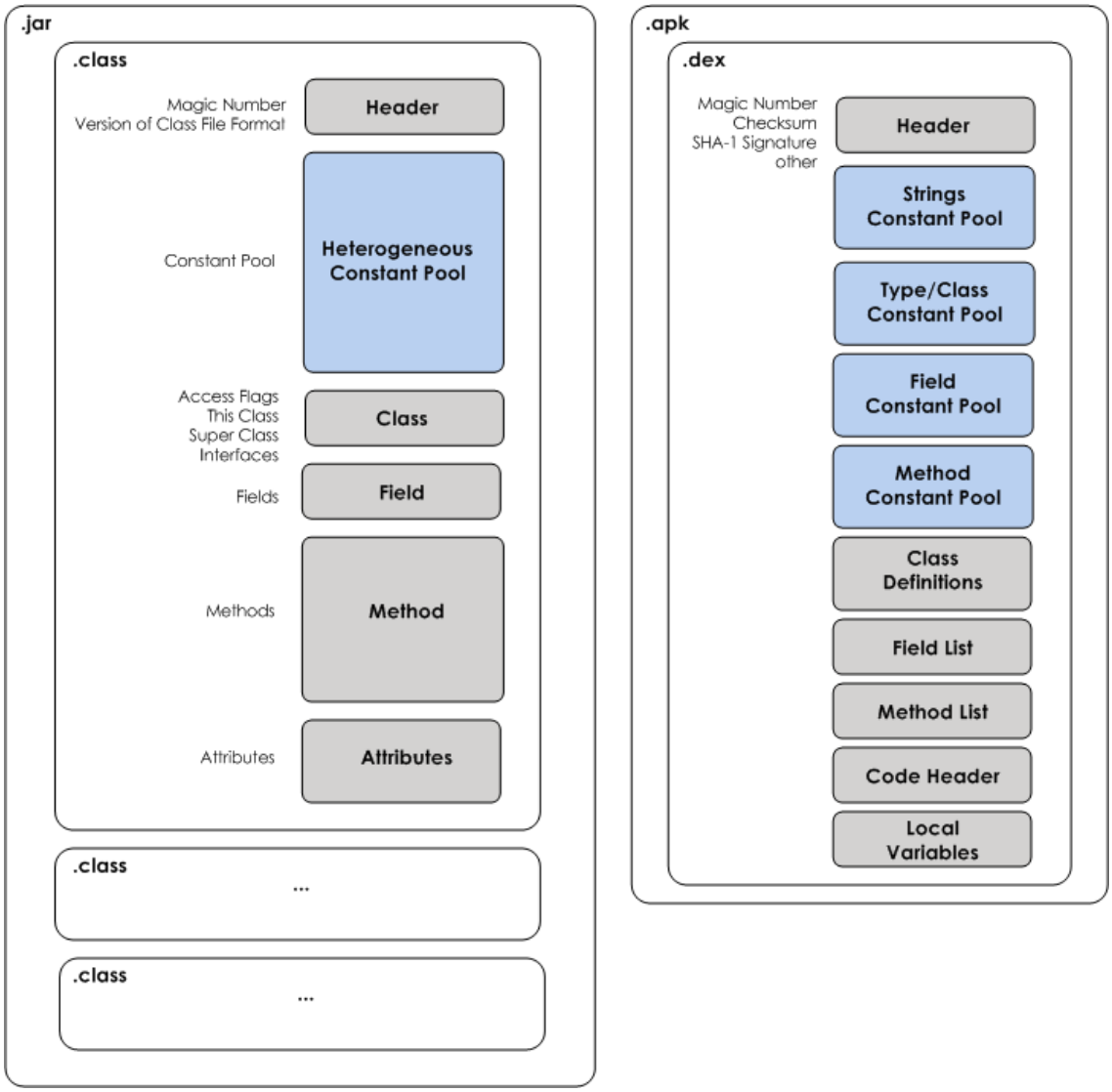
The remainder of this paper surveys the major design choices of the Dalvik VM that were influenced by the requirements outline previously, mainly the .dex file format, the Zygote, the register-based architecture, and security.

The Dex File Format

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

On the Android platform, Java source code is still compiled into .class files. But after .class files are generated, the "dx" tool is used to convert the .class files into a .dex, or Dalvik Executable, file. Whereas a .class file contains only one class, a .dex file contains multiple classes. It is the .dex file that is executed on the Dalvik VM.

The .dex file has been optimized for memory usage and the design is primarily driven by sharing of data. The following diagram contrasts the .class file format used by the JVM with the .dex file format used by the Dalvik VM.[4][5]



The .dex file format uses shared, type-specific constant pools as it's primary mechanism for conserving memory.[6] A constant pool stores all literal constant values used within the class. This includes values such as string constants used in your code as well as field, variable, class, interface, and method names. Rather than store these values throughout the class, they are always referred to by their index in the constant pool. The constant pools in the .class and .dex files are highlighted in blue above. In the case of the .class file, each class has its own private, heterogeneous constant pool. It is heterogeneous in that all types of constants (field, variable, class, etc.) are mixed together. Contrast this to the .dex file that contains many classes, all of which share the same type-specific constants pools. Duplication of constants across .class files is eliminated in the .dex format.

You may ask why sharing of a single heterogeneous pool is not enough and why type-specific pools were used. The type-specific pools actually allow for further elimination of repetition by making the constants more granualr. Consider the two method signatures `(Ljava/lang/String;Ljava/lang/Object;)Ljava/lang/String;` and `(Ljava/lang/String;Lcom/example/Foo;)Ljava/lang/String;`. This is a

representation in a .class file. The Ljava/lang/String constants are repeated. In a .dex file, the Ljava/lang/String constant is stored once and there is a reference from each place where it is used.

By allowing for classes to share constants pools, repetition of constant values is kept to a minimum. The consequence of the minimal repetition is that there are significantly more logical pointers or references within a .dex file compared to a .class file.

If the primary goal of utilizing shared constant pools is to save memory, how much memory is actually being saved? Early in the life of the .class file format, a study found that the average size of a .class file is actually quite small. But since the time to read the file from storage is a dominant factor in startup time, the size of the file is still important. When analyzing how much space each section of the .class file takes on average:

> the biggest part of the Java class files is the constant part [pool] (61 percent of the file) and not the method part that accounts for only 33 percent of the file size. The other parts of the class file share the remaining 5 percent.[7]

So it is clear that optimization of the constant pool can result in significant memory savings. The Android development team found that the .dex file format cut the size in half of some of the common system libraries and applications that ship with Android.[8]

Code	Uncompressed JAR File (bytes)	Compressed JAR File (bytes)	Uncompressed dex File (bytes)
Common System Libraries	21,445,320 (100%)	10,662,048 (50%)	10,311,972 (48%)
Web Browser App	470,312 (100%)	232,065 (49%)	209,248 (44%)
Alarm Clock App	119,200 (100%)	61,658 (52%)	53,020 (44%)

It must be noted that these memory sharing optimizations do not come for free. Garbage collection strategies must respect the sharing of memory. Garbage collections are independent between applications, even though they are sharing some memory, since each application is in a separate process, with a separate VM and separate heaps. The current strategy in the Dalvik garbage collector is to keep mark bits, or the bits that indicate that a particular object is "reachable" and therefore should not be garbage collected, separate from other heap memory.[9] If the mark bits were stored with the objects on the heap, all shared objects would be immediately "unshared" during each garbage collection cycle when the garbage collector traverses the heap and sets the mark bits. Remember that the sharing is predicated on the shared memory being read-only. Separating the mark bits from the other heap memory avoids this trap.

The Zygote

Since every application runs in its own instance of the VM, VM instances must be able to start quickly when a new application is launched and the memory footprint of the VM must be minimal. Android uses a concept called the Zygote to enable both sharing of code across VM instances and to provide fast startup time of new VM instances. The Zygote design assumes that there are a significant number of core library classes and corresponding heap structures that are used across

many applications. It also assumes that these heap structures are generally read-only. In other words, this is data and classes that most applications use but never modify. These characteristics are exploited to optimize sharing of this memory across processes.

The Zygote is a VM process that starts at system boot time. When the Zygote process starts, it initializes a Dalvik VM, which preloads and preinitializes core library classes. Generally these core library classes are read-only and are therefore a good candidate for preloading and sharing across processes. Once the Zygote has initialized, it will sit and wait for socket requests coming from the runtime process indicating that it should fork new VM instances based on the Zygote VM instance. Cold starting virtual machines notoriously takes a long time and can be an impediment to isolating each application in its own VM. By spawning new VM processes from the Zygote, the startup time is minimized.

The core library classes that are shared across the VM instances are generally only read, but not written, by applications. When those classes are written to, the memory from the shared Zygote process is copied to the forked child process of the application's VM and written to there. This "copy-on-write" behavior allows for maximum sharing of memory while still prohibiting applications from interfering with each other and providing security across application and process boundaries.[10][11]

In traditional Java VM design, each instance of the VM will have an entire copy of the core library class files and any associated heap objects. Memory is not shared across instances.

Register-based Architecture

Traditionally, virtual machine implementors have favored stack-based architectures over register-based architectures. This favoritism was mostly due to "simplicity of VM implementation, ease of writing a compiler back-end (most VMs are originally designed to host a single language and code density (i.e., executables for stack architectures are invariably smaller than executables for register architectures)."[12] The simplicity and code density comes at a cost of performance. Studies have shown that a

> register-based architecture requires and average of 47% less executed VM instructions than the stack based [architecture]. On the other hand the register code is 25% larger than the corresponding stack code but this increased cost of fetching more VM instructions due to larger code size involves only 1.07% extra real machine loads per VM instruction which is negligible. The overall performance of the register-based VM is that it take[s], on average, 32.3% less time to execute standard benchmarks.[13]

Given that the Dalvik VM is running on devices with constrained processing power, the choice of a register-based VM architecture seems appropriate. Although register-based code is about 25% larger than stack-based code, the 50% reduction in the code size achieved through shared constant pools in the .dex file offsets the increased code size so you still have a net gain in memory usage as compared to the JVM and the .class file format.

Security

Android's security architecture ensures

> no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc.[14]

These features are implemented across the operating system (OS) level and framework level of Android and not within the Dalvik VM. As mentioned previously, each application is run within its own instance of the VM, which itself is running in its own OS process. Each process is assigned its own user ID that can only access a limited set of features within the system and can only access its own data. Therefore, an application cannot interfere with other applications. If additional security permissions are required, the application can request them through its Manifest file. These permissions are verified at application installation time by the framework and are enforced by the OS at runtime.

To address memory constraints and allow for fast startup times, Dalvik shares core, read-only libraries between VM processes. The sharing is done securely by giving the VM processes permission to read the code but not edit it.

Since it is not possible for applications to interfere with each other based on the OS level security and Dalvik VMs are confined to a single OS process, Dalvik itself is not concerned with runtime security. Although Dalvik is not relied upon for security, it is interesting to note that most of the standard Java security classes remain in the Android distribution. These include the java.lang.SecurityManager and some of the classes in the java.security package. In standard Java environments, the SecurityManger plays the role analogous to the OS process-level security in Android. The SecurityManager typically controls access to resources external to the JVM such as files, processes, and the network. In the Android distribution, the standard security framework is apparently present for applications to use within their own application space but is neither fully implemented nor configured (no java.policy files are present) for interprocess security.[15]

Conclusion

If you align the Android platform to the JME CDC (the Connected Device Configuration or the Configuration for More Capable Devices and SmartPhones, which is based on the JVM) as opposed to the JME CLDC (the Connected Limited Device Configuration, a configuration for much more resource constrained devices which is based on an optimized, yet-still standard, kVM), Google's choice to implement their own non-standard virtual machine appears to be sound from a technical perspective. But the question remains as to whether or not it would have been better for Google to work within the standards process (the JCP or Java Community Process) to get the JME platform back on track and push the standards toward a better future. But given the snails pace at which the standards process moves and the head start that competitors like Apple's iPhone had, working with the standards may not have been an option.

[1] "Device Requirements" Android Source Code <git://android.git.kernel.org/platform/development/pdk/docs/guide/system_requirements.jd>

[2] "Dalvik" Android Source Code <git://android.git.kernel.org/platform/development/pdk/docs/guide/dalvik.jd>

[3] "What is Android?" Android Developers <http://developer.android.com/guide/basics/what-is-android.html>

[4] Lindholm, Tim and Yellin, Frank. The Java Virtual Machine Specification. Second Edition. Sun Microsystems, 1999: Chapter 4

[5] Pavone, Michael. "Dex File Format" <http://www.retrodev.com/android/dexformat.html>

[6] Bornstein, Dan. "Dalvik VM Internals" 2008 Google I/O Session Videos and Slides <http://sites.google.com/site/io/dalvik-vm-internals>

[7] Antonioli, Denis and Pilz, Markus. "Analysis of the Java Class Format" Technical Report 98.4 April 1998

[8] Bornstein

[9] Bornstein

[10] Brady, Patrick. "Anatomy & Physiology of an Android" 2008 Google I/O Session Videos and Slides <http://sites.google.com/site/io/anatomy--physiology-of-an-android>

[11] Bornstein

[12] Jones, Derek. "Register vs. stack based VMs" <http://shape-of-code.coding-guidelines.com/2009/09/register-vs-stack-based-vms/>

[13] Security Engineering Research Group, Institute of Management Sciences. "Analysis of Dalvik Virtual Machine and Class Path Library" 2009

[14] "Security and Permissions." Android Developers <http://developer.android.com/guide/topics/security/security.html>

[15] "Usage of SecurityManger in Androids Permission-System." Android Security Discussions <http://groups.google.com/group/android-security-discuss/browse_thread/thread/847e99bfb2276eff>