International Journal For Technological Research In Engineering
Volume 1, Issue 8, April-2014 ISSN (Online): 2347 - 4718

A REVIEW ON THE FUNCTIONALITY OF DALVIK VIRTUAL MACHINE PRESENT IN ANDROID OPERATING SYSTEM

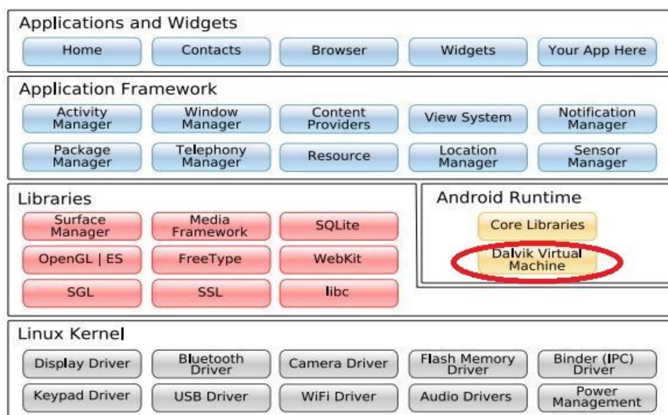
Zeshan I.Khan[1], Vijay Gulhane[2]
[1] Student, [2]Associate Professor
Department of Computer Science and Engineering
Sipna College of Engineering and Technology, Amravati, Maharashtra, India

Abstract: Dalvik Virtual Machine which plays an important role in the efficient execution of the application in Android Operating System made by Google. Instead of using a Standard JVM, the decision of making an alternative virtual machine while using Java code in the android application development suggest many reasons for the efficiency as well as fast execution. The Report helps to show the overall architecture, functionality, advantages behind using Dalvik Virtual Machine which is treated as the heart of Android Operating System.
Keywords: Dalvik, Stack, Register, Dex, Virtual Machines

I. INTRODUCTION

Dalvik is the process virtual machine (VM) in Google's Android Operating System. It is the software that runs the apps on Android devices. Dalvik is thus an integral part of Android, which is typically used on mobile devices such as mobile phones and tablet computers as well as more recently on embedded devices such as smart TVs and media streamers. Programs are commonly written in Java and compiled to byte code. They are then converted from Java virtual machine-compatible Java class files to Dalvik-compatible .dex (Dalvik Executable) and odex (Optimized Dalvik Executable) files before installation on a device, giving rise to the related terms odexing and de-odexing. The compact Dalvik Executable format is designed to be suitable for systems that are constrained in terms of memory and processor speed. Dalvik is open-source software. It was originally written by Dan Bornstein, who named it after the fishing village of Dalvík in Iceland.

II. ANDROID ARCHITECTURE

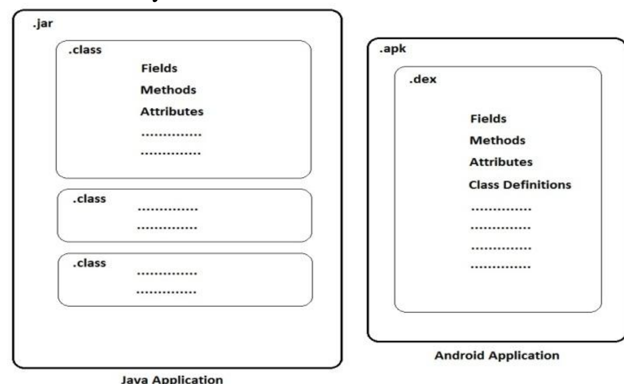


III. ABOUT DALVIK

"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. 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.

IV. THE DEX FILE FORMAT

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 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.



 Copyright 2013.All rights reserved.

V. ANALYZING EFFICIENT NATURE OF DALVIK

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. 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.

Code	**Un-Compressed Jar**	**Compressed Jar**	**dex file**
Web Browser App	470,312 (100%)	232,065 (49%)	209,248 (44%)
Alarm Clock App	119,200 (100%)	61,658 (52%)	53,020 (44%)

VI. VMS PROCESS

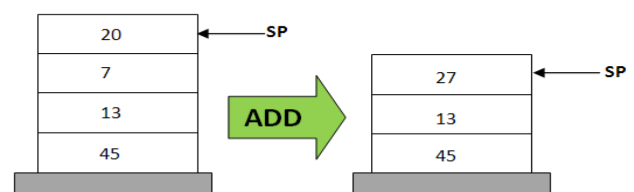
Since every application runs in its own instance of the VM, VM instances must be able to start quickly when a new application is launched and the memory footprint of the VM must be minimal. Android uses a concept called the Zygote to enable both sharing of code across VM instances and to provide fast 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 pre initializes 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. 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.

VII. USING REGISTER-BASED ARCHITECTURE RATHER THAN STACK-BASED ARCHITECTURE

Traditionally, virtual machine implementers 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., executable for stack architectures are invariably smaller than executable for register architectures)." The simplicity and code density comes at a cost of performance. Studies have shown that a register-based architecture requires an 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. 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.

VIII. STACK BASED VIRTUAL MACHINES

A stack based virtual machine implements the general features described as needed by a virtual machine in the points above, but the memory structure where the operands are stored is a stack data structure. Operations are carried out by popping data from the stack, processing them and pushing in back the results in LIFO (Last in First Out) fashion. In a stack based virtual machine, the operation of adding two numbers would usually be carried out in the following manner (where 20, 7, and 'result' are the operands):



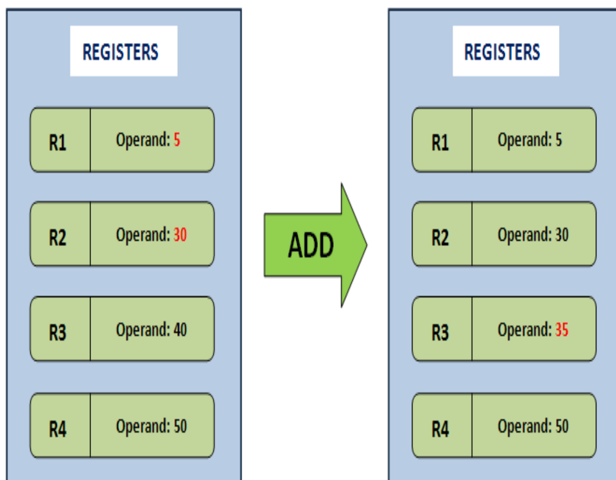
 Copyright 2013.All rights reserved.

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

Because of the PUSH and POP operations, four lines of instructions is needed to carry out an addition operation. An advantage of the stack based model is that the operands are addressed implicitly by the stack pointer (SP in above image). This means that the Virtual machine does not need to know the operand addresses explicitly, as calling the stack pointer will give (Pop) the next operand. In stack based VM's, all the arithmetic and logic operations are carried out via Pushing and popping the operands and results in the stack.

IX. REGISTER BASED VIRTUAL MACHINES

In the register based implementation of a virtual machine, the data structure where the operands are stored is based on the registers of the CPU. There is no PUSH or POP operations here, but the instructions need to contain the addresses (the registers) of the operands. That is, the operands for the instructions are explicitly addressed in the instruction, unlike the stack based model where we had a stack pointer to point to the operand. For example, if an addition operation is to be carried out in a register based virtual machine, the instruction would more or less be as follows:

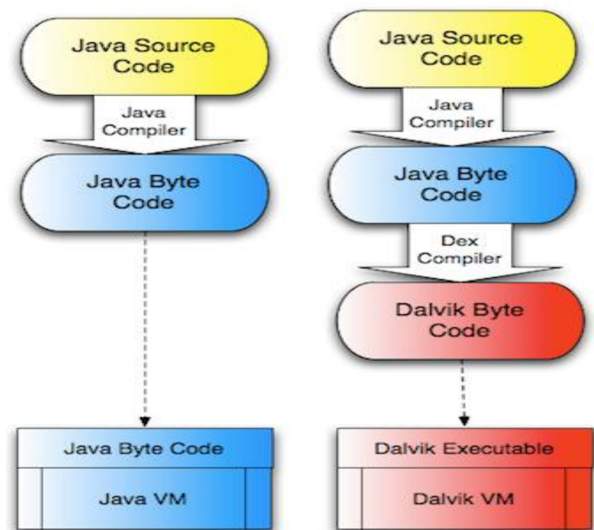


ADD R1, R2, R3; # Add contents of R1 and R2, store result in R3

As it is mentioned earlier, there is no POP or PUSH operations, so the instruction for adding is just one line. But unlike the stack, we need to explicitly mention the addresses of the operands as R1, R2, and R3. The advantage here is that the overhead of pushing to and popping from a stack is non-existent, and instructions in a register based VM execute faster within the instruction dispatch loop. Another advantage of the register based model is that it allows for some optimizations that cannot be done in the stack based approach. One such instance is when there are common sub expressions in the code, the register model can calculate it once and store the result in a register for future use when the sub expression comes up again, which reduces the cost of recalculating the expression. The problem with a register based model is that the average register instruction is larger than an average stack instruction, as we need to specify the operand addresses explicitly. Whereas the instructions for a stack machine is short due to the stack pointer, the respective register machine instructions need to contain operand locations, and results in larger register code compared to stack code.

X. CONCLUSION

The use of Dalvik Virtual Machine with Register Based Architecture to convert a .class file into a .dex file format which helps the application to execute in a fast manner as well as in a own separate process.



REFERENCES

[1] David Ehringer- "The Dalvik Virtual Machine Architecture".
[2] http://en.wikipedia.org/wiki/Dalvik_ (software).
[3] Dalvik (http://source.android.com/devices/tech/dalvik/).
[4] Stack based and Register based Architecture (http://markfaction.wordpress.com/2012/07/15/stack-based-vs-register-based-virtual-machine-architecture-and-the-dalvik-vm/).

 Copyright 2013.All rights reserved.