COSC 530: Semester Project
Dalvik Virtual Machine

By

Laxman Nathawat

Yuping Lu

Chunyan Tang

Ye Sun

Date

December 3, 2013

Table of Contents

1. Introduction -- 2

2. Virtual Machine Types --- 3

3. JVM vs. DVM --- 6

4. Dalvik Instruction Set -- 9

5. Dalvik Performance Analysis --- 13

6. Conclusion --- 17

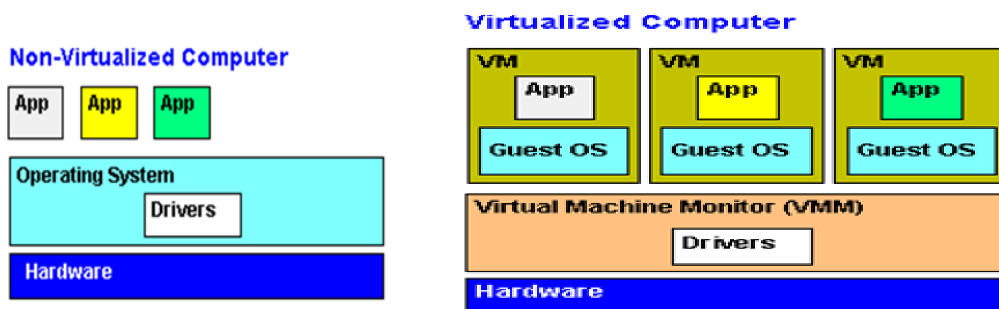
7. References -- 18

1. Introduction

The concept of Virtual Machines (VMs) is very old in the field of computing and in the broadest sense encompasses all emulation techniques to provide a standard software interface. In recent years, virtualization has gained momentum because of multiple factors such as security failures in standard operating systems, increased importance of security and reliability, and increasing use of cloud computing where unrelated users may share same hardware. Another factor that contributed to popularity of VMs is substantial increase in processor speeds, improvements in memory hierarchies and improvements in VM software stack to make VM overheads reasonable.

The operating system (OS) running on virtual machine is known as guest OS. The OS over all virtual machines is called host OS. Multiple VMs each run their own OS (guest OS). This is often used in server combination, which allows different server services to run in separate VMs on the same physical machine without interference instead of running them each in individual physical machine. Compared to the conventional platform where a single OS utilizes all the hardware resources, multiple virtual machines run on a single physical machine, all share the hardware resources, and run their own OS individually.

The VM supporting software is called Virtual machine monitor (VMM) or hypervisor. It manages the mapping of virtual resources to physical resources, which may be time-shared, partitioned, or emulated in software. The software enhances the performance of the VM's guest OS and provides much more control over the VM interface. The following images illustrate three applications running in a regular computer versus in a virtualized computer.



The hypervisor is the most important part in VM.

2. Virtual Machine Types

Increased use of virtualization at the desktop, server, data center and cloud computing level has led to development new VMs and improvement of existing VMs. Some of these VMs are open-source based whereas others are closed-source and proprietary in nature. VMs can also differ in terms of their instruction set design and the level of software interface provided through different emulation techniques.

The virtual machines which have similar instruction set architecture as the underlying hardware are known as system-level virtual machines. A system VM (also known as hardware VM) is a software designed for hardware virtualization. It provides a system platform at the binary instruction set architecture (ISA) that supports the execution of a complete operating system. It allows the sharing of the underlying physical machine resources among different virtual machines, whereas each VM has its own operating system.

The advantages of using a system VM are: (1) Improved stability and security. It provides multiple operating system environments that can co-exist on the same computer in strong isolation from each other. Single operating system is independent from configurations of other operating systems. This feature increases isolation and security in modern systems. (2) Consolidation. It provides a platform to run programs where the real hardware is not available for use or provide a more efficient use of computing resource including energy consumption and cost effectiveness by time-sharing a single computer between several single-tasking operating systems. (3) Flexibility. A virtualized machine can host numerous versions of an operating system, which allows developers to test their programs in different OS environments on the same machine and crashing in one virtual machine will not bring down the whole system. In addition, VM is relatively easy to move from one server to another to balance the workload, migrate to faster hardware, and recover from hardware failure. Examples of available system VMs are IBM VM/370, VMware ESX server, and Xen.

Xen is open-source virtualization software that started as a research project at the University of Cambridge. It is widely supported by most of the leading computer vendors such as IBM, Citrix,

Red Hat, HP and Intel. Over the years it has included several modifications and features to improve overall performance of the virtualization software. Some of the improvements are:

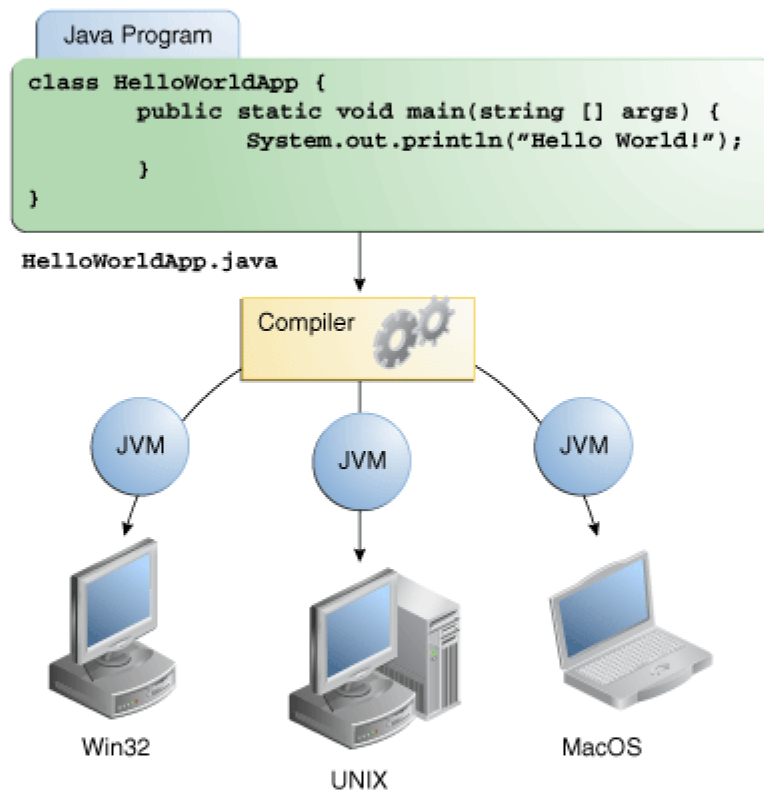
- Use of paravirtualization
- Mapping itself to upper 64 MB address space of VM (to avoid TLB flushing)
- Allows guest OS to allocate pages
- Uses four levels of 80x86 protection levels. VMM runs at the highest level (0) and the guest OS running at the next level (1).
- Uses driver domains and guest domains to simplify IO handling.

VMWare ESX Server virtualization software is based on VMWare ESX hypervisor. The ESX based virtualization runs on bare-metal by using its own kernel, called vmkernel (microkernel) on the top of Linux kernel. The Linux kernel is started at the boot time and serves as the service console. The ESXi variant of ESX does not use Linux kernel as the service console but solely depends on vmkernel which provides three interfaces – hardware, guest OS and service console to interact with entities outside virtual environment.

In addition to system VMs there is another categories of VMs called process VMs (also known as application VMs). Process VM run as a normal application inside an operating system and supports a single process. It is created when the process is started and destroyed when it exits. It provides a platform-independent programming environment that abstracts underlying hardware or operating system, and allows a program to execute in the same way on any platform. Therefore, process VM provides a higher level abstraction than the low-level instruction set architecture abstraction of the system VM. Process VMs are implemented using an interpreter and achieve performance comparable to compiled programming languages using just-in-time compilation. Examples of process VM include Java VM, Dalvik VM and Microsoft .net Framework.

The Java VM is stack-based virtual machine. Once Java compiler translates Java source code into .class (byte code), the JVM runtime environment executes byte code or .jar files emulating JVM instruction set by interpreting or by Just-in Time (JIT) compiling.

As shown in the following picture, the byte code generated by the compiler can be used by multiple Java virtual machines running on different operating systems. For example, byte code generated on Windows OS can be fed to JVM running on Linux OS without the need to recompile.



Dalvik Virtual Machine (DVM) is register-based and the use of registers instead of stack is mostly due to use of DVM for mobile devices. Android is an open source mobile platform from Google and is based on Linux kernel with DVM as the process or application VM. Unlike regular desktop computing, mobile devices have special design constrains such as:

- Limited memory
- Limited processor speed
- Variety of screen sizes and resolutions
- No swap space
- Battery power

The DVM architecture design takes into account most the above mentioned constraints making it different from JVM. The next section of the report exclusively talks about key differences in the design of DVM and JVM.

3. JVM vs. DVM

Java Virtual Machines and Dalvik Virtual Machines have significant differences in the architecture design and functionality. We present comparison of two VMs using memory usage, architecture design, compilation techniques, and library support as parameters.

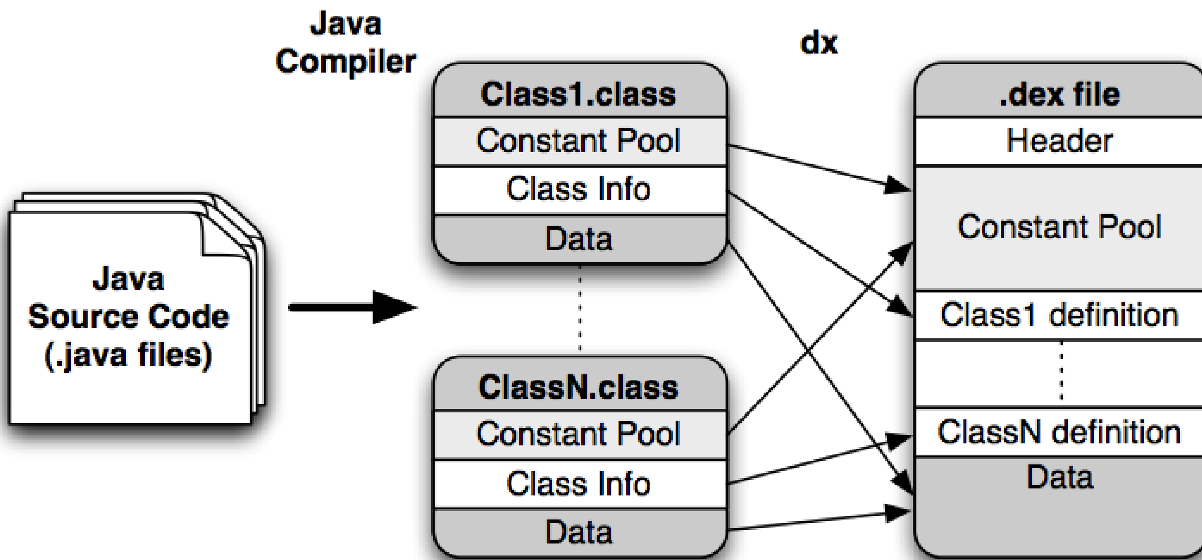
Memory Usage Comparison

Java Virtual machine (JVM) uses heap memory for its application. It has a built-in garbage collector that manages the internal memory. When a java program needs memory, it requests memory from JVM. In case there is no memory left, JVM automatically reclaims memory for reuse using garbage collector without memory allocation and deallocation. This feature eliminates memory leaks and other memory-related problems. However, JVM uses most of its resources on garbage collection, which leads to serious performance problem. For example, JVM has trouble releasing more of its memory when an "out of memory" exception is thrown. JVM uses big proportion of its memory for runtime libraries created in shared memory. On the contrary, in Dalvik VM, programs are commonly written in Java, compiled to byte code, and then converted from JVM-compatible .class files to Dalvik Executable files, which can be executed directly. The compact Dalvik Executable format has low memory requirement, which is suitable for systems with limited memory and processor speed, i.e. mobile phones, tablet computers, embedded devices such as smart TVs.

Architecture Comparison

The JVM architecture is designed to support most of the popular operating systems whereas DVM architecture is specifically targeted for the Android platform. Since mobile devices run in a constraint environment, they must make efficient use of storage, memory, battery power, and processor power. Dalvik's register-based architecture allows Android to be more efficient and faster compared to the stack-based design of the JVM. Also, Android platform is designed to run apps from thousands of users and vendors; it must provide a high-level of security for individual apps as well as for the platform itself. Android provides security by giving each app its own virtual machine which is different than the JVM approach where all applications share same virtual machine.

The following picture shows a dex file which combines multiple .class files into a single file. Different items such as constants, variables, methods, and classes are grouped into separate sections in the dex file and then are accessed by respective classes through pool indexing.



Because each app in Android runs in its own process with its own instance of the Dalvik virtual machine, a device can run multiple VMs efficiently with minimum memory. This feature in part possible due to use of Dalvik Executable (.DEX) file format on DVM.

Multiple Instance and JIT Comparison

JVM runtime executes .class or .jar files using a just-in-time compiler (JIT). JIT causes delay in initial execution of an application due to the time it takes to load and compile the byte code. In the worst case, it may crash the system if resources become unavailable for applications. These become an obvious disadvantage when it is used in limited system resources such as tablets and cell phones. Dalvik uses ahead-of-time optimization that involves the instruction modification. Therefore, it allows multiple instances of VM to run simultaneous with low memory requirement.

Reliability Comparison

In current standard Java runtime systems, the failure of a single component can have significant impacts on other components. In the worst case, a malicious or erroneous component may crash the whole system. On the other hand, Dalvik runs every instance of VM in its own separate process. Separate processes prevent all applications from crashing in case if the VM for a specific app crashes.

Supported Libraries Comparison

The Dalvik VM like JVM has built-in support for core java programming packages. In addition to core packages, Dalvik has its own set of packages such as com.google.* and android.*. The following table lists subset of packages for Dalvik and standard Java.

Libraries	**Dalvik**	**Standard Java**
java.io	Yes	Yes
java.net	Yes	Yes
android.*	Yes	no
com.google.*	Yes	No
javax.swing.*	No	Yes

4. Dalvik Instruction Set

Dalvik instructions are longer and more complex than the JVM instructions. Dalvik VM can use up to 256 registers and currently has 226 instructions. The following conventions are used for Dalvik instruction format:

- Arguments which name a register have the form "v*X*".
- Arguments which indicate a literal value have the form "#+*X*".
- Arguments which indicate a relative instruction address offset have the form "+*X*".
- Arguments which indicate a literal constant pool index have the form "*kind@X*", where "*kind*" indicates which constant pool is being referred to. Each opcode that uses such a format explicitly allows only one kind of constant.
- Registers are 32 bits wide. For 64-bit values adjacent registers are used.
- Bitwise data is represented in-line in the instruction stream.
- Pseudo instructions are used for variable length data. For example, fill-array-data, represents a pseudo instruction.
- Type specific opcodes are suffixed with their types. For example, add-int.
- Type general opcodes for 64-bit data are suffixed with –wide.

The following table summarizes opcode mnemonics, their bit sizes and brief description.

Mnemonic	Bit Size	Description
b	8	immediate signed **b**yte
c	16, 32	**c**onstant pool index
f	16	inter**f**ace constants (only used in statically linked formats)
h	16	immediate signed **h**at (high-order bits of a 32- or 64-bit value; low-order bits are all 0)
i	32	immediate signed **i**nt, or 32-bit float
l	64	immediate signed **l**ong, or 64-bit double
m	16	**m**ethod constants (only used in statically linked formats)
n	4	immediate signed **n**ibble
s	16	immediate signed **s**hort
t	8, 16, 32	branch **t**arget
x	0	no additional data

In the table below, we summarize some of the instructions that are available in Dalvik VM.

Instruction Type	**Instruction Format**	**Instruction Description**
Copy	move-wide/16 vAAAA, vBBBB	Copy contents of register A into register B. Both registers are of size 16 bits.
Move	move-result-wide vAA	Move the double-word result of the most recent invoke-kind into the indicated register pair A (8 bits). This must be done as the instruction immediately after an invoke-kind whose (double-word) result is not to be ignored; anywhere else is invalid.
Move	move-result-object vAA	Move the object result of the most recent invoke-kind into the indicated register A (8 bits). This must be done as the instruction immediately after an invoke-kind or filled-new-array whose (object) result is not to be ignored; anywhere else is invalid.
Write	move-exception vAA	Save a just-caught exception into the given register A (8 bits). This should be the first instruction of any exception handler, whose caught exception is not to be ignored, and this instruction may only ever occur as the first instruction of an exception handler; anywhere else is invalid.
Read	return-object vAA	Return from an object-returning method. A is the return value register (8 bits)
Read	return-wide vAA	Return from a double-width (64-bit) value-returning method. A is the return value register-pair (8 bits).
Move	const/16 vAA, #+BBBB	Move the given literal value (sign-extended to 32 bits) from register B (16 bits) into the specified register A (8 bits).
Write	new-instance vAA, type@BBBB	Construct a new instance of the indicated type, storing a reference to it in the destination register A (8 bits). The type must refer to a non-array class. B: type index
Write	new-array vA, vB, type@CCCC	Construct a new array of the indicated

		type and size. The type must be an array type. A: destination register (8 bits) B: size register C: type index
Write	fill-array-data vAA, +BBBBBBBB	Fill the given array with the indicated data. The reference must be to an array of primitives, and the data table must match it in type and size. A: array reference (8 bits) B: signed "branch" offset to table data (32 bits)
Branch	goto +AA	Unconditionally jump to the indicated instruction. A: signed branch offset (8 bits)
Branch	packed-switch vAA, +BBBBBBBB	Jump to a new instruction based on the value in the given register, using a table of offsets corresponding to each value in a particular integral range, or fall through to the next instruction if there is no match. A: register to test B: signed "branch" offset to table data (32 bits)
Branch	sparse-switch vAA, +BBBBBBBB	Jump to a new instruction based on the value in the given register, using an ordered table of value-offset pairs, or fall through to the next instruction if there is no match. A: register to test B: signed "branch" offset to table data (32 bits)
Branch	cmpl-float (*lt bias*) vAA, vBB, vCC cmpl-float (*gt bias*) vAA, vBB, vCC	Perform the indicated floating point, storing 0 if the two arguments are equal, 1 if the second argument is larger, or -1 if the first argument is larger. The "bias" listed for the floating point operations indicates how NaN comparisons are treated: "gt bias" instructions return 1 for NaN comparisons, and "lt bias" instructions return -1. A: destination register (8 bits)

		B: first source register or pair C: second source register or pair
Conditional branch	if-eq vA, vB, +CCCC if-ne vA, vB, +CCCC	Branch to the given destination if the given two registers' values compare as specified. A: first register to test (4 bits) B: second register to test (4 bits) C: signed branch offset (16 bits)
Indirectly branch	invoke-virtual {vD, vE, vF, vG, vA}, meth@CCCC	Call the indicated method. The result (if any) may be stored with an appropriate move-result* variant as the immediately subsequent instruction. B: argument word count (4 bits) C: method index (16 bits) D..G, A: argument registers (4 bits each)
Unary operation	int-to-long vA, vB int-to-float vA, vB int-to-double vA, vB int-to-byte vA, vB	Perform the identified unary operation on the source register, storing the result in the destination register. A: destination register or pair (4 bits) B: source register or pair (4 bits)
Binary operation	add-int vAA, vBB, vCC sub-int vAA, vBB, vCC mul-int vAA, vBB, vCC div-int vAA, vBB, vCC	Perform the identified binary operation on the two source registers (register B and C), storing the result in the destination register. A: destination register or pair (8 bits) B: first source register or pair (8 bits) C: second source register or pair (8 bits)

5. Dalvik Performance Analysis

Dalvik VM includes several features for performance optimization, verification, and monitoring. We discuss subset of performance parameters in this report.

- Dalvik Executable (DEX)
- Byte code optimization
- Byte code verification
- Execution modes
- Dalvik Debug Monitor ("DDM")

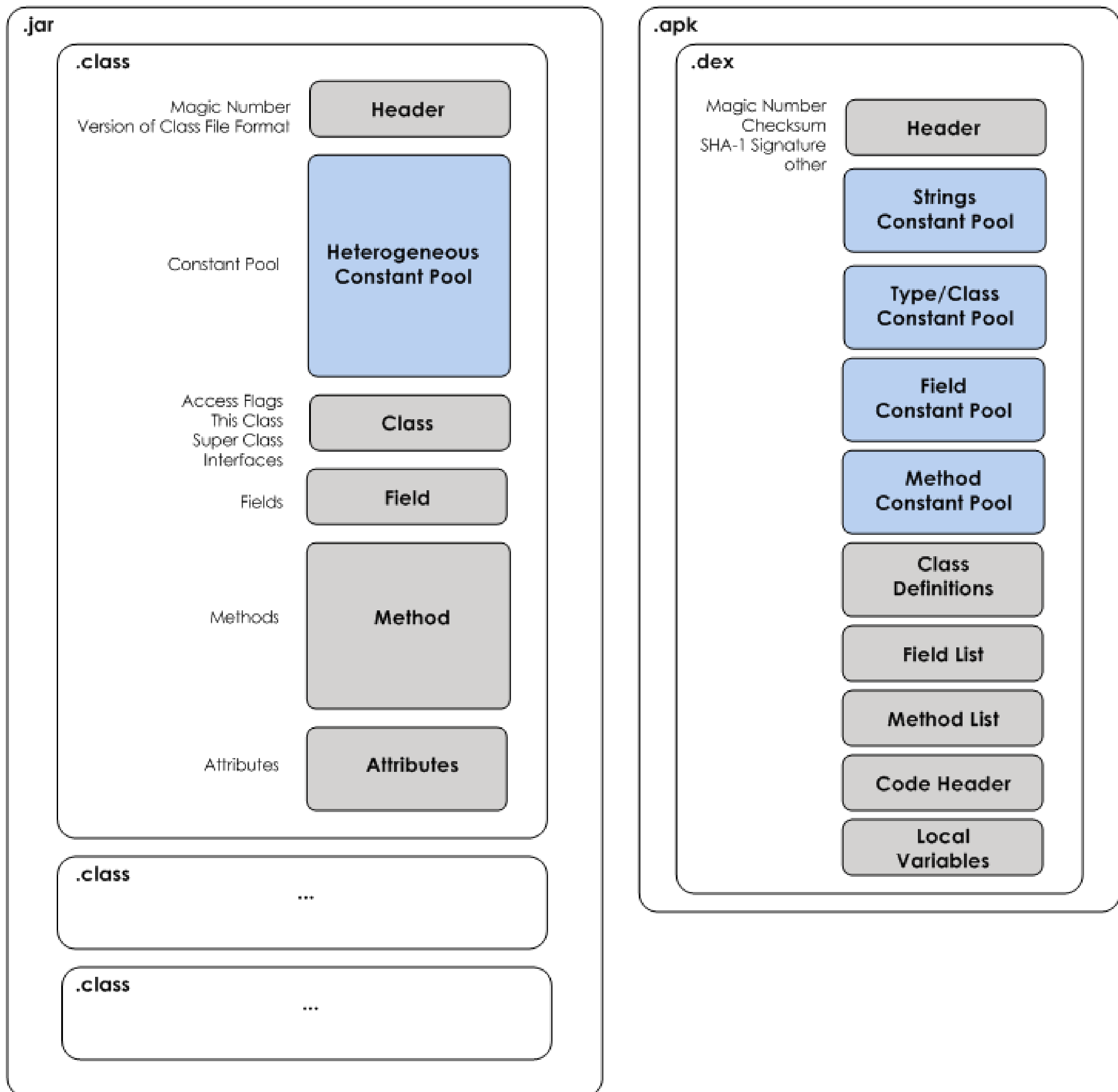
Dalvik Executable (.DEX)

Java source code is compiled by the Java compiler into .class files. Then the dx (dexer) tool, part of the Android SDK processes the .class files into a file format called DEX that contains Dalvik byte code. The dx tool eliminates all the redundant information that is present in the classes. In DEX all the classes of the application are packed into one file. The following table provides comparison between code sizes for JVM jar files and the files processed by the dex tool. The table compares code sizes for system libraries, web browser applications, and a general purpose application (alarm clock app). In all cases dex tool reduced size of the code by more than 50%.

Code	Uncompressed JAR file	Compressed JAR file	Uncompressed dex file
Common System Libraries	100%	50%	48%
Web Browser App	100%	49%	44%
Alarm Clock App	100%	52%	44%

In standard Java environments each class in Java code results in one .class file. That means, if the Java source code file has one public class and two anonymous classes, let’s say for event handling, then the java compiler will create total three .class files.

The compilation step is same on the Android platform, thus resulting in multiple .class files. But after .class files are generated, the "dx" tool is used to convert all .class files into a single .dex, or Dalvik Executable, file. 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.



Byte Code Optimization

Since mobile devices have limited RAM and disk storage capacities, every attempt should be made to reduce the size of application code and related data. Some of the optimization goals addressed in Dalvik VM are:

- The overhead in launching a new app must be minimized to keep the device responsive.
- Class data must be shared between multiple processes to minimize total system memory usage.
- Storing class data in individual files results in a lot of redundancy, especially with respect to strings. To conserve disk space we need to factor this out.
- Parsing class data fields adds unnecessary overhead during class loading. Accessing data values (e.g. integers and strings) directly as C types is better.
- Byte code optimization is important for speed and battery life.
- For security reasons, processes may not edit shared code.

The typical VM implementation uncompresses individual classes from a compressed archive and stores them on the heap. This implies a separate copy of each class in every process, and slows application startup because the code must be uncompressed (or at least read off from disk in many small pieces). On the other hand, having the byte code on the local heap makes it easy to rewrite instructions on first use, facilitating a number of different optimizations.

These goals led to following optimization decisions:

- Multiple classes are aggregated into a single "DEX" file.
- DEX files are mapped read-only and shared between processes.
- Byte ordering and word alignment are adjusted to suit the local system.
- Byte code verification is mandatory for all classes but “pre-verify” is preferred.
- Optimizations that require rewriting byte code must be done ahead of time.

One way to optimize classes is to load them all into the VM and verify them for optimization but this approach can lead to difficulty in allocation of resources associated with native shared libraries. Instead dexopt tool is used for optimization which performs abbreviated VM initialization and loads zero or more dex files from the bootstrap class path for optimization. After process completion, dexopt tool releases all resources. File locking mechanism is used to prevent multiple VMs invoking dexopt tool simultaneously on same set of files.

Byte Code Verification

Byte code verification can be can be enabled for all classes, disabled for all, or enabled only for "remote" (non-bootstrap) classes. It should be performed for any class that will be processed with the DEX optimizer, and in fact the default VM behavior is to only optimize verified classes. Once the verified and optimized DEX files have been prepared, verification incurs no additional overhead except when loading classes that failed to pre-verify. If DEX files are processed with verification disabled, and the verifier is turned on later, application loading will be noticeably slower (perhaps 40% or more). There are certain types of checks that verifier does not perform. For example,

- Operand stack size is not verified.
- Type restrictions on constant pool are not verified.
- Since VM does not support subroutines, limits on JSR and ret do not apply during verification.

Execution Mode

The Dalvik VM supports multiple (three) execution modes to optimize performance for different types of tasks. The three execution modes are – fast, portable and debug. The fast mode is optimized for the current platform of the Android and may consist of hand optimized assembly code. The portable mode is used to target multiple platforms and is not specifically optimized for single platform. The debug mode is a variation of the portable mode. It allows profiling and single-stepping. Command line options can be given to the zygote process of the VM to switch between different modes.

Dalvik Debug Monitor

The Dalvik Debug Monitor (DDM) tool allows monitoring live states of the Dalvik VM. It works on the client-server architecture and can be used to monitor multiple VMs running on a device connected through wireless network connection or through USB. DDM can be used to monitor thread states and overall heap status for VMs. It can also be used to monitor app logcat, load averages and virtual memory usage for a VM. DDM server is written in Java language for portability and uses Java Debug Wire Protocol (JDWP).

6. Conclusion

Increased security concerns and improvements in processor speeds and memory hierarchies have led to popularization of virtual machines on almost all types of computing devices – mobile, desktop, and servers. Virtual machines can be categories into two groups – system-level and application-level. System-level virtual machines emulate entire operating system and the user gets the illusion of having entire machine to himself/herself. On the other hand, application-level or process-level virtual machine runs as an application of a process within an operating system.

Java Virtual Machine differs from Dalvik Virtual Machine is many ways. One, JVM is stack-based whereas DVM is register-based. Two, JVM keeps byte code files separate whereas DVM uses DEX tool to combine multiple byte code files into a single file. Three, JVM does not perform optimization on byte code files whereas DVM uses dexopt tool to optimize byte code to make it suitable for memory, storage and battery power constraint mobile devices.

In terms of architecture, DVM supports up to 256 registers and provides about 226 opcodes for different types of instructions. Registers in Dalvik VM are 32-bit wide and follow destination-then-source ordering for instruction arguments. The Dalvik architecture also supports multiple instructions for variety of tasks such as arithmetic operations, branching and data copying.

Dalvik VM incorporates series of features for performance optimization, verification and monitoring. Some of the critical performance features included are Dalvik executable file format, byte code optimization, byte code verification, multiple execution modes, and the use of the Dalvik Debug Monitor (DDM).

The work in this report can be extended by doing performance analysis of an actual Java application and by comparing byte code size and execution times on DVM and JVM. Another possible suggestion for further work is to compare DVM with ART (Android Runtime VM). ART is currently in experimental phase and uses Ahead-of-time (AOT) compilation to pre-compile the byte code into machine language at the time of app installation. It is expected to drop the runtime to half but the running process will occupy 10-20% more storage space.

7. References

1) John L. Hennessy and David A. Patterson. Computer Architecture – A Quantitative Approach, 5th edition, Morgan Kaufmann Publishing, 107 - 111.

2) http://source.android.com/devices/tech/dalvik/instruction-formats.html

3) http://imsciences.edu.pk/serg/wp-content/uploads/2010/10/1st_Analysis-of-Dalvik-VM.pdf

4) http://jawadmanzoor.files.wordpress.com/2012/01/android-report1.pdf

5) http://en.wikipedia.org/wiki/Dalvik_(software)

6) http://www.netmite.com/android/mydroid/dalvik/docs/dalvik-bytecode.html

7) http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf

8) http://www.netmite.com/android/mydroid/dalvik/docs/dex-format.html

9) http://androidaio.com/google-introduces-artandroid-runtime-in-kitkat/

10) http://xenproject.org/

11) http://www.vm.ibm.com/

12) http://docs.oracle.com/javase/tutorial/

13) http://www.pcmag.com/encyclopedia/term/53927/virtual-machine

14) www.vmware.com