Constructing an Environment and Providing a Performance Assessment of Android's Dalvik Virtual Machine on x86 and ARM

Goutham Selvakumar

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfilment of the requirements for the degree of Master of Science.

Thesis Committee:

The Thesis Committee for Goutham Selvakumar certifies
That this is the approved version of the following thesis:

Constructing an Environment and Providing a Performance Assessment of Android's Dalvik Virtual Machine on x86 and ARM

Committee:

Chairperson

Date Approved

Acknowledgements

I would like to take this opportunity to thank my advisor Dr.Prasad Kulkarni for guiding me right from the beginning till the end of this research. This project is result of his vision and accumulated knowledge from his past. I am glad to have such a mentor who is very kind, extremely patient, flexible and highly knowledged. He is the sole motivation for completing this research and writing this report.

I would like to thank all my committee members, Dr.Xin Fu and Dr.Victor Frost, who helped me to make this possible. I would like to thank my parents, friends and family, whom are the very reason for me being in this position today. They were always available whenever I needed them. I would like to thank, University of Kansas for its research program and all the facility it provided to help me gain knowledge in the recent cutting edge technology. I am glad that I got the assistant-ship which help me to focus more on academics by backing me financially.

Finally, I would like to thank all the teachers in my life, who shaped me to become who I am now, without their dedication and encouragement all this would have been next to impossible.
Thank you all.

Abstract

Android is one of the most popular operating systems (OS) for mobile touch screen devices, including smart-phones and tablet computers. *Dalvik* is a process virtual machine (VM) that provides an abstraction layer over the Android OS, and runs the Java-based Android applications. The first goal of this project is to construct a development environment for conveniently investigating the properties of Android's Dalvik VM on contemporary x86 and ARM architectures. The normal development environment restricts the Dalvik VM to run on top of Android, and requires an updated Android image to be built and installed on the *target* device after any change to the Dalvik code. This update-build-install process unnecessarily slows down any Dalvik VM exploration. We have now discovered a configuration that enables us to study the Dalvik VM as a stand-alone application on top of the Linux OS on x86 machines.

The second goal of this project is to understand the translation/compilation sub-system in the Dalvik VM, experiment with various modifications to determine the best translation parameters, and compare the Dalvik VM's just-in-time (JIT) compilation characteristics (such as quality of code generated and compilation time) on the x86 and ARM systems with a state-of-the-art Java VM. As expected, we find that JIT compilation is able to significantly improve application performance over basic interpretation. Comparing Dalvik's generated code quality with the Java HotSpot VM, we observe that Dalvik's ARM target is a much more mature compared to Dalvik-x86. Therefore, Dalvik's simple trace-based compilation generates code quality that is much worse than HotSpot on the x86, but achieves better performance on ARM. Finally, our experiments also reveal effective JIT compilation parameters for the Dalvik VM, and their effect on benchmark performance and memory usage.

Contents

List of Figures

Chapter 1

Introduction

Android is an open-source operating system (OS) initially developed by Android Inc. and Google. It is currently one of the most popular OS for touchscreen-based mobile devices, including smart-phones and tablet computers. Android is built on top of the Linux operating system kernel, and allows developers and manufacturers to freely modify and distribute the OS for their specific devices. Thus, close variants of the Android OS are currently deployed by several device manufactures, including Motorola, Samsung, HTC, and Google.

The Android OS does not allow *native* execution of any programs/applications. Instead, applications are developed in the Java language [9], and are securely executed within a *sand-boxed* process virtual machine (VM) [20]. This virtual machine used by Android is called *Dalvik*. Thus, the performance, usability, acceptance, and competitiveness of the Android platform is intimately linked to the quality of the run-time environment facilitated by the Dalvik VM.

Given this importance of the Dalvik VM, it is important to have an accessible, efficient, and easy to use framework to study and change the code and properties of the Dalvik VM, and measure (the changes to) its performance characteristics.

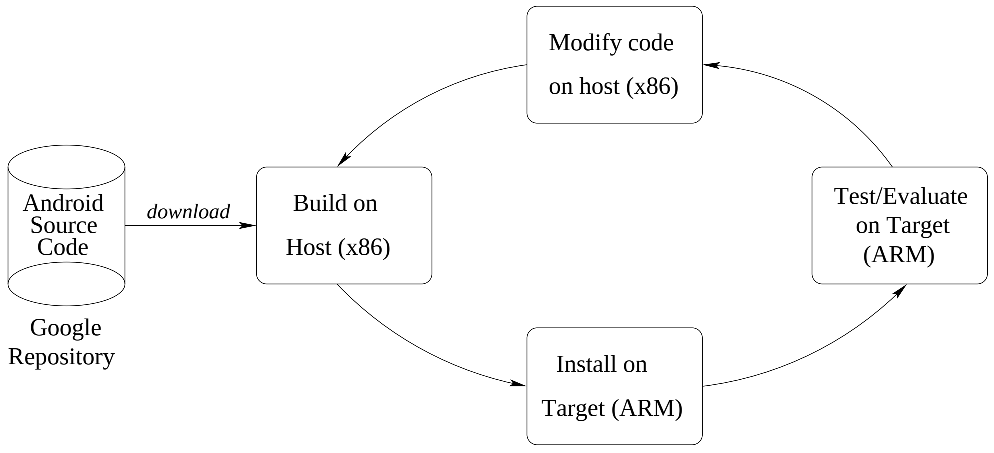

Figure 1.1. Building and evaluating Android OS or Dalvik VM

However, the Dalvik VM is not distributed as a stand-alone component, and can only execute within the context provided by the Android OS. Likewise, the Android open-source community provides no instructions for only building/installing the Dalvik VM sub-component on the host machine.

An even greater challenge is caused by the existing Android build process that is focused solely on installing Android on the supported target devices. Currently, the tasks of compiling the Android source code files and building the OS image need to be performed on x86 machines. This OS image (embedding the Dalvik VM) is then installed on the corresponding hardware device. We can then install and run the applications on the target device. Thus, there is no existing documentation to run the Dalvik VM *natively* on x86 (Linux) machines (without first installing the Android OS). While there exists an indirect mechanism for *emulating* the Android OS on x86, accurately measuring the performance characteristics of Dalvik benchmarks is very hard (and slow) in an emulated environment.

Figure 1.1 illustrates performance evaluation process enabled by the documented instructions for building, installing, and running the Android OS. Building

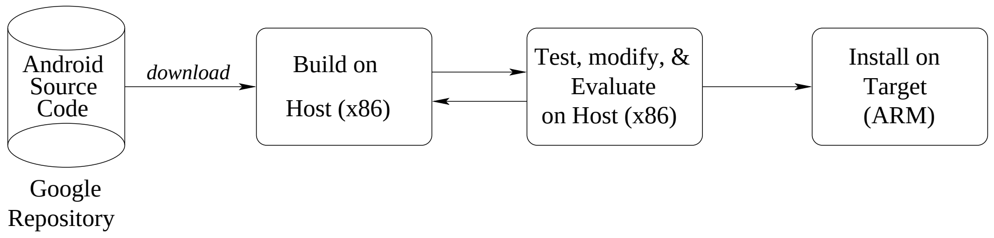

Figure 1.2. Our modified process for building and evaluating Android OS or Dalvik VM

and Installing the updated version of Android after change to the Dalvik source code is highly tedious and time consuming. It is clear that this process is very clumsy, frustrating and slow. Therefore, our first goal for this project was to determine and provide an environment to run the Dalvik VM natively on x86-Linux machines, without the need to first install the Android OS. We have now constructed this framework, which allows us to quickly and effectively build and evaluate the performance characteristics of the Dalvik VM in a similar fashion to any other Java virtual machine. An illustration of our current Dalvik VM evaluation framework on x86-Linux is presented in Figure 1.2.

The other goal of this project is to understand the internal organization and implementation details of the Dalvik VM, quantify its performance characteristics, and study the effect of modifying the VM properties on overall program speed and memory consumption. We perform an extensive *white-box* evaluation of the Dalvik VM, and compare its performance with the sophisticated HotSpot Java virtual machine [13] on both the x86 and the ARM. Our study complements and expands other recent Dalvik VM evaluations [17] by employing a wider selection of benchmarks, exploring different aspects of the Dalvik VM, and conducting the evaluation on two dominant processor architectures (x86 and ARM), instead of just conducting the analysis on the ARM platform.

Thus, the successful implementation of this project enables more comprehensive updates and evaluations of the Dalvik VM in the future, as well as indicates the direction to take to improve future Dalvik VM performance. The rest of this project report is structured as follows. In the next chapter, we describe relevant background concepts and present related works. In Chapter 3 we present the steps we take to construct our stand-alone evaluation framework for the Dalvik VM on x86. We describe the results of our evaluation of the Dalvik VM performance in Chapter 4. We describe some future research to extend this project in Chapter 5 and present our overall observations and conclusions in Chapter 6.

Chapter 2

Background and Related Work

Our project needs familiarity with several technical concepts, including just-in-time compilation, staged compilation, profiling, etc. We explain these relevant background concepts in this chapter. We also present details regarding the Android OS architecture stack and the placement of the Dalvik virtual machine in this stack. Additionally, we also discuss related research results from other projects evaluating Dalvik VM performance.

2.1 Just-in-Time Compilation

Managed languages, such as Java, Java-script, and C#, were designed with the *compile once run anywhere* philosophy. Statically at compile time, programs written in such languages are converted into a platform-independent *intermediate* representation. Since the binary distribution format for such programs is specially designed to be platform-independent, they can portably execute on all platforms that provide a run-time environment (or virtual machine) for their execution. Such portable program execution is extremely important for applications distributed over the Internet.

Program emulation performed by the VM can take the form of interpretation or binary translation [20]. However, interpretation is inherently slow. Therefore, most of the performance-aware VMs also incorporate a module to compile the program to native code before its execution at run-time. Since this compilation occurs at run-time (in parallel with program execution) it is called dynamic or *just-in-time* (JIT) compilation [2].

Since JIT compilation happens at run-time, it can adversely affect overall program performance and cause non-deterministic application pauses if performed injudiciously and especially on single-core machines. The ParcPlace Smalltalk VM [5] followed by the Self-93 VM [11] pioneered many adaptive optimization techniques to reduce such adverse effects of JIT compilation. For example, the technique of *selective* compilation uses online profiling to detect and only compile the subset of *hot* program methods in a program. [1, 11, 14, 19]. This technique is based on the observation that most applications spend a large majority of their execution time in a small portion of the code [1, 3, 12].

Thus, selective compilation reduces compilation overhead while achieving application efficiency gains comparable to full compilation. Most VMs deploy selective compilation with a *staged* emulation model [10]. With this model, each method is initially interpreted or compiled with a fast non-optimizing compiler at program start to improve application response time. Later, the virtual machine attempts to determine the subset of hot methods to selectively compile, and then compiles them at higher levels of optimization to achieve better program performance. Both the Dalvik VM and HotSpot Java VM employ selective compilation and staged emulation.

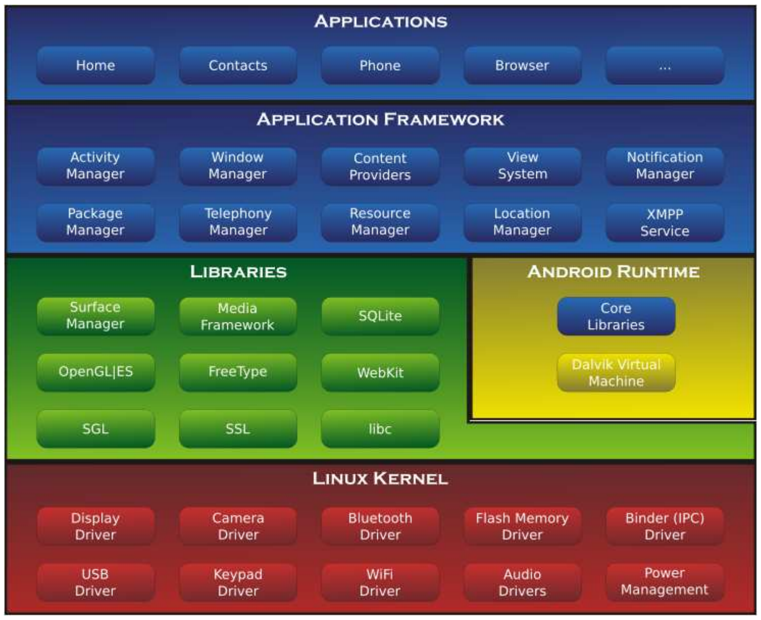

Figure 2.1. The Android operating system architecture. [8]

2.2 Android Platform Architecture

Figure 2.1 shows the system architecture and various components of the Android OS software stack [7]. The Linux kernel functions as the base layer for Android that interacts with the hardware. Android's native libraries in the next layers are specific to each device platform and are written in C/C++ and compiled to native code. The Dalvik VM provides the run-time environment for Android, and is the most important component for the purpose of this project. The Dalvik VM is a variant of a Java virtual machine, but does not run Java *classfiles*. Instead, it converts the stack-based classfiles into a register-based intermediate format, called the *dex* files. Some more details about the Dalvik VM and *dex* files is presented in the next section. Finally, the higher layers include the application framework that provide tools used by the applications in the top-most layer of the Android OS architecture.

2.3 The Dalvik Intermediate Code and Virtual Machine

DalvikVM is a Process Virtual Machine that instantiates an application, optimizes and verifies it in an individual execution environment for each process. [8] It uses a register based IR, which uses registers to encode the instructions in the intermediate binary file. In contrast, JVM is a stack based IR that uses the stack to decode the instructions. JVM opted stack based architecture because that gives a simple implementation for a virtual machine. But a register based VM has the advantage of reducing the number of binary instructions executed as it does not need to load each operand to and from the stack. But a register based IR has the overhead of encoding more operands in each instruction, which increases the static binary file. DalvikVM made the instruction set more precise by fetching two instructions per cycle making each line 25% larger in size. Overall this helps to improve the efficiency of the program execution relative to the stack based VM. DalvikVM opted for a register based model mainly because of its intrinsic performance advantage over the conventional stack based model used by JVM. Moreover byte code verification is much faster in a register based VM, where the integrity checks are highly simplified. Additionally, malfunctioning code due to transmission or storage discrepancy, it is easier to handle in a register based VM rather than in the stack machine. Dalvik also reduces the executable size to minimize the memory footprint, which is one of the key factors in embedded operating system like Android. DalvikVM is designed to operate on custom made byte code which is stored in a specific format called dex file rather than the conventional Java byte code. DEX files is interpreted and optimized for each application during the install and first run. Since the Android uses the java coded application, the Java class-file has to be converted to the Dalvik recognized DEX file format for

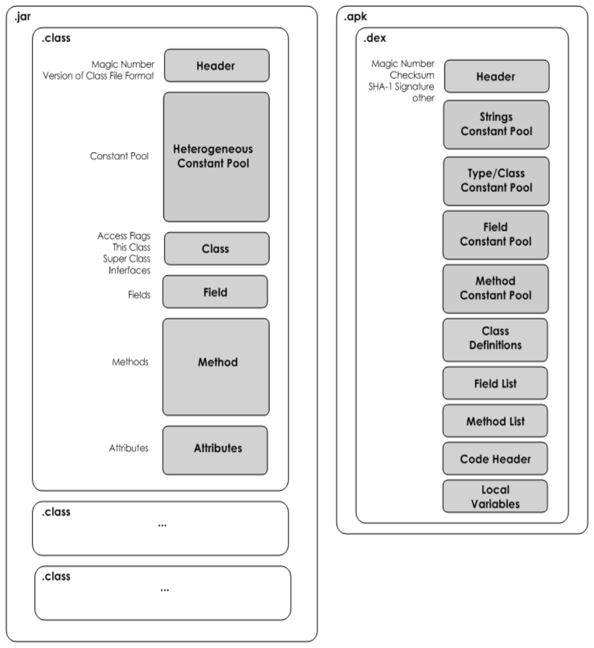

Figure 2.2. The Comparison between DEX and CLASS file format [8]

execution. The dex tool is used to convert the java byte code zip/jar files to the DEX files. The dex utility command to convert the Java butecode class-file is:

```
dx --dex --output=example.jar example.class
```

DEX file has individual constant space as shown in the figure 2.2 which helps to minimize memory footprint. This implicit typing helps to reduce the method signature redundancy. dx is capable of aggregating various DEX files into a single DEX file format. DEX files are made read only and they are shared across the applications by the inter-process communication handle of Android called binders. Byte code alignment, verification and the possible optimization are done well ahead and stored as the optimized DEX file in the dalvik-cache for faster run time execution. So this optimization is done 'just in time' when the application is first run after the installation. The Android system creates this optimized DEX file and stores it to the dalvik-cache in 'system' folder. This folder can be found in the $ANDROID_DATA/data/dalvik-cache. The dexopt tool is used to do the byte-code validation and alignment before executing the code from the DEX file format. The verifier is run to check whether the DEX file has valid legal instructions. This check avoids run time error due to misalignment and code sizing problems. Verification is more important because the interpreter can avoid lots of potential error scenarios if checked by the verifier before hand. Verification further helps to improve the inter application security as Dalvik runs in a sandbox. Since the Android ecosystem allows the sharing of certain libraries across the application space there could be problems of concurrency. So this modules are made read only to increase the efficiency to enable fast access by the Zygote layer. The necessary file that has to be rewritten is handled separately using the file lock mechanism.

2.4 Related Works on Evaluating Dalvik VM Performance

DalvikVM is a nascent virtual machine that has been developed mainly to improve the Android run time performance which runs on ARM. DalvikVM was first introduced with the interpreter only implementation in the older Android versions. Later with the Android 2.2 Froyo release, it also included the Just-in-Time Compiler (JITC) to improve program performance. Researchers have concluded some preliminary studies of Dalvik VM to compare its efficiency with other existing Java virtual machines for standard benchmark programs. A blog written by Oracle employees compares the performance of Dalvik with its own

Java SE implementation. [18] Android and Java run time have different library implementation, so many benchmarks do not run on both of these target VMs. Since Android and Java do not share same graphics library, benchmarks using graphics API have to be avoided. Additionally the Android version of Linux does not support *glibc*. After resolving all such compatibility issues their early study evaluated the performance of benchmarks like SciMark, CaffeineMark 3.0 , Tegra2, kBench on Dalvik and Java SE virtual machines. A developmental ARM beagleboard was used to run the benchmarks. They observed that Java SE virtual machine achieves 2 to 3 times better program run-time compared to the Dalvik VM.

Another evaluation of Android Dalivk VM was done with the Hotspot implementation of java virtual machine. [17] The Android Gingerbread Version 2.3 and an implementation of HotSpot called PhoneMe was used to run some benchmarks. Both the virtual machines are installed on the same board and to evaluate them. Benchmarks from Embedded Microprocessor Benchmark Consortium (EEMBC) was used. The observation from these tests showed that the Dalvik interpreter slightly outperforms the HotSpot Interpreter. However, when both the virtual machines are run in the JITC mode, HotSpot performance efficiency is about 3 times compared to the Dalvik. Moreover, they found that Dalvik generated code size is larger due to worst code quality and trace chaining. All these experiments only used the ARM target. Since Android is gaining more popularity in tablets and porting of Android to x86 is done by many enthusiasts. we in this thesis, also compare the performance of Dalvik with HotSpot on x86.

Chapter 3

Environment to run Dalvik on x86 and ARM

Dalvik, a process virtual machine is an integral part of the Android OS. We configure the entire development environment of the latest Android Open Source Program (AOSP) version 4.1 called *Jelly Bean*. The AOSP is forked from `https://source.android.com/source/downloading.html` as per the given instructions. The first step is to install a concurrent version management tool called *Repo*. Repo helps to fetch a particular branch of the Android source tree by using the *repo sync* option with the master branch details. Since the code is huge it takes a significant amount of time to download the source files which totals about 13GB in size. Then, the required dependencies like the python run-time and other tools are configured for the host system on which the code is to be built.

The Android source is built after updating the environment using the provided `envsetup.sh` script. This script has the necessary Android build paths and the toolchains that are required for the host machine and the build target. The required target build can be chosen by using the command called *lunch*. This script will prompt the user with the options to build the target for the emulator, ARM, x86, Virtual Box, and development boards like pandaboard, etc. This script also supports enabling the debugging mode on the target. For instance, the *full-eng-x86* option builds for the x86 target with debug mode enabled. This selection can be further narrowed by allowing more user customization by using the command *choosecombo*. This command will prompt the user for a build type selection (release, debug), followed by the target product and architecture and its variants. This script will prepare a configured parametrized file that will be used to build the Android environment for the target accordingly to the selected options. Finally, the source can be built by using *make*, which can also enable multi-core CPUs to be simultaneously involved in the build process.

After the environment is built, Android along with the Dalvik VM sub-system, is packaged as an image file for the target installation. This process entails many checks for building and validating the Android source with the various target specific system parameters for the specified target architecture and the product. As mentioned earlier, this update-build-install cycle takes a long time. So any changes done to the Dalvik VM require the same procedure to be repeated every time.

Obviously, a re-compilation with *make* for only small code updates significantly reduces the amount of time required to rebuild the changed code, provided that target architecture and the product configured remain the same. The Android OS image is then installed on the target device. The Dalvik VM has to be instantiated for a particular process and its functionality has to be checked to see the particular feature change works as intended. Since the Dalvikvm is targeted to the ARM machine, this whole setup has to be ported to the target hardware for testing.

After building, the target images like system.img, boot.img, userdata.img are located in the `out/target/product/` directory. These images have to be copied and installed to the target machine in the necessary file partition format like `/system` and `/data`. The images have to be transferred to the Pandaboard in a specific file format with manual configurations of the partitions. Creating the proper partitions, like `/system`, `/data`, `/sdcard` again takes a long time and needs to take care of many subtle details. Similarly, the process of changing the Dalvik source code and then porting it to the hardware, followed by testing and verifying the functionality is very time consuming and tedious.

Therefore, running and testing the Dalvik functionality on the x86 emulator is a viable solution. The Android OS build can be targeted to the emulator and the functionality can be checked using a particular application .apk. The applications can be either installed on the emulator via Google play or similar media or can be copied to the `/system/bin` folder. Then a copy of ODEX (optimized DEX file) has to be updated to the dalvik-cache to install it manually. The emulator can interface with the *abd* tool and the activity can be monitored and logged on to the *logcat*. Unfortunately, this emulator configuration cannot be used for performance evaluation. To reduce this tedious work process, we have now simplified the test process by configuring Dalvik to run on native x86. After testing, the final configured versions. Only final version can then be ported to the target device (ARM).

Once the Android environment is configured the rebuild time can be substantially reduced by just making and building the Dalvik source and its dependent library. The following script snippet cleans the Dalvik dependent library and then makes the *libdvm* and *dalvikvm* libraries.

```
make clean-libdvm clean-libdvm_assert clean-libdvm_sv clean-libdvm_interp
make -j4 libdvm
make -j4 dalvikvm
```

After this step, Dalvik can be natively tested on x86 and iterated till the final version and then ported to the target making the process relatively simpler.

3.1 Standalone Execution of DalvikVM

The Dalvik virtual machine can be run as a standalone entity without the Android OS environment when some dependent library files and environment variables are exported. The standalone Dalik VM is capable of instantiating a particular application and executing it in its own environment. This setup can expedite the developmental process of Dalvik.

After making the necessary code changes the Dalvik VM module alone can be built and compiled. Then, the same can be executed in native x86 environment to verify and validate its correctness. This will improve the development process reducing the time spent on administrative work. This process helps to make a simplified setup environment, and can be used to optimize a particular application on Dalivik VM or developing more Dalvik features on top of the existing code.

This initialization script is configured to export and set the necessary *bootclass-path* and other environment variables to execute the Dalvik VM in the standalone mode.

```
#!/bin/sh
# Base directory, at top of source tree is replaced
# with absolute path.
# Configure root dir of interesting stuff.
export LD_LIBRARY_PATH=/vendor/lib:/system/lib
root=‘pwd‘
export ANDROID_ROOT=$root

# configure bootclasspath
bootpath=/system/framework

export BOOTCLASSPATH=/system/framework/core.jar
:/system/framework/core-junit.jar
:/system/framework/bouncycastle.jar
:/system/framework/ext.jar
:/system/framework/framework.jar
:/system/framework/telephony-common.jar
:/system/framework/mms-common.jar
:/system/framework/android.policy.jar
:/system/framework/services.jar
:/system/framework/apache-xml.jar

# This is where we create the dalvik-cache directory;
# make sure it exists
export ANDROID_DATA=/tmp/dalvik_$USER
mkdir -p $ANDROID_DATA/dalvik-cache

exec ./bin/dalvikvm -cp /home/go/vm/hello.dex Hello $@
```

The Android root is the base directory for the deployment, and the base framework and library files are expected to be located in fixed directories starting from this root. Every other file reference is relative to this location. The dependencies of the *bootclasspath* varies from device to device. These dependencies can be found from the `init.rc` file located `/out/target/product/generic_x86/root/init.rc` in the target source. The Android data comes into picture when the DEX files are optimized and stored during the first run of the program. The `dalvik-cache` is the directory inside the Android data which is exclusively accessed by the application at run-time. This ODEX helps to save more memory space by storing the optimized version of the byte file and also drastically reduces the application initialization time which is of more importance in the embedded systems. These optimizations focus on reducing the power consumption and improve the battery backup, while lowering the memory utilization.

3.2 Dalvik Debugging Environment

Dalvik VM allows source level debugging with any tool that supports the Java Debug Wire Protocol (JDWP). One such debugger used here is jdb. Debugging terminal can be used to instantiate the debugging over the TCP via the adb shell for remote debugging. The Dalvik VM has implementations of the Dalvik Debug Monitor, which exposes the source of Dalvik to hook it up for detailed analysis whenever needed. The JIT profile tracking can be closely monitored using such a debugger in real-time. The debug server is made to run and listen on the given port 8000, and then the Dalvik VM is hooked to the running application. Then, the client is attached to the listening port to start recording the debug messages and to create traces for later analysis.

This is how a jdb debugger can be attached to the running program in Dalvikvm.

```
run -agentlib:jdwp=transport=dt_socket,address=8000,suspend=y
,server=y -cp /home/go/vm/bin/loop.dex looper $@
jdb -attach localhost:8000
```

Dalvik can also execute under well know debugger, like *gdb*, for source level debugging. The debugger is a separate module in Dalvik VM that implements the generic JDWP protocol. Any more changes to those files can be done as needed. The source location for this debugger is */dalvik/vm/jdwp*. The Debugger.cpp file in the `dalvik/vm` is the bridge point to connect the debugger to the DalvikVM.

3.3 Important Dalvik VM Files

Some important Dalvik virtual machine files we used and updated include:

1. */dalvik/vm/Globals.h* file is the global header where the necessary variables are declared to retain the global scope across the library. DvmGloblal, DvmJitGlobal struct maintain the state wherever the vm starts or shutdown by tracing state parameters.

2. */dalvik/vm/Init.cpp* file is used for the initialization and shutdown of the virtual machine and as well as handling the command line arguments during the invocation. This is the file used to expose the command line arguments to the user initialization.

3. */dalvik/vm/Thread.cpp* is used to manage the system threads that are implemented as native pthread library. This is where Dalvik spawns compiler threads, GC threads and other system threads and associates them with the

thread structure created. There are a total of seven threads in the Dalvik VM that do the intended work.

4. */dalvik/vm/compiler/Compiler.cpp* is where the compiler en-queues the work and where the code cache is filled with the trace recorded. The code cache allocation and reset are also done here. Profiling is tracked by the JIT profile table as and when needed.

5. */dalvik/vm/interp/Jit.cpp* file implements the trace based JIT, which records the trace as code cache when the necessary criteria are met (threshold). The Jit traces are recorded in two levels.

The JIT threshold parameters and the code cache size are initialized to default values if not set earlier in the following files.

1. */dalvik/vm/compiler/codegen/x86/CodegenInterface.cpp* this file has the target specific configuration for the x86 machines. This also has the trace profiling methods.

2. */dalvik/vm/compiler/codegen/arm/arm7-a/ArchVariant.c* this file has the target specific configuration for the ARM machines. There are four version of ARM where the ArchVariant.c has to be updated accordingly. This also has the trace profiling methods.

3.4 Using Linaro image tool

Linaro is the Android alliance formed from the partnership of the hardware manufactures and the software developers primarily to reduce the fragmentation of Android [16]. The Android fragmentation is a serious problem for the developmental community that needs to be addressed to make the Android OS a better ecosystem to increase the flexibility and resilience among varied devices where it is used.

Linaro's image tool helps us to move the Android OS images to the sdcard with the pre-configured default memory layout. This layout can be customized by altering the code in `linaro_image_tools/media_create/android_boards.py` if necessary. So our files produced for the specific target can be easily moved by utilizing this tool. Example Linaro image tool usage to format the SDcard using the android images is shown here.

```
linaro-android-media-create
--mmc /dev/mmcblk0
--dev panda intel jelly bean android x86
--boot ./out/target/product/pandaboard/boot.tar.bz2
--system ./out/target/product/pandaboard/system.tar.bz2
--userdata ./out/target/product/pandaboard/userdata.tar.bz2
```

The adb shell provides a way to interact with the Android board file transfer and further debugging and necessary data gathering from the system remotely. A physical connection can be used to obtain an Android terminal. Then interaction with the system is done over the USB serial cable. Any terminal emulator like *minicom* can help to establish this connection.

3.5 Compiler Code Flow

The interpreter code of the Dalvik virtual machine is written in both the C and assembly languages. The assembly level implementations are intended for better performance. Each routing in the assembly is crafted to fit in the same memory bounds to be more efficient. The interpreter is very fast and frequent checking of the traces for the compilation criteria will reduce its robustness. So the compilation criteria are only checked when a particular trace is executed N number of times. The N is the threshold value which is used to mark the trace to the compilation queue. For x86 architecture the default value for this compilation threshold is 255. The different ARM boards have a range of values from 40 to 200 accordingly to the specific memory and processor configurations.

The `/dalvik/vm/mterp/x86/footer.S` is the assembly file that is used to jump to the JIT from the interpreter [6]. The interpreter is designed to select the mostly hot traces as well as some normal traces . Then the hash table which tracks the profile count of each trace and records its PC value when the criteria is met. So, on the next trigger this hash table is checked for the PC value which indicates the translation is already requested for a particular trace. If there is a hit this goes to the next level of the translation process where the traces pass the second level filtering. If there is a miss, then the interpreter control is passed back to the trace selection mode. This is done in the small code chunk called *common-update-profile* in the `footer.s`. So, this code triggers a reset of the counter associated with each trace to zero.

This translation request queue is intentionally less aggressive, meaning there are many chances for non-hot traces to be marked as hot. To ensure only the hot methods are chosen correctly from the second level a check is done here. These second-level checks happen during the positive scenario where the trace is hot and needs to be translated and the recorded trace executes correctly without any exceptions. If there is any exception in the execution of the trace the JIT code has to be returned back to the interpreter in the normal trace selection mode. To move from the JIT code cache to the interpreter, the VM can take two frequent paths named *dvmJitToInterp**. One of the path is *dvmJitToInterpNormal*, which is used based on the profiling threshold when the code has a conditional branch. The other path would be *dvmJitToInterpTraceSelect* method, which is chosen during unconditional branch or a direct invocation.

So after a trace is passed from the level one criteria it is marked for the compilation in the hash table which is indexed with the PC value of the trace being requested. So this is the decision making criteria for the level two trace selection to enter the compilation queue. Now the traces entering this level are entered into the hash table and the control goes back to the interpreter for further recording of the traces. This is done in the *dvmJitCheckTraceRequest* method in the `/dalvik/vm/interp/Jit.cpp` file. This method uses a Filter-key which serves as the filter head to process the selective trace to compilation. The filter key is the combination of the two values of the filter criteria, the Program Counter (PC) and the method pointer. If the target system architecture and the type of the application run on it is already know these values can be tuned and tailored for particular execution profile.

At the same time the type of the execution profiles can be varied. It can either be configured for spiky or the flat program profile. The spiky execution profiles have small number of selective hot methods/traces that need to be cached and compiled. As this criteria requires the selection process to be more stringent and selective, a perfect match of the PC value of the trace is required. So these will have better performance with the PC key.

```
intptr_t filterKey = (intptr_t)self->interpSave.pc value
```

The other execution profile is the flat scenario where many methods are hot in a short period of time. This profile requires more methods to be eventually compiled. So a heuristic solution to this problem would be to find all the possible hot traces in the particular method and compile them all anticipating that they will be required in the future. This requires the method pointer to be selected, and all the traces within it are selected and compiled as Filter-key.

```
intptr_t filterKey = (intptr_t)self->interpSave.method
```

But, the applications used can have varied profiles and we need a combination of both to find an optimal heuristic value for overall performance gain during JIT compilation. For the existing filtering scheme, the higher order bits are chosen from the method pointer and the lower order bits from the Dalvik PC value. This would help us to get the average performance gain for both sets of applications. The `JIT_TRACE_THRESH_FILTER_PC_BITS` is used to select the filter criteria by changing the Dalvik PC slice. The wider the slice, the lesser the compilation. The default value of this Dalvik PC slice is 4. In our experiments, we vary this number from 0 to 32 to check for the various other possible configuration runs and their effect on performance. Choosing just the method key as the filter will be beneficial for the flat execution profiles. On the other hand choosing just the Program Counter(PC) will be beneficial for the spiky execution profiles as this will make the selection more specific to the hot traces.

Finally, the recorded trace is added to the compile queue. In short, until this point the trace which is recorded is profiled to a value 2N+1 times where the N is the target threshold value. Later, the compiled code is added to the code cache. The code cache is by default set to the 512k in Android. This value can be altered to accommodate more compiled code in the cache. The upper bound

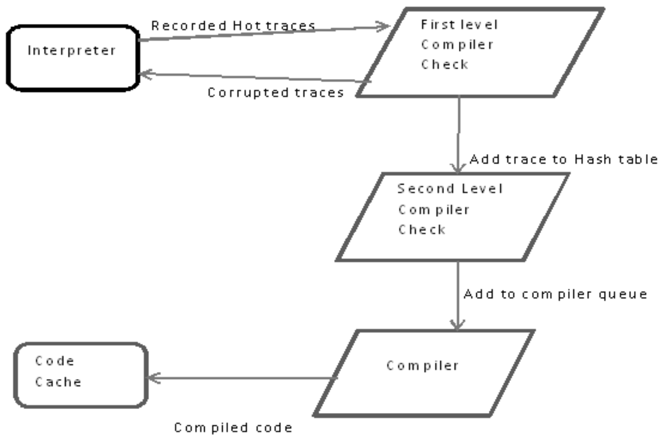

Figure 3.1. Overall Compiler Code Flow.

of the code cache size is set to make sure that each process does not use much memory for just caching hot traces. This value can only be changed in accordance with the maximum heap size allowed. All possible combinations are tried to find the optimal threshold value that can be configured for a particular target. To do this dynamically we need the profile information before hand. A high level control-flow of the JIT compiler is illustrated in Figure 4.1.

3.6 Dalvik Command Line Arguments

Dalvik can be instantiated by passing certain parameters during program start-up, which changes its behaviour accordingly. The following flags recognize more generic arguments. The classpath (-cp) flag is used to indicate the location of the dex file to Dalvik. Other flags are used to alter the heap size during startup: *-XmsN* (min heap, must be multiple of 1K, >= 1MB) and *-XmxN* (max heap, must be multiple of 1K, >= 2MB). Dalvik can be indicated to run in three modes by setting these flags: *-Xint* (extended to accept ':portable', ':fast' and ':jit'). Usually the default Dalvik executes with JIT compilation enabled.

There are other arguments unique to Dalvik which alter the JIT profiling, code checking and validation etc. For example, *-Xjitverbose* flag is used to log the JIT profile stats in verbose mode for analysis and further optimization purpose. This has to be coupled with exporting the ANDROID_LOG_TAGS during the execution. The ANDROID_LOG_TAGS can be customized for each application that runs. The log level can be changed to a specific demon or application as follows:

```
ANDROID_LOG_TAGS="ActivityManager:I MyApp:D *:S".
```

The *-Xjitthreshold:decimalvalue* flag can be set in accordance to the target machine as needed. Additionally, there are many other arguments that can help to optimize a particular method by its signature, class name etc.

We added some command line arguments to access the JIT compiler by changing the threshold and filter criteria of JITC on the go. These include

1. *-XjitfiltermethodKey* enables to set the Filter-key as just method pointer. This allows the VM to execute the whole method containing traces that are not yet labelled as hot. This heuristic helps to compile more trace assuming that running application as the flat execution profile.

2. *-XjitfilterpcKey* enables to set the Filter-key as the trace's program counter, which selects particularly the relatively hot traces that stand apart from

others. This Filter-key will be more effective on applications having the spiky execution profile.

3. *-Xjitdpcslice:decimalvalue* alters the size of the Dalvik PC slice that has to be selected for the Filter-key. The default value is 4. The lower bound is 0 and the upper bound is 32. This can be altered in this range to chose an optimal filter criteria for the second level JITC parameter.

4. *-Xjitdisablel2* disables the level two trace selection completely. So this means that the traces are profiled only N+1 times before it is sent to the compiler queue for caching.

The max heap used for the execution is calculated from pooling the current heap size used in every run periodically. This value is stored in the global space allocated in the *DvmGlobal* structure. To measure the compiler execution, garbage collector, and VM thread run-times, we add a code snippet to get the thread ID during their assignment from the `Thread.cpp` file. The IDs are stored in the library scope `Globals.h`. When these threads are signalled for termination, each thread's time is greped from the `/proc` file system for the execution time. All the threads are signalled before the garbage collector thread that is done later in time. The Dalvik VM shutdown path is traced and the run-times are calculated right before each thread termination.

Chapter 4

Assessing Dalvik VM Performance

In this chapter we report our observations from experiments comparing Dalvik VM performance to that on Java Hotspot's virtual machine on both the x86 and ARM architectures. We notice that the results differ quite significantly on these two architecture. We suspect that this difference is on account of Dalvik being optimized more for the ARM, which is its primary target architecture.

4.1 Experimental Environment

In this section we describe the hardware, OS, and benchmark setup we employ for our experiments comparing the Dalvik VM with the standard HotSpot JVM.

4.1.1 Hardware/OS Setup

For building the Android OS from source we use a 64-bit Intel Core i7-2630QM CPU @ 2.00GHz 8-core x86 machine running the Linux Ubuntu 12.04.2 LTS

(Precise Pangolin) operating system. This machine has a total installed physical memory of 6GB.

For our experiments on the ARM architecture we employ a developmental ARM 'PandaBoard ES', which has an ARMv7 dual-core CortexA9 processor with 1GB DDR2 physical memory. The pandaboard has on-board 10/100 Ethernet, 2x USB 2.0 High-Speed host ports for IO, and an HDMI video out port. The pandaboard is configured to run both the Android as well as the Ubuntu OS. On ARM-Ubuntu we use the HotSpot JVM (Java version 1.6.0_27) that is installed by default on Ubuntu 12.04.1. We built the Android OS (with all our Dalvik VM updates) from source on the x86, and then install this updated version of Android on the pandaboard by using OS image installation tools distributed by Linaro [16].

We conduct all our x86 experiments on a single dedicated machine. This machine has an 8-core x86 64-bit processor with 2GB of physical memory.

4.1.2 Benchmark Setup

To accurately compare the Dalvik and HotSpot virtual machines we use and adapt Java programs from standard Java benchmark suites, including Caffeine [4], SPEC jvm98 [22], and SPEC jvm08 [21]. To correctly run these benchmark programs on the Android Dalvik platform, we convert each program into the Android compatible DEX file format by using a tool called dx. The DEX jar file includes all the internal DEX files and manifest files information needed for identification of items inside the archive. The CaffeineMark contains benchmarks that run smaller (*sieve*, *loop*, *logic*, *method*, *float*, and *Graphics*) programs. The overall score is the Geomean of these values. From the jvm98 suite we use the *202_jess*, *205_raytrace*, *209_db*, *213_javac*, *227_mtrt* and *228_jack* benchmark programs. From jvm08 we use the *crypto.signverify*, *scimark fft*, *scimark monte_carlo*, *scimark sor* and *scimark sparse* programs. For our startup runs, each benchmark is run 10 times and the average time is reported. For each comparison experiment we plot a specific Dalvik VM performance property as compared to the same property on the HotSpot Java VM. So, if the result of the comparison is exactly one, this implies that both VMs deliver comparable performance.

4.2 Performance Comparison of Hotspot and Dalvik Virtual Machines

Interpreter performance is an important aspect for VMs employing selective just-in-time (JIT) compilation. For such VMs all program code needs to be interpreted until it is selected for compilation. Given this importance of interpreter performance, both the Dalvik and HotSpot interpreters are written primarily in assembly, with some C language code. In this section, we first evaluate the performance of Dalvik's interpreter. For these experiments we disable JIT compilation in both the HotSpot and Dalvik VM, and then compare the program run-time in interpretation mode on both VMs on the x86 and ARM architectures.

Figure 4.1 compares program run-times on the Dalvik VM with their corresponding run-times on the HotSpot VM. In this graph, any bar-plot lower than one implies that Dalvik achieves a performance gain over hotspot. On the other hand, bar-plots greater than one indicate poorer Dalvik VM performance. The chart shows the interpreter comparison results for both X86 and ARM architectures.

We find that the Dalvik interpreter is very competitive, and is able to improve upon the highly optimized HotSpot VM for all our programs. This improvement may be a result of the register-based intermediate format (IR) used by the Dalvik DEX code (as opposed to the stack-based IR used by HotSpot). The interpreter performance is dominated by the issue, decode, and dispatch Intel jelly bean android x86 overhead for each executed instruction, and is affected less by any other optimization. A three-address register-based IR is able to express the same computation is much fewer instructions compared to a zero-address stack-based IR. Therefore, by reducing the number of interpreted instructions, the Dalvik VM is able to improve interpreter VM performance.

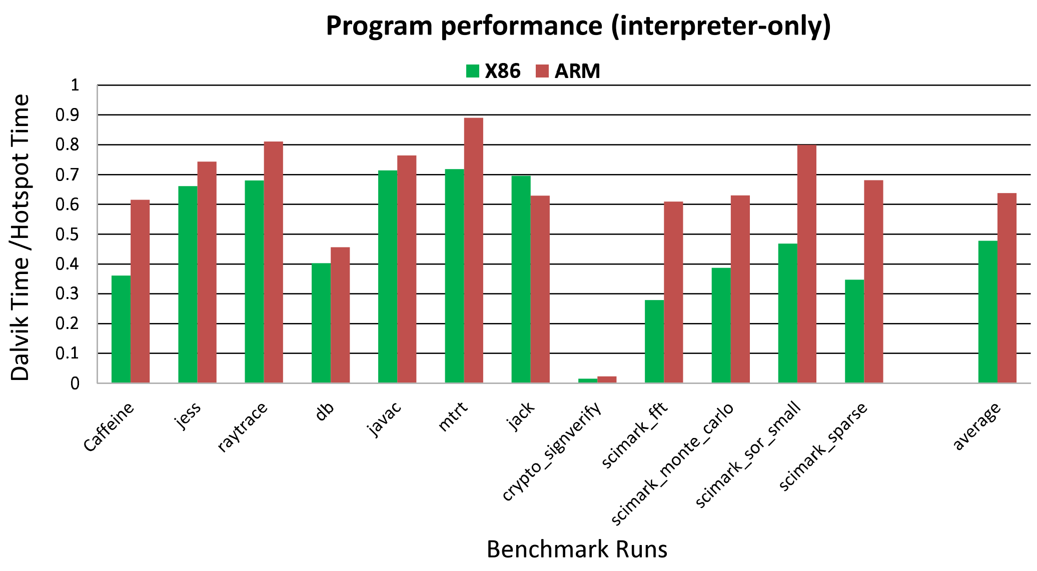

Figure 4.1. The Interpreter Performance of Hotspot and Dalvik.

Figure 4.2 shows the benefit of JIT compilation to program start-up performance. This experiment compares the program run-time using the Dalvik VM with its interpreter-only configuration to the program run-time with the default

Dalvik VM that compiles and optimizes the hot program traces dynamically. As expected, we find that several applications see a significant positive performance impact with JIT compilation enabled. This experiment also shows that both the x86 and ARM architecture targets witness a similar performance gain for each benchmark program. On average, the performance benefit ranges from 44% to 50% on the x86 and ARM respectively.At the same time, it is surprising to find that many applications, including *javac*, *mtrt* and *jack*, do not benefit much from JIT compilation.

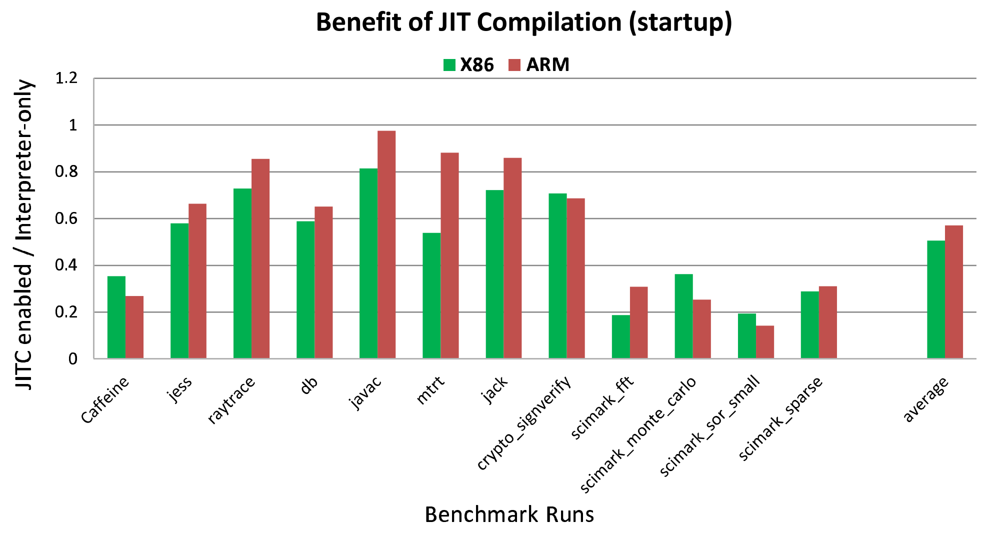

Figure 4.2. The Dalvik Interpreter versus Compiler performance.

Figure 4.3 compares the steady-state program run-time (with JIT compilation enabled) with interpreted performance for Dalvik on x86.Steady-state time measures the program run-time after all/most compilation activity is over. For our steady-state experiments we run 5 iterations during each benchmark run and the time of the last iteration is considered as the program run-time. Interestingly, we find that with the Dalvik VM on x86 the program run-time is not much different in the start-up and steady-state modes. This result suggests that compilation of hot methods happens very early in the Dalvik VM. Additionally, since compilations happens in its own thread (which will get its own dedicated core on our math-core test machines), it does not interfere with application performance.

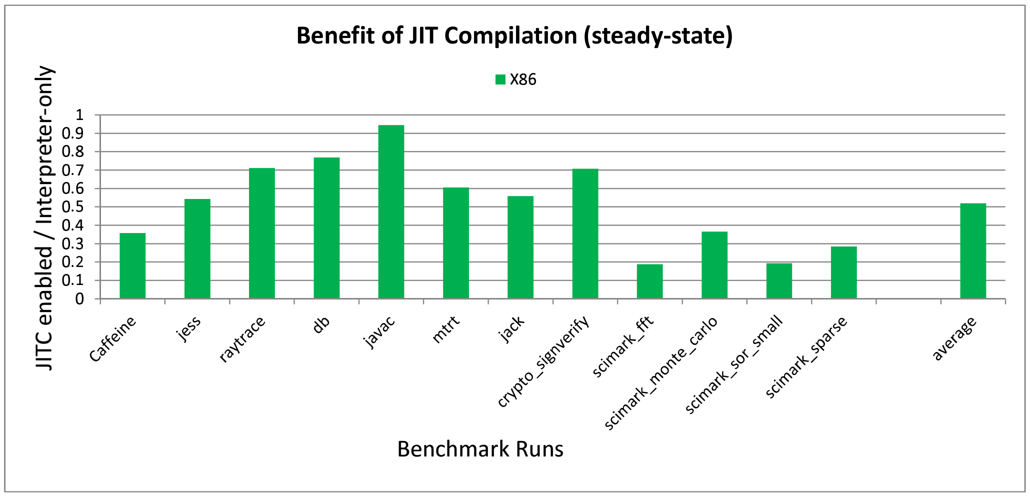

Figure 4.3. The Dalvik Interpreter versus Steady State Compiler performance on X86.

Figure 4.4 illustrates the results of experiments comparing the program start-up performance with the default Dalvik VM configuration (JIT compilation enabled) to the default HotSpot setting on the x86 and the ARM. These results are

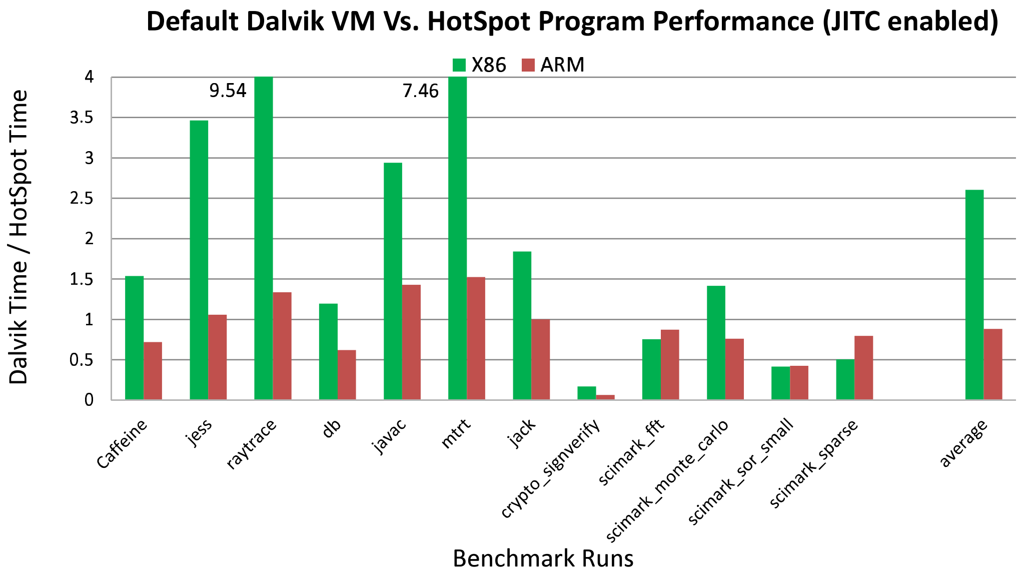

Figure 4.4. The Compiler performance of Hotspot versus Dalvik.

quite interesting, and must be interpreted with the knowledge that traditionally, while the ARM has been the primary target for Android's Dalvik VM, the x86 may have been a preferred target for Oracle's HotSpot over the ARM. Therefore, we find that while the default HotSpot configuration severely outperforms Dalvik on the x86, the Dalvik VM is able to achieve better program performance on the ARM. We also note that our results comparing the Dalvik VM with HotSpot on the ARM are contrary to earlier experiments [18]. We believe that the difference may be on account of using different benchmarks and/or different builds/versions of both Dalvik and HotSpot.

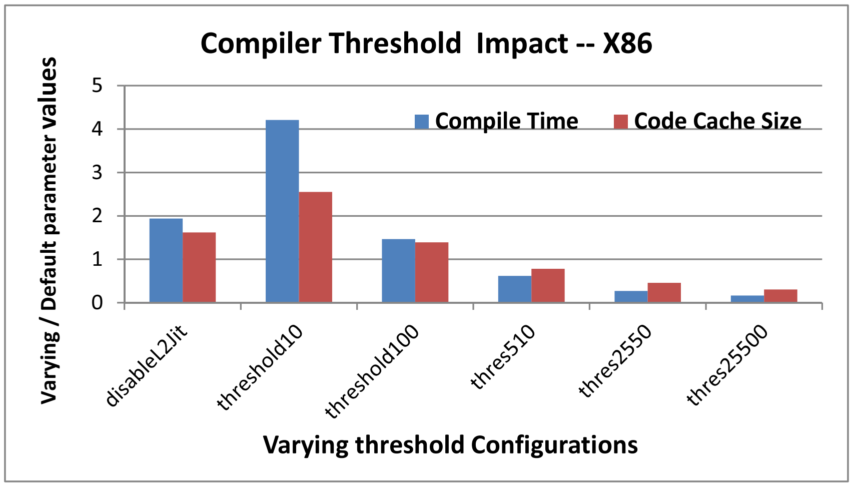

Figure 4.5. The Impact of JitThreshold on Dalvik's compiler efficiency on X86.

4.3 Dalvik Compiler Parameter Tuning

As explained earlier, Dalvik uses a selective compilation model, with the VM using interpretation to initially run the program, and then compiling the *hot* program sections to optimized native code. The compilation efficiency of the Dalvik virtual machine depends on many parameters. Some of the parameters that directly affect the JIT compilation are the selective compilation thresholds chosen for compiler, along with the values of program counter (PC) and the method pointer of the traces that are also used to determine when and what program traces to compile. We modified the Dalvik VM to allow experimentation with several different parameter configurations to find the selective compilation parameter set that should be used for ideal program performance for our benchmark suite.

Our first set of experiments studies the effect of using different selective compilation thresholds on overall program performance. The compilation threshold

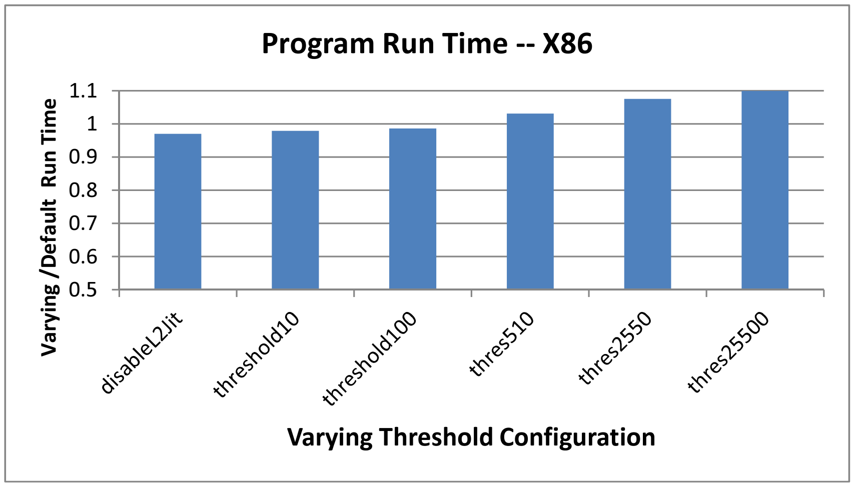

Figure 4.6. The Impact of JitThreshold on Dalvik's Application run time on X86.

determines whether a particular recorded trace is hot or cold, and predicts whether that trace will be executed frequently in the future based on the past profile information. It has been shown in several other works that the compilation threshold can have a huge impact on program run-time, and therefore, determining the ideal threshold value is important [15]. A low threshold value will select more traces for compilation and increase the compilation overhead with diminishing returns for benefiting application run-time. In contrast, an excessively high threshold value will result in the program running in slow interpretation mode for a longer duration.Therefore, finding the optimal threshold for the execution of the program is very essential. Unfortunately, there is no scientifically known rule to set this parameter since it is dependent on many factors. So empirical analysis of various application profile can give us an average value that we can used for a specific architecture.

To assess the impact of JIT compilation threshold on program performance we evaluate different JIT configurations. Our first configuration disbales the second level compilation check performed by Dalvik (see Chapter 3 for a description of the compilation flow in Dalvik).With no second level check, the VM should allow many more methods to be sent for compilation. The remaining configurations select different threshold values. The threshold value of 255 is used as the default on x86 by the Dalvik VM.The threshold values are incresed beyond the default 255. The figure 4.5 shows that the compile time and code cache size reduces when the threshold value is increased. Accordingly the figure 4.6 shows application run time increases towards right, as more time is spent on interpretation.

Figure 4.5 compares the compile time and code cache size impact of different selective compilation configurations to the default setting used by Dalvik. As expected, this figure shows that disabling the level two check and lowering the compilation threshold increases both the compile time as well as the size of the code cache by allowing more traces to be compiled. Thus, for the very aggressive threshold value of 10, the more traces compiled causes a significant spike in compile time and code cache size.

This increase in the compile time may be justified if it produces a corresponding gain in overall program performance. Figure 4.6 compares the overall program performance with the variations in JIT compilation parameters compared to the default values used by Dalvik on the x86 architecture. Interestingly, we find that these more aggressive compilation configurations have a minimal impact on pro-

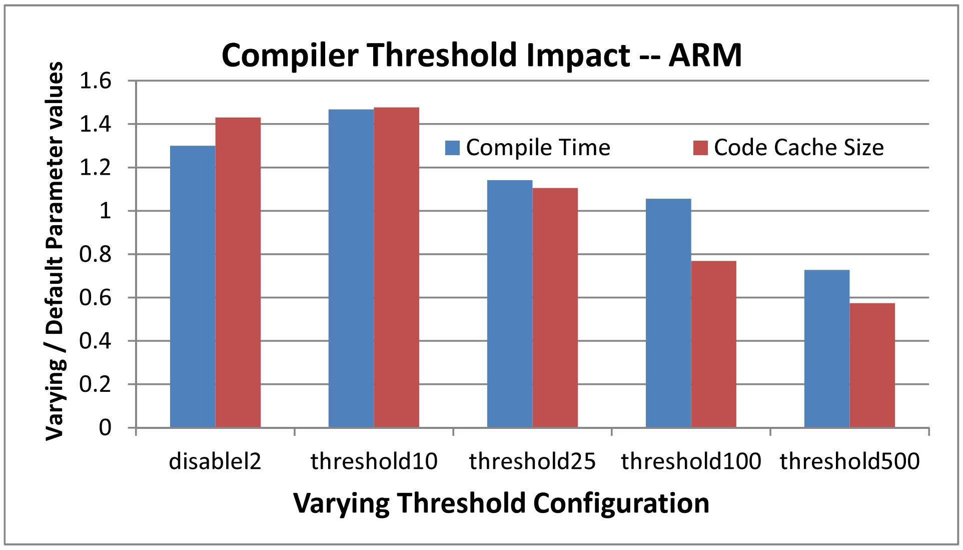

Figure 4.7. The Impact of JitThreshold on Dalvik's compiler efficiency on ARM.

gram performance. Thus, these results show that there is little motivation for more aggressive JIT compilation for Dalvik on the x86. Similarly, Figures 4.7 and 4.8 illustrate the tradeoffs of varying JIT compilation threads for Dalvik on the ARM architecture. Again, the first configuration in each plot enables more aggressive compilation by disabling the second level compiler checks. The remaining configurations select different threshold values. The threshold value of 40 is used as the default on ARM, which is more aggressive compared to x86. Results in 4.7 show the expected outcome of a corresponding increase or decrease in compile time and code-cache size as compilation is made more or less aggressive respectively. However, 4.8 again shows that a change in compiler aggressiveness does not have

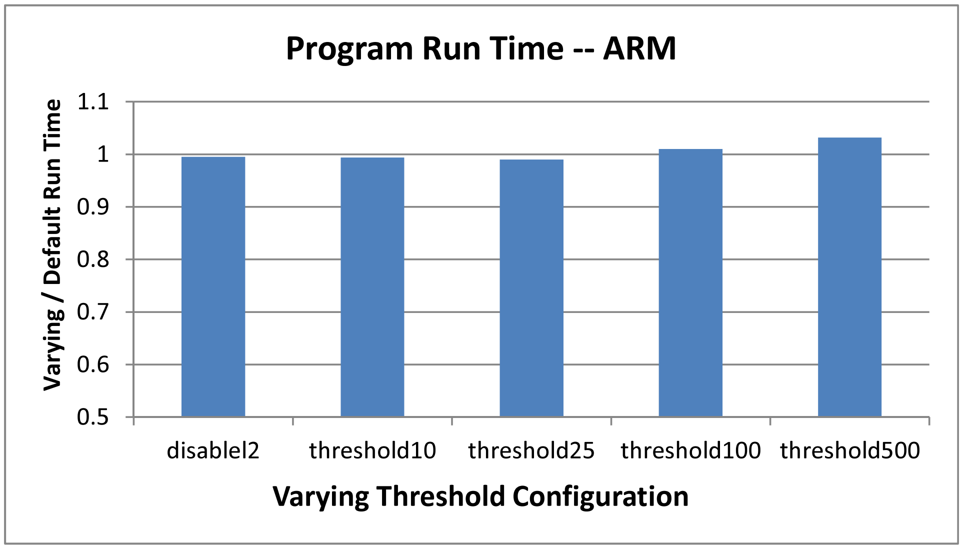

Figure 4.8. The Impact of JitThreshold on Dalvik's Application run time on ARM.

a noticeable impact on run-time (start-up) performance. This observation indicates that these is potential for reducing Dalvik JIT compiler aggressiveness on ARM to reduce memory consumption without affecting performance.

Our next set of experiments explores the effect of changes in the *Filter-key* used by the level two selective compilation check. The Dalvik PC value is changed from 0 to 32 in units of 4 to accommodate all possible combination of parameters. It is recommended in the Dalvik source code that the *Filter-key* should contain only the PC values for program with significantly spiky profile, and only the method pointer for programs with significantly flat profile. In lieu of such information, the default Dalvik configuration uses a combination of both these parameters. For these experiments we vary this Filter-key value along with exploring the effect of different selective compilation thresholds on x86 and ARM.

In total, we evaluated around sixty different JIT compilation configurations to determine the best setting on x86. Our observation from average performance of configuration profiles are show in the figures 4.9 and 4.10. The configurations are compared with the default Dalvik configuration with JIT enabled as baseline. The figure 4.9 shows the run time benchmark numbers where as figure 4.10 shows the compiler run time for the same. We find that the average does not vary much from one configuration to other. However we observed two significant benchmarks javac and jess have varied impact for a selected configuration with threshold value set as 10. With this configuration , the JavaC performance degrades heavily. This degradation is much worse than the interpreter performance. On the other hand for the same configuration profile jess benchmark seems to have a performance gain of about 0.2 times. This clearly shows how the application category affects the JITC compiler parameters.

JITC also indirectly affects the parameters like code-cache size, heap size and compiler queue size, which determines the compiler high watermark for queueing the traces. There is a physical maximum of 1024M of code-cache size for each process run by Dalvik to conserve memory resource on embedded systems. The heap size affects the internal code cache efficiency. Low code-cache size may lead to frequent reset of code cache, which flushes all the compiled code and starts the JIT compilation from the beginning. This causes the loss of all the profiled information, which now will need to be recollected, increasing the JIT overhead drastically for the program run.

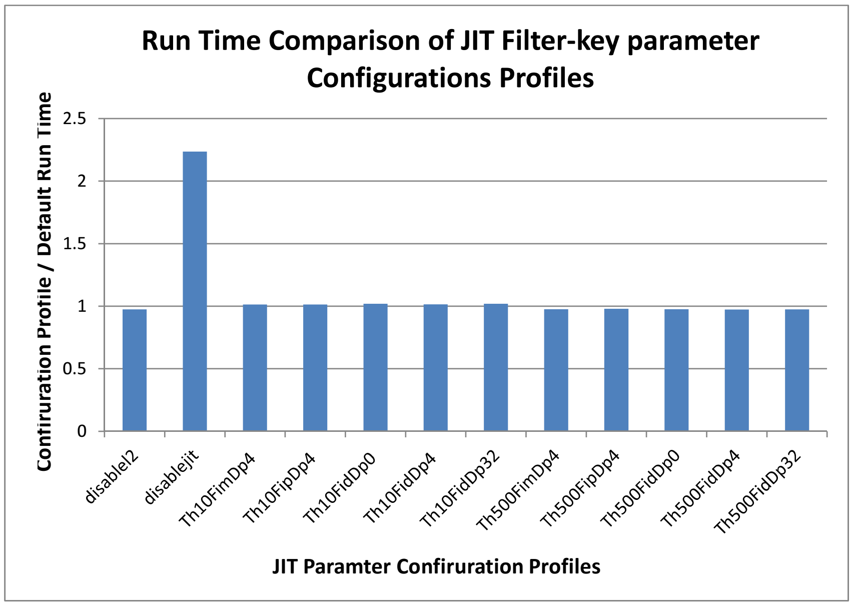

Figure 4.9. The Run time performance comparison of various JIT configurations

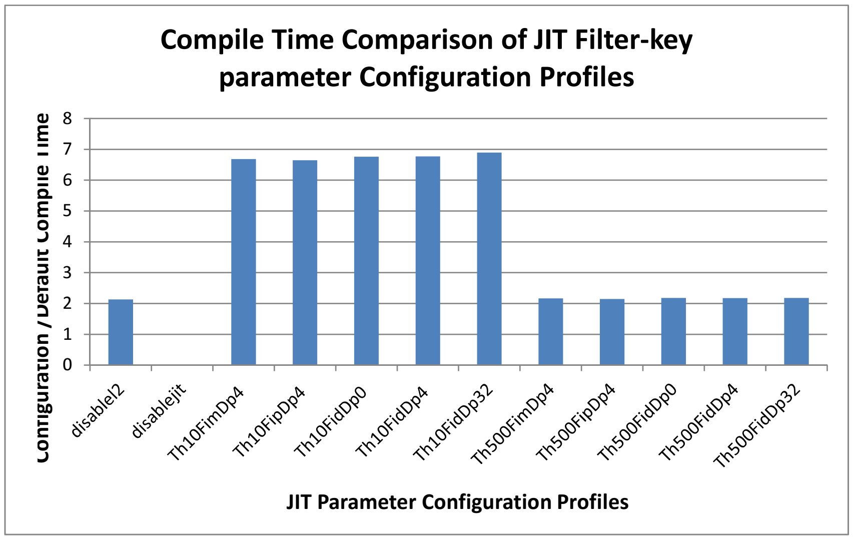

Figure 4.10. The Compile time performance comparison of various JIT configurations

Chapter 5

Future Work

We believe that the Dalvik VM environment constructed during this research for x86 and ARM targets will enable much future research into exploring and improving application execution performance on mobile and embedded devices. First, we will perform a more in-depth exploration of Dalvik VM performance properties using additional benchmarks and by exposing more tunable VM parameters. Second, the constructed Dalvik VM environment will allow implementation and testing of ideas to improve memory and power consumption during application execution on mobile devices. For example, we will investigate approaches to find the ideal *working set* of the program along with tracking changes to this set so as to only compile and maintain the necessary native code in Dalvik's code cache to reduce per-program memory consumption. Finally, we will explore mechanisms to customize optimization phase selections and other JIT compilation settings to improve overall program performance of the Dalvik virtual machine.

Chapter 6

Conclusions

The primary goal of this project was to construct an environment to investigate and improve the performance characteristics of Android's Dalvik virtual machine on contemporary x86 and ARM architectures. As part of this project we had to learn the process of building Android, running a stand-alone version of the Dalvik VM on x86-Linux machines, and installing updated variants on Android and Dalvik on the ARM platform. We have been successful at achieving this goal and now have the knowledge to update and execute the Dalvik virtual machine on both the x86 and ARM target architectures. Our ability to isolate and run the Dalvik VM on x86-Linux machines enables us to shorten the update-build-install cycle necessary to explore Dalvik performance properties on other target architectures.

The second goal of this project was to understand the internal organization of Dalvik's dynamic compilation and execution engine, and evaluate and compare its runtime performance with other standard Java virtual machines. As part of this study we have now performed the first open performance assessment of the Dalvik VM on the x86 and compared it with Oracle's HotSpot JVM. Additionally, we also quantify Dalvik VM performance for Android's Linaro build on our ARMv7 pandaboard platform. We have discovered that the register-based intermediate format used by Dalvik's DEX input code enables the VM to achieve much better interpreter performance compared to the HotSpot JVM that uses the standard JVM stack-based bytecode classfile representation. We also found that while Dalvik needs further performance enhancements to make it competitive with the HotSpot JVM on x86-Linux, its performance is very competitive and often exceeds that achieved by Oracle's HotSpot JDK on the ARM. Thus, the success of this project provides a robust open-source environment that will allow much future research on JIT compilation in Android's Dalvik VM for contemporary architectures.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, February 2005.

[2] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[3] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 13–20, 2000.

[4] CaffeineMark. Caffeinemark benchmarks. http://www.benchmarkhq.ru/cm30/, 1997.

[5] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[6] Google. Google android building and development group. `https://groups.google.com/d/topic/android-platform/D1NpRKAiOgg/discussion`.

[7] Google. Google io developer conference. `http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html`.

[8] Google. Google io developer conference on dalvikvm internals. `https://sites.google.com/site/io/dalvik-vm-internals/`.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (3rd Edition)*. Prentice Hall, third edition, June 14 2005.

[10] G. J. Hansen. *Adaptive systems for the dynamic run-time optimization of programs.* PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1974.

[11] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Language Systems*, 18(4):355–400, 1996.

[12] D. E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java hotspot^{TM}client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[14] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, December 2000.

[15] P. A. Kulkarni. Jit compilation policy for modern machines. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 773–788, 2011.

[16] Linaro. Automated android installation using linaro image tool. `https://wiki.linaro.org/Linaro-Image-Tools`.

[17] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon. Evaluation of android dalvik virtual machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 115–124, New York, NY, USA, 2012. ACM.

[18] Oracle. Oracle blog comparing java se and dalvik virtual machines. `https://blogs.oracle.com/javaseembedded/entry/how_does_android_22s_performance_stack` Nov. 2010.

[19] M. Paleczny, C. Vick, and C. Click. The Java hotspottm server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Re-*

search and Technology Symposium, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[20] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[21] SPEC2008. Specjvm2008 benchmarks. http://www.spec.org/jvm2008/, 2008.

[22] SPEC98. Specjvm98 benchmarks. http://www.spec.org/jvm98/, 1998.