Implementation of JVM Tool Interface on Dalvik Virtual Machine

Chien-Wei Chang[1], Chun-Yu Lin[1], Chung-Ta King[1], Yi-Fan Chung[1], Shau-Yin Tseng[2]

[1]Department of Computer Science
National Tsing Hua University, Hsinchu, Taiwan

[2]SoC Technology Center
Industrial Technology Research Institute, Hsinchu, Taiwan

ABSTRACT

Mobile devices such as cell phones, GPS guiding systems, and mp3 players, now become one of the most important consumer electronic products. Being an embedded system, mobile devices are highly integrated in software and hardware for robustness, high performance, and low cost. The problem is that this also makes it very difficult to understand the internal interactions of hardware as well as software modules in such devices and to identify performance bottlenecks and design faults. Profiling helps developers to understand the behaviors of a system, especially during the development of new platforms. Android is a new software platform intended for mobile devices. It is composed of Linux and a Java virtual machine called Dalvik. The ability to profile Android helps developers to familiarize with Android's features and optimize their applications. In this paper, we discuss the development of a profiling tool interface, JVM TI, on Android. With this tool interface, developers can profile their Java code running on Dalvik using JVM TI.

Keywords: JVM TI, JAVA, Dalvik virtual machine, Google Android, Embeded systen, Profiling tool.

I. Introduction

Android [8] is a new software platform for mobile devices introduced and developed by Google, and later Open Handset Alliance [9]. It is one of the first platforms for mobile devices that is open source [7]. Android platform consists of a Linux operating system, core libraries of the Java programming language, an Java virtual machine (Dalvik virtual machine), and some key applications (e.g. browser, maps).

Dalvik virtual machine (DVM) is a major component of Google's Android platform. It is optimized for low memory requirements and is designed to allow multiple VM instances to run at the same time. The virtual machine runs Java applications. However, Dalvik virtual machine is different from standard Java virtual machine in some ways. First, most virtual machines use a stack-based architecture, but Dalvik is a register-based architecture. Second, Dalvik runs Java applications which have been transformed into the Dalvik Executable (.dex) format. These two major differences make Dalvik different from a standard Java virtual machine. Therefore, tools developed on standard Java virtual machines, such as profiling tools, cannot be ported to DVM directly without modifications.

Java virtual machine tool interface (JVM TI) [1][10] is a native programming interface on Java virtual machine. The interface provides functionalities to inspect the state of a virtual machine, gather information during run time, and also control the execution of applications running on the Java virtual machine. JVM TI has been used by many profiling tools such as HProf [12], YAJP [5], Jboss [4], etc. But JVM TI is not available in all implementations of the Java virtual machine. Dalvik is one of the virtual machines that do not support JVM TI.

In this paper, we discuss implementation of JVM TI on Dalvik virtual machine. With the tool interface, programmers can easily develop their own tools or use available profiling tools based on JVM TI to profile Dalvik virtual machine. But our implementation has some limitations on load-time byte-code instrumentation. We will discuss the details in the following sections.

II. Implementation of JVM TI

A. Design Philosophy

JVM TI is not supported on original Dalvik virtual machine, so we need to design a new subsystem on Dalvik for JVM TI. In this paper, we propose a well implemented JVM TI on Dalvik. Since Dalvik virtual machine is a small, simple, and memory efficiency virtual machine. We follow some basic principle to design our Java virtual machine tool interface on Dalvik virtual machine.

- Low overhead: As Dalvik is a simple virtual machine, the tool interface runs on it should not cause too much overhead.
- Memory efficiency: Since Dalvik targets on mobile devices which has memory constraint, JVM TI should not take too much memory space
- Easy to upgrade: With the upgrade of Android, our interface should be easy to port on latest version. As a result, we gather our code independent of Dalvik source code.

B. Architecture of JVM TI

Our implementation of JVM TI consists of two parts, "virtual machine events and handler" and "tool interface provided functions". Figure 1 shows the architecture of our implementation. We add two subsystems into Dalvik, event handler and implementation of each function. The grey blocks

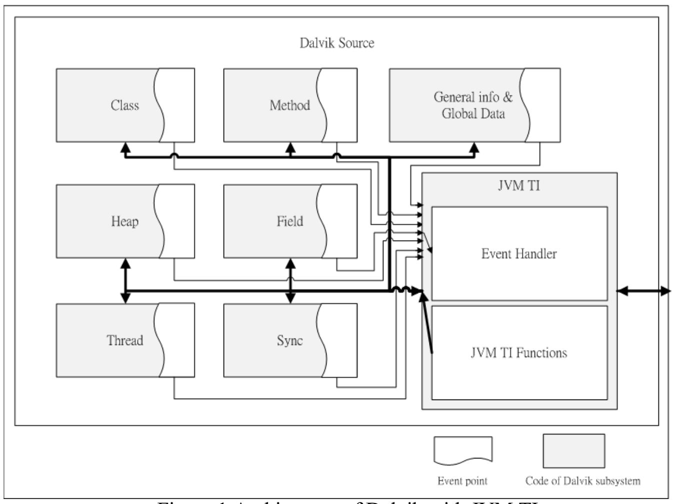


Figure 1 Architecture of Dalvik with JVM TI

are each subsystem source code of Dalvik. For example, block "Thread" means the implementation of Java thread in Dalvik, including structure of Java thread, and functions to maintain Java thread such as creation, stop etc. The white block of each grey block means what we add or modify to the subsystem, including event and additional functions or structure to support JVM TI.

C. *JVM TI Events Implementation*

We separate event handler for every different event instead of a single handler for every event. This lowers the overhead of event handler, because we need not to switch from different event. And more importantly, variables need to pass to every callbacks are quite different, thus this design is more suitable for this requirement. This design is also convenient to user apply our JVM TI on newer version of Dalvik virtual machine when Android upgrades. Users simply instrument the separate event handler function call to the location we define, and get the same result.

Table I lists all the events available in Dalvik. Most events here are implemented as mentioned above and follow the definition that Sun defines. The only difference is the "class file load hook" event, which will be discussed later.

The functions that JVM TI provides can be classified into twenty categories. The sixteen categories implemented are listed in Table II.

D. *Challenge and Contribution of Implementation*

- Class File Load Hook, Byte-code Instrumentation

Class file load hook event is triggered when the virtual machine obtains a class file data, but before it constructs the in-memory representation for that class. At this event Dalvik permit agents to modify the original class file, which is called "load-time byte-code instrumentation". [13]

There are some problems when we implement this event and do load-time byte-code instrumentation. First, the way Dalvik load a class is different from standard virtual machine. Dalvik loads a single dex file into a read-only memory for a Java application at initialization of Dalvik. When a class loads into Dalvik, the different data will point to the different location of dex file.

TABLE I. IMPLEMENTATION OF JVM TI EVENTS

Events	
Class File Load Hook	Monitor Wait
Class Load	Monitor Waited
Class Prepare	Object Free
Data Dump Request	Object Allocation
Exception	Thread Start
Exception Catch	Thread End
Garbage Collection Start	VM Initialization Event
Garbage Collection End	VM Death Event
Monitor Contended Enter	VM Start Event
Monitor Contended Entered	

TABLE II. IMPLEMENTATION OF JVM TI FUNCTIONS

Events	
Memory Management	Heap
Thread	Local Variable
Thread Group	Object
Stack Frame	Class
Field	Event Management
Method	Timers
Raw Monitor	System Properties
JNI Function Interception	General Information

To solve this problem, we modify the way Dalvik loads methods. We make a copy of every method when it is loaded if agent registers class file load hook event. And makes the class point to original location of these methods, and then call the callbacks of this event to agent. An agent can modify the method because we've copied methods to writeable space. After returning from the callback, we check whether the method is modified or not. If agent did not modify the methods, we free the copies to get back memory space. If agent modifies the methods, we make this class point to the copies. This way causes some memory overhead.

This method also has it limitation. Dalvik is a register based virtual machine, when class files transform into single dex file, dex tool sets number of registers to every method. The number of registers to every method will be kept in dex file and used as verification. The verification will check dex file directly, and we cannot modify it. So if the number of registers of byte-code to be instrumented greater than number of registers set to method by dex tool, the instrumentation cannot be done. This causes the limitation of byte-code instrumentation in Dalvik. A better solution here needs to modify the dex tool to reserve the registers for byte-code instrumentation.

Second, Dalvik executes different file format from standard "class" file, and Dalvik runs its own byte-code. This means we cannot use the library that Sun provides to do byte-code instrumentation.

We solve this problem as follow: We compile the tracker class (the method call to be injected to select class) with user program, and use "dexdump" tool to find out the position of the tracker class to know the byte-codes of method call to the tracker class. The position of tracker class would be different from applications through the transform of dex tool. This means the byte-codes for the method calls differs from application to application. As a result, we need to modify the method for different applications now.

- Agent Data Management

Agents can tag data to target object or thread as a tracker. The original structure to present object and thread of Dalvik do not reserve the space to keep information like this, so we need to modify the original structure of thread and object. We reserved four bytes as pointer to agent data and another four bytes to reserve additional data that agents need to keep in structure of thread and object.

- Object Monitor Information

Another structure of Dalvik we modify is object monitor. The Java object monitors of Dalvik are implemented by pthread. Thus, the monitor structure is simple and do not keep information that JVM TI support. Such as, threads that waiting to own this object are unknown, and threads that waiting to be notified by this monitor are also not keep in Dalvik.

We modify the object monitor structure of Dalvik. We add two lists to maintain the n threads that waiting to own the object and waiting to be notified by the monitor and then calculate the number of threads. We also need to modify the functions that provide monitor functionality of Dalvik to keep the information of monitor accurate. In order to lower the overhead of overall execution time, the information we add will be kept if agents registered the monitor events.

- Event Callbacks Scheduling

JVM TI agents need to synchronize the execution of event callbacks, because callbacks always need to modify the global data. JVM TI provides raw monitor to synchronize the execution of event callbacks. The Java thread in Dalvik is implemented by pthread, so as agent thread. So we create a structure to maintain the raw monitor and implement it by pthread library. Since raw monitor needs not to provide any information to agent as object monitor does. We simplify the raw monitor structure, and use functions that pthread provides to implement the functionality such as notify, wait etc.

III. Evaluation

TABLE III. SEVEN DEMO AGENTS

Agents	
Version Check	GC Test
Heap Viewer	Method trace
Method Instrument	Waiters
Heap Tracker	

A. *JVM TI Verification*

We implement our JVM TI on Dalvik version 1.4.4. We run the whole system on emulator version 1.0 that Google provide. In order to verify our JVM TI, we use seven demo agents that Sun provides. These seven agents shown in Table III not only show part of abilities of JVM TI, but users can also extend these seven agents to satisfy their requirement. To verify our implementation, we compare our result with JVM TI implementation on Sun jdk1.6.0_06.

We use a simple multithread Java program as our target application on Dalvik and Sun JVM. This program uses two threads to do a simple calculation ten times and each thread does garbage collection at time equals five. We use this program because of its simplicity, we can predict how it work on VM with JVM TI, and the events of running this program are enough to show these agents.

B. *Memory Space Overhead*

- Static memory overhead:

The original image of Dalvik VM is 47.6MB. With our implementation JVM TI on Dalvik, the image increase to 51.8MB.

- Run-time memory overhead:

We add some data into Dalvik global data in order to keep the events registered by agents, agent data, and JVM TI environment. We list these in Table IV. Also, we modify the internal structure to represent Dalvik class object, thread, and monitor. We list the original size of this part in Table V and show the percentage of increasing size.

C. *Execution Time Overhead*

We use free Java benchmarks "Caffeine" and "SciMark 2.0" as normal Java applications to calculate the extra execution time when application running with agents. SciMark is a Java benchmark for scientific and numerical computing. Caffeine is a JVM benchmark. It scores VM with testing method call speed, calculate, loops, floating point calculations.

Since we would like to observe the overhead of events and functions of JVM TI, agents include byte-code instrumentation are not included here because the overhead will include the additional Java method execution.

TABLE IV. MEMORY OVERHEAD FOR JVM TI

Required Variables	Bytes
JVM TI Global Variable	32
JVM TI Environment Data	117(#functions)*4+8 = 476
Event Callbacks	20(#events)*4 = 80
Event Registrations	20(#events)*4 = 80
Total	628

TABLE V. ADDITIONAL MODIFICATION OF DALVIK

Structure	Original	Increase	Percentage
ClassObject	184 bytes	8 bytes	4%
Thread	280 bytes	8 bytes	3%
Monitor	44 bytes	16 bytes	27%

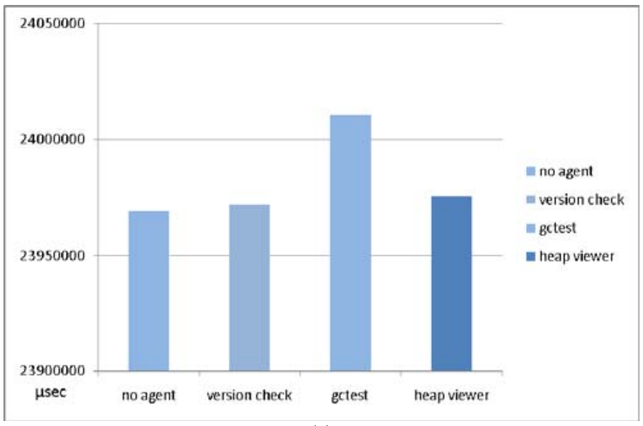
We execute each benchmark and each agent ten times and calculate the average. Figure 2 shows the result of our test. VersionCheck always causes about two thousands µsec. The heapViewer needs to scan though the whole heap, so the time would be different because of the size of the heap. It causes about six thousands µsec in these two benchmarks. Caffeine generates garbage collections eight times in average and one time for SciMark. Because the average execution time for each GC is five thousands µsec, the gctest implementation causes averagely four thousands µsec overhead for each benchmarks.

IV. CONCLUSIONS AND FUTURE WORKS

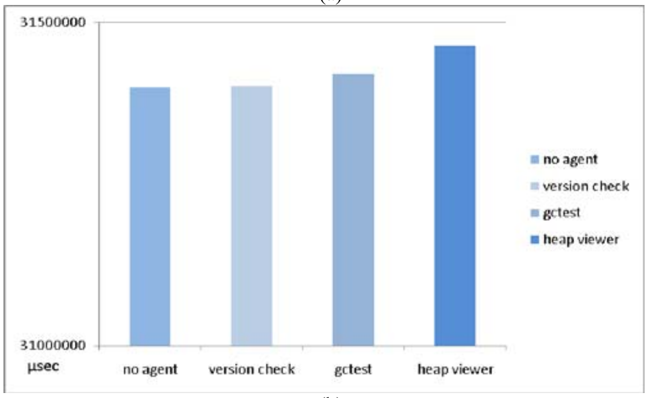
Through good profiling methods, developers of embedded systems can reduce the efforts to identify the design and development problems of the systems, and optimize the design. A general purpose tool interface helps developers achieve this goal.

With our implementation of JVM TI on Dalvik, profiling tools for Java based on JVM TI can be used on Dalvik. Furthermore, with the events and functions that JVM TI supports, users can develop their own tool on Dalvik. The results show our JVM TI causes little overhead on processing time and memory space.

Although we have implemented some simple functions to do the load-time byte-code instrumentation, the functions are only for our test cases only. We still need a complete library to support full functionality to support automatic byte-code instrumentation for all Java applications. The method call to tracker class would be different for different applications, so a good solution would need the information from dex tool or instrument byte-codes through dex tool. Providing a complete byte-code instrumentation library is the major future work.



(a)



(b)

Figure 2 Execution time of *Caffine* (a) and *SciMark* (b)

REFERENCE

[1] Fred Long. "Software Vulnerabilities in Java", Technical Note, CMU/SEI-2005-TN-044, 2005

[2] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. Addison-Wesley, Reading, Massachusetts, 1996

[3] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley, Reading, Massachusetts, 1996

[4] Jboss profiler: http://www.jboss.org/jbossprofiler/

[5] YajP – Yet Another Java Profiler: http://yajp.sourceforge.net/

[6] Dalvik Debug Bridge Monitor Service: http://developer.android.com/guide/developing/tools/ddms.html

[7] Android Open Source Project. Open Handset Alliance: http://source.android.com/license

[8] Official Android website: http://www.android.com/

[9] Open Handset Alliance, website: http://www.openhandsetalliance.com/

[10] Java Virtual Machine Tool Interface Spec: http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html

[11] Java Software, Sun Microsystems, Inc. Java Development Kit (JDK). http://java.sun.com/products/jdk/1.2

[12] HPROF: A Heap/CPU Profiling Tool in J2SE 5.0: http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[13] Renaud Pawlak, "Spoon: Compile-time Annotation Processing for Middleware," IEEE Distributed Systems Online, vol. 7, no. 11, 2006, art. no. 0611-oy001.