AndroScope for Detailed Performance Study of the Android Platform and Its Applications

Myeongjin Cho, Seok Joong Hwang, *Student Member*, IEEE, Ho Jin Lee, Minseong Kim and Seon Wook Kim, *Senior Member*, IEEE
School of Electrical Engineering, Korea University

***Abstract*--This paper presents a performance analysis tool for the Android platform and its applications, called *AndroScope*, which collects performance data from both Dalvik VM and native libraries and provides GUI integrated with the Android SDK.**

I. INTRODUCTION

Traceview [1] is one of the Android SDK tools, and the tool provides a graphical viewer for execution time logs of method activities (enter and exit) created by Dalvik VM (Virtual Machine). Traceview helps us profile the performance of Android applications.

Recently, applications using Java Native Interface (JNI) and native libraries have been developed actively for faster processing than pure Java applications. But Dalvik VM cannot trace native libraries (Dalvik VM can trace only Java applications) because Android libraries are written in C/C++. In addition, Traceview cannot perform low-level analysis, e.g., does not provide more detailed performance data such as instructions per cycle and cache miss ratio, because Dalvik VM only supports time-based trace logs. To overcome these limitations, more advanced trace tools than Traceview are required.

ARM Streamline [2] is one of the solutions for low-level performance analysis for the Android platforms, but not for Java applications. Furthermore, ARM Streamline only supports the ARM architecture, not non-ARM based Android devices, e.g., MIPS and Intel Androids [3].

In this paper, we introduce a performance trace tool for the Android platforms, called *AndroScope (Android Microscope)* that supports tracing not only Android applications but also Android platforms, and provides low-level performance analysis through hardware performance counters (HPC). Also, we present GUI which is based on Traceview to show more detailed data such as HPCs, memory usages, etc. and handles mass trace log for profiling and performance analysis. Our tool is integrated with Android SDK for user's convenience. To our best knowledge, *AndroScope* is the only existing tool to support all the above functionalities.

This paper is organized as follows. Section II introduces an overview of our tool. Sections III and IV show our advanced trace mechanism and GUI in detail. Finally, Section V makes conclusion.

II. OVERVIEW OF ANDROSCOPE

Our system has the following distinguished features from the original Dalvik VM trace and Traceview: 1) HPC trace, 2) native libraries trace, 3) fast data processing for mass trace log, and 4) advanced Traceview. To support these features, we modified Dalvik VM and Traceview as shown in Fig 1.

This research has been supported by LG Electronics.

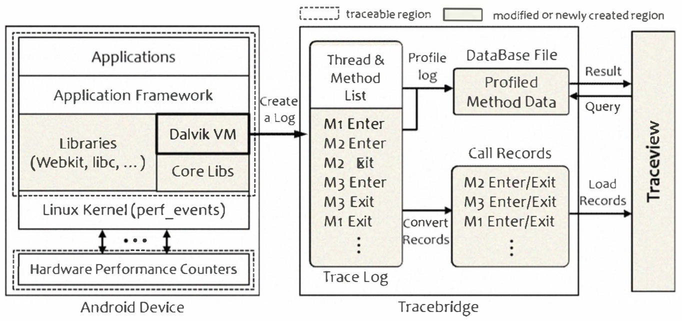


Fig. 1. Organization of *AndroScope*.

Our system consists of Dalvik VM, tracebridge, and Traceview. Dalvik VM is in charge of tracing performance data and creating a log, and traceable regions are enclosed by dashed lines in Fig 1. The tracebridge was added for fast data processing of mass trace log. The trace log is partitioned into two parts, profiled method data and call records. Traceview is in charge of displaying the traced logs.

III. PERFORMANCE DATA COLLECTION

A. *Trace with Hardware Performance Counter*

For low-level performance analysis, our tool collects HPC data in addition to time. The HPC is a set of special purpose registers built into most modern microprocessors to store the counts of hardware events such as cache miss, CPU cycles and instructions architecturally executed. These HPC events can provide detailed low-level performance analysis to users. For instance, we can easily derive instructions per cycle (IPC) that is a basic performance factor from collected CPU cycles and executed instructions events.

To support the HPC trace, we used *perf_events* [4] which is a driver to access HPCs in the Linux kernel. Note that low version kernels, e.g., 2.6.32 used in Froyo, require a ported *perf_events* of Linux kernel 2.6.35 which is used for Gingerbread.

B. *Trace of Native Libraries*

The original trace mechanism is able to trace activities of Java methods and Java native methods written in Java Native Interface. But the mechanism cannot know what happens inside of Java natives and other functions including Dalvik VM itself. Our advanced trace mechanism overcomes these limitations through inserting instrumentation codes at the beginning and the end of native functions in order to collect trace data.

We extended the GCC compiler front-end to automatically insert instrumentation codes. The extended compiler provides two kinds of ways to control the instrumentation: compiler options and function attributes. We used three pre-existing GCC options and one more new option: 1) *instrument-functions* for full source instrumentation, 2) *instrument-functions-exclude-function-list*, 3) *instrument-functions-exclude-file-list*, and 4) *instrument-functions-minimum-line*. The *instrument-functions-minimum-line* is to discard trivial functions based on source line counts. The extended compiler accepts two function attributes which are specified in source code: *__attribute__ ((do_instrument))* and *__attribute__ ((no_instrument))* which are to turn the instrumentation on and off for the corresponding functions regardless of the compiler options.

The followings were excluded in the compiler-based instrumentation for the native trace: The instrumentation code itself, the PThread library, and naked functions. Tracing of the instrumentation code itself is obviously not intended and incurs malfunction in the trace. Until initialization of a thread local storage (TLS) completes in PThread creation, the trace component cannot identify a thread id of a newly created thread which is one of the trace record entries. A naked function which is specified by a function attribute does not have prologue and epilogue; such a function is usually used for writing a function only with inline assembly codes. Therefore, inserting instrumentation codes at the beginning and the end of a naked function breaks calling convention.

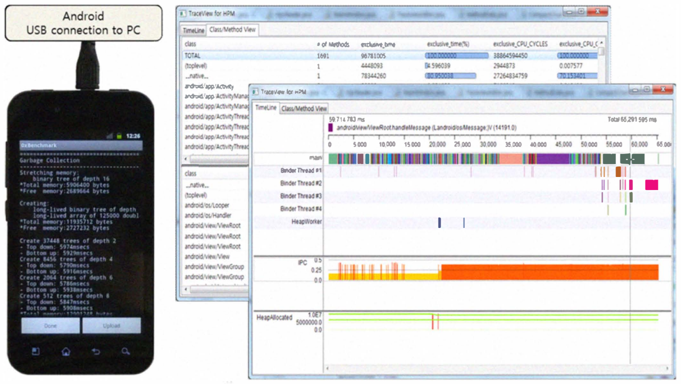
C. *Selective Trace*

The change of the instrumentation options (compiler flags and function attributes) incurs recompilation of applications and libraries, which is a time-consuming task. So our trace mechanism supports runtime filtering for selective trace with an external configuration file (*trace_method.conf*). This file has methods information such as class, method name, and signature which identify a specific method. If the file has no information, all methods are traced.

IV. TRACEBRIDGE AND GRAPHICAL USER INTERFACE

In order to provide detailed performance analysis, we implemented a nice graphical user interface based on Traceview. Traceview analyzes the trace log file and displays a timeline panel which describes when each thread and method starts and stops. Also, Traceview displays a profile panel that provides a summary of what happened inside a method such as inclusive/exclusive time, the number of calls and so on.

However, Traceview has performance problems because it is written in Java and an eclipse plugin. So we implemented a middleware which is named *tracebridge* for fast data processing. The *tracebridge* analyzes a trace log file and creates call records which have start/end information of function calls and creates a database file which has statistics information of each class and function with HPC events. In other words, *tracebridge* is in charge of data processing and GUI based on TraveView is only responsible for displaying results.



```
0xBenchmark
**********************************
Garbage Collection
----------------------------------
Stretching memory:
    binary tree of depth 16
*Total memory:5906400 bytes
*Free  memory:2689664 bytes

Creating:
    long-lived binary tree of depth
    long-lived array of 125000 doubl
*Total memory:11935712 bytes
*Free  memory:2727232 bytes

Create 37448 trees of depth 2
- Top down: 5974msecs
- Bottom up: 5929msecs
Create 8456 trees of depth 4
- Top down: 5790msecs
- Bottom up: 5916msecs
Create 2064 trees of depth 6
- Top down: 5786msecs
- Bottom up: 5938msecs
Create 512 trees of depth 8
- Top down: 5847msecs
- Bottom up: 5908msecs
```

Fig. 2. An example which is traced on a shown Android phone and an analyzed result in *AndroScope*. The number of call records in the example is about 6 millions.

Fig. 2 shows the GUI of our tool. The *AndroScope* GUI consists of a timeline view (the front image) and a profile view (the back image). The timeline view contains timelines for execution time, HPCs, and heap memory usage. The timeline class was optimized for mass data because the timeline of Traceview is terribly slow to display huge amount of data. And the profile view contains detail information of each class and function. Because the results can be classified by class or method, the profile view helps us which class and method are most time/HPCs consuming. Also, we can find out the ratio of native methods to Java methods in an application. Moreover, because our trace mechanism can trace the Dalvik VM and Android libraries, we can find out which libraries and parts of Dalvik VM are used frequently and most time-consuming through analyzing native method. Consequently, it is possible to optimize the Dalvik VM and the Android libraries at the same time. If you find hot spots and weak points, a related source code can be opened by double click on the method name.

V. CONCLUSION

We introduced our *AndroScope* to provide low-level performance analysis on the Android platforms with HPCs and to support to trace Java applications, native applications, Dalvik VM, and even Android libraries. To provide native trace, we implemented the instrumentation interface in GCC and allowed the selective instrumentation of specific methods for reducing trace overheads. Through our tool, we can get a chance to optimize Android platforms as well as applications.

REFERENCES

[1] Traceview, http://developer.android.com/guide/developing/tools/traceview.html
[2] ARM Streamline Performance Analyzer, http://www.arm.com/products/tools/software-tools/ds-5/streamline.php
[3] MIPS Android, http://developer.mips.com/android/
[4] Performance Counters for Linux, v8. http://lwn.net/Articles/336542