AndroScope: An Insightful Performance Analyzer for All Software Layers of the Android-Based Systems

Myeongjin Cho, Ho Jin Lee, Minseong Kim, and Seon Wook Kim

Android has become the most popular platform for mobile devices. However, Android still has critical performance issues, such as "application not responding" errors and hiccups resulting from garbage collection. Many phone vendors have tried to resolve the problems by characterizing and improving the performance. However, there are few insightful performance analysis tools for the Android-based systems. This paper presents AndroScope, which is a performance analysis tool for both the Android platform (Dalvik virtual machine, core libraries, Android libraries, and even Linux kernels) and its applications. To the best of our knowledge, this is the first tool to collect and analyze performance data from all the software layers of the Android-based systems. AndroScope offers a trace mechanism to collect such deep and wide performance data as hardware performance counters, time, and memory usage. In addition, the tool includes TraceBridge, which is a middleware for the fast handling of mass logs. Moreover, AndroScope offers an integrated graphical user interface with the Android software development kit to display a great volume of the detailed performance data.

Keywords: Performance analysis, Android, Dalvik virtual machine, instrumentation, hardware performance counter.

Manuscript received Apr. 3, 2012; revised Nov. 12, 2012; accepted Sept. 25, 2012.

This work was supported by the IT R&D program of MKE/KEIT [10041686, Cooperative Control Communication/Security Technology and SoC Development for Autonomous and Safe Driving System].

Myeongjin Cho (phone: +82 10 5269 2116, linux@korea.ac.kr), Ho Jin Lee (deathfielder@korea.ac.kr), Minseong Kim (kissofgod@korea.ac.kr), and Seon Wook Kim (corresponding author, seon@korea.ac.kr) are with the School of Electrical Engineering, Korea University, Seoul, Rep. of Korea.

http://dx.doi.org/10.4218/etrij.13.0112.0203

I. Introduction

Many consumer electronics, especially mobile platforms and TVs, have been developed with Android. Recently, many vendors have been relying upon hardware solutions of a high performance and intelligence level to solve performance issues in the Android systems, such as slow execution, high power consumption, inefficient multithread scheduling, and so on. Performance analysis for the Android-based systems is needed because there are few insightful performance analysis tools. The performance analysis tool is very important for system developers to understand program behaviors and evaluate how well programs interact with hardware. Therefore, the tool makes it possible for the developer to use lower-end hardware to reduce cost by tuning both hardware and software, thus improving the performance.

Traceview and ARM Streamline [1] are representatives of the popularly used performance analysis tools for Android. Traceview is one of the Android software development kit (SDK) tools, and it provides a graphical viewer for execution time logs of method behaviors (their entries and exits) created by Dalvik virtual machine (VM). Recently, applications using Java Native Interface (JNI) [2] and native libraries written in C/C++ have been developed for faster processing than is achieved using pure Java applications. However, Dalvik VM cannot trace the native libraries but, rather, can only trace Java code sections. Also, Traceview cannot perform low-level analysis; for example, it does not provide more detailed performance data, such as instructions per cycle (IPC), cache miss ratio, and so on, because Dalvik VM only supports time-based trace logs. ARM Streamline is one of the solutions for low-level performance analysis for the ARM-based platforms but not for Android. The tool provides low-level performance

data, such as hardware performance counters (HPCs), kernel events, and memory usage. However, ARM Streamline does not profile Java applications for the Android applications. Furthermore, ARM Streamline only supports the ARM-based Android. In other words, ARM Streamline cannot be used for the MIPS [3] and Atom-based Androids. The tools for MIPS and Atom [4] are not yet known to the public.

In this paper, we introduce a performance analysis tool for the Android platform, called "AndroScope" (Android microscope) [5]. AndroScope provides a trace mechanism for tracing not only the Android applications but also all software layers of the Android platform, that is, Dalvik VM, core libraries, Android libraries, and even Linux kernels. Additionally, the mechanism provides low-level performance analysis through tracing HPCs, memory usage, and so on. Also, AndroScope offers a middleware for the fast handling of mass logs and an advanced graphical user interface (GUI), which is based on Traceview. Our tool is integrated with the Dalvik Debug Monitor Server (DDMS) for the user's convenience. To the best of our knowledge, our proposed AndroScope is the first tool to collect and analyze low-level performance data from all the software layers of the Android-based systems, that is, both the Android platform and its applications. Also, AndroScope can be easily ported to any platforms with a minor change (only HPC driver) of its implementation.

This paper is organized as follows. Related work is described in section II, and an overview of our proposed AndroScope is given in section III. Sections IV, V, and VI respectively present performance data collection mechanisms, middleware for fast trace log processing, and an advanced graphical viewer for mass logs. Our tool is evaluated in section VII, and conclusions are drawn in section VIII.

II. Related Work

1. Dalvik-VM-Based Profiling Tools

Dalvik VM provides a method-by-method trace mechanism for debugging the Android applications and profiling performance by looking for their method activities, such as method enter or exit. When the method activities are detected, Dalvik VM records data log that includes the thread ID, method ID, method actions (entry or exit), and time delta from start time of the trace, as shown in Fig. 1. If the thread and method names appear for the first time during the trace, their information is added to the thread and method lists in the key log, as shown in Fig. 2. When the trace is finished, the data and the key log are combined into one trace file.

There are two tools for getting meaningful information from the trace file, the GUI-based Traceview and the console-based

```
* File format:
* header
* record 0
* record 1
* ...
*
* Header format:
* u4 magic 0x574f4c53 ('SLOW')
* u2 version
* u2 offset to data
* u8 start date/time in usec
*
* Record format:
* u1 thread ID
* u4 method ID | method action
* u4 time delta since start, in usec
```

Fig. 1. Data log formats.

```
*version
1
clock=global
*threads
1 main
6 JDWP Handler
5 Async GC
4 Reference Handler
3 Finalizer
2 Signal Handler
*methods
0x080f23f8 java/io/PrintStream write ([BII)V
0x080f25d4 java/io/PrintStream print (Ljava/lang/String;)V
0x080f27f4 java/io/PrintStream println (Ljava/lang/String;)V
0x080da620 java/lang/RuntimeException     <init>      ()V
[...]
0x080f630c android/os/Debug startMethodTracing ()V
*end
```

Fig. 2. Example of key log.

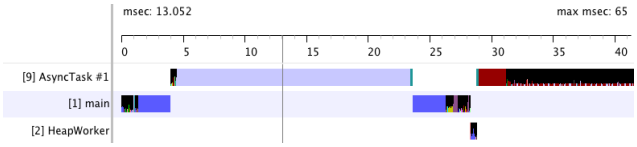

Fig. 3. Timeline view of Traceview. Left side indicates thread ID and its name in each row. Right side shows start and stop times of each thread and method. Each method is represented by a unique color.

Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
0 (toplevel)	100.0%	2060.709	0.0%	0.272	3+0	686.903
1 java/lang/Thread.run ()V	99.4%	2048.673	0.0%	0.013	1+0	2048.673
2 java/util/concurrent/ThreadPoolExecutor$Worker.ru	99.4%	2048.660	0.0%	0.003	1+0	2048.660
3 java/util/concurrent/ThreadPoolExecutor.runWorke	99.4%	2048.657	0.0%	0.016	1+0	2048.657
4 java/util/concurrent/FutureTask.run ()V	99.4%	2048.572	0.0%	0.002	1+0	2048.572
5 java/util/concurrent/FutureTask$Sync.innerRun ()V	99.4%	2048.570	0.0%	0.018	1+0	2048.570

Fig. 4. Profile view of Traceview, including method name, inclusive elapsed time, exclusive elapsed time, and number of calls.

dmtracedump. Traceview is a graphical viewer to load the trace file and display such basic information as the enter and exit times of each method. To display the profiled information, such as an elapsed time of a method, Traceview makes a call stack from the data log and then calculates the elapsed time, the number of calls of a method, and the context switching. Traceview offers a timeline view (Fig. 3) to show when each

thread and method starts and stops and a profile view (Fig. 4) to provide summaries of each method.

Dmtracedump is a console-based profiling tool for the Android applications in the Android SDK. Dmtracedump produces a profiling result as a text or HTML file, which contains the inclusive and exclusive elapsed time and the number of calls of each method. Also, dmtracedump generates a call graph.

2. ARM Streamline Performance Analyzer

ARM Streamline is one of the solutions for system-wide software profiling and performance analysis for the ARM-based platforms. This tool has not been developed for the Android platform. One of the strengths of ARM Streamline is fast profiling. ARM Streamline uses a sampling mechanism for collecting performance data. Due to the difference of profiling methodologies, ARM Streamline is much faster but less accurate than the Dalvik VM trace mechanism. Its other strength is the ability to support various low-level functionalities that are related to HPC and kernel events, as shown in Figs. 5 and 6. However, to analyze the performance of the Android platform and its applications, we should use Traceview with ARM Streamline together. Obviously, it is too difficult and inconvenient for most users.

Fig. 5. Timeline view of ARM Streamline performance analyzer.

Self	Instances	Function Name	Location	Image
24.59%	25	[libc.so]	<anonymous>	<anonymous>
24.04%	2	[libskia.so]	<anonymous>	<anonymous>
18.91%	1	[kernel]	<anonymous>	<anonymous>
8.52%	1	[idle]	<anonymous>	<anonymous>
8.42%	28	[libdvm.so]	<anonymous>	<anonymous>
3.96%	20	[dev/ashmem/dalvik-jit-code-cache]	<anonymous>	<anonymous>
2.14%	1	[libGLESv1_CM_POWERVR_SGX530_125.so]	<anonymous>	<anonymous>
1.63%	2	[gator]	<anonymous>	<anonymous>
1.27%	18	[libandroid_runtime.so]	<anonymous>	<anonymous>
1.15%	18	[libutils.so]	<anonymous>	<anonymous>

Fig. 6. Function view of ARM Streamline performance analyzer.

III. Overview of AndroScope

To support insightful performance analysis for all layers of the Android platform and its applications, we developed the AndroScope performance analyzer based on the existing Dalvik VM trace mechanism and Traceview by adding the following attractive features: 1) collection of HPC events, such as IPC, branch miss prediction, and L1 data cache miss for low-level performance analysis; 2) compiler instrumentation and trace of native codes to collect performance data from all software components of the Android platform, that is, the application layer of the Android stack including Dalvik VM, core libraries, and Android libraries (Webkit, SQLite, libc and so on), as shown in Fig. 7; 3) support of fast data processing for mass trace log; and 4) various timeline, profile, and graphical views for deep and intelligent (that is, insightful) performance analysis. The organization of AndroScope is shown in Fig. 7.

Our system consists of four processing components: instrumentation, data collection, data processing, and

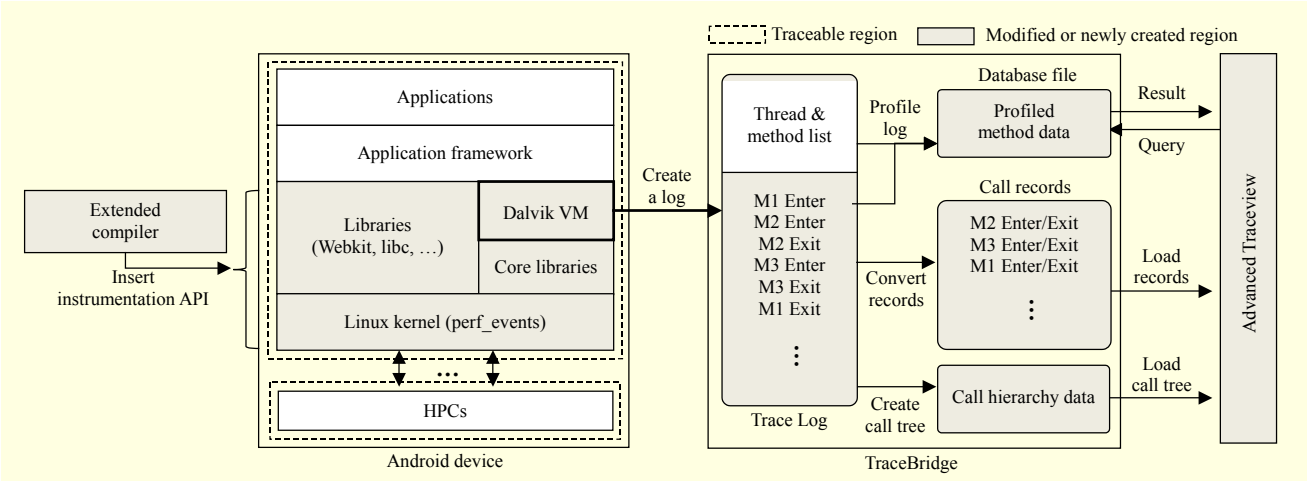

Fig. 7. Organization of AndroScope. Gray regions represent modified or newly created regions to support HPC and native trace. Components surrounded by dashed lines are traceable regions by AndroScope. Linux kernel is not traced and instrumented because doing so significantly increases trace overhead.

visualization. Dalvik VM is in charge of instrumenting Java methods, collecting their performance data, and creating a log. The traceable regions are enclosed by dashed lines in Fig. 7. Also, our modified GNU compiler collection (GCC) compiler system can instrument native codes automatically and selectively either by itself or a programmer's declaration options. TraceBridge is added for the fast handling of a mass trace log. The trace log is partitioned into three parts: profiled data, call records, and call hierarchy data. Traceview is responsible for displaying each part in different views.

IV. Performance Data Collection and Trace

1. Trace with Hardware Performance Counter

For low-level performance analysis, AndroScope collects HPC data in addition to time in the original Dalvik VM trace mechanism. There is a set of special purpose registers built into most modern microprocessors to keep a count of hardware events, such as CPU cycles, instructions architecturally executed, data cache misses, and so on. These HPC events can provide the user an opportunity for detailed low-level performance analysis. For instance, we can easily derive IPC, that is, a basic performance factor from the collected CPU cycles and executed instruction events.

To support the HPC trace, we use perf_events, which is a driver to access HPCs in the Linux kernel, as shown in Fig. 7. We add perf_events of Linux kernel 2.6.35 for Gingerbread to lower-version kernels, for example, 2.6.32 for Froyo, because perf_events of the lower-version kernels were not available on our experimental mobile platform to use the ARM Cortex A8 architecture [6]. The ARM Cortex A8 architecture provides 50 HPC events, and the representative HPC events are listed in Table 1. A user can trace HPCs selectively by enabling them on our AndroScope GUI.

Table 1. Representative HPC events of ARM Cortex A8 [6].

Event name	Description
CPU_CYCLES	CPU cycles
INSTR_EXECUTED	Instructions architecturally executed
L1_DATA_MISS	L1 data misses
L1_INST_MISS	L1 instruction misses
L1_INST	L1 instruction cache accesses
L2_CACHE_MISS	L2 cache misses
PC_BRANCH_MIS_PRED	Branches mispredicted or not predicted
PC_BRANCH_TAKEN	Branches predicted taken

2. Trace of Native Libraries and Instrumentation

The Dalvik VM trace mechanism is able to trace activities of Java methods and Java native methods written in JNI. Therefore, the mechanism cannot provide users with what happens inside of Java natives and other functions, including Dalvik VM itself. Our advanced trace mechanism overcomes these limitations by inserting instrumentation codes at the beginning and the end of native functions to collect trace data from any code.

We extend the GCC 3.4.4 compiler front end to automatically and selectively insert instrumentation codes for the user's convenience. The compiler provides two ways to control the instrumentation: by compiler options and by function attributes. We use three pre-existing GCC options with a new one, as shown in Table 2.

Table 2. Instrumentation options for native trace.

Option	Explanation
instrument-functions	Full source instrumentation
instrument-functions-exclude-function-list	Function list to be excluded
instrument-functions-exclude-file-list	File list to be excluded
instrument-functions-minimum-line	Functions to be excluded by the minimum number of function lines (new option)

The instrument-functions-minimum-line is to discard trivial functions based on source line counts because an overhead of the instrumented trace code has a decisive effect on the function performance in the case of a small function. For example, as a default, native functions with less than eight lines are excluded. The extended compiler accepts two function attributes, which are specified in source code: __*attribute*__ ((*do_instrument_function*)) and __*attribute*__ ((*no_instrument_function*)), which are to turn the instrumentation on and off for the corresponding functions regardless of the compiler options.

The following elements are excluded in the compiler-based instrumentation for the native trace, as shown in Fig. 8: 1) instrumentation code itself, 2) PThread library [7], and 3) naked functions. The tracing of the instrumentation code itself is obviously not intended and incurs a malfunction in the trace. Until a "thread local storage" is completely initialized in the PThread creation, the trace component cannot identify a thread ID of a newly created thread, which is one of the trace record entries. The naked function, which is specified by a function attribute, does not have a prologue and an epilogue; such a function is usually used for writing a function only with inline assembly codes. Therefore, inserting instrumentation codes at

```
/* To register a hook function */
__attribute__((no_instrument_function))
void deeptrace_register_hook_functions(
    void (*hook_function) (void**, int, void*)
)
{
    if(hook_function) {
        deeptrace_hook_function = hook_function;
    }else{
        /* if hook_function is null, use the default dummy hook function */
        deeptrace_hook_function = deeptrace_dummy_hook_function;
    }
}

/* Will be called at the beginning of functions */
__attribute__((no_instrument_function))
void deeptrace_func_enter(void **info, void *object)
{
    deeptrace_hook_function(info, 0/*enter*/, object);
}

/* Will be called at the end of functions */
__attribute__((no_instrument_function))
void deeptrace_func_exit(void **info)
{
    deeptrace_hook_function(info, 1/*exit*/, 0);
}
```

Fig. 8. Example of function attributes. This code is instrumentation API which has to be excluded for native trace.

the beginning and the end of the naked function breaks calling convention.

3. Selective Trace at Runtime

The change of the instrumentation options (compiler flags and function attributes) incurs recompilation of applications and libraries, which is an extremely time-consuming task. The compilation of full mobile software takes several hours in general. Therefore, our trace mechanism supports runtime filtering for selective tracing with an external configuration file (*trace_method.conf*). The file contains method information, such as a class name, a method name, and a signature that identifies a specific method. If the file has no information, all methods are traced. For example, specific methods in a critical path can only be traced by the selective trace.

4. Trace of Startup Codes

To trace methods, a process ID of an application is required because we need to know which process will be traced. Therefore, the application must already be started before we start a trace. However, sometimes we need to know the performance in the part of a startup code of the application. So, our trace mechanism supports startup trace with an external configuration file (*trace_startup.conf*). This file specifies an application's information, such as an application name and a name of a trace log file to be created. If an application is registered in the *trace_startup.conf*, a trace will be started simultaneously when the application is started.

```
* Record format:
* u1 CPU ID
* u1 thread ID
* u4 method ID | method action
* u4/EBV time delta since start, in usec
* u4/EBV heap size (Optional)
* u4/EBV heap allocated (Optional)
* u4/EBV heap object (Optional)
* u8/EBV HPC values (Optional)
```

Fig. 9. Modified record format for our trace mechanism.

5. Heap Information and Holding Thread

Java applications allocate a significant amount of heap memory and undergo garbage collection frequently. However, the garbage collection does not guarantee that it always performs at the right time, and the Android system usually holds a larger amount of heap memory. Therefore, the heap memory behavior is an important performance factor in the Android platforms. Our trace mechanism provides such heap memory information as heap size, allocated heap size, and the number of allocated objects. Also, it provides information when garbage collection has been performed.

Also, the original trace mechanism provides method activity and its thread information. When an application runs for a long time, a high number of threads are frequently created and terminated, which often results in the disappearance of thread information. Therefore, our trace mechanism provides a holding thread option so as not to terminate threads until their trace finish is guaranteed. There is a limitation on the number of threads under Linux. Note that we can see the limitation in /proc/sys/kernel/threads-max. The default is the number of memory pages divided by four, and we can modify the value as necessary.

6. Compact Log

Our trace log is much bigger than an original log because HPC events and native traces can generate long records, as shown in Fig. 9. For instance, if we collect four HPC events and heap information, an extended record length is 54 bytes and the original length is 9 bytes (thread ID, method ID and action, and time delta). Therefore, an extended log size is about six times bigger than the original log. To reduce the size of the trace log, our trace mechanism provides a compact log option.

An extensible bit vector (EBV) [8] is a data structure with an extensible data range. If the compact-log option is turned on, heap information and HPC values are encoded by EBV-8. Each byte contains a single extension bit that indicates whether there is another byte or not. For example, as shown in Fig. 10, 128 uses 2 bytes and the most significant bit in the first byte is 1 to indicate that the next byte exists. Therefore, a short integer value, such as CPU ID or thread ID, is not encoded by EBV-8

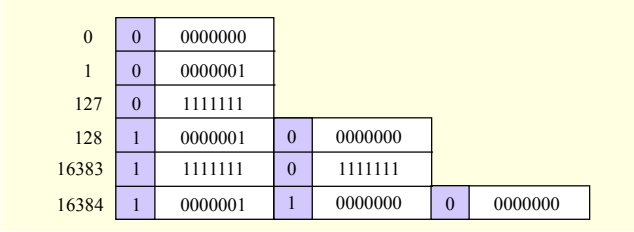

Fig. 10. EBV-8 word format.

because there is either no advantage or overhead is incurred when we use 2 bytes to express a 1-byte integer value. Method ID is also excluded because it is a fixed range and a 4-byte value.

Generally, time and HPC values always increase the log file size linearly. To reduce the log more effectively, delta value (difference between current and previous value) are encoded by EBV-8.

V. TraceBridge

1. TraceBridge for Fast Data Processing

To provide detailed performance analysis, we extend log data and implement a GUI based on Traceview, which we will hereafter refer to as the advanced Traceview. However, the original Traceview has been implemented without considering the mass trace log. Actually, its default maximum log size is only 8 MB. Traceview is written in Java and run as an Eclipse plugin, and it suffers from performance degradation due to limited heap space.

Therefore, to resolve the problem, we implement a middleware called "TraceBridge" to process the collected data fast and to handle the mass data log. TraceBridge analyzes a trace log file and creates call records that have start and end information of function calls and a database file that has statistics information of each class and function with HPC events. Also, TraceBridge builds a call hierarchy file for displaying the graphical call graph in Traceview. Each node of the graph represents a method with its name and execution time, and an edge of the graph shows the call relationship between methods. TraceBridge is written in C for Linux and Windows, and it can process a mass log extremely fast. For example, 300-MB log data processing takes only about one minute.

2. Restoring Corrupted Call Stack

If a native trace option is turned on, data processing for building a call stack sometimes fails due to *setjmp* and *longjmp* [9] in the native libraries. The setjmp and longjmp are used to manipulate the flow of the execution sequence in C. The longjmp concept is similar to using *goto*, but the longjmp can jump across functions. Therefore, these functions must be handled carefully to maintain the call stack.

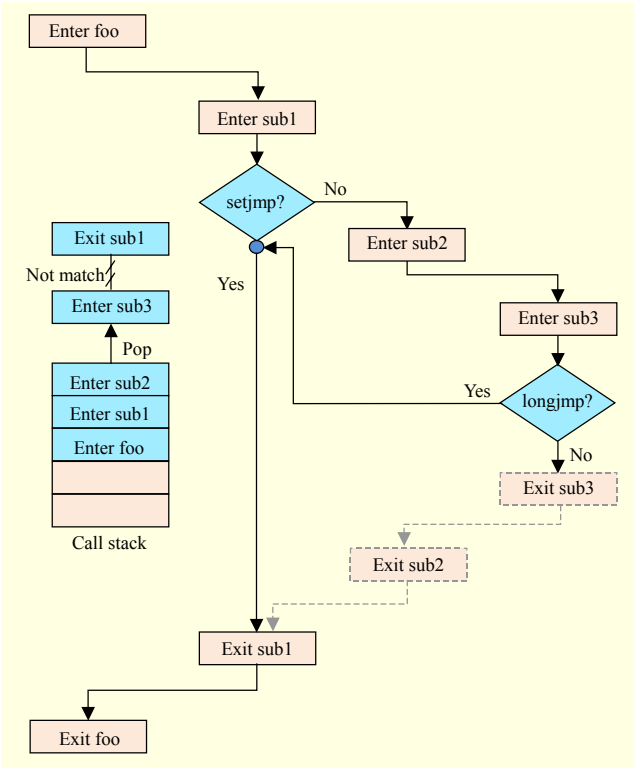

Fig. 11. Example of call stack corruption by setjmp and longjmp.

Figure 11 shows an example of setjmp and longjmp execution to corrupt a call stack build. In TraceBridge, a call stack is made by manipulating a sequence of method and function call records. Fundamentally, an entry and an exit of a method should be in pairs, that is, *enter foo* → *enter sub1* → *enter sub2* → *enter sub3* → *exit sub3* → *exit sub2* → *exit sub1* → *exit foo*. As shown in Fig. 11, the entry of the sub3 function cannot make a pair with its exit because, when the enter sub3 is popped, it is compared with the exit sub1. Therefore, we address this problem and implement a handling policy for setjmp and longjmp. If the call stack is not matched due to longjmp, dummy exits are added to make pairs. In this case, exit sub3 and exit sub2 are added in sequence and their exit time is set to the same time as exit sub1.

3. Segment Partitioning Algorithm for Context Switching

A segment is the smallest unit for displaying a method call in the timeline view. However, an entry and an exit of the segment does not simply imply an entry and an exit of a call. If a context switching occurs during a call, the call can be partitioned into more than one segment. However, we cannot

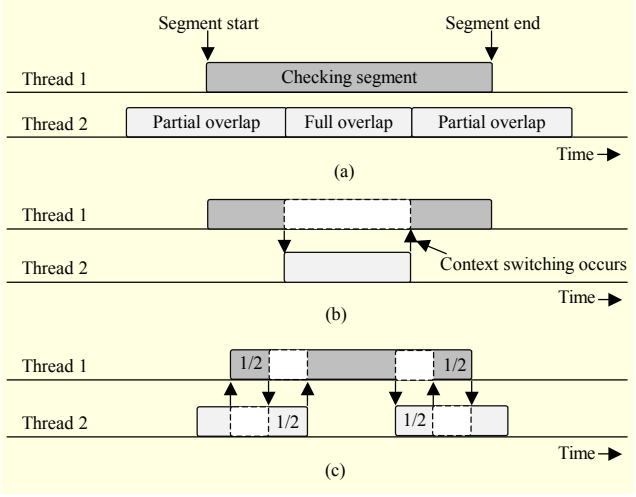

Fig. 12. Segment partitioning algorithm for context switching: (a) context switching cases, (b) full overlap case, and (c) partial overlap case.

know the exact context switching events because the trace mechanism is based on the method-by-method tracing. Thus, we need to develop a segment partitioning algorithm for context switching, and the algorithm is explained in Fig. 12.

All segments are scanned from left to right and top to bottom. If one segment is selected, we can identify the possible context switching cases by finding segments to overlap their execution time with the selected segment.

There are two scenarios of context switching, as shown in Fig. 12(a). In the case of the full overlap, as shown in Fig. 12(b), the overlapped period of the checking segment is removed, and it is assumed that the context switching occurs between the overlapped segments. Regarding Fig. 12(c), we only know the following information: 1) Thread 2 runs first; 2) Thread 1 runs before Thread 2 finishes, that is, a context is switched at this time; and 3) Thread 2 finishes before Thread 1 finishes, that is, a context is switched a moment prior. However, we cannot know exactly when the context is switched. Therefore, the idea that the overlapped segments are partitioned into two parts with a half-length between the start time of Thread 1 and end time of Thread 2 is manufactured.

VI. Advanced Traceview

The original Traceview analyzes the trace log file and displays the analyzed data. However, Traceview takes so much time to analyze a huge amount of call records. To solve the problem, the advanced Traceview is only responsible for displaying profiled results due to TraceBridge's data processing. As a result, Traceview becomes much lighter and performs faster.

A screenshot of the advanced Traceview is shown in Fig. 13,

Fig. 13. Screenshot of advanced Traceview integrated into DDMS: (a) trace interface and timeline view and (b) profile view.

and we present distinguished features and functionalities of the advanced Traceview compared to the original version in the following subsections.

1. Advanced Timeline View for Mass Records

The advanced Traceview includes a timeline view that displays call records with times and their names in a time-based sequence and shows their context switching. As the size of a call record increases, the timeline view on the original Traceview is extremely slow. When we are zooming in or out of the timeline segments for detailed analysis, we often encounter this problem. Moreover, whenever the user moves the mouse pointer over an area of the timeline view, all of the segments are redrawn. In the case of small logs, this problem is a minor issue. However, in the case of mass logs, we must improve the performance to remove a flickering. Therefore, to resolve the problem, we implement a double buffering.

2. HPC and Heap Memory Views

The advanced Traceview also provides HPC and heap memory views in the time sequence, as shown in Fig. 13. The HPC view shows the collected HPCs and their derived events, such as IPC, and the heap memory view displays heap memory

information, such as heap size, allocated heap size, the number of objects, and garbage collection. In addition, both the HPC and the heap memory views are linked with the timeline view. Therefore, when we are zooming, scrolling, and moving horizontally, all the views are affected together.

3. Class and Method View with Java and Native Method

The advanced Traceview provides a profile view that is partitioned into a class view and a method view. The class view presents statistics information, such as the total number of methods, an accumulative time, and HPC event values grouped by a class. Also, the method view shows method information and detailed performance data, such as the number of calls, elapsed time, HPC event values, and so on, for all the methods in the selected class. In addition, we can determine which method is a JNI or native function through the JNI and Class columns. If a method is native, its class name is shown as __native__.

4. Graphical Call Tree Diagram of a Method

The advanced Traceview provides a graphical call tree diagram of each method. TraceBridge creates a call hierarchy file to contain methods and their call relationship information. The call tree diagram presents call paths from a root (toplevel) to its children inside the selected method. Since the diagram shows both Java and native functions, we can know the full path from a toplevel to native functions in detail.

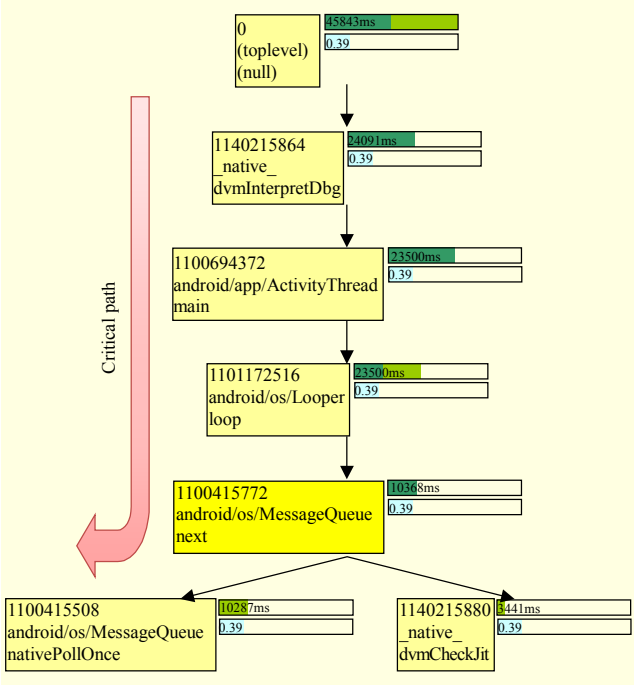

Fig. 14. Example of call tree diagram of deskclock application.

Each node has a method ID, a method name, HPCs, and an inclusive time in the method and function. However, in the case of a long trace, the call graph is too big to display. Therefore, the threshold can be assigned by a TraceBridge option from 0% (nothing) to 100% (everything) to display brief call graphs.

Figure 14 shows an example of the call tree diagram of *android/os/messageQueue.next*. Each time highlighted in dark green is the inclusive time of its dominant child and the time highlighted in light green is its inclusive time. The blue bar indicates an HPC value, which is an IPC in this case. Also, from the graph in Fig. 14, we can easily find the critical path.

VII. Evaluation

This section evaluates the performance of AndroScope. Also, we compare AndroScope with the ARM Streamline, which is a commercial tool and the most popular tool. The evaluation host system has two 2.6-GHz cores with 3 GB of main memory on 64-bit Linux and a target system that is a commercial product that has a 1-GHz core with 512 MB of main memory on Gingerbread.

Figure 15 shows the experimental environment with a real commercial device and a web browser application. The Android phone is connected to a PC through the USB interface, and the USB debugging mode is turned on. To trace the Android platform and its applications, we can use the *Trace Config* button, which is able to set the trace configuration for enabling and selecting HPC events, turning on native trace options, and so on. Also, the *Trace Start/End* button can be used to start collecting and analyzing the performance data.

1. Log Size Reduction

Figure 16 shows log sizes of applied and unapplied log compactions in the case of tracing four HPC events (dcache

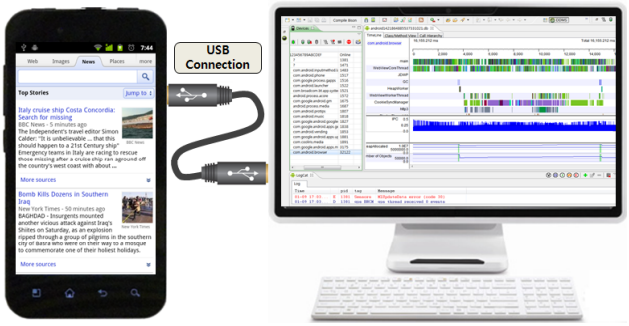

Fig. 15. Android phone used for experiment and analyzed result of web browser application by AndroScope. AndroScope view is matched with left side of Fig 13.

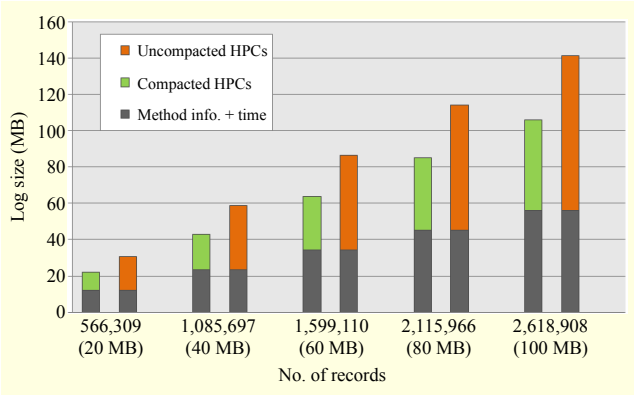

Fig. 16. Log size comparison of compact log and normal log in case of tracing with four HPC events.

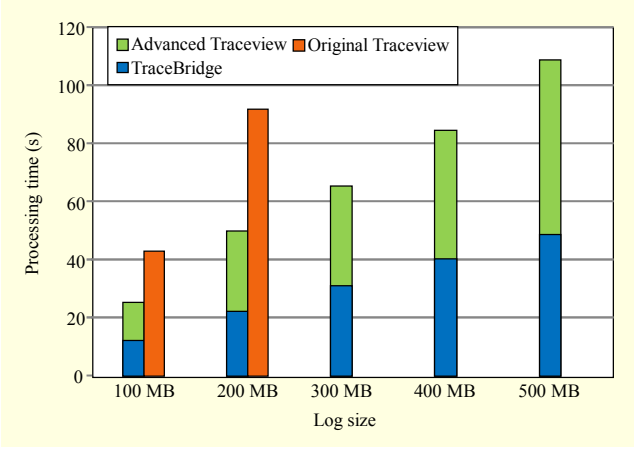

Fig. 17. Comparison of data processing performance for mass log.

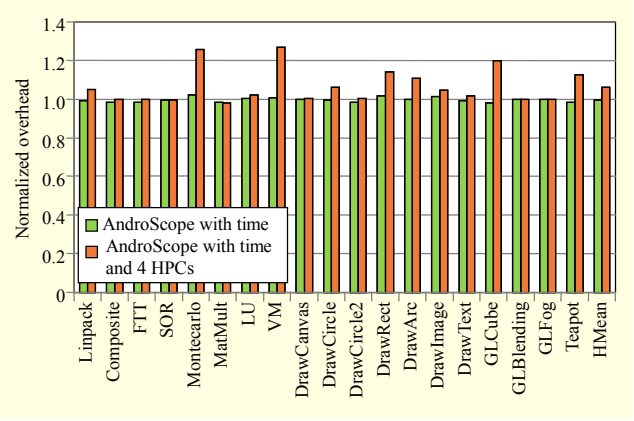

Fig. 18. Execution slowdown according to HPC data collection only in Java methods. All results normalized over originally built-in Java trace with elapsed time.

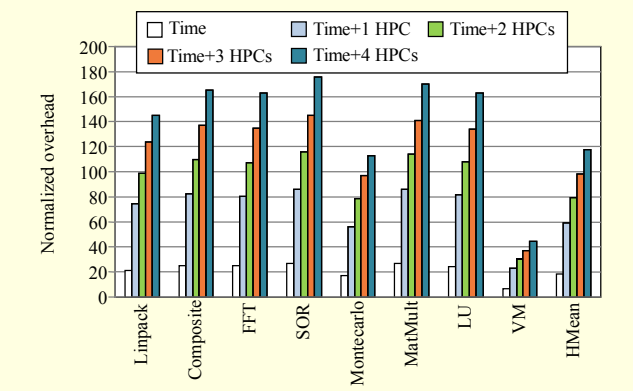

Fig. 19. Execution slowdown according to instrumentation of both Java and native methods. All results normalized over time collection by originally built-in Dalvik VM.

read/write, instructions, and cycle events). Remember that only HPC values are encoded by EBV-8. As a result, the log sizes of only four HPCs and total information are reduced by about 44% and 26% on average, respectively. The reduction ratio of the total log size is proportional to the number of pieces of EBV-format data.

2. Data Processing Performance

We measure the data processing time according to the log size, and the comparison result is shown in Fig. 17. It should be noted that the original Dalvik VM trace cannot profile the performance data above 300 MB. It is because the Android target system does not have enough memory to collect huge amounts of data. Our system flushes the collected data into flash memory on the target instead of keeping it in DRAM to avoid the memory shortage problem. Also, the original Traceview cannot handle the mass data due to the *Out of Memory Error : Java heap space*. The advanced Traceview provides much better performance than the original Traceview because the data size is reduced by TraceBridge in advance, that is, before displaying the data.

Despite the performance improvement of Traceview, the performance display still occupies about half of the total execution time. Traceview loads and converts large amounts of data, which requires a large amount of heap space to store it. However, there is limited heap space for execution. This is why Traceview cannot be further enhanced.

3. Data Collection and Instrumentation Overhead

Figure 18 shows the execution slowdown of time and HPC data collection via only Java methods by our AndroScope with respect to time traces supported originally by Dalvik VM. To measure the HPC data collection overhead, we use 0xbench [10], which is an integrated Android benchmark suite. We optimize the built-in trace mechanism by Dalvik VM, and there is therefore a little speedup in the AndroScope time trace. When four HPC events are collected, including elapsed time, the overhead is only about 6% on average.

Figure 19 shows the execution slowdown due to instrumentation overhead in both Java and native methods. In this case, every static and shared library whose lines are more than eight are instrumented. Therefore, a huge amount of native methods are traced along with Java methods. In general, the instrumentation overhead is proportional to the number of traced methods. Due to this reason, the execution time of the time collection in Java and native methods is 18 times slower than that of only Java methods. When four HPC events are collected, including elapsed time, the data collection is only 117 times slower on average. However, all libraries are traced in this case. We can significantly decrease the instrumentation overhead by excluding unimportant libraries.

4. Comparison with Typical Performance Analysis Tool

Table 3 compares two tools, AndroScope and ARM Streamline. Our tool can be used for any CPU platform with a minor change in implementation, that is, by porting the perf_events driver in the Linux kernel, whereas ARM Streamline can only be used for an ARM-based platform. Also, there is a difference in the profiling methodology. AndroScope uses a trace mechanism whereas ARM Streamline conducts sampling. Due to the profiling mechanism, AndroScope is much better than ARM Streamline in terms of accuracy. On the other hand, ARM Streamline is much faster than AndroScope. Additionally, AndroScope supports the Java trace, whereas ARM Steamline is not capable of doing so. It should be noted that ARM streamline is only applicable for the ARM-based systems, not for the Android platform. Therefore, performance analysis of Java applications is not possible using ARM Streamline.

Table 3. Comparison of AndroScope and ARM Streamline.

Functionality	AndroScope	Streamline
Platform support	Any	ARM only
Profiling methodology	Trace	Sampling
Accuracy	High	Normal
Overhead	Normal	Low
JAVA trace	Yes	No
Native trace	Yes	Yes
Kernel trace	Yes	Yes
HPC events	Yes	Yes
Heap memory usage	Yes	Yes
Call graph	Yes	Yes
Multicore support	Yes	Yes
Integrated into Eclipse	Yes	Yes

VIII. Conclusion

We introduced AndroScope to provide low-level performance analysis of the Android platform with HPCs, memory usage, and so on, and to support tracing Java applications, Dalvik VM, core libraries, Android libraries, and even Linux kernels. To the best of our knowledge, AndroScope is the first tool to collect and analyze performance data from all software layers of the Android platform.

To provide a native trace, we implemented the instrumentation interface in GCC and allowed the selective instrumentation of specific methods to reduce tracing overhead. Also, we offered TraceBridge for handling mass logs efficiently and the advanced Traceview for visualizing the performance data in many ways.

References

[1] *ARM DS-5 Using ARM Streamline*, ARM DUI 0482F, ARM Ltd., UK, 2011.

[2] R. Gordon, *Essential JNI: Java Native Interface*, Upper Saddle River, NJ: Prentice-Hall, Inc., 1998.

[3] MIPS Technologies, Inc., "Getting Started with Android," Sunnyvale, CA, USA, 2011. http://developer.mips.com/android/getting-started-with-android/

[4] Intel Corporation, "Android 3.2 on Intel Architecture," Santa Clara, CA, USA, 2012. http://software.intel.com/en-us/articles/android-32-on-intel-architecture

[5] M. Cho et al., "AndroScope for Detailed Performance Study of the Android Platform and Its Applications," *IEEE Int. Conf. Consum. Electron.*, 2012, pp. 412-413.

[6] M. Baron, "Cortex-A8: High Speed, Low Power." *Microprocessor Report*, vol. 11, no. 14, 2005, pp. 1-6.

[7] D. Buttlar, J. Farrell, and B. Nichols, *PThreads Programming: A POSIX Standard for Better Multiprocessing*, Sebastapol, CA: O'Reilly Media Inc., 1996.

[8] *EPC Tag Data Standard*, EPCglobal, 2010.

[9] W.R. Stevens, *Advanced Programming in the UNIX Environment*, Reading, MA: Addison-Wesley, 1993.

[10] 0xbench. http://code.google.com/p/0xbench/

Myeongjin Cho received his BE, MS, and PhD from the School of Electrical Engineering of Korea University, Seoul, Rep. of Korea, in 2006, 2008, and 2013, respectively. Currently, he is a post-doctoral researcher at Korea University. His research interests include Android performance analysis and optimization, high performance computing, and microarchitecture. He is a member of ACM and IEEE.

Ho Jin Lee received his BE from the Electrical Engineering Department of Korea University, Seoul, Rep. of Korea, in 2011 and is working on his MS in the same department. His research interests include microarchitecture, mobile device performance analysis, and compilers.

Minseong Kim received his BE from the Electrical Engineering Department of Korea University, Seoul, Rep. of Korea, in 2011 and is working on his MS in the same department. His research interests include low-power SoC design and high-performance microprocessor architecture.

Seon Wook Kim received his BE from the Electronics and Computer Engineering Department of Korea University, Seoul, Rep. of Korea, in 1988. He received this MS from the Electrical Engineering Department of Ohio State University, Columbus, OH, USA, in 1990 and his PhD from the Electrical and Computer Engineering Department of Purdue University, West Lafayette, IN, USA in 2001. He was a senior researcher at the Agency for Defense Development from 1990 to 1995 and a staff software engineer at Intel/KSL from 2001 to 2002. Currently, he is a professor with and the associate dean of the College of Engineering, Korea University. His research interests include compiler construction, microarchitecture, and SoC design. He is a senior member of ACM and a member of IEEE.