A Systematic Debugging and Performance Analysis Framework for Android Platforms

Juhyun Kyong
School of Computer Science,
Kookmin University,
Seoul, Korea
juhyun.kyong@gmail.com

Minsuk Lee
Dept. of Computer Engineering,
Hansung University,
Seoul, Korea
minsuk@hansung.ac.kr

Gu-Min Jeong
School of Electrical Engineering,
Kookmin University,
Seoul, Korea
gm1004@kookmin.ac.kr

Chanik Park
Samsung Electronics,
Hwasung City, Korea,
ci.park@samsung.com

Sung-Soo Lim
School of Computer Science,
Kookmin University,
Seoul, Korea,
sslim@kookmin.ac.kr

ABSTRACT

In this paper, we describe a new debugging and performance analysis method to obtain a fast and efficient Android-based development. Based on a USB-based kernel debugging, we can debug a kernel simply through a single USB interface. Also, by use of a whole layer performance analyzer, performance bottlenecks can be found quickly and accurately.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids and Testing tools

General Terms

Measurement, Performance, Design, Reliability, and Experimentation.

Keywords

Performance analysis, Systematic debugging, Android platform.

1. INTRODUCTION

Due to the multi-layered architecture of Android platform, we need systematic tools for the debugging and the performance analysis. Above all, these tools should provide the monitoring of the application behavior and the analysis of the performance bottlenecks hierarchically. Current tools, available today, have the following two problems: First, since the existing system level debugging methods make the debugging time consuming with the complicated equipment and the software configuration, we need a rather simpler system-level debugging method using the basic configurations of target devices. Second, since interactions and correspondences among the different software layers significantly affect the performance degradation, there should be performance analyzer tools considering the whole layers in order to find the performance bottleneck.

In this paper, we propose a debugging and performance analysis framework for Android-based smart devices supporting both a convenient system-level debugging method and a whole layer hierarchical performance analysis. We have implemented a system-level debugging method not based on the complicated JTAG interface but based on the simple USB interface. Using a whole layer performance analysis tool, we can trace and analyze the application execution. Through the trace and the analysis, we can find exact locations of the performance bottlenecks within software layers from Android application threads to Linux kernel services.

We have implemented these debugging and performance analysis frameworks as a single tool. The effectiveness and validity of the proposed method has been shown through the experiment of the commercial Android smartphones.

The remainder of this paper is organized as follows: In Section 2, we describe the structures of the USB-based debugging framework and the whole layer hierarchical performance analyzer. In Section 3, we show the implementation results of the debugging and performance tools and the conclusion follows in Section 4.

2. DEBUGGING AND PERFORMANCE ANALYSIS FRAMEWORK

2.1 USB-Based Kernel Debugging Framework

Generally, kernel-level debugging methods [3][4] are required since most of critical system failures and problems are caused by interactions within different software layers including kernel layer. There have been two methods for the kernel-level debugging, which are KGDB based method through a serial port in the conventional embedded systems and JTAG based hardware debugging method.

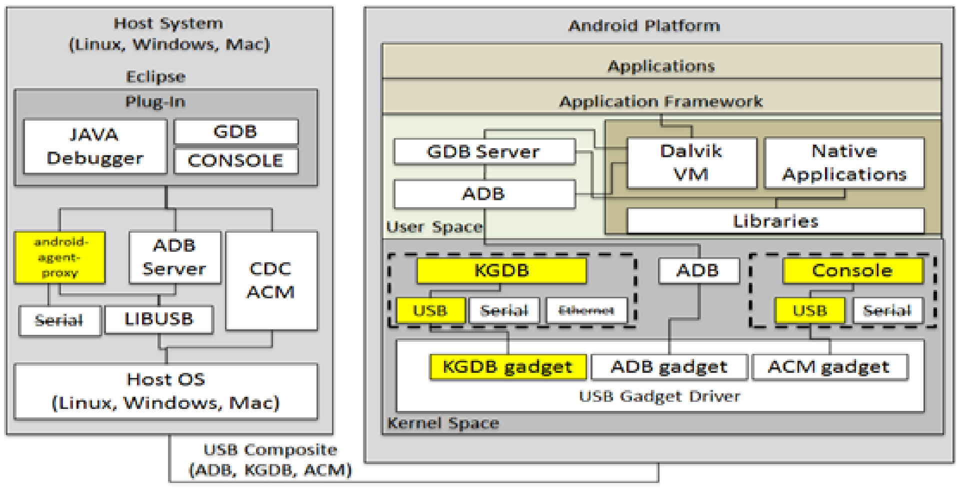


Figure 1 USB-based debugging framework

However, these two methods may be inappropriate for the Android based smartphones, since 1) the serial port-based method does not provide the debugging console when the port is connected for GDB processing and 2) the hardware debugging equipment and tools are not always available. Thus, we propose a kernel debugging framework, which is fully based on USB interface and provides both debugging protocol communication channel and debugging console through a single USB interface.

We have implemented the proposed USB-based kernel debugging framework by modifying both host and device side. For a host side, we redesign an android-agent-proxy to support USB interface providing both debugging channel and debugging console, while we modify the KGDB to support USB interface onto KGDB gadget module at device side.

Figure 1 shows the overall architecture of the proposed USB-based kernel debugging framework. As shown in Figure 1, the android-agent-proxy, KGDB and Console modules are modified to provide USB support. Since we use the existing protocol in ADB(Android Debugging Bridge), all the functions in ADB can be supported in the new debugging framework.

One of the most important things for the development is how to implement USB driver for the USB-based kernel debugging framework. Figure 2 shows the way of the USB driver implementation. Unlike the application-level remote debugging

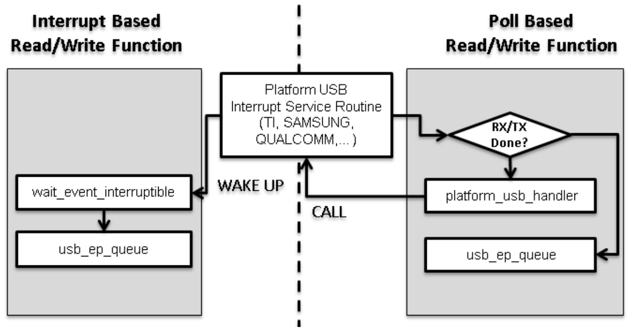


Figure 2 Polling-based USB driver

using GDBSERVER, the KGDB performs debugging functions through exception handling when the debugging event occurs. Original implementation of USB transmission based on interrupt processing basically complicates USB-based KGDB processing since both operations are exception-handling based and could not be combined together at the same time. Therefore, the USB device driver should be re-implemented using polling method instead of interrupt basis.

2.2 A Whole Layer Performance Analyzer

Android application framework based on Dalvik VM, native system calls processing, and kernel events handling [5] have their own performance analysis tools and methods without any consideration on correspondences among the layers.

For a whole layer performance analysis, we should integrate these separate methods and make information exchange channels among layers. The offline analysis could combine the results extracted from each software layer analysis. By combining and integrating available open source performance analysis tools, we implemented a whole layer performance analysis tool set.

Also, for the Android framework layer profiling, DDMS has been slightly modified and, for the kernel layer profiling, *Ftrace* [1][2] has been used. Since the *Oprofile*[6] uses a time-based sampling, it often misses the important kernel events during analysis. Considering these facts, we selected *Ftrace* instead of *Oprofile* as a kernel layer profiler. While *Ftrace* is accurate in performance profiling, there should be additional costs to perform kernel events profiling. Therefore, minimizing the profiling cost as well as maintaining the accuracy of event logging have been a primary issue in performance profiling at kernel layer.

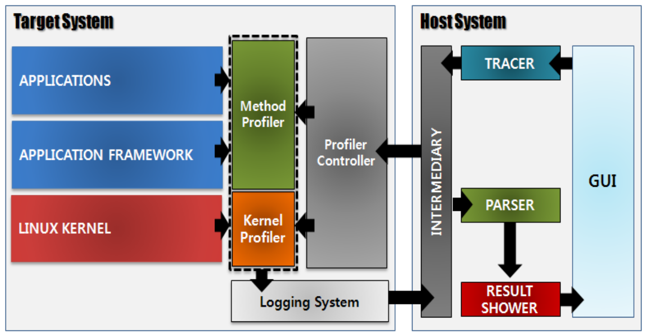


Figure 3 The whole layer performance analyzer

Figure 3 shows the architecture of the proposed tool set. One of the important features for the proposed tools is that the target Android applications do not need to be modified to enable analysis. Profiler-enabled Android platform with a slight modification gets the user-provided target Android methods to be traced at run-time and the tracing for the methods are automatically enabled.

Users can specify the target methods to analyze from GUI of the host tool and the target method information is transferred to target system through Intermediary interface between the GUI and Profiler controller. Profiler controller actually enables logging systems in both application and kernel layers. Method profiler performs a method profiling at Android framework level while Kernel profiler performs a kernel event logging. The logged event data are collected together in Logging system to be transferred to hosts system for an offline analysis. The analysis results are displayed using diverse presentation methods by GUI system at host.

3. IMPLEMENTATION

The proposed USB-based kernel debugging method and the whole layer hierarchical performance analysis framework have been implemented separately and integrated into one single debugging and performance analysis environment. Figure 4 shows the proposed debugging and performance analysis environment. As shown in this figure, the USB interface is the only requirement to perform debugging and performance analysis.

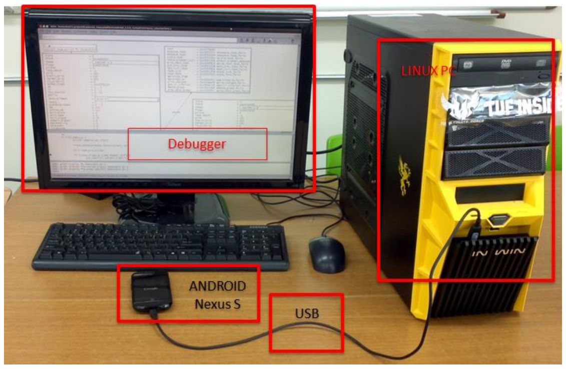
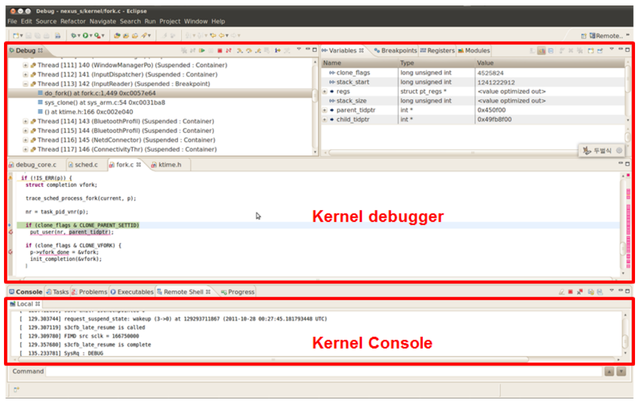


Figure 4 Debugging and performance analysis environment

Since the proposed debugging method provides both debugging channel and debugging console using a single USB interface, both functions could be used in a single debugger window. Figure 5 shows the kernel debugger in Eclipse.



```
Thread [111] 140 (WindowManagerPo) (Suspended : Container)
Thread [112] 141 (InputDispatcher) (Suspended : Container)
Thread [113] 142 (InputReader) (Suspended : Breakpoint)
    do_fork() at fork.c:1,449 0xc0057e64
    sys_clone() at sys_arm.c:54 0xc0031ba8
    () at ktime.h:166 0xc002e040
Thread [114] 143 (BluetoothProfil) (Suspended : Container)
Thread [115] 144 (BluetoothProfil) (Suspended : Container)
Thread [116] 145 (NetdConnector) (Suspended : Container)
Thread [117] 146 (ConnectivityThr) (Suspended : Container)
```

Name	Type	Value
clone_flags	long unsigned int	4525824
stack_start	long unsigned int	1241222912
regs	struct pt_regs *	<value optimized out>
stack_size	long unsigned int	<value optimized out>
parent_tidptr	int *	0x450f00
child_tidptr	int *	0x49fb8f00

```
if (!IS_ERR(p)) {
  struct completion vfork;

  trace_sched_process_fork(current, p);

  nr = task_pid_vnr(p);

  if (clone_flags & CLONE_PARENT_SETTID)
    put_user(nr, parent_tidptr);

  if (clone_flags & CLONE_VFORK) {
    p->vfork_done = &vfork;
    init_completion(&vfork);
  }
```

```
[  129.303744] request_suspend_state: wakeup (3->0) at 129293711867 (2011-10-28 00:27:45.181793448 UTC)
[  129.307119] s3cfb_late_resume is called
[  129.309780] FIMD src sclk = 166750000
[  129.357680] s3cfb_late_resume is complete
[  135.233781] SysRq : DEBUG
```

Figure 5 Kernel debugger in Eclipse

The whole layer performance analyzer could show the behaviors of the activities from all the layers including application thread-level, libraries-level, and kernel event-level. Careful analysis of the hierarchical trace information would be able to accurately locate the performance bottlenecks or problematic parts.

Figure 6 shows the offline visualization results of the whole layer performance analysis data. In the figure, the response times of the events in each layer are shown.

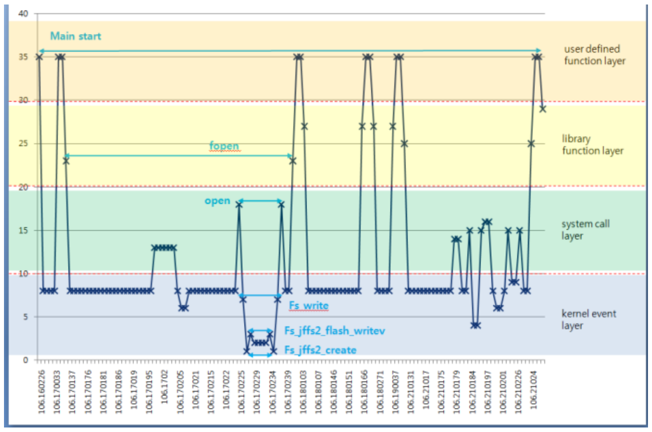


Figure 6 Visualized performance analysis data

4. CONCLUSION

The paper describes two additional debugging and performance analysis support methods: USB-based debugging method and hierarchical performance trace and analysis. The combination of the two proposed methods could lead to significantly enhanced systematic debugging and performance analysis environment. From this, we can obtain faster and more accurate results for the identification of problematic parts in the target software.

5. REFERENCES

[1] Richard Moore, A universal dynamic trace for Linux and other operating systems, *In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[2] Tim Bird, Measuring Function Duration with Ftrace, Proceedings of the Linux Symposium, 2009

[3] Jason Wessel, "The State of Kernel Debugging Technology", LinuxCon 2010, 2010.

[4] Kgdb: linux kenrel source level debugger, "http://kgdb.linsyssoft.com/, 2001.

[5] M. Desnoyers and M. Dagenais, "LTTng: Tracing across execution layers, from the Hypervisor to user-space," *in Proc. of the Linux Symposium*, Ottawa, Canada, Jul. 2008.

[6] J. Levon and P. Elie, Oprofile: A system profiler for linux. http://oprofile.sf.net, September 2004.