國立交通大學

資訊學院　資訊學程

碩 士 論 文

Android 系統上, 開機、網頁瀏覽、串流播放的執行時間剖析和瓶頸分析

Booting, Browsing and Streaming Time Profiling and Bottleneck Analysis on Android-Based Systems

研 究 生：杜之雄

指導教授：林盈達　教授

中 華 民 國　九 十 九 年 六 月

Android 系統上, 開機、網頁瀏覽、串流播放的執行時間剖析和瓶頸分析

Booting, Browsing and Streaming Time Profiling and Bottleneck Analysis on Android-Based Systems

研 究 生：杜之雄　　Student：Tzu-Hsiung Du

指導教授：林盈達　　Advisor：Ying-Dar Lin

國 立 交 通 大 學

資訊學院　資訊學程

碩 士 論 文

A Thesis

Submitted to College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

June 2010

Hsinchu, Taiwan

中華民國九十九年六月

Android 系統上,開機、網頁瀏覽、串流播放的執行時間剖析和瓶頸分析

學生: 杜之雄　　指導教授: 林盈達

國 立 交 通 大 學　　資 訊 學 院　　資 訊 學 程 碩 士 班

摘要

Android 系統在開機、網頁瀏覽及串流播放這三種使用情境下的效能表現顯得不盡理想。在 Android 系統上剖析一種使用情境的執行時間，會發現面臨到三個主要的特性:第一、一個使用情境的執行流程會跨不同軟體層呼叫不同元件，第二、任一軟體層都是由一種以上的程式語言所構成，第三、系統有限的儲存空間。本論文提出一個可以在*多語言多軟體層平台*上找出執行時間瓶頸的*分階段反覆插入剖析方法*。由單一的模組開始，進而逐漸地加入不同軟體層的不同模組來剖析，以避免在剖析過程中產生大量不必要的資料，最後再把不同軟體層執行時所輸出的資料合併分析，藉此找出執行時間瓶頸。此方法被實做測試在一台 Android 的產品上，實驗結果顯示 72%的開機時間花在 user space 環境的初始化，而在 user space 初始化中，44.4%的時間花在啟用背景服務程序及背景管理程序，37%花在先行載入 Java classes 和 resources.實驗結果並顯示，網路是影響網頁瀏覽最主要的因素。在載入一個 2128 kB 大小的網頁的實驗環境下，螢幕繪圖顯示系統僅佔總執行時間的 5%。在 Wi-Fi 環境下播放一段 22 MB 的串流短片，系統需要花 5.7%的時間來做撥放準備的動作。影片資料下載加上資料解碼部分的執行時間共佔了這段準備時間的 72%。

關鍵字: Android，系統開機，網頁瀏覽，串流播放，時間剖析

Booting, Browsing and Streaming Time Profiling and Bottleneck Analysis on Android-Based Systems

Student: Tzu-Hsiung Du Advisor: Dr. Ying-Dar Lin

Degree Program of Computer Science

National Chiao Tung University

Abstract

Android-based systems perform slowly in three perceptible scenarios: booting, browsing, and streaming. Time profiling on Android devices encounters three unique properties: 1) the execution flow of a scenario invokes multiple software layers, 2) each software layer is implemented in different programming languages, and 3) log space is limited. This thesis proposes a staged iterative instrumentation approach that starts profiling a scenario from a single module, restrainedly profiles more modules and layers to avoid enormous irrelevant profiling results, and finally consolidates the profiling results from different layers to find out the bottlenecks. Experiments on the off-the-shelf Android product showed that 72% of booting time is spent on the initialization of user-space environment; specifically, 44.4% of user-space initialization time is to start Android services and managers, and 39.2% is for preloading Java classes and resources. Experimental results also showed that the networking technology is the most significant factor influencing the browsing performance on Android. The time of drawing screen only takes less than 5% of total time for browsing a 2128 kB web page. In the streaming scenario, video preparation causes 5.7% time overhead for playing a 22-MB video file over Wi-Fi connection. Execution time of Video-downloading and data-decoding take 72% ratio of preparation time.

Keywords: Android, booting, browsing, streaming, time profiling.

Contents

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 BACKGROUND **4**

2.1 ANDROID ARCHITECTURE 4

2.2 PROFILING TOOLS ON ANDROID 6

CHAPTER 3 STAGED ITERATIVE INSTRUMENTATION APPROACH **10**

CHAPTER 4 IMPLEMENTATION ON BOOTING, BROWSING, AND STREAMING **13**

4.1 GENERIC IMPLEMENTATION 13

4.2 BOOTING 14

4.3 BROWSING 16

4.4 STREAMING 17

CHAPTER 5 EXPERIMENTAL RESULTS AND OBSERVATIONS **20**

5.1 TESTBED 20

5.2 BOOTING SCENARIO 21

5.3 BROWSING SCENARIO 22

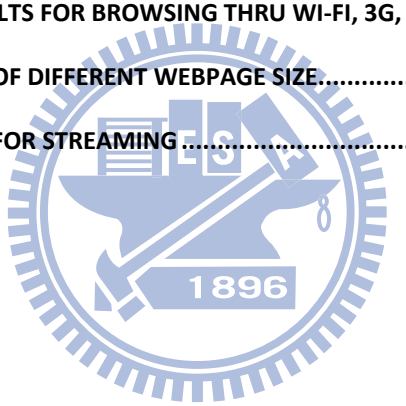
5.4 STREAMING SCENARIO 25

CHAPTER 6 CONCLUSIONS AND FUTURE WORKS **27**

REFERENCE **29**

List of Figures

FIGURE 1 ANDROID ARCHITECTURE 4
FIGURE 2 STAGED ITERATIVE INSTRUMENTATION APPROACH 12
FIGURE 3 BOOTING PROCEDURES 15
FIGURE 4 BROWSING PROCEDURES 17
FIGURE 5 STREAMING PROCEDURES 18
FIGURE 6 TESTBED 20
FIGURE 7 PROFILING RESULTS FOR BOOTING 21
FIGURE 8 DISTRIBUTION OF BOOTING TIME 22
FIGURE 9 TIME PROFILING RESULTS FOR BROWSING THRU WI-FI, 3G, AND 2.75G NETWORKS 24
FIGURE 10 PROFILING RESULTS OF DIFFERENT WEBPAGE SIZE 25
FIGURE 11 PROFILING RESULTS FOR STREAMING 25



List of Tables

TABLE 1 BOOTING TIME AND BROWSING EXPERIENCE ON POPULAR SMARTPHONES.....1

TABLE 2 COMPARISON OF PROFILING TOOLS.....9

Chapter 1 Introduction

Internet-connectable devices, like smartphones, set-top boxes, and netbooks, provide users with the ability to use Internet at anytime in anyplace. Among these Internet-connectable devices, the smartphones ported with Android, a platform for Internet-connectable devices, are expected to catch the most eyeballs in next few years. Android is an open source platform derived from Linux in 2009. Designing a device upon Android can reduce the license royalty for manufacturers. Academia also benefits from using Android to develop new features and prove experimental thoughts due to its open source [1]. Despite the advantages, however, Android-based devices perform slowly in three perceptible scenarios: booting, browsing, and streaming. Booting time is the first perception to users trying a new smartphone; browsing and streaming are two common usage scenarios among smartphone subscribers [2]. Table 1 compares the time spent on booting and browsing on popular off-the-shelf products. Though all products have similar hardware capabilities, the execution on Android products takes a longer time than that on iPhone.

Table 1 Booting time and browsing experience on popular smartphones

Product name	CPU Frequency	Arch	RAM	OS	Boot time(s)[i]	Load page(s)[ii]
HTC G1	528 MHz	ARM11	192M	Android 1.5	42	9.5
HTC Hero	528 MHz	ARM11	288M	Android 1.5	68	9.4
Samsung i7500	528 MHz	ARM11	128M	Android 1.5	60	7.4
iPhone 3GS	600 MHz	ARM Cortex	256M	OS X 3.1	24	5.4

Previous work has intensively studied the performance enhancement technologies for the three scenarios [9-13]. Those literals share three common procedures. First, profiling

[i] The measured time is the interval which starts from pressing the power button and ends at the first GUI window appearing.
[ii] Load www.yahoo.com via Wi-Fi interface.

tools are used to trace the execution flows and running time on a target scenario. Then, execution flows are redesigned to speed up the scenario. Finally, the performance improvement of the redesigned system is proven by profiling again. Apparently, profiling tools play an important role in performance enhancement.

The work [3-4] categorized profiling tools into two types: sampling techniques and instrumentation techniques. Sampling techniques, e.g., OProfile [5] and gprof [6], collect the process-usage statistics by periodically checking which program, or more detailed, which function is occupying CPUs. Instrumentation techniques, e.g., LTTng [7] and KFT [8], insert profiling code into the source code of targeted programs, so the profiling results can be collected during execution.

Two issues are arisen from previous work. First, each profiling tool has different characteristics and therefore only targets on a specific layer or few layers. On the other hand, Android is a complicated system with multiple layers, from application layer, i.e., Java-derived applications, Java virtual machine, toward Linux kernel. In particular, Galenson [9] showed that browsing in Android invokes six layers. Second, previous researchers validated their ideas on development boards [10-14] or emulators [9]. However, hardware is optimized for the commercial products, so the performance results obtained from development boards and emulators may be different from the one in the off-the-shelf products. In addition, unlike a development board equipping sufficient log space, the hardware optimization may leave only limited space on products, e.g., 64 kB log buffer on the HTC Dream smartphone. As a result, using profiling tools having no problem on development boards may encounter the out-of-resource problem on products.

This work proposes a profiling approach to find out the *true* bottlenecks crossing multiple layers for the three scenarios, i.e., booting, browsing, and streaming, on real-world Android products. The common profiling procedures for arbitrary scenarios are

revealed. For each scenario, specific profiling procedures are then developed. The procedures are validated on the off-the-shelf products, and the bottlenecks of each scenario are identified.

The rest of this thesis is organized as follows. Chapter 2 briefs Android architecture and various profiling tools. Chapter 3 proposes the methodologies of profiling procedures, and then Chapter 4 details the implementation. Chapter 5 presents the experimental environment and discusses the profiling results. Finally, Chapter 6 concludes this thesis and offers directions for future research.

Chapter 2 Background

The chapter briefs Android architecture first and then describes and compares various profiling tools in section 2.2.

2.1 Android Architecture

As shown in Figure 1, Android consists of four major software layers which are written in three different programming languages, i.e., Java, C++, and C. From the hardware-dependent toward the user-controllable, the four layers are: Linux kernel, running environment, application framework, and applications.

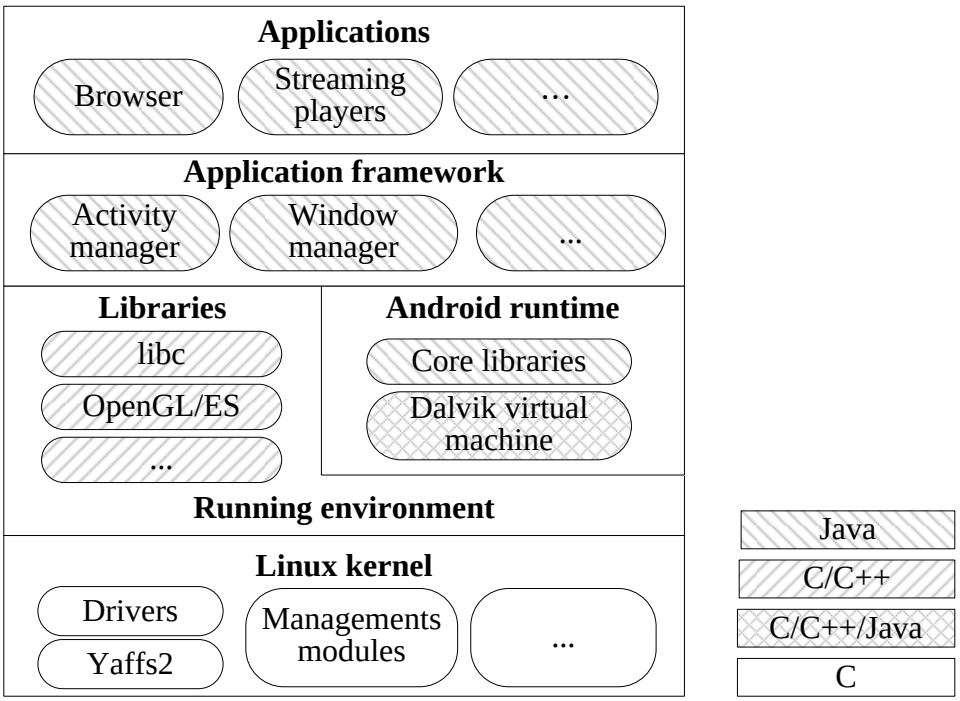


Figure 1 Android Architecture

Linux kernel layer

Android kernel was derived from Linux 2.6 kernel, so it inherits lots of Linux advantages including numerous device drivers and the core operating system functionalities, e.g., memory management, process management, and networking. The inheritances have been proven and shown robust over time. Android also modifies Linux kernel for accommodating the tightly resource-constraint factors in embedded devices,

such as using the Yaffs2 [15], which is optimized filesystem for NAND flash. The whole kernel is written in C programming language, so most of profiling tools which are used in common Linux distributions can be adopted in Android.

Running environment layer

Running environment layer includes two major components, native libraries and Android runtime. Native libraries contain a set of C and C++ libraries, such as libc and OpenGL/ES, to provide common routines for upper layers. Android runtime includes Dalvik virtual machine (VM) and core libraries. Dalvik VM is derived from Java VM which is written in C, C++ and Java. The core libraries, which are written in Java, contain common Java classes for application development. The Android runtime was designed specifically for Android to meet the needs of running in a resource-limited embedded device.

Application framework layer

Application framework layer contains reusable components which can be used by applications. Components in this layer are written in Java, C++ and C programming languages. Activity manager which manages the life cycle of applications is the major component of this layer. In Android, only one foreground graphic-user-interface (GUI) application is allowed to exhibit at one time, and other running applications are managed by the activity manger in the background. Besides, window manager draws the graphic component, such as status bar, for a foreground GUI.

Applications layer

All user-visible applications on Android are written in Java programming language. They include applications written by Google, and applications written by open-source communities and commercial developers.

2.2 Profiling Tools on Android

Profiling tools are used to detect the hotspots of a program or set of programs for alleviating performance issues. A hotspot is a piece of code that is frequently executed or whose execution time is extremely long. The bottleneck in this paper is a piece of code whose long-time execution making users intolerable. Therefore, the first step to find bottlenecks is to use profiling tools to identify hot spots. This section briefs seven profiling tools commonly used in Linux or Android platform. Specifically, OProfile [5] Bootchart [16] and printk [17] are general profiling tools supporting both Linux and Android. The Debug class [18] and Log utilities [19] are profiling tools specific to Android. Linux Trace Toolkit Next Generation (LTTng) [7], Kernel Function Tracer (KFT) [8], vertical profiling tool [20] are another three profiling tools commonly used in Linux, yet they are not ported to Android.

OProfile

OProfile benefits from a kernel driver which utilizes the performance counters, presented in most modern CPUs, to record CPU-related events, such as cache misses, and the hardware timer periodically collecting function-executing information of either user-space programs or kernel. The profiling overhead using OProfile is low [21]. However, the resolution of call graph, i.e., relationships between function calls, depends on the granularly of hardware timer, and therefore may be inaccurate. Porting effort of the kernel driver is the major obstacle of using OProfile. Fortunately, OProfile has already ported on Android.

Bootchart

Bootchart also uses sampling techniques to profile the booting processes of Linux. During booting, a script is run by the user-space program, *init*, to periodically gather statistics on process level. Like OProfile, the overhead of Bootchart is also small, i.e., less

than 7% of CPU utilization [22]. However, the profiling results are too rough to find out the true bottlenecks in Android. The reasons are twofold. First, most of Android services and managers are threads, whereas Bootchart can only present the relation of processes. Second, Bootchart is executed after kernel triggering the first user-space program, init, so Bootchart cannot provide the complete profiling information of kernel booting.

Debug Class

The Android built-in Java Debug class, i.e., android.os.Debug, provides developers a way to create log and trace the execution of an Android application. The source code of applications must be instrumented with specific code. The Debug class does not satisfy our profiling objectives due to two disadvantages. First, the class cannot profile native libraries which are written in C and C++. Second, the class overwrites an existing profiling result for every execution of the instrumented source-code block. As a result, the Debug class cannot profile a service-type program which needs to be executed several times during a scenario.

Log Utilities

The Log utilities, including the Java Log class, i.e., android.util.Log, and C/C++ native library, i.e., liblog, are also built-in Android. The utilities can record log during the execution of user-space applications. Like using the Debug class, the source code of targeted programs has to been instrumented by specific code provided by the Log utilities. Comparing to the Debug class, the Log utilities have two advantages. First, the utilities can be used with not only Java but also C and C++ programs. Second, the utilities do not overwrite an existing output, so a program can be executed several times during profiling.

LTTng and *KFT*

LTTng and KFT are two profiling tools for tracing kernel performance. Both of them leverage the instrumentation techniques. LTTng provides a programming interface to

instrument the source code, while KFT uses a compiler-assisted capability, i.e., the -finstrument-functions flag in GNU compiler collection (gcc), to automatically instrument profiling routines to every entry and exit of kernel functions. When the execution reaches an instrumentation point, a log event is recorded. The overhead of LTTng is proportional to the number of instrumentation, whereas the overhead of KFT is extremely high, i.e., over 100% of overhead in our measurement, because KFT instruments all kernel functions.

printk

Printk is a log-recording function built-in in Linux kernel. Kernel developers can insert the functions anywhere in the kernel source to record log. The advantage of using printk is that the function does not have any kernel-version capability issue. On the other hand, the need of human effort to instrument is its major drawback.

Vertical profiling

Vertical profiling tool uses hardware performance monitors [23] and specific Java virtual machine, Jikes RVM, to observe events of the system on various software layers. Hardware performance monitors are supported on most modern processors by different application program interface (API). Vertical profiling tool can do time profiling on a multi-layer and multi-language platform with low system overhead. The main drawback of this tool is the hardware/software dependence issue, thus it is hard to port on different platform such as an embedded system with few performance counters in CPU.

Table 2 summarizes the pros and cons of the abovementioned profiling tools. Besides vertical profiling tool, no single tool can profile the complete scenario, cross multi-language multi-layer, on Android. But vertical profiling tool is not generic enough to port on different platform and hardware. Next chapter proposes approach to meet this requirement.

Table 2 Comparison of profiling tools

Profiling technique	Tool	Android support	Advantages	Drawbacks
Sampling	OProfile	Yes	● Low overhead	● Not support Java on Android
	Bootchart	Yes	● Low overhead ● Friendly GUI	● No kernel-space booting log ● No information for threads
Instrumentation	Debug class	Yes	● Call graph	● Not support C,C++ ● Trace log will be overwritten ● Human effort for instrumentation
	Log utilities	Yes	● Support Java/C/C++	● Human effort for instrumentation
	LTTng	No	● Low overhead	● Not support Java on Android
	KFT	No	● Provide complete call graph ● Kernel profiling	● High overhead ● Not support user-space profiling
	printk	Yes	● No kernel-version capability issue ● Kernel profiling	● Human effort for instrumentation ● Not support user-space profiling
Sampling/Instrumentation	Vertical profiling	No	● Low system overhead ● Support Java/C/C++ ● Support Kernel-space and user-space profiling	● Hardware dependence ● No embedded environment verification

Chapter 3 Staged Iterative Instrumentation Approach

Designing a time profiling approach in Android shall consider three unique properties: 1) multi-layers, 2) multi-programming-languages, and 3) limited log space.

Multi-layers multi-languages

The proposed approach takes multi-layer multi-language features into account by mixing using different profiling tools for different modules on an execution flow. A consequent issue is that a process must be developed to consolidate all profiling results from different profiling tools. As a result, a time-stamped tag is embedded in each log to help consolidation.

Limited log space

To conquer the limited log space, the proposed approach starts profiling a scenario from a single module, and restrainedly profiles more modules and layers to avoid enormous irrelevant profiling results. Specifically, the approach first selects the earliest executed module of a scenario, e.g., the web browser for the browsing scenario, and other modules suggested by domain experts. All functions *S* of the selected modules *M* are instrumented with checkpoints. Then the source codes of *M* are rebuilt and the scenario is executed to obtain profiling results. The profiling results may contain unnecessary checkpoint *S'*. Therefore, the checkpoints *S* shall be refined to reduce output results. Consequently, the source codes are rebuilt, and the scenario is executed again to get refined profiling results. The abovementioned steps describe one profiling stage. The huge amounts of logs are refined by one or more *iteration*, the iterative procedures from step 6 to step 10, in one stage. However, the selected modules *M* may not cover all required modules in a complete execution flow of a scenario. For such case, modules M^+ and checkpoints S^+ in adjacent layers are included to profile. New stage can then start. The

approach continues until no new module can be added to *M*. Figure 2(a) depicts the approach.

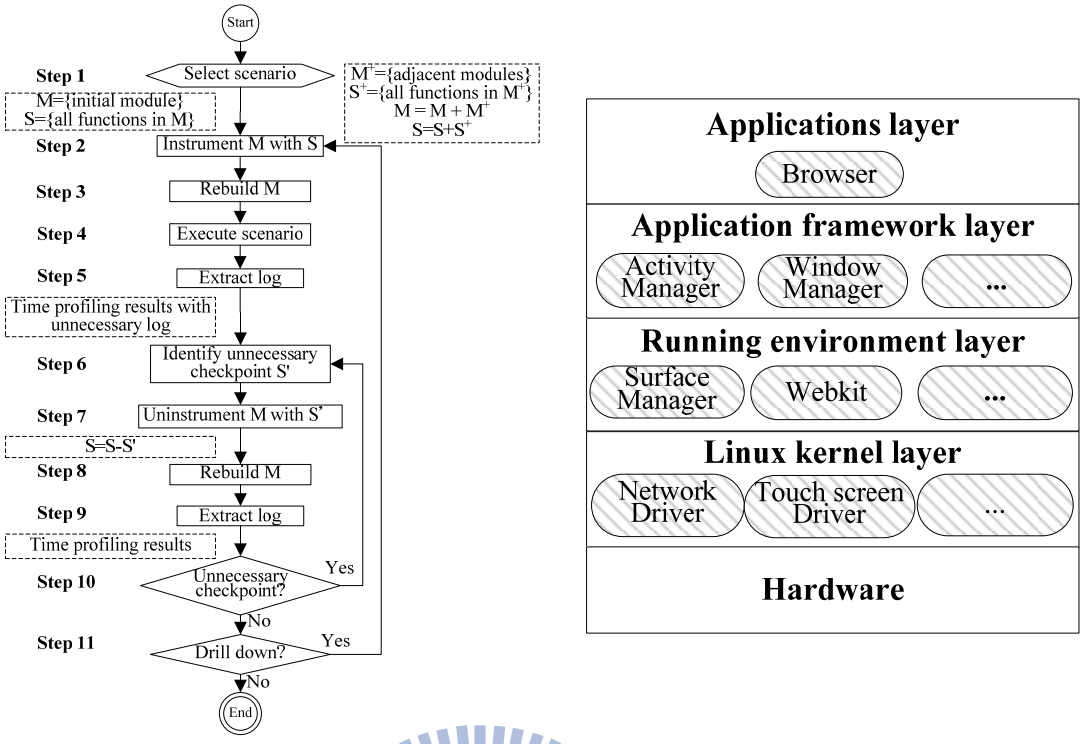
An example of approach: Browsing

The proposed approach is demonstrated by the example of profiling web browsing. For loading a web page, the web browser invokes many modules located in different layers. These modules includes, but not limited to belong to, the web browser on the application layer, the activity manager and window manager on the application framework layer, the surface manger (a module for screen drawing) and the Webkit (a web-browser engine) on the running environment layer, and the network driver and touch screen driver on the kernel layer. Figure 2(b) shows the modules invoked by web browsing.

In the first stage of our approach, *M* only contains the web browser, and *S* is all functions of the browser. The step 6 to step 10 in Figure 2(a) removes unnecessary functions from *S* iterative, so the checkpoints and corresponding profiling results are reduced after iteration. The step 11 determines that activity manager and window manager can be added, which causes an additional stage with M^{+}={activity manager, window manager} and *M*={browser, activity manager, window manager} to profile the scenario again. The stages end after all invoked components in the kernel layer are profiled.

Potential issue

This approach removes the unnecessary checkpoints depending on the questions we are interesting in manually. It may cause mis-uninstrument issue if the checkpoints have been removed before is an important points on later stage. Although this kind of issue doesn't appear in our experiment, we assume that it may be triggered on specify situation. If we suffer this kind of issue, we can roll back to Step 2 to instrument related modules M again anytime on this procedure.



(a) Procedure of staged iterative instrumentation profiling approach (b) Major Layers and Modules related to Android Browsing

Figure 2 Staged iterative instrumentation approach

Chapter 4 Implementation on Booting, Browsing, and Streaming

This chapter details the implementation of the staged instrumentation approach in Android. The generic implementation techniques, including the selection of profiling tools, log consolidation, and instrumentation automation, are introduced first. Then, the specific implementation techniques for three scenarios, i.e., booting, browsing, and streaming, are described in turn.

4.1 Generic Implementation

Selection of Profiling Tools

Though many profiling tools are available on Android, some tools among them are superior to others because they can profile multiple layers. The support of multi-layer profiling reduces the profiling complexity in twofold. First, the learning time on profiling tools is shortened and the code-modification complexity is decreased. Second, the formats of profiling results are simplified. This thesis selects the Log utilities for profiling user-space modules, because the utilities can profile all user-space modules no matter what programming languages they are implemented by and which layers they reside in. For kernel-profiling, this work picks the printk function, since the code-style of using printk is similar to the use of Log utilities.

Logs Consolidation

The profiling results of both Log utilities and the printk function have already embedded the time information. Unfortunately, the time resolutions, baselines, and formats of the two profiling tools are different. The time resolution of Log utilities is microsecond, its format is YY/mm/dd transformed from the output of `cpu_clock` function, and the value is bound for local CPU. On the other hand, the time resolution of printk is nanosecond, its format is a numerical number transformed from the output of `current_kernel_time` function, and the value is bound for kernel. To obtain more

precise results, we unify the time format of both profiling tools with the printk format.

Auto Instrumentation

Source-code instrumentation is a heavy and boring routine, so we implemented an instrumentation-automation script to insert the specific profiling code into specific files or directories, i.e., turning-on checkpoints, and remove the inserted profiling code, i.e., turning-off checkpoints. The script works for C, C++, and Java, and can instrument the source code with the format in either the Log utilities or printk. As a result, the procedures of a profiling iteration, i.e., adding or removing modules from the *M* and *S*, is easy to achieve by the turning on and turning off.

4.2 Booting

Booting Time

The booting time in this work is defined as the period starting from pushing the power bottom on the target Android platform and ending at the finish of drawing booting screen, i.e., the function, SurfaceFlinger::bootFinished(), in SurfaceFlinger.cpp. As default factory setting, Wi-Fi service and telecom service are disabled on our testing environment.

Booting Procedures

After the power button is pressed, Initial Program Loader (IPL) checks hardware and loads the Second Program Loader (SPL). SPL loads the compressed kernel image from flash to memory, decompresses kernel, and then executes the first kernel function, start_kernel. Kernel first initializes lots of data structures and tasks, and loads drivers. Next, the first user-space program, init, is executed. The init starts the service_manager (the core Android service taking responsibility for the registration of other Android services), daemon programs (like the Volume Management daemon, vold), zygote, (the parent process of all Java virtual machines), and media_server (a server managing

multimedia framework on Android). Zygote immediately preloads some Java classes and resources for the acceleration of Java applications, and then starts the system_server process. The system_server is a process managing all Android services. It scans the flash to find out all installed applications, and then creates threads for Android services. One of the services important for booting is the SurfaceFlinger, a screen-drawing service. The booting procedure ends at the finish of bootFinished() routine of the SurfaceFlinger service.

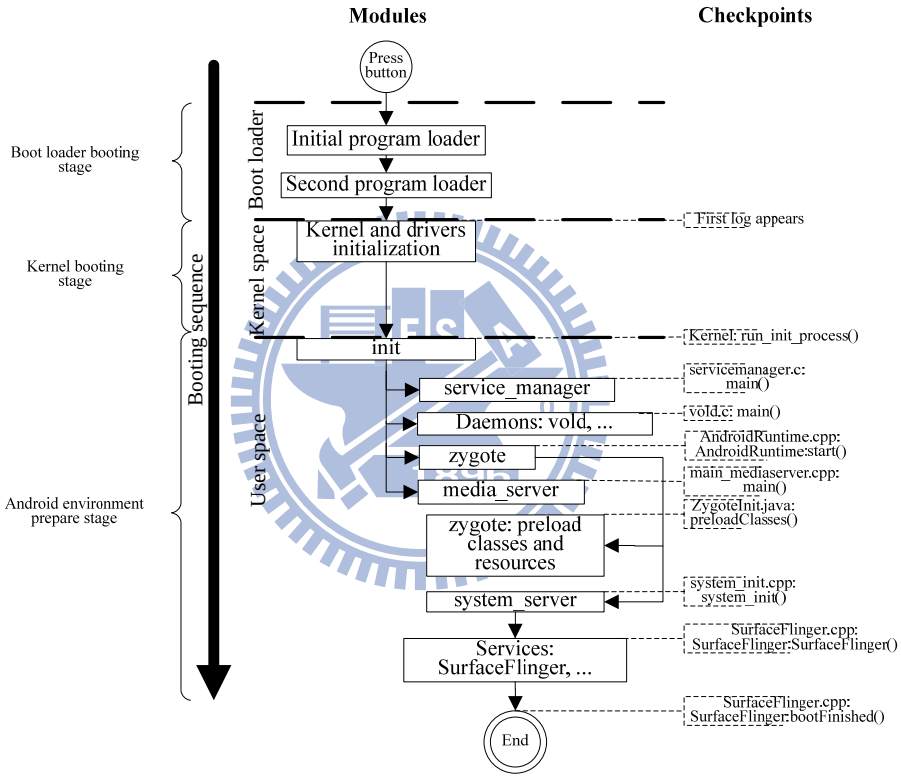


Figure 3 Booting procedures

Checkpoints

According to the booting procedures described above, we divided the booting sequence into three blocks: boot loader, kernel space, and user space. The first three checkpoints are set to the boundaries of the three blocks. Then, we used the staged instrumentation approach incrementally profiling the booting scenario, and identified the following seven important checkpoints: the first service (service_manger), the first

daemon (vold), the parent process of virtual machines (zygote), the multimedia server (media_server), the Java classes and resources preloading routine, the core of system services (system_server), and the first system service launched by system_server. Figure 3 draws the important modules for booting procedures, and the corresponding checkpoints.

4.3 Browsing

Browsing Time

The booting time in this work is defined as the period starting from touching the "go" button on screen in the web browser after a URL has been specified, and ending at completely loading the specified webpage, i.e., the function, onPageFinished(), in BrowserActivity.java, is called.

Browsing Procedures

Pressing the go bottom on screen first generates a system event. The event is handled by the touch screen driver in kernel, and passed to the activity manager (ActivityManager) and the web browser. The web browser can then know the specified URL and send the webpage request to Webkit, which is the built-in web browser engine in Android. Webkit uses HTTP protocol to request the desired webpage, and then parses the received webpage into a Document Object Model (DOM) tree. The detail of the DOM is beyond the scope of this work, so we omit the process. Usually, a webpage contains multiple element resources, e.g., image files, which shall be loaded together with the target webpage, so Webkit continues queuing and downloading the remaining element resources while drawing the layout on screen with using skia, a 2D drawing library.

Checkpoints

When applying the staged instrumentation approach on browsing scenario, the module set *M* in first iteration contains only the web browser. After multiple iterations, we identified the following important checkpoints: the interrupt request (IRQ) handler of

touch screen driver, the event handlers in activity manager and web browser, the Webkit-invoking function in web browser, the HTTP request and response functions in Webkit, the DOM-tree building and parsing functions in Webkit, and the screen layout functions in Webkit. Figure 4 illustrates the important modules for browsing procedures, and the corresponding checkpoints.

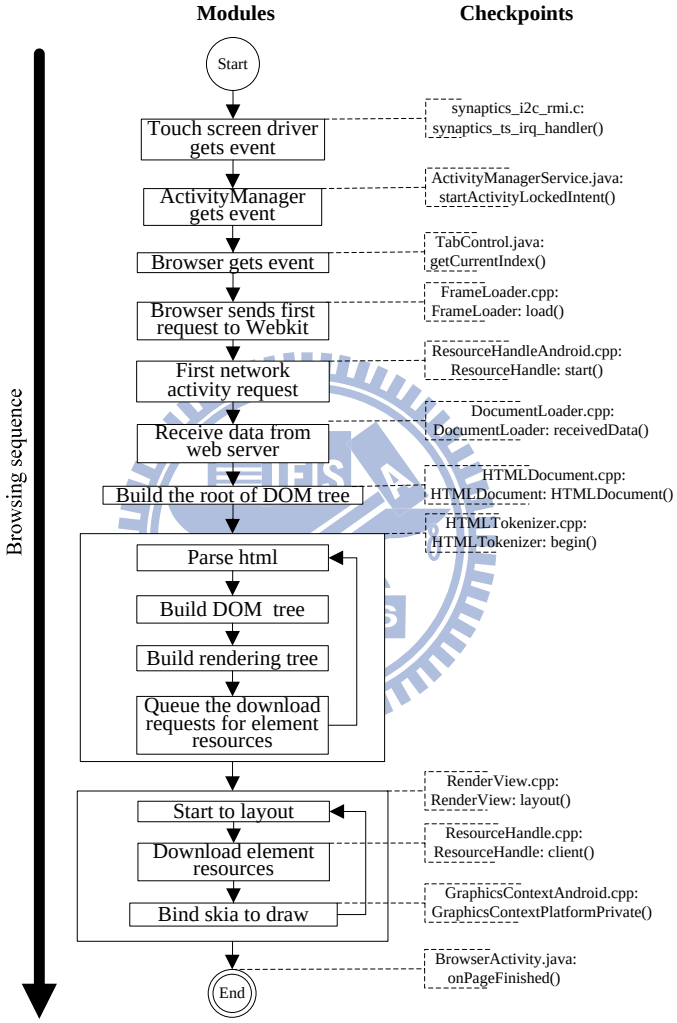


Figure 4 Browsing procedures

4.4 Streaming

Streaming Time

The built-in streaming player on Android is called YouTube. Before profiling the streaming scenario, we first connected to Internet and executed YouTube. YouTube

showed a list of video icons on screen. The booting time in this work is defined as the period starting from touching a video icon on screen, and ending at finishing the video play, i.e., the function, stayAwake(), in MediaPlayer.java, is called.

Streaming Procedures

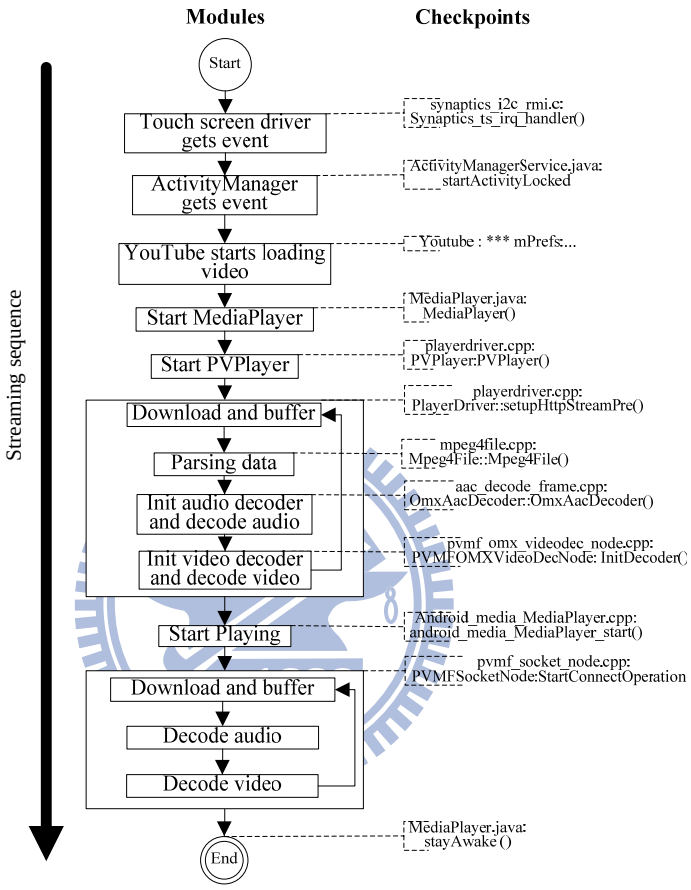


Figure 5 Streaming procedures

Like the browsing scenario, touching a video icon on screen also generates a system event, which is handled by the touch screen driver and passed to the activity manager and YouTube in turn. YouTube then sends the video request to MediaPlayer, which is a service taking responsibility for all kinds of media-playing requests. MediaPlayer dispatches the video-playing request to the PVPlayer, the built-in video player. Next, the PVPlayer continues downloading and decoding video from the Internet, and showing video on screen. At the end of play, MediaPlayer is notified by its stayAwake function.

Checkpoints

YouTube does not open its source, so we cannot apply the staged instrumentation approach on YouTube directly. Therefore, the approach starts from application framework layer instead. Thus, at the first iteration, *M* includes modules in application framework layer. Besides, YouTube, though released in binary form, still leaves some log data, which are taken as the checkpoints during our profiling. After multiple iterations, we identified the following important checkpoints: IRQ handler of touch screen driver, the event handler in activity manager, the log messages indicating that YouTube starts loading video, the constructors of MediaPlayer and PVPlayer, the video-downloading, audio-decoding, video-decoding functions in PVPlayer, the video-playing function in MediaPlayer, and finally, the end function, stayAwake, indicating the terminate of video play. Figure 5 illustrates the important modules for streaming procedures, and the corresponding checkpoints.

Chapter 5 Experimental Results and Observations

This chapter presents the profiling results obtained from an off-the-shelf Android device for the three user-perceptible scenarios, i.e., booting, browsing, and streaming. The testbed is first described. Then, important time-consuming functional blocks for each scenario are discussed in turn.

5.1 Testbed

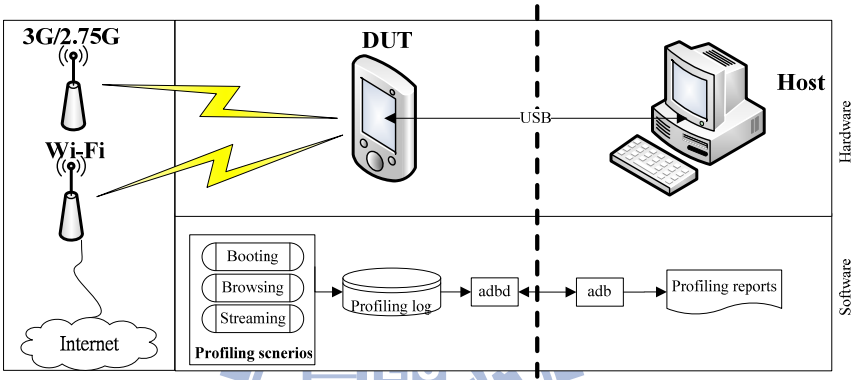


Figure 6 Testbed

The experiments in this work were conducted on an Android-based DUT (Device under test), Android Dev Phone 1. The DUT is an Android-based smartphone, whose root permission is left unlocked for developers to rewrite any program or the whole system on the platform. The hardware is 100% same as another off-the-shelf Android smartphone, HTC Dream, so we regard it as a commercial product. The version of Android platform is AOSP 1.6 (Android Open Source Project). Beside the DUT, a host machine is installed to provide us an operating interface to the DUT. Recall that the Android platform allows only one foreground GUI program to be executed at one time. Hence, the host machine provides an alternative, command-line, interface to on-line operate profiling tools, and to collect profiling results during the execution of a scenario. The operating interface on the host is a client program called adb (Android Debug Bridge), and the corresponding server

on the DUT is adbd. DUT and the host connect to each other thru the USB interface. Finally, the EDGE (2.75G), WCDMA (3G), and IEEE 802.11g (Wi-Fi) networks are accessible in the testbed environment. Figure 6 depicts the testbed.

5.2 Booting Scenario

Questions to Answer

We use the approach in this work to answer two questions on booting time. First, what is the accurate execution time of bootloader, kernel-space initialization, and user-space initialization? Second, which services or functions dominate booting time?

Profiling Results

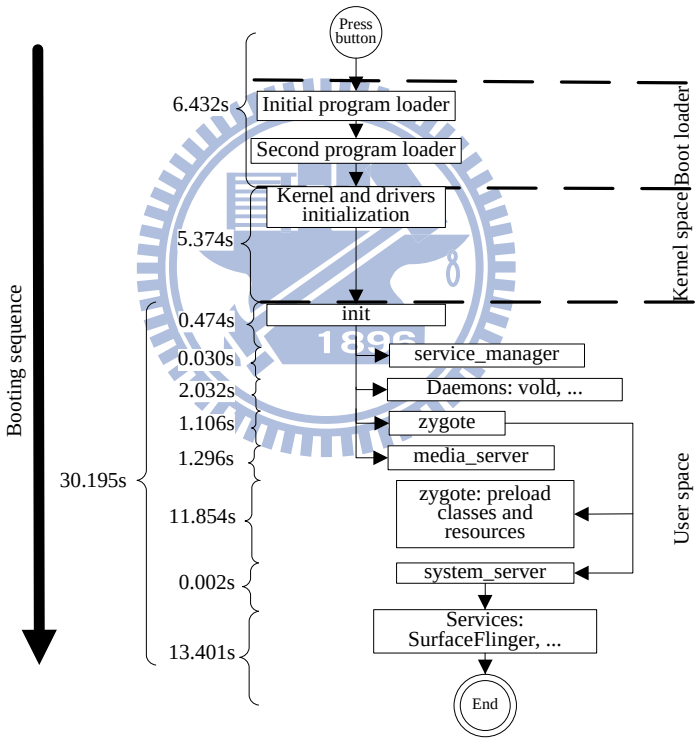


Figure 7 Profiling results for booting

Figure 7 shows the profiling results for the booting scenario with the checkpoints defined in chapter 4. As shown in Figure 8, the initialization of user-space environment takes 72% of total booting time. Drilling down the user-space initialization, the major time-consuming parts are Java classes and resources preloading by zygote (39.2% of user

space) and the startup of services and managers (44.4% of user space).

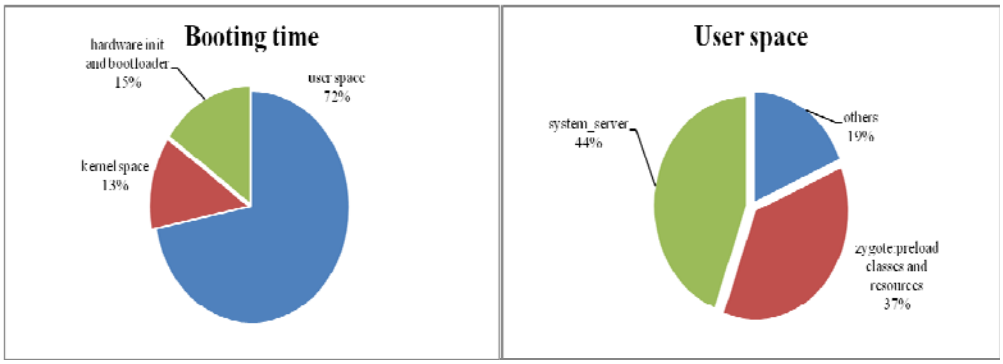


Figure 8 Distribution of booting time

Bottleneck Discussion

Preloading Java classes and resources takes lots of booting time. Apparently, reducing the number of preloaded classes and resources can efficiently decrease the booting time. However, the startup time for an application after booting is increased, because the Java virtual machine must spend time on loading those un-preloaded classes and resources. Therefore, tradeoff between booting time and startup time must be evaluated.

The startup time of services and managers also takes considerable booting time. In current Android design, the desktop screen is shown after all services and managers are ready. However, for many users, the later the screen shows, the slower the booting performance is. Thus, a booting-sequence redesign, which draws the desktop screen as early as possible and postpones launching the desktop-unrelated services, may be admired by users.

5.3 Browsing Scenario

Questions to Answer

Three questions are answered after profiling by our approach. First, what is the accurate execution time spent on each invoked module on Figure 4? Second, is there any hardware acceleration, i.e., Graphic Processing Unit accelerated (GPU-accelerated), on

rendering procedure during browsing? Third, as well know rendering and networking function is the major function on browsing. We are interesting in which of them dominates browsing time?

Different networking technologies

Figure 9 presents the time profiling results of loading a 2128-kB webpage which contains eight attached images. The experiments were conducted under different networking technologies, including Wi-Fi, 3G, and 2.75G.

The results showed that the browsing performance with 2.75G is slower than the performance under other networks. Browsing with Wi-Fi exhibited the fast performance. In addition, the results showed that 90% browsing time consumed by downloading element resources, no matter what kinds of networks the DUT connected to. The function of downloading element resources takes responsibility for using HTTP protocol to request the attached web resources, e.g., the images in our experiments, from web server. Apparently, the performance of the function depends on the networking technology in use. We conclude that the networking technology in use is the most significant factor influencing the browsing performance on Android.

The source code of GPU related drivers have been instrumented in this experimentation, but we didn't found any GPU related output during browsing procedure. Thus we could also verify GPU didn't be invoked during the rendering procedure of browser. If system invoke GPU to support browsing, it may takes addition energy consumption and addition offload time between CPU and GPU. The results showed that only 5% browsing time consumed by rendering function. Therefore, we can understand that there isn't sufficient niche to invoke GPU during browsing.

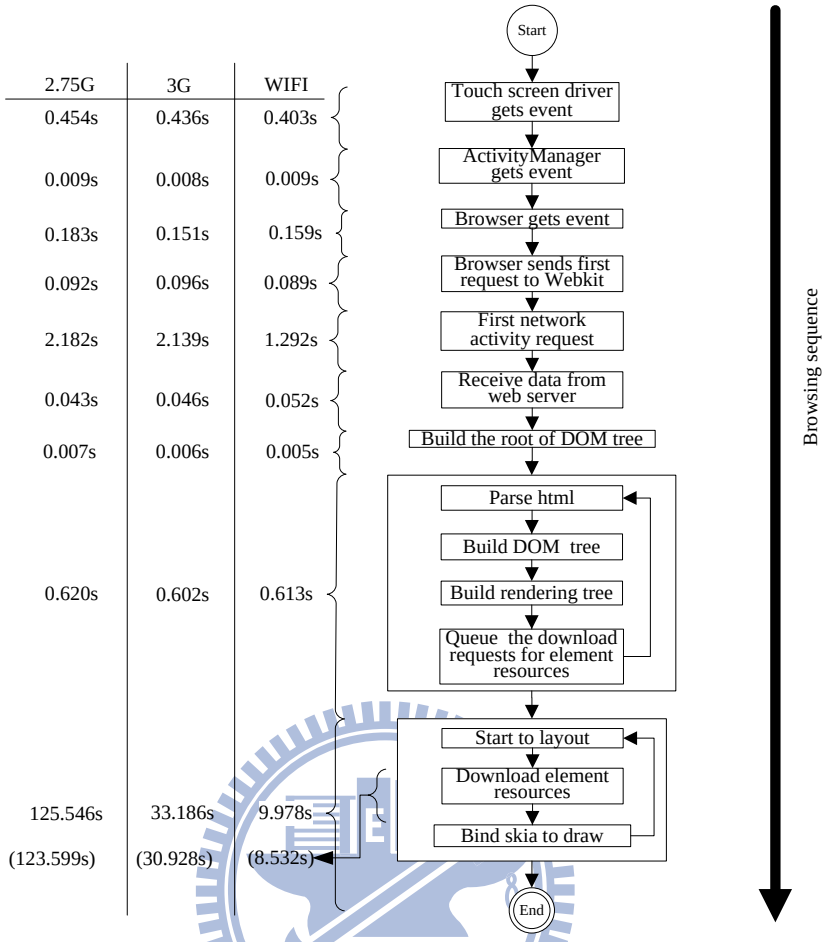


Figure 9 Time profiling results for browsing thru Wi-Fi, 3G, and 2.75G networks

Different size of single component in a webpage

Figure 10 presents the experimental results of the performance for different total webpage size under Wi-Fi networks. The webpage size is adjusted by the size of the attached web image. The “others” part denotes the sum of all execution time excluding the time for downloading element resources. We collapse most functions into the “others” part, because the total time of these functions are small and steady no matter what the webpage size is. Again, the function of downloading element resources dominates the execution time of browsing, which indicates that the network transmissions are the most important factor affecting the performance of web browsing.

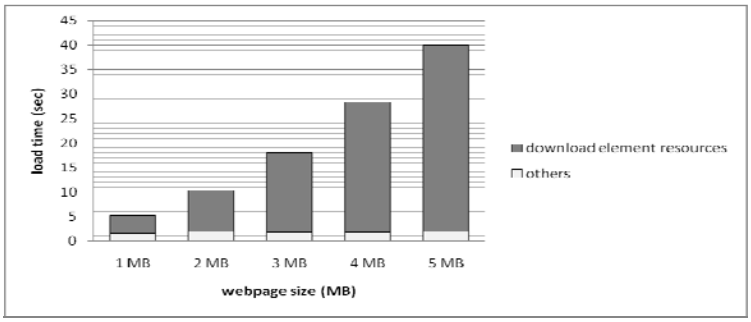


Figure 10 Profiling results of different webpage size

5.4 Streaming Scenario

Questions to Answer

We interest in three questions in streaming. First, what is the accurate execution time spent on each invoked module on Figure 5? Second, what is the execution time from the selected video icon be touched to the video be played (preparation time)? Third, which functions dominate preparation time?

Profiling Results

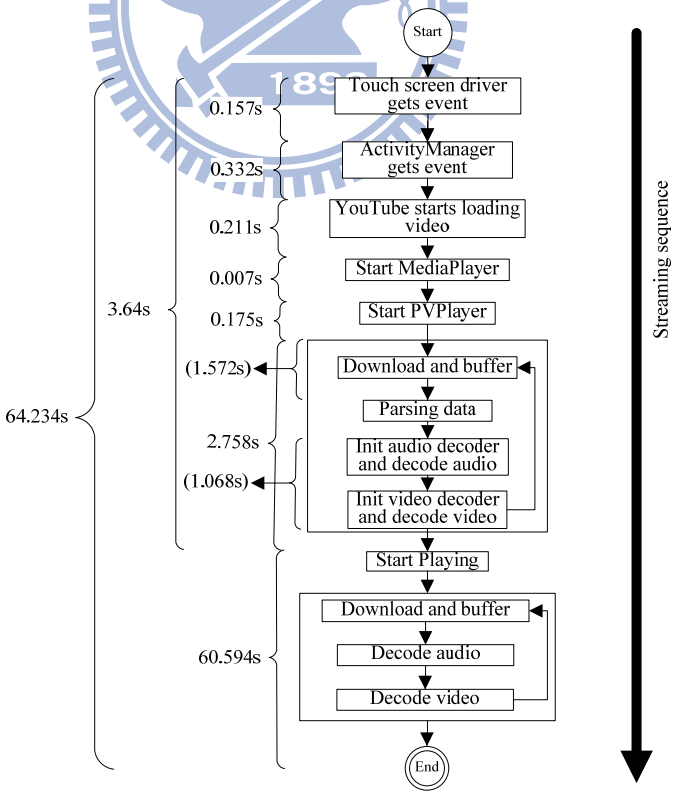


Figure 11 Profiling results for streaming

Figure 11 shows the time profiling results of playing a one-minute MPEG-4 video from YouTube site. The file size of the video is 22 MB, and the DUT accessed the Internet thru the Wi-Fi interface. No jitter happened during experiments. Therefore, the streaming performance sensed by user in this experiment is the time spent on video preparation, i.e., 3.64 seconds. Obviously, two bottlenecks residing in video preparation are the video-downloading (1.572 seconds) and data-decoding (1.068 seconds). The data-decoding include audio decoding (0,515 seconds) and video decoding (0.553 seconds).

Chapter 6 Conclusions and Future Works

In this thesis, we designed a staged instrumentation approach for time profiling on multi-language multi-layer platform. The approach was practiced on the off-the-shelf product to examine the execution flows of three common scenarios, booting, browsing, and streaming.

As well know, user-space initialization time dominate the booting time, but no literals present the complete numeric distribution of total Android booting time. This work showed that 72% of Android booting time is spent on the initialization of user-space environment, 13% is on kernel initialization, and 15% is for boot loader. Drilling down the booting time in user space, 44.4% is used by starting Android services and managers, and 39.2% is used by preloading Java classes and resources. For the browsing scenario, the experimental results exposed that the networking technology in use is the most significant factor influencing the browsing performance on Android. The time of drawing screen only takes less 5% of total time for browsing a 2128 kB web page. This result can explain why the GPU-acceleration is unnecessary for rendering function of browser. In the streaming scenario, video preparation causes 5.7% time overhead for playing a 22-MB video file over Wi-Fi connection. The execution time of Video-downloading and data-decoding take 72% ratio of preparation time.

Though the proposed profiling approach can profile multi-language multi-layer platform, the profiling results still need manual analysis. In the future, we will design tools for analysis automation. Besides, this approach can integrated with energy profiling to measure the energy consumption of functions on system runtime. More scenarios, like the GPS navigation, and platforms, such as Android on set-top boxes, will also be profiled. The GPU didn’t supported on current version of Android browser for rendering function,

but it may supported on new version ones. We will also profile the browsing time with GPU and browsing time without GPU in future work.

Reference

[1] J. Aguero, M. Rebollo, C. Carrascosa, and V. Julian, "Towards on embedded agent model for Android mobiles," in *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services* pp. 1-4, 2008.

[2] comScore, "80 percent of iPhone Users in France, Germany and the UK Browse the Mobile Web " [Online]. Available: http://www.comscore.com/Press_Events/Press_Releases/2008/07/iPhone_Users_in_Europe_Browse_the_Web.

[3] C. Ghoroghi and T. Alinaghi, "An introduction to profiling mechanisms and Linux profilers." [Online]. Available: http://www.docstoc.com/docs/7671023/An-introduction-to-profiling-mechanisms-and-Linux-profilers.

[4] S. Shende, "Profiling and Tracing in Linux," in *Proc. Second Extreme Linux Workshop #2, USENIX Annual Technical Conference*, pp. 26-30, 1999.

[5] "Oprofile: A System Profiler for Linux." [Online]. Available: http://oprofile.sourceforge.net/.

[6] S. Graham, P. Kessler, and M. Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, p. 126, 1982.

[7] M. Desnoyers and M. Dagenais, "LTTng: Tracing across execution layers, from the Hypervisor to user-space," in *Proceedings of the Ottawa Linux Symposium*, pp. 101-105, 2008.

[8] "Kernel Function Trace." [Online]. Available: http://elinux.org/Kernel_Function_Trace.

[9] J. Galenson, C. Jones, and J. Lo, "Toward a Browser OS Designing a next-gen mobile OS with web technologies," Department of Cumputer Science, University of California, Berkeley 2008.

[10] H. Jo, H. Kim, H. Roh, J. Lee, and S. Maeng, "Improving the Startup Time of Digital TV," *IEEE Transactions on Consumer Electronics*, vol. 55, p. 722, 2009.

[11] C.-M. Huang and Y. Huang, "A Timing Analysis of Booting Procedures on Embedded Linux Systems," Electrical and Control Engineering, National Chiao Tung University, 2009.

[12] C. Park, K. Kim, Y. Jang, and K. Hyun, "Linux Bootup Time Reduction for Digital Still Camera," in *Proceedings of the Ottawa Linux Symposium vol.2*, pp. 231-240, 2006.

[13] C. Yang and Y. Huang, "An Empirical Analysis of Embedded Linux Kernel 2.6.14

to Achieve Faster Boot Time," Master, Electrical and Control Engineering, National Chiao Tung University, 2006.

[14] R. Gau, "Implementation of Lightweight Graphic User Interface Engine for Portable Multimedia Player on Arm-based Embedded System," Master, CSIE, National Central University, 2009.

[15] "YAFFS2 (Yet Another Flash File System)." [Online]. Available: http://www.yaffs.net/yaffs-2-specification-and-development-notes.

[16] Z. Mahkovec, "Bootchart." [Online]. Available: http://www.bootchart.org/.

[17] "Printk Times." [Online]. Available: http://elinux.org/Printk_Times.

[18] "Android Debug." [Online]. Available: http://developer.android.com/reference/android/os/Debug.html.

[19] "Android Log." [Online]. Available: http://developer.android.com/reference/android/util/Log.html.

[20] M. Hauswirth, P. Sweeney, A. Diwan, and M. Hind, "Vertical profiling: understanding the behavior of object-priented applications," *ACM Sigplan Notices*, vol. 39, pp. 251-269, 2004.

[21] W. Cohen, "Tuning programs with OProfile," *Wide Open Magazine*, vol. 1, pp. 53-62, 2004.

[22] "Using Bootchart on Android." [Online]. Available: http://elinux.org/Using_Bootchart_on_Android.

[23] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind, "Using hardware performance monitors to understand the behavior of java applications," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - vol. 3*, pp. 5-5, 2004.