Journal of Network and Computer Applications ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems

Ying-Dar Lin [a], Cheng-Yuan Ho [b], Yuan-Cheng Lai [c,*], Tzu-Hsiung Du [a], Shun-Lee Chang [a]

[a] Department of Computer Science, National Chiao Tung University, Hsinchu County 300, Taiwan
[b] Information and Communications Technology Lab, Microelectronics and Information Systems Research Center, National Chiao Tung University, Hsinchu County 300, Taiwan
[c] Department of Information Management, National Taiwan University of Science and Technology, Taipei County 106, Taiwan

ARTICLE INFO

Article history:
Received 17 July 2012
Received in revised form
27 January 2013
Accepted 17 February 2013

Keywords:
Android
Booting
Browsing
Streaming
Time profiling

ABSTRACT

Android-based systems perform slowly in three scenarios: booting, browsing, and streaming. Time profiling on Android devices involves three unique constraints: (1) the execution flow of a scenario invokes multiple software layers, (2) these software layers are implemented in different programming languages, and (3) log space is limited. To compensate for the first and second constraints, we assumed a staged approach using different profiling tools applied to different layers and programming languages. As for the last constraint and to avoid generating enormous quantities of irrelevant log data, we began profiling scenarios from an individual module, and then iteratively profiled an increased number of modules and layers, and finally consolidated the logs from different layers to identify bottlenecks. Because of this iteration, we called this approach a staged iterative instrumentation approach. To analyze the time required to boot the devices, we conducted experiments using off-the-shelf Android products. We determined that 72% of the booting time was spent initializing the user-space environment, with 44.4% and 39.2% required to start Android services and managers, and preload Java classes and resources, respectively. Results from analyzing browsing performance indicate that networking is the most significant factor, accounting for at least 90% of the delay in browsing. With regard to online streaming, networking and decoding technologies are two most important factors occupying 77% of the time required to prepare a 22 MB video file over a Wi-Fi connection. Furthermore, the overhead of this approach is low. For example, the overhead of CPU loading is about 5% in the browsing scenario. We believe that this proposed approach to time profiling represents a major step in the optimization and future development of Android-based devices.

1. Introduction

Devices such as Smartphones, set-top boxes, and netbooks provide users with the ability to access the Internet at anytime, from anywhere. Among these Internet connectable devices, Smartphones operating under Android, an open source platform developed from Linux in 2009, are expected to garner the most attention in coming years. Designing a device on the Android operating system reduces licensing fees, and developers benefit from the ability to develop new features and test their innovations in an open source environment (Pieterse and Olivier, 2012). However, Android-based devices suffer from poor performance in three areas: booting, browsing, and streaming. Boot-up time is the first perception users have when trying a new Smartphone, and browsing and streaming are two usage scenarios commonly encountered by Smartphone subscribers (Comscore.com, 2008). We compared the time spent booting and browsing using popular off-the-shelf products. Although all of these products have similar hardware capabilities, the execution on Android-based products takes much longer than similar applications on iPhones. Previous researchers have worked intensively on improving the performance in these three areas (Singh et al., 2011; Zhao et al., 2011; Trestian et al., 2012), and all of these studies have shared three common procedures. First, profiling tools were used to trace the flow of execution and the running time of targeted tasks. Second, the flow of execution was redesigned to reduce the time required to perform the three tasks. Finally, the improvement in performance was evaluated by profiling the system again. Clearly, profiling tools play an important role in the enhancement of performance. Profiling tools can be categorized into two types: instrumentation and sampling techniques (Ghoroghi and Alinaghi, http://www.docstoc.com/docs/7671023/An-introduction-to-profiling-mechanisms-and-Linux-profilers; Patel and Rajawat, 2011). Instrumentation techniques, such as debug classes (Yoon, 2012;

* Corresponding author. Tel.: +886 2 27376794; fax: +886 2 27376777.
E-mail address: laiyc@cs.ntust.edu.tw (Y.-C. Lai).

1084-8045/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jnca.2013.02.024

Android Developer, http://developer.android.com/reference/packages.html), log utilities (Yoon, 2012; Android Developer, http://developer.android.com/reference/packages.html), printk (Printk Times, http://elinux.org/Printk_Times), Linux Trace Toolkit Next Generation (LTTng) (Toupin, 2011), and Kernel Function Trace (KFT) (http://elinux.org/Kernel_Function_Trace) insert profiling code into the source code of targeted programs, thereby enabling the profiling results to be collected during execution. On the other hand, sampling techniques, such as OProfile (Levon, http://oprofile.sourceforge.net/) and Bootchart (Mahkovec, http://www.bootchart.org/), collect the process-usage statistics by periodically checking which program or process is occupying the CPU.

Two difficulties have consistently plagued previous studies. One problem is the fact that Android is a complex system with multiple layers, and the characteristics of each profiling tool limit it to a specific layer or layers within the software, as explained in Section 2. The other problem is that previous researchers have validated their ideas on development boards or emulators, despite the fact that hardware is meant to be optimized for commercial products. This has resulted in discrepancies between the performance results obtained in the lab and those provided by off-the-shelf products. Unlike a development board equipped with sufficient log space, hardware optimization may leave only limited space on the end-product, e.g., a 64 KB log buffer on the HTC Dream Smartphone. As a result, profiling tools work very effectively on development boards but often encounter out-of-resource problems on the devices for which they were intended.

This work proposes a novel approach to profiling across multiple layers to identify true bottlenecks in booting, browsing, and streaming using real-world Android based devices. We revealed common profiling procedures used for arbitrary scenarios and developed specific profiling procedures for each scenario. All procedures were validated on off-the-shelf products, to identify the true bottlenecks of each scenario. This work has the following major contributions: (1) we propose a staged iterative instrumentation approach, which has properties of limited log space, multi-layers, and multi-programming-languages; (2) we solve the implementation issues of this approach for time profiling booting, browsing, and streaming using real-world Android devices; and (3) we conduct extensive evaluations in booting, browsing, and streaming scenarios and identify the bottlenecks in these scenarios.

The remainder of this paper is organized as follows. In Section 2, we briefly describe the Android architecture and various profiling tools. In Sections 3 and 4, we present our proposed methodologies and the means by which the profiling procedures are implemented. In Section 5, we present the experimental environment and discuss the profiling results. Finally, in Section 6, we off conclusions and suggest directions for future research.

2. Background

This section briefly describes the Android architecture and various profiling tools.

2.1. Android architecture

As shown in Fig. 1, Android software comprises four major layers, written in three different programming languages: Java, C++, and C. From basic hardware compliance to the level controlled by users, the four software layers are the Linux kernel, running environment, application framework, and applications.

1. Linux kernel layer
   The Android kernel was derived from the Linux 2.6 kernel, so it inherits many advantages of Linux such as numerous device drivers and core operating system functionalities, e.g., memory management, process management, and networking. In order to accommodate the tightly constrained resources associated with embedded devices, Android adds new modules into its kernel or modifies some parts of the Linux kernel. For example, Android includes Yet Another Flash File System, 2nd edition (Yaffs2), an optimized file system for NAND flash, but the Linux kernel did not. Accordingly, Android can run on many different types of devices. Moreover, most of the profiling tools used in common Linux distributions can be adopted for Android since the entire Linux kernel is written in C programming language.
2. Running environment layer
   The 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, providing common routines for upper layers. In contrast, Android runtime is designed specifically for Android to meet the needs of operating in a resource-limited embedded device. It includes Dalvik virtual machine (VM) and core libraries. Dalvik VM is derived from Java VM and written in C, C++, and Java while the core libraries are written in Java containing common Java classes for the development of applications.
3. Application framework layer
   The application framework layer contains reusable components, accessible to applications. Components in this layer are written in Java, C++, and C programming languages. Among the components in this layer, activity and window managers are the two most important. The former manages the life cycle of applications, while the latter draws graphic elements, such as status bars, to provide a foreground graphical-user-interface (GUI). Furthermore, in Android, only one foreground GUI application may be displayed at a given time, while other running applications are managed by the activity manger in the background.
4. Applications layer
   All user-visible applications on Android can be developed by anyone, like commercial developers, open-source communities, and Google, and are written in Java programming language.

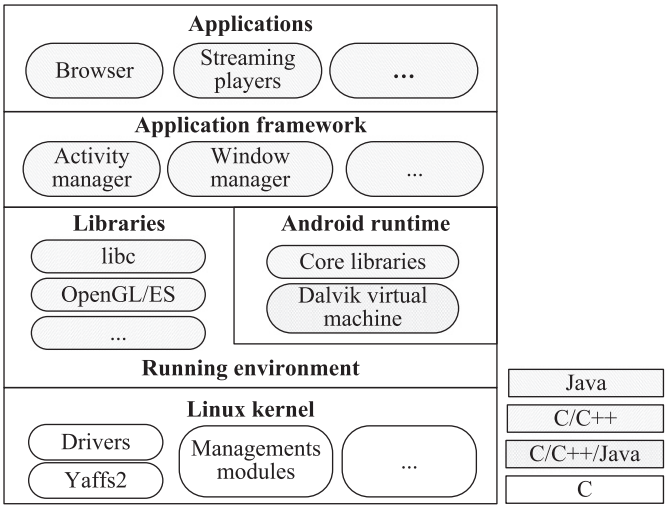

Fig. 1. Android architecture.

2.2. Profiling tools on android

Profiling tools are used to detect hotspots in a program or a set of programs to alleviate performance issues. A hotspot is a piece of code that is frequently executed or the execution time of which

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

is extremely long. Furthermore, in this article, a piece of code with an extended execution time, which generally makes users be intolerable, is defined as a bottleneck. This subsection reviews nine profiling tools commonly used on a Linux or an Android platform. Among these nine profiling tools, printk (Printk Times, http://elinux.org/Printk_Times), OProfile (Levon, http://oprofile.sourceforge.net/), and Bootchart (Mahkovec, http://www.bootchart.org/) are applicable in both Linux and Android. Debug classes (Yoon, 2012; Android Developer, http://developer.android.com/reference/packages.html), log utilities (Yoon, 2012; Android Developer, http://developer.android.com/reference/packages.html), Wlayer (Lee and Lim, 2011), and Profiledroid (Wei et al., 2012) are four profiling tools specific to Android while LTTng (Toupin, 2011) and KFT (Kernel Function Trace, http://elinux.org/Kernel_Function_Trace) are commonly used in Linux, not ported to Android yet.

1. Debug class
   Java Debug class built into Android, such as android.os.Debug, provides developers with the means to create logs and trace the execution of Android applications. However, the source code of applications must be implemented with specific code when Debug class is used. In addition, because of the two disadvantages, Debug class does not satisfy the profiling objectives of booting, browsing, or streaming in this work. First, it is unable to profile native libraries written in C and C++. Second, it overwrites existing profiling results for every execution of the instrumented source-code block. As a result, Debug class is unsuited to a service-type program, which needs to be executed several times during the three scenarios mentioned.
2. Log utilities
   Log utilities, including Java Log class (android.util.Log) and C/C++ native library (liblog), are also built into Android. These utilities are capable of recording logs during the execution of user-space applications. Log utilities have two advantages, compared to Debug class. One advantage is that the utility is not limited to Java, but can also be used with C and C++ programs. The other advantage is that these utilities do not overwrite an existing output, making it possible to execute a program several times during a single profiling session. However, Log utilities require human input for instrumentation and the source code of targeted programs has to be instrumented by specific code provided by Log utilities.
3. printk
   printk is a log-recording function built into the Linux kernel. Kernel developers are able to insert this function anywhere in the kernel source to record logs. The advantage of using Printk is the fact that the functionality does not have any kernel-version compatibility issues; however the need for human interaction is a major drawback.
4. LTTng and KFT
   LTTng and KFT are two profiling tools for tracing Linux kernel performance. Both LTTng and KFT belong to instrumentation profiling techniques. In detail, 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, an event will occur and the related information of that event is recorded into a log. The overhead of LTTng is proportional to the number of instrumentation whereas the overhead of KFT is extremely high, over 100% of overhead in our measurement. This is because KFT needs to instrument all kernel functions.
5. OProfile
   OProfile benefits from a kernel driver and a hardware timer. The former utilizes the performance counters found in most modern CPUs to record CPU-related events, such as cache misses while the latter periodically collects the function-executing information of either user-space programs or the kernel. Although the profiling overhead of using OProfile is low, the resolution of call graphs, i.e., the relationships between function calls, depends on the granularity of the hardware timer and may therefore be inaccurate. Furthermore,

Table 1
Comparison of profiling tools.

Profiling technique	Tool	Android support	Advantages	Drawbacks
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
	printk	Yes	• No kernel-version capability issue • Kernel profiling	• Human effort for instrumentation • Not support user-space profiling
	LTTng	No	• Low overhead	• Not support Java on Android
	KFT	No	• Provide complete call graph • Kernel profiling	• High overhead • Not support user-space profiling
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
	Wlayer	Yes	• Provide whole layer analysis • Support Kernel-space and user-space profiling	• High overhead • No information for threads
	Profiledroid	Yes	• Multi-layer profiling	• High overhead • No information about consumed time

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

the porting effort of the kernel driver is a major obstacle in using OProfile.

6. Bootchart
   Bootchart is a sampling technique and it profiles the processes involved in booting into Linux. While booting, a script is run by the user-space program, init, to periodically gather statistics at the process level. The overhead of Bootchart is low; however, the profiling results are too rough to identify true bottlenecks in Android. The reasons for this are two-fold. First, most Android services and managers are threads, whereas Bootchart only presents processes. Second, Bootchart is executed after the kernel triggers the first user-space program, init, so it is unable to provide complete profiling information of kernel booting.
7. Wlayer
   WPlayer provides a whole layer performance analysis tool for Android platforms. It combines and integrates available open source performance analysis tools. It is composed of method profiler, kernel profiler, and profiler controller. Profiler controller actually enables logging systems in both application and kernel layers. Method profiler performs method profiling at Android framework level while kernel profiler performs kernel event logging. Since additional costs to perform kernel event profiling are significant in Wlayer, it suffers from high overhead in profiling.
8. Profiledroid
   Profiledroid is a comprehensive, multi-layer system for monitoring and profiling applications on Android. It profiles applications at four layers: (a) static, or application specification, (b) user interaction, (c) operating system, and (d) network, with different tools at different layers. Profiledroid can do profiling on a multi-layer platform and can be used to find inconsistencies between application profile and behavior patterns in a systematic way. However, its main drawback is that it does not provide any profiling results about consumed time. Also its overhead is high because it adopts different profiling tools at different layers and analyzes the generated results in different layers to find their inconsistencies.

Table 1 summarizes the advantages and drawbacks of the abovementioned nine profiling tools. As previously discussed, no single profiling tool is capable of profiling the entire processes that requires attention, across multiple languages, and multiple layers on Android. Accordingly, in the next section, we propose a staged iterative approach to instrumentation profiling.

3. Staged iterative instrumentation approach

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

3.1. Limited log space

To overcome the difficulties imposed by the limited log space and avoid generating enormous quantities of irrelevant data during profiling, the proposed approach begins profiling a scenario from a single module, moving on to profile more modules and layers in a constrained manner. Specifically, the earliest executed module of a scenario is first selected. The web browser for the browsing scenario, or any other modules suggested by domain experts are examples. Figure 2(a) depicts the procedures, 11 steps of the proposed approach, staged iterative instrumentation profiling. Step 1 is to select a scenario to profile, and then, in step 2, all functions S of the selected modules M are equipped with checkpoints. The source code of M is rebuilt, the scenario is executed, and the profiling results are obtained respectively in

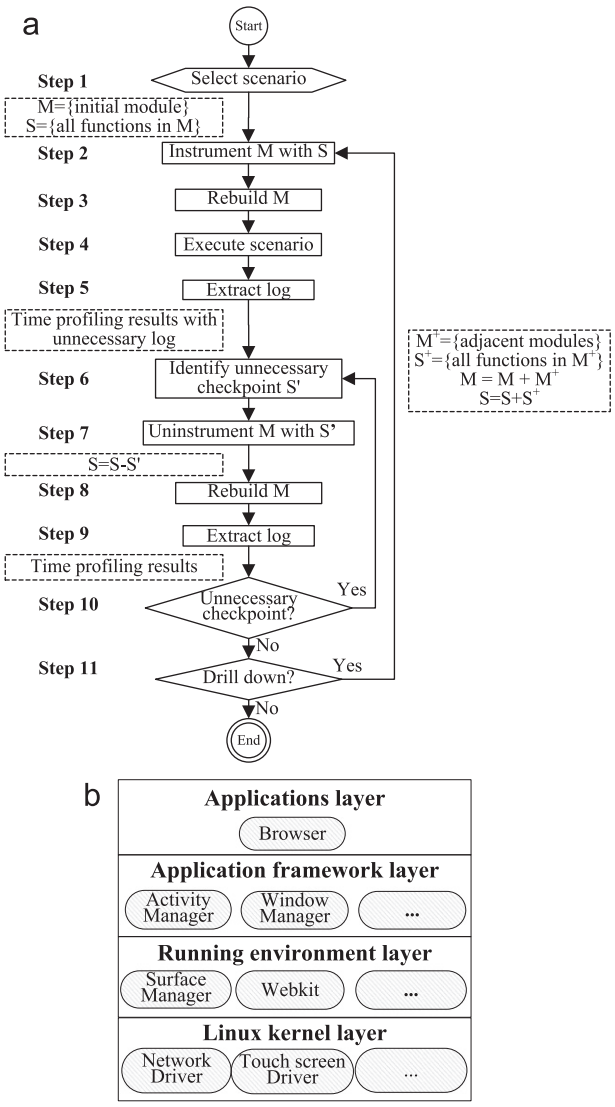

Fig. 2. Staged iterative instrumentation approach. (a) Procedure of staged iterative instrumentation profiling. (b) Major layers and modules related to Android browsing.

steps 3, 4, and 5. After step 5, checkpoints S are refined in the following manner as to reduce unnecessary checkpoint S'. In this manner, the source codes are rebuilt and the scenario is executed again to obtain refined profiling results. The enormous logs are refined through one or more iterations, the iterative procedures from step 6 to step 10, in a single stage. However, the selected modules M do not necessarily cover all required modules in a complete execution flow of a scenario. In such cases, modules M^+ and checkpoints S^+, in adjacent layers, are included in the profile. Therefore, a new stage will then begin. This process continues until no new modules can be added to M.

3.2. Multi-layers multi-languages

The proposed approach takes into account the problems associated with multiple layers and languages by combining various profiling tools for each of the modules in an execution flow. As Fig. 2(a) shows, the selected module M with necessary checkpoints is executed and its corresponding profiling results are obtained with a loop from step 2 to step 11. When a module M^+ in the same layer adopts the other language, the corresponding instrumentation of this language will be inserted in a new loop. Thus our time-profiling tool has the property of multi-programming-languages. On the

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

other hand, when a module M^+ is in the lower layer, the corresponding instrumentation for this module is also inserted in a new loop for further observation. Thus our time-profiling tool has the property of multi-layers. One resulting issue was that a process had to be developed in which a proper time-stamped tag, which is specific to adopted language, should be embedded in each log to facilitate consolidation of all profiling results from each of the profiling tools.

3.3. Example using this approach: browsing

We demonstrated the proposed approach by presenting an example of profiling the processes involved in web browsing. Web browsers invoke many modules located in various layers to open a web page. These modules includes, but are not limited to, the web browser in the application layer, the activity and window managers in the application framework layer, the surface manger (a module for screen drawing), the Webkit (a web-browser engine) in the running environment layer, and the network driver and touch screen driver in the kernel layer. Figure 2(b) shows the modules invoked by web browsing.

In the beginning, M contains only the web browser and S is all of the functions of the browser. Then, the proposed approach removes unnecessary functions from S iteratively as shown in steps 6–10 in Fig. 2(a), to reduce the number of checkpoints and corresponding profiling results. Step 11 determines whether the activity manager or window manager can be added, resulting in 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 have been profiled.

3.4. Potential issue

The proposed approach removes unnecessary checkpoints depending on the questions which developers are interesting in manually. However, if these removed checkpoints are important to the later stages, the profiling results will be bias. This issue is called mis-uninstrument problem. The solution of this issue is that the approach rolls back to step 2 and re-instruments related modules M.

4. Implementations on three scenarios

This section details the implementation of the staged instrumentation approach on Android. First, we introduced generic implementation techniques, including the selection of profiling tools, log consolidation, and the automation of instrumentation. We then describe specific implementation techniques for each of the three scenarios, booting, browsing, and streaming.

4.1. Generic implementation

1. Selection of profiling tools
   A number of profiling tools on Android are superior to others because of their ability to profile multiple layers. Multi-layer support reduces complexity two-fold. First, it shortens the time required for users to learn how to use profiling tools and decreases the complexity involved in modifying code. Second, it simplifies the form of the results of profiling. Accordingly, log utilities and printk are selected to profile user-space modules and kernels, respectively. This is because the former is capable of profiling all user-space modules and the code-style of printk is similar to that of log utilities.
2. Logs consolidation
   Both log utilities and printk include time information in their profiling results, but unfortunately, time resolution, baselines, and formats of log utilities' and printk's time information are different. The time resolution of log utilities is in microseconds, and its format is yy/mm/dd transformed from the output of the cpu_clock function, and its value is bound to the local CPU. On the other

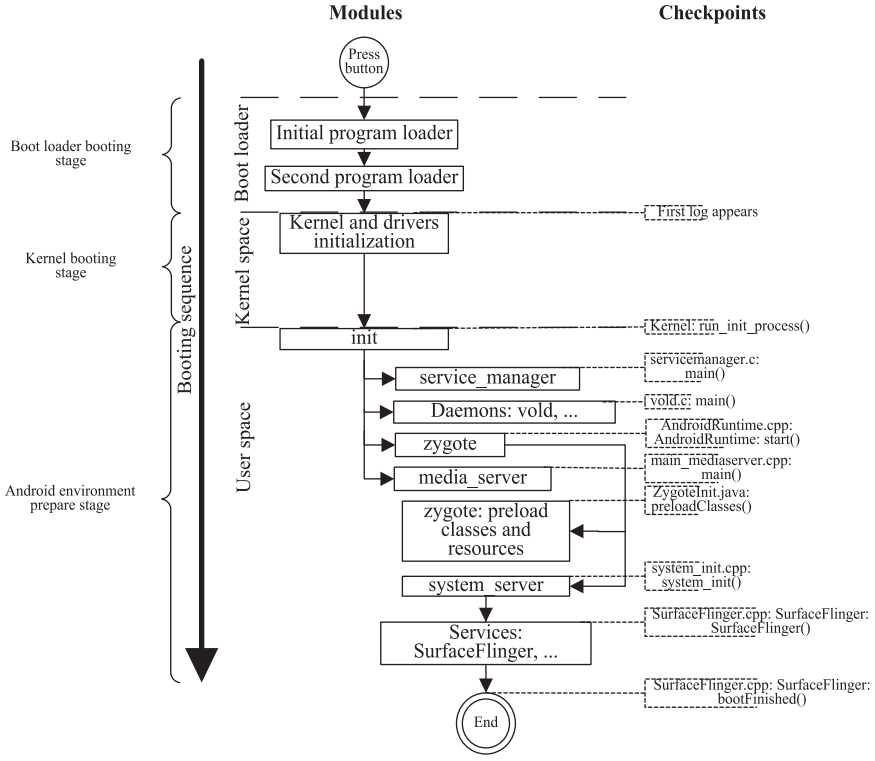

Fig. 3. Booting procedures.

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

hand, printk's time resolution is in nanoseconds, its format is a numerical number transformed from the output of the current_kernel_time function, and its value is bound to the kernel. To obtain more precise results, we had to unify the time formats with the format of printk.

3. Auto instrumentation
Because the instrumentation of source-code was an exhaustive routine, an instrumentation-automation script is used to insert 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 used for C, C++, and Java was able to instrument the source code with the format in either log utilities or printk. As a result, it was easy to achieve the procedures of a profiling iteration such as adding or removing modules from M and S.

4.2. Booting

1. Boot time
We defined boot time as the period from when the power bottom is pressed until the boot screen has been completely

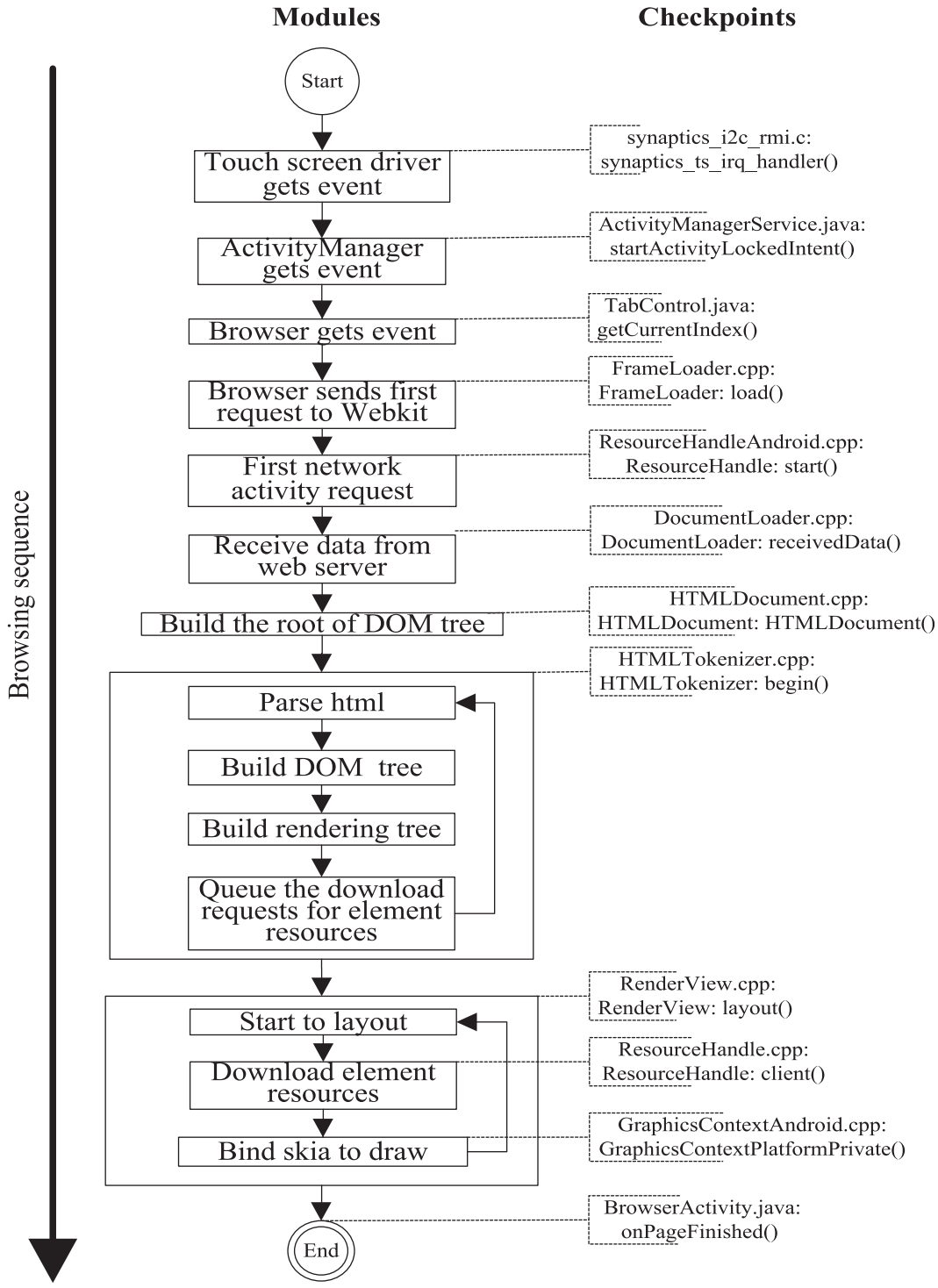

Fig. 4. Browsing procedures.

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

drawn, with all Wi-Fi telecom services disabled at their default factory settings.

2. Booting procedures
   After the power button is pressed, the Initial Program Loader (IPL) checks the hardware and loads the Second Program Loader (SPL), which in turn, loads the compressed kernel image from flash to memory, decompresses the kernel, and executes the first kernel function, start_kernel. The kernel first initializes a great many data structures and tasks, and loads drivers. Next, the first user-space program, init, is executed. Init starts the service_manager (the core Android service responsible for the registration of other Android services), daemon programs, zygote (the parent process of all Java virtual machines), and media_server (managing multimedia framework on Android). When zygote starts, it immediately preloads a number of Java classes and resources for the acceleration of Java applications, and then starts the system_server, a process managing all Android services. System_server scans the flash memory to identify all installed applications, and creates threads for Android services. The booting procedure ends at the finish of the bootFinished routine of the screen-drawing service, SurfaceFlinger.
3. Checkpoints
   According to the above booting procedures, the boot sequence is divided into three areas: boot loader, kernel space, and user space. The first three checkpoints are set at the boundaries of these three areas. The staged instrumentation approach is then used to identify the following seven important checkpoints: the first service (service_manger), the first daemon, 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 illustrates the important modules involved in the booting procedure. Moreover, each block in Fig. 3 has a corresponding checkpoint as mentioned before (the same for browsing and streaming).

4.3. Browsing

1. Browsing time
   Browsing time is defined as the period from when the "go" button is touched on screen in the web browser after a URL has been specified, and ending after completely loading a specified webpage.
2. Browsing procedures
   Pressing the "go" bottom of the screen generates a system event. In particular, this event is handled by the touch screen driver in the kernel, and passed to the activity manager (ActivityManager) and the web browser. The web browser knows the specified URL and sends the webpage request to Webkit, a built-in web browser engine in Android. Webkit then uses HTTP protocol to request the desired webpage, parsing it into a Document Object Model (DOM) tree. The DOM is beyond the scope of this paper, so the details of this process are omitted. Usually, a webpage contains multiple elements loaded together with the target webpage. Webkit continues queuing and downloading the remaining elements while drawing the layout on the screen using a 2D drawing library, skia.
3. Checkpoints
   In the beginning, the module set M contains only the web browser. After multiple iterations, the following important checkpoints are identified as illustrated in Fig. 4: the interrupt request (IRQ) handler of the touch screen driver, the event

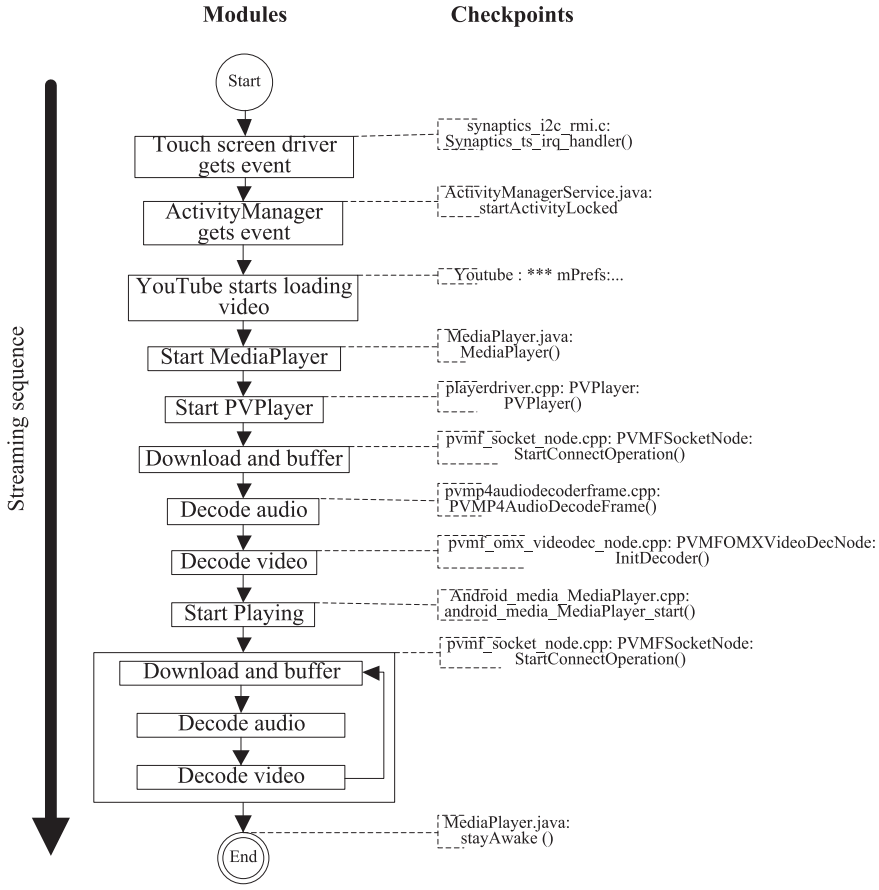

Fig. 5. Streaming procedures.

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

handlers in the activity manager and web browser, the Webkit-invoking function in the web browser, the HTTP request and response functions, the DOM-tree building and parsing functions, and the screen layout functions in Webkit.

4.4. Streaming

1. Streaming time
   The built-in streaming player on Android is called YouTube. Before profiling the streaming scenario, we had to connect to the Internet and execute YouTube, showing a list of video icons on the screen. The streaming time is defined as the period from when a video icon is touched, until the video finishes playing.
2. Streaming procedures
   As with the browsing scenario, touching a video icon 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, a service for every variety of media-playing requests. MediaPlayer dispatches the video-playing request to PVPlayer, the built-in video player. PVPlayer continues downloading and decoding video from the Internet, while displaying video on the screen until the end of play, as notified by the stayAwake function of MediaPlayers.
3. Checkpoints
   YouTube does not release its source, and therefore, the staged instrumentation approach cannot be applied to YouTube directly. During the first iteration of the proposed approach, M includes modules in the application framework layer, and YouTube still leaves a certain amount of log data, which is interpreted as checkpoints during profiling. After multiple iterations, the following important checkpoints are identified as shown in Fig. 5: the IRQ handler of the touch screen driver, the event handler in activity manager, the log messages indicating that YouTube has started loading video, the constructors of MediaPlayer and PVPlayer, the video-playing function in MediaPlayer, the video-downloading, audio-decoding, and video-decoding functions in PVPlayer, and finally, the end function, stayAwake, indicating the termination of the video.

5. Experimental results and observations

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

5.1. Testbed

Experiments were conducted on an Android-based Smartphone, Android Dev Phone 1, referred to hereafter as DUT. The reason for selecting this DUT was that the root permissions had been unlocked for developers to rewrite any program. In addition, the entire system on the platform and the hardware of this DUT was the exact same as that of another off-the-shelf commercial Android Smartphone, the HTC Dream. The version of the platform used for this DUT was Android Open Source Project (AOSP) 1.6. The reasons for selecting this platform were that we want to prove the proposed approach can work in an elder Android platform and this platform is the first Android version supporting CDMA/EVDO, 802.1x, and VPNs networking technologies.

Recall that the Android platform allows only one foreground GUI program to be executed at one time. Hence, to operate profiling tools on-line and collect profiling results during the execution of a scenario, a host machine was needed to provide an alternative, command-line, and operational interface. In addition, the operating interface on the host was a client program called

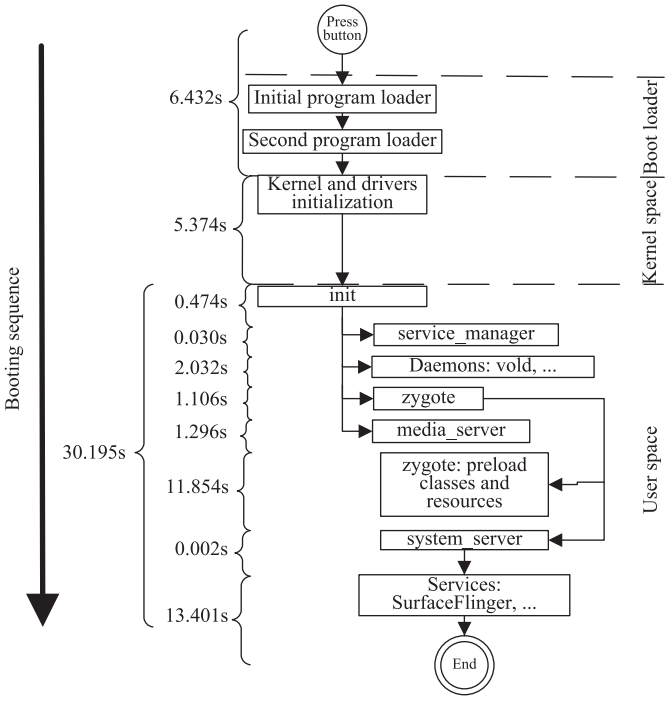

Fig. 7. Profiling results of booting procedures.

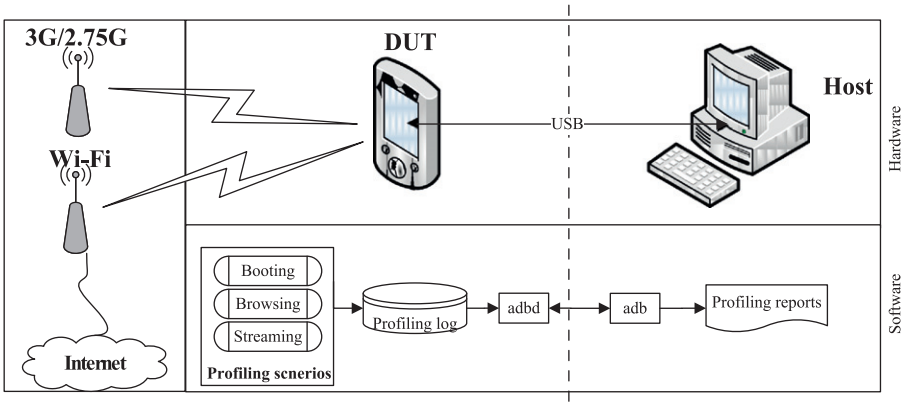

Fig. 6. Testbed.

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

Android Debug Bridge (adb), and its corresponding server on the DUT was adbd. DUT and the host connected to each other through the USB interface. Finally, the EDGE (2.75G), WCDMA (3G), and IEEE 802.11 g (Wi-Fi) networks were accessible in the testbed environment. Figure 6 depicts the described testbed.

5.2. Booting scenario

Figure 7 shows the profiling results for booting with the checkpoints defined in Section 4. The initialization of the boot loader, kernel-space, and user-space environment respectively required approximately 15%, 13%, and 72% of total boot time as shown in Fig. 8(a). Furthermore, when drilling down the user-space initialization, we could observe that two major time-consuming processes, the startup of services and managers (44.4% of user space) and Java classes and resource preloading by zygote (39.2% of user space), as shown in Fig. 8(b).

The reason that the preloading of Java classes and resources and the startup of services and managers were bottlenecks during boot-up was that the former attempts to shorten the time to start applications, while the latter tries to ready all services and managers before the desktop screen appears.

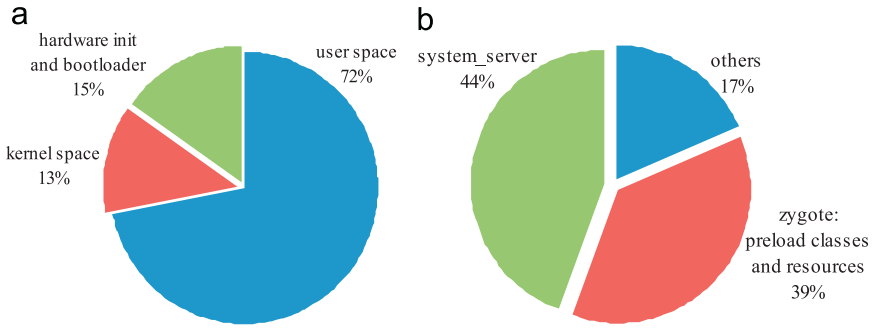

Fig. 8. Detailed profiling results of booting: (a) distribution of booting time and (b) distribution of user-space initiation time.

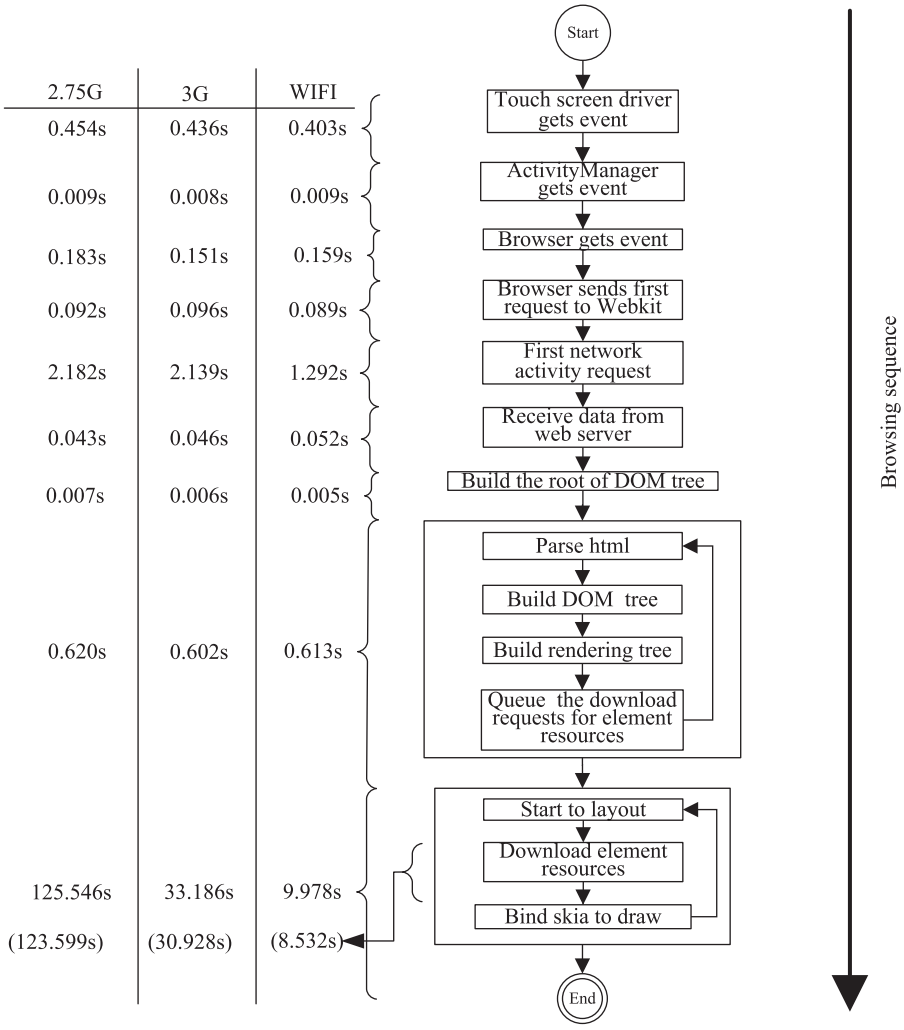

Fig. 9. Profiling results of browsing procedures.

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

It appears that reducing the number of preloaded Java classes and resources could considerably decrease the boot time. However, the startup time for applications would be increased because the Java VM would have to spend time loading the unpreloaded Java classes and resources. Hence, it is a tradeoff between boot time and startup time. On the other hand, in current Android designs, the desktop screen is shown after all services and managers are ready. However, for many users, the later the screen appears, the slower the boot performance is. Thus, if designers of DUTs want to attract attention for the speed of their device, they could have the desktop screen drawn as early as possible and postpone launching services unrelated to the desktop, when redesigning the boot-sequence.

5.3. Browsing scenario

Figure 9 presents the results of time profiling after loading a 2128 KB webpage with eight attached images. The experiments were conducted using three different networking technologies, Wi-Fi, 3G, and 2.75G. The results show that the browsing performance with 2.75G and Wi-Fi was the slowest and the fastest among three networking technologies, respectively. The results also show that downloading element resources consumes 90% of the browsing time, regardless of the network the DUT connects to. Accordingly, on Android, networking technology is the most significant factor influencing browsing performance.

There are two interesting things. One is that the overhead of this proposed approach was about 5% in CPU loading, from 63% to 68%. The other is that the source code of the drivers related to the Graphic Processing Unit (GPU) were also instrumented in this experimentation, but with no GPU related output during browsing procedures. Furthermore, results of the experiment showed that the rendering function consumed only 5% of the browsing time. It would be easy to believe that the GPU is not invoked during the process of rendering the browser.

Figure 10 presents the experimental results measuring the performance of rendering webpages of different sizes under Wi-Fi networks. The size of webpages is adjusted according to the size of a single attached web image. The "others" denote the sum of all execution times excluding the time for downloading webpages. From Fig. 10, it can be seen that regardless of the size of the webpage, the time of "others" is small and steady. On the other hand, network transmission is the most important factor influencing the performance of web browsing.

5.4. Streaming scenario

Figure 11 shows the time profiling results of playing a one-minute MPEG-4 video from the YouTube site. The video file size was 22 MB and the DUT accesses Internet through the Wi-Fi

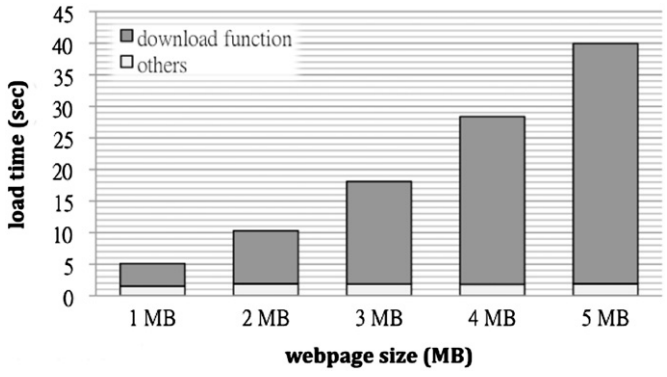

Fig. 10. Profiling results of webpages of various sizes.

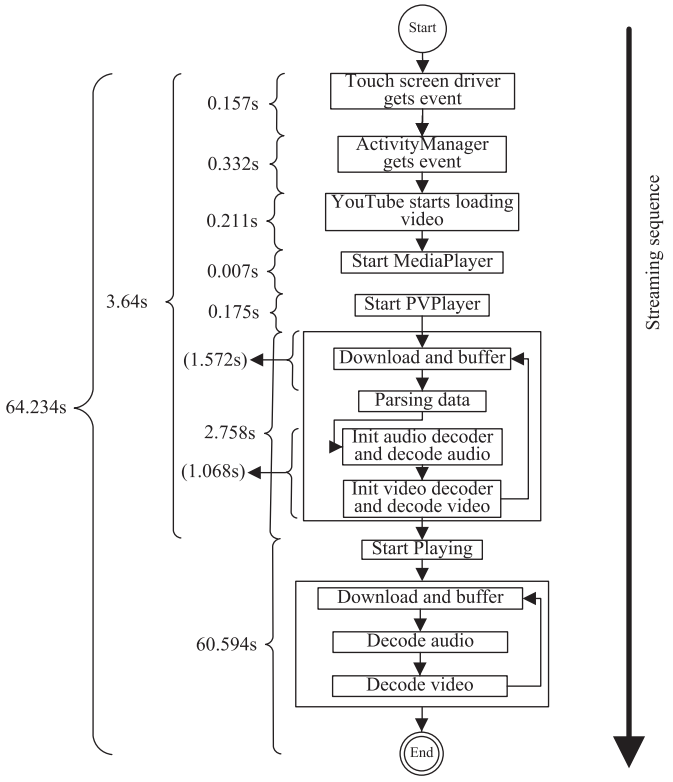

Fig. 11. Profiling results of streaming procedures.

interface. No jitter occurred during the experiments, so the streaming performance perceived by users was the time spent preparing the video. In other words, users needed to wait 3.64 s to watch a video after touching the play icon. Obviously, two bottlenecks residing in video preparation are video-downloading (1.572 s) and data-decoding (1.068 s), accounting for 77% (=(1.572+1.068)/3.64) of the preparation time. Furthermore, the data-decoding included audio decoding (0.515 s) and video decoding (0.553 s).

6. Conclusions

In this work, we developed a staged instrumentation approach to time profiling using a multi-language, multi-layer platform. Then this approach for time profiling booting, browsing, and streaming on off-the-shelf products was implemented and the execution flow on these scenarios was investigated to verify its effectiveness.

Although it is well known that the user-space initialization dominates the time required to boot, until now, no studies have provided a complete numeric distribution of Android booting time. This paper first illustrates how 72% of Android booting time is spent on initializing the user-space environment, 13% on kernel initialization, and 15% on the boot loader. Furthermore, in user space, the time required for starting Android services and managers, and preload Java classes and resources are 44.4% and 39.2%, respectively. As for browsing performance, the experimental results indicated that networking technology is the most significant factor influencing the speed of the Android system. The time required to draw the screen takes less than 5% of the total time required to browse a 2128 KB webpage. This also explains why GPU-acceleration is unnecessary for rendering browser functions. In the streaming scenario, the Android-based device required 3.64 s for video preparation before it was able to play a video file

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024

over Wi-Fi connections. In addition, the execution time of video-downloading and data-decoding accounted for 77% of preparation time.

The proposed approach to time profiling is capable of integrating energy profiling tools to measure energy consumption during system runtime. In the future, other scenarios like GPS navigation, and Android on set-top boxes, will also be profiled. Although the proposed approach is capable of profiling multi-language, multi-layer platforms, the profiling results still require manual analysis. Therefore, if an automatic analysis tool could be developed, the entire experimental process could be automated.

Acknowledgments

This work was supported in part by Institute for Information Industry under the project of embedded software and living service platform and technology development, which was subsidized by the Ministry of Economy Affairs, Taiwan. It was also supported in part by D-Link Corp. and Realtek Seminconductor Corp.

References

Android Developer. Available 〈http://developer.android.com/reference/packages.html〉.

Comscore.com. 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〉, July 2008.

C. Ghoroghi, T. Alinaghi. An introduction to profiling mechanisms and Linux profilers. Available 〈http://www.docstoc.com/docs/7671023/An-introduction-to-profiling-mechanisms-and-Linux-profilers〉.

Kernel Function Trace. Available 〈http://elinux.org/Kernel_Function_Trace〉.

J. Levon. Oprofile: a system profiler for Linux. Available 〈http://oprofile.sourceforge.net/〉.

N. Lee, S.S. Lim. A whole layer performance analysis method for Android platforms. In: Proceeding of IEEE symposium on embedded systems for real-time multimedia; October 2011. p. 1.

Z. Mahkovec. Available 〈http://www.bootchart.org/〉.

H. Pieterse, M.S. Olivier. Android botnets on the rise: trends and characteristics. In: Proceeding of information security for South Africa (ISSA); August 2012. p. 1-5.

Patel R, Rajawat A. A survey of embedded software profiling methodologies. International Journal of Embedded Systems and Applications 2011;1(2):19–40.

Printk Times. Available: 〈http://elinux.org/Printk_Times〉.

G. Singh, K. Bipin, R. Dhawan. Optimizing the boot time of Android on embedded system. In: Proceeding of IEEE 15th international symposium on consumer electronics; June 2011. p. 503–508.

R. Trestian, A.N. Moldovan, C.H. Muntean, O. Ormond, C.M. Muntean. Quality utility modelling for multimedia applications for Android mobile devices. In: Proceeding of IEEE international symposium on broadband multimedia systems and broadcasting; June 2012. p. 1–6.

Toupin D. Using tracing to diagnose or monitor systems. IEEE Software 2011;28(1):87–91.

X. Wei, L. Gomez, I. Neamtiu, M. Faloutsos. In: Proceedings of the 18th annual international conference on mobile computing and networking; August 2012. p. 137–148.

Yoon H-J. A study on the performance of Android platform. International Journal on Computer Science and Engineering 2012;4(4):532–7.

B. Zhao, B.C. Tak, G. Cao. Reducing the delay and power consumption of web browsing on smartphones in 3G networks. In: Proceeding of 31st international conference on distributed computing systems; June 2011. p. 413–422.

Please cite this article as: Lin Y-D, et al. Booting, browsing and streaming time profiling, and bottleneck analysis on android-based systems. Journal of Network and Computer Applications (2013), http://dx.doi.org/10.1016/j.jnca.2013.02.024