2016 IEEE International Conference on Computational Science and Engineering, IEEE International Conference on Embedded and Ubiquitous Computing, and International Symposium on Distributed Computing and Applications to Business, Engineering and Science

An Android Systrace Extension for Tracing Wakelocks

Bui Duc Binh
Department of Computer Science and Engineering
Waseda University
Tokyo, Japan
binh@kasahara.cs.waseda.ac.jp

Kimura Keiji
Department of Computer Science and Engineering
Waseda University
Tokyo, Japan
kimura@apal.cs.waseda.ac.jp

***Abstract*—Most of Android users have experienced issues with the battery life. One cause of battery drainage is the usage of the Wakelocks, which keep the CPU in working mode to enable applications to perform work in the background, such as communicating with their servers or collecting GPS information. Without acquiring the Wakelocks, the application might degrade its user experience. For instance, an SNS application might receive messages from other users with delay. However, the improper usage of Wakelocks could result in poor battery life. Being aware of the behavior and the usage of Wakelocks in particular applications in real-time can help Android developers to solve the problem of improper Wakelock usage. This paper introduces a tool for fine-grain tracing of both application and kernel Wakelocks by extending the Android Systrace. This tool enables developers to have a more detailed view of their application Wakelocks as well as the system Wakelocks so that they can achieve better power optimization.**

Keywords—Android; Wakelock; Tracing tool; Systrace; Power consumption

I. INTRODUCTION

Android smart devices have become one of the most exciting trends in the global market due to their affordable price but powerful performance. However, the demand for high performance leads to an enormous amount of power dissipation in Android devices. Since the capacity of the battery is limited, many Android users face issues with poor battery life, degrading the user experience. Therefore, it is important to analyze the power consumption problem on the Android platform.

One cause of the battery draining problem is that there are too many processes running in the background, which are invisible to users but still use a lot of system resources. These processes consume battery life even when the device screen is turned off. In order to have the background processes work properly, it is necessary to keep the CPU in running mode. The Android OS provide a mechanism called Wakelock [1] [2] to fulfill that requirement. An application must acquire a Wakelock to force the device to stay on when it wants to start a process in the background; otherwise, the process might not function correctly and affect the user experience as the system might unexpectedly enter low power mode. When a Wakelock is acquired, the system keeps the device awake according to the level requested. There are 4 types of Wakelock as shown in Table 1.

TABLE Ⅰ: 4 Types of Wakelock

Type	CPU	Screen	Keyboard
PARTIAL_WAKE_LOCK	ON	OFF	OFF
SCREEN_DIM_WAKE_LOCK	ON	DIM	OFF
SCREEN_BRIGHT_WAKE_LOCK	ON	BRIGHT	OFF
FULL_WAKE_LOCK	ON	BRIGHT	BRIGHT

The important point here is that a Wakelock must be released as soon as possible when the application no longer needs it to perform any task. Unreleased Wakelocks, requiring too many Wakelocks, or acquiring Wakelocks for a long time would significantly affect the device battery life. Therefore, analyzing the usage of the Wakelock helps developers to understand unexpected battery drainage and support them in debugging and optimizing their Android applications.

Previous works have taken several approaches to manage Wakelocks. In [11] [14] [15], the authors analyze the application data flow, detect a Wakelock bug at compile time, and notify developers of the bugs. [12] and [13] are attempts to detect Wakelock misuse at runtime and notify users of the detections. However, none of these works could show an overall picture of Wakelock usage for a given period of time, which, we believe, is extremely important to analyze the Android Wakelock. This paper introduces a tool to trace Android Wakelocks at a fine-grained level by extending the Android Systrace tool [3]. This tool enables developers to have a more detailed understanding of their application Wakelocks as well as the system Wakelocks in order to achieve better power optimization.

The rest of this paper is structured as follows. Section 2 presents the proposed tool architecture. Section 3 shows the evaluation result, and Section 4 gives the conclusion of the paper.

II. ARCHITECTURE

A. Android Systrace

The tool we are developing is based on the Android Systrace tool, which helps to analyze the performance of the application and the Android system by collecting trace data from the Android kernel and application threads. This data can help developers to understand the behavior of the Android system and applications processes. The combined data is displayed in a

978-1-5090-3593-9/16 $31.00 © 2016 IEEE
DOI 10.1109/CSE-EUC-DCABES.2016.175

webpage that allows developers to zoom in, zoom out, move left and move right. Fig. 1 shows an example view of the Android Systrace. Each time slice presents a task at a given period of time. The Android Systrace provides various kind of tracing data categories. For example, these categories include graphics, audio, video, dalvik VM, CPU scheduling, CPU frequency, CPU load and so on. The Android Systrace can help developers diagnose various application issues such as UI performance of the application [4].

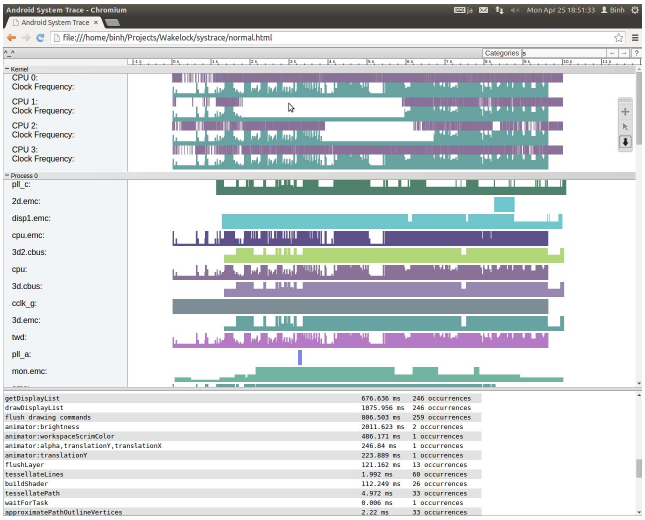

Figure 1. The Android Systrace Tool Example

Android Systrace is a two-stage tool: data collecting and data processing. The data is collected and stored in a trace log file by the debug tracing kernel module. The data to be collected is specified as arguments to a Python script [6]. Android Systrace tool uses a built-in command, called atrace, to collect the tracing data. Once the trace data to collect is selected using atrace, the kernel begins collecting trace information. After a period of time, all data will be transferred from the Android device to an external computer using the adb tool. Then, those data are formatted and displayed in a browser.

B. *The Extended Android Systrace Tool Architecture*

In order to collect Wakelocks usage data, we need to inform the kernel to obtain Wakelocks trace. We also need to modify the final data formatter to display the Wakelocks information correctly in the browser.

There are two types of the Android Wakelocks: kernel Wakelocks and application Wakelocks. Kernel Wakelocks are acquired at the kernel level while application Wakelocks are acquired at the application level. Tracing both types of Wakelocks is necessary. For the kernel Wakelocks, we put tracepoints [7] in the wake_lock() and wake_unlock() functions which are the entrypoints of the Wakelock management in the power module of the Linux kernel. We also define a new trace function and trace format for the Wakelocks trace in <trace/events/sched.h>. It provides a sysfile for enabling Wakelocks tracing.

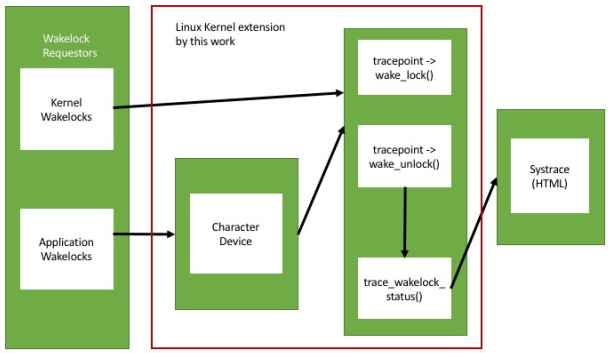

Figure 2. The Architecture of Extended Android Systrace

For the application Wakelocks, we need to create a bridge to transfer the Wakelocks information from userspace to kernel space. Since Android applications are usually developed in Java, the information on the application Wakelocks is firstly generated in userspace. Using JNI [8], we are able to pass all tracing information to the native environment. After that, the Wakelocks information is sent to the kernel space by writing it to a new character device. The character device driver will handle the information sent to the kernel space. We put a Wakelock tracepoint in that driver, hence the kernel will be able to capture the Wakelocks information from Android applications. Fig. 2 shows the full picture of the extended Android Systrace Architecture.

III. TRACING RESULTS

In this paper, we implemented the experiments to capture the Wakelock trace in various periods of time. We also created a test application to check the presence of the Wakelock acquired by the test application in the extending Android Systrace.

A. *Evaluation Environment*

All experiments are implemented in the Nexus 7 2012 installed with Android OS 4.4.4. We make some modifications to the Android source code (AOSP) and the Android kernel and recompile the whole system in the Ubuntu 12.04 environment. The test application is developed in Android Studio v1.4 [9]. A Wakelock is acquired for 20 minutes and then released.

B. *Evaluation Results*

Fig. 3, 4, 5 and 6 show the tracing results in the period of 2 minutes, 20 minutes, 1 hour and 2 hours, respectively. Fig. 7 shows the tracing result of executing the test application. The results indicate that the developed extensions effectively present an overall picture of Wakelocks usage. We can observe Wakelocks precision at the granularity of microseconds and how the kernel and the Android applications are acquiring and releasing Wakelocks. The trace diagram contains many time slices with labels. The time slices length represents the acquiring time, and the label represents the package name or the process. The areas with red marks in Fig. 3, 4, 5 and 6 are the examples of time slices. By looking at a time slice and its label, developers are able to study the behavior and the usage of Wakelocks in applications of interest.

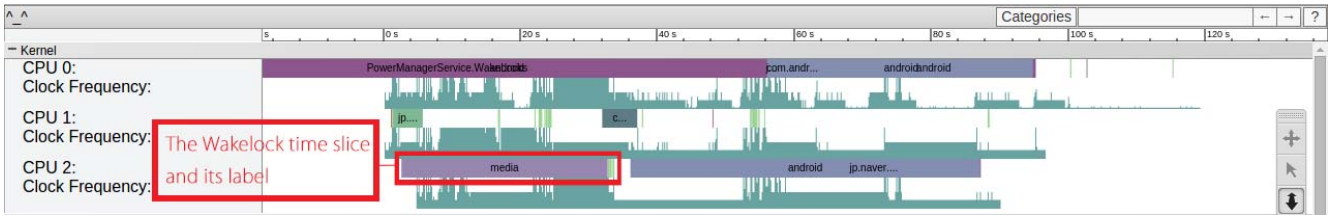

Figure 3. The Extended Android Systrace (2 minutes)

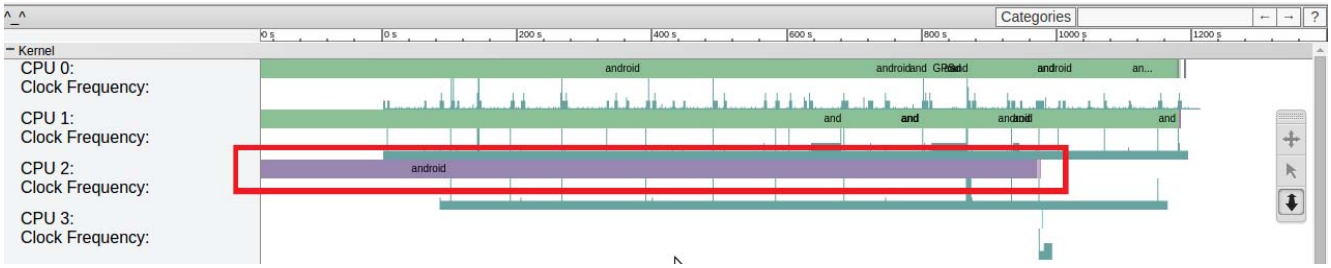

Figure 4. The Extended Android Systrace (20 minutes)

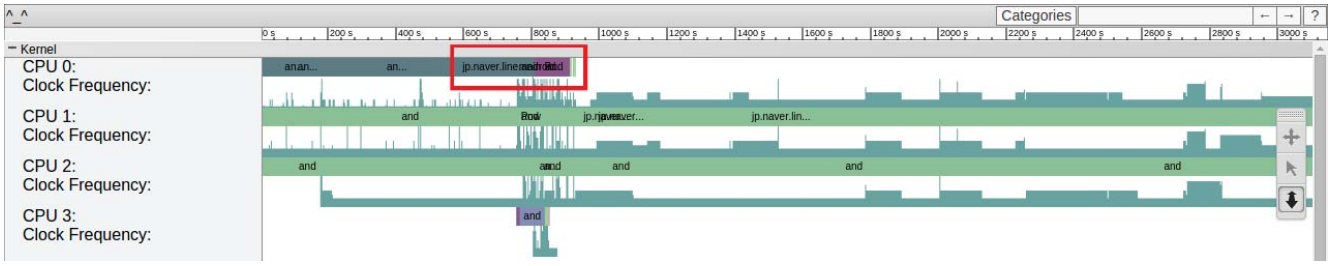

Figure 5. The Extended Android Systrace (1 hour)

Figure 6. The Extended Android Systrace (2 hours)

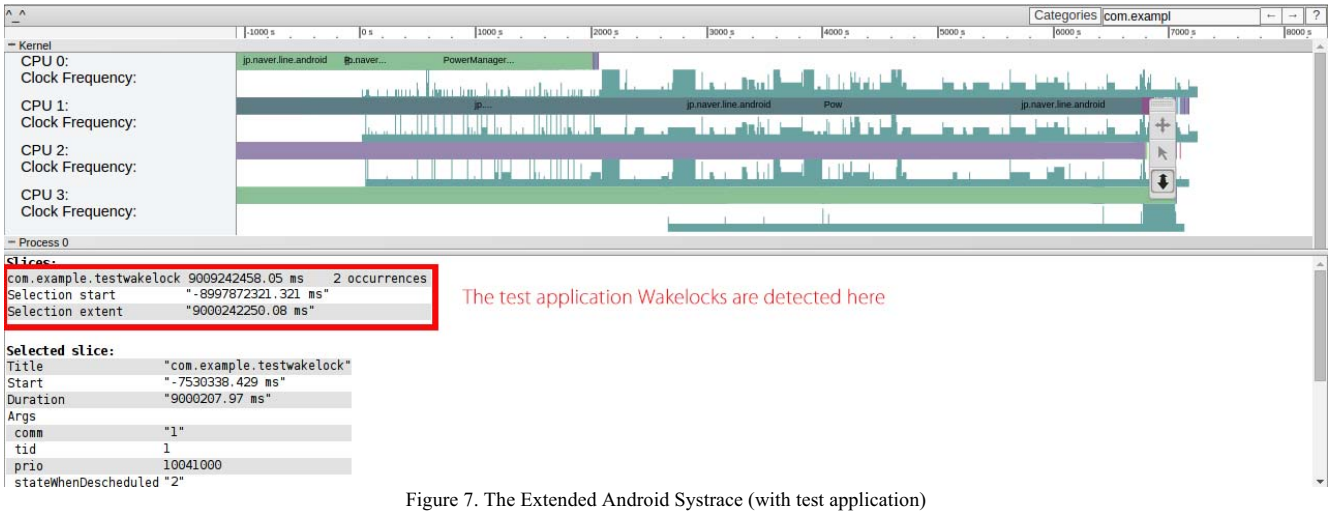

Figure 7. The Extended Android Systrace (with test application)

TABLE II : Comparison between Kernel Wakelock and Application Wakelock acquiring time

	Call times	Acquiring Time(ns)
Kernel Wakelock	9268	2.15637E+10
Application Wakelock	466	8.88196E+18

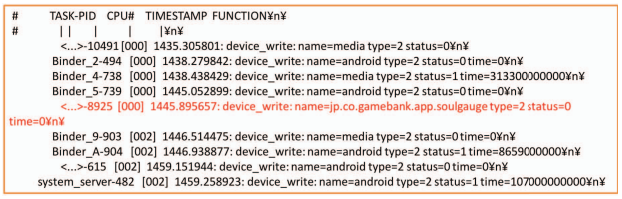

Figure 8. The Detection of Wakelock Bug in a Commercial Application

In Fig. 3, the "media" time slice shows a Wakelock acquired by "media", which is an Android system process. Similarly, in Fig. 4, we can observe an Android system process acquiring Wakelock, namely "android". Fig. 5 shows an application Wakelock time slice. The application name can be derived from the label "jp.naver.line.android", which implies the Line application. In the extended Android Systrace, the green slices represent application Wakelocks while the pink slices represent kernel Wakelocks.

If there are many Wakelocks in a systrace, developers can use the search function, which is located in the right corner of the browser. Wakelock usage can be searched by package name. Fig. 7 shows an example of the search function. In Fig. 7, it is clear that our tool can trace the Wakelock acquired by the developed application, named com.example.testWakelock.

Table 2 shows that the total time spent in kernel Wakelock is significantly smaller than the total time spent in application Wakelocks. This observation can hint developers to focus on optimizing application Wakelocks over kernel Wakelocks.

In this thesis, we actually used our tool to verify applications from Google Play store. Fig. 8 shows that we successfully detected a Wakelock bug in a commercial application [17]. It can be seen that a Wakelock was acquired but not released. The releasing trace is not captured by our tool, which will collect both Wakelock acquiring and releasing in normal case.

IV. CONCLUSION

Acquiring Wakelocks is necessary to execute background tasks properly on an Android platform. However, careless management of Wakelocks can drastically drain the battery life and degrade the user experience. Although some previous works try to handle improper use of Android Wakelock by detecting bugs at compile time or runtime, none of those works present an overall picture of the Wakelock usage. This paper introduces a tool for tracing Android Wakelocks at a fine-grained level by extending the Android Systrace tool. The experimental results show that our tool could capture both kernel Wakelocks and application Wakelocks at microsecond granularity. In addition, we could detect Wakelock bug from a real Android application by the developed tool. In future work, we plan to continue development of this tool by automatically identifying Wakelock bugs and offering hints to help developers optimize their application power consumption.

REFERENCES

[1] Android PowerManager.
http://developer.android.com/intl/ja/reference/android/os/PowerManager.html

[2] Android Wakelock.
http://developer.android.com/intl/ja/reference/android/os/PowerManager.htmlWakelock

[3] Android Systrace.
http://developer.android.com/intl/ja/tools/help/systrace.html

[4] Analyzing UI Performance with Systrace.
http://developer.android.com/intl/ja/tools/debugging/systrace.html

[5] Android Open Source Project (AOSP).
https://source.android.com/source/index.html

[6] Android systrace.py.
https://chromium.googlesource.com/android_tools/+/master/sdk/platform-tools/systrace/systrace.py

[7] Linux Tracepoint.
https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

[8] Android NDK JNI.
http://developer.android.com/intl/ja/ndk/index.html

[9] Android Studio.
http://developer.android.com/intl/ja/tools/studio/index.html

[10] Pratiksha S. Patil, Jinalkumar Doshi, Dayanand Ambawade, "Reducing power consumption of smart device by proper management of Wakelocks" Advance Computing Conference (IACC), 2015 IEEE International. Banglore, pp. 883–887, June 2015

[11] Faisal Alam, Preeti Ranjan Panda, Nikhil Tripathi, Namita Sharma, Sanjiv Narayan, "Energy Optimization in Android Applications through Wakelock Placement" 2014 Design, Automation & Test in Europe Conference & Exhibition. Dresden, pp. 1-4, March 2014

[12] Xigui Wang, Xianfeng Li, Wen Wen, "WLCleaner: Reducing Energy Waste Caused by Wakelock Bugs at Runtime" Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on. Dalian, pp. 429-434, August 2014

[13] Kwanghwan Kim, Hojung Cha, "Wake-scope: Runtime Wakelock Anomaly Management Scheme for Android Platform" EMSOFT '13 Proceedings of the Eleventh ACM International Conference on Embedded Software. Montreal, pp. 1-10, September 29-October 04, 2013

[14] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, Yuvraj Agarwal, "Towards verifying android apps for the absence of no-sleep energy bugs" Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems. Hollywood, CA, pp. 3, October 2012

[15] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, Samuel P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps" Proceedings of the 10th international conference on Mobile systems, applications, and services. Low Wood Bay, Lake District, UK, pp. 267-280, June 2012

[16] Line application.
https://play.google.com/store/apps/details?id=jp.naver.line.android

[17] SoulGauge application.
https://play.google.com/store/apps/details?id=jp.co.gamebank.app.soulgauge