# Analyzing GUI Running Fluency for Android Apps

Tian Huang Zhenyu Zhang Xue-Yang Zhu*
State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
(+86)10-6266{1630, 1630, 1655}
{huangt, zhangzy, zxy}@ios.ac.cn

## ABSTRACT

Android as a free open platform has become increasingly popular and been widespread adopted in mobile, tablet, and other devices. However, a great number of issues, such as inadequate quality and the fragmentation phenomenon, have emerged, enhancing the difficulty of developing. Among them, the running fluency of Android apps directly affects user experience directly. As a result, it is of great significance to detect and analyze it.

The frame rate and 16-ms-per-frame benchmark are the most popular metrics to evaluate and measure the smooth performance of Android application GUIs and to test the quality of apps by developers. However, very few studies have analyzed the performance and consider the adequate usage of frame rate before extensively applying it. Further, current tools provided by Google or third-party cannot obtain the frame rate and rendering time for the system with multiple applications.

In this work, we focus on the performance issue, revisit and analyze various factors that Android apps do not run smoothly, along with Android graphic system. After that, we present ARFluency --- a tool to measure and automatically analyze the system and applications without modifying the source code of the Android apps. We also conduct an experiment to validate our tool using realistic Android apps. Experimental results show that although even the apps running fluently do have problematic frames. However, the metrics of frame rate cannot accurately reflect the performance of Android applications.

## CCS Concepts

**• Software Engineering →Software/Program Verification—Reliability • Software Engineering →Testing and Debugging—Testing tools, Tracing.**

## Keywords

Android; Performance analysis; Running fluency; FPS

## 1. INTRODUCTION

Android [7] created by Google and the Open Handset Alliance has become a rather popular platform for mobile applications (apps) commonly used in a variety of devices including mobile, tablet, etc. It is reported that the number of those devices ranging in different sizes of screens has reached ten billion [5]. Meanwhile, an enormous amount of applications have been developed. As subscribes are growing in number and market share is expanding, a great number of issues have emerged, especially GUI lagging, memory bloat, and energy leak [2]. Such performance bugs bring in bad user experience.

Android operation system is continuously updated. Every version provides better performance on various aspects such as user interface (UI), battery life, user control, etc. Current Marshmallow based on API level 23 runs much more fluently than the first one released. Besides, developers tend to develop and optimize their apps so as to get more online downloads because apps of good quality have advantages over those of bad performance in market competition and the "survival of fittest" circumstances. Android is always on the road toward high performance.

However, as Android system grows rapidly, various problems appear, e.g., device-specific problems and software related issues, resulting in *fragmentation* (see Figure 1). These problems have deteriorated the difficulty of developing apps, as well as testing them. For example, an app runs fluently on one device may not perform well on another. Furthermore, any occasion where performance issues of an app occur may lead to uninstallation, especially GUI lagging (e.g. screen tearing, standstill) and poor responsiveness. To users, whether an Android app runs fluently or not extremely matters.

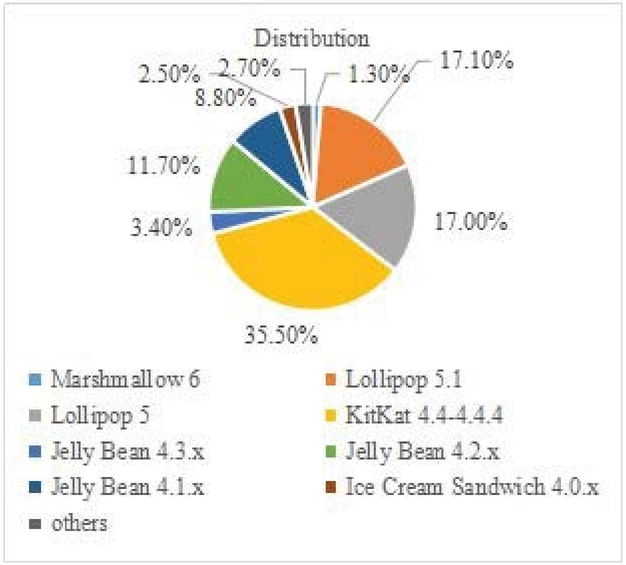


**Figure 1. Distribution of Android OS in market share.**

*MSCC'16,* July 05 2016, Paderborn, Germany
© 2016 ACM. ISBN 978-1-4503-4343-5/16/07…$15.00
DOI:http://dx.doi.org/10.1145/2940353.2940356

* Corresponding author.

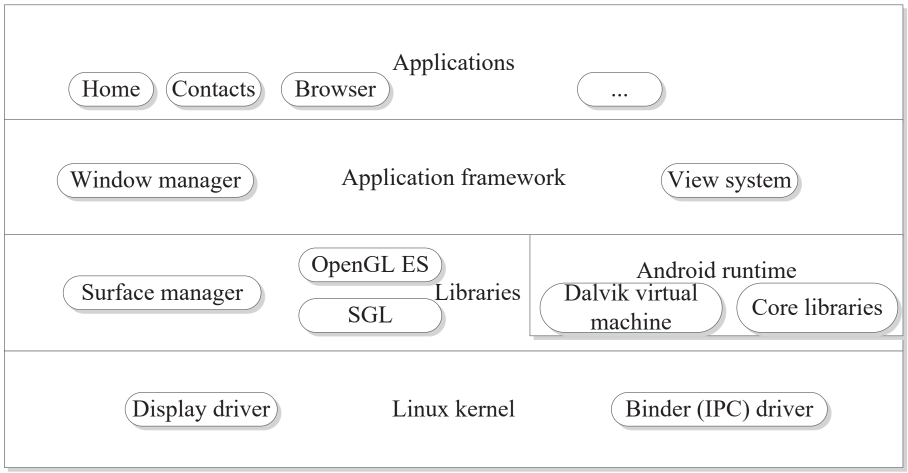


**Figure 2. Android system architecture.**

In this work, we focus on the performance of Android application GUIs. By introducing the phenomena where the apps do not run smoothly from the users' perspective, we analyze Android graphic system used to present graphics to the screen. Then we discuss two available metrics to evaluate and measure the performance of Android application GUIs, namely the *frame rate* and the *16-ms-per-frame* benchmark. Although frame rate is commonly used as an important factor to measure the performance on automated testing platforms (ATPs), we revisit whether the frame rate accurately reflects the performance. After that, we implement our tool --- ARFluency to access the frame rate and the rendering time usage of Android apps to investigate tools provided by Google and thirty-party. Finally, we conduct an experiment to use realistic apps downloaded from the Android markets to test our tool.

Contributions of the work are as follows.

1) This is the first work focusing on running fluency of Android application GUIs from the users' perspective.
2) This is the first work choosing and comparing both the frame rate and the16-ms-per-frame as metrics.
3) We implemented a tool ARFluency to obtain the frame rate and rendering time usage of Android apps.
4) We conducted an experiment to use realistic Android apps from Google Play to validate our tool.

The remainder of this paper is organized as follows. Section 2 discusses the performance related including the phenomenon such as screen tearing, Android graphic system and metrics, along with some tools available. Section 3 presents our tool ARFluency. Section 4 conducts an experiment and reports observations. Section 5 introduces related work. Section 6 concludes the paper.

## 2. BACKGROUND AND MOTIVATION

In this section, we first introduce Android and Android systems as the background of this work, together with interested performance metrics. After that, we explore existing tools and motivate our work.

### 2.1 Android

From the users' perspective, running fluency emerges as a comprehensive performance of a system that delivers information quickly. In Android, it refers to the responsiveness and rendering performance of Android app GUIs. A most visible manifestation of poor responsiveness is an "Application Not Responding" error (ANR [3]). When the application does not respond to user input (e.g., screen tough), within certain time (5 seconds in an Activity or 10 seconds in a BroadcastReceiver), an ANR dialog should be presented to the user by *Android run-time* [10]. Such errors create a highly negative user experience, and the efforts to avoid them is of great importance [11].

Frame rate (Frames per Second, FPS) is the rate at which an imaging device generates consecutive images, while the refresh rate is the rate at which the display hardware refreshes the images called frames. The vertical synchronization technique (V-Sync) is introduced since Android 4.1, aiming at limiting the frame rate lower than the refresh rate [8].

Screen tearing results from two frames appearing on the screen at the same time. After introducing V-Sync, screen standstill comes out instead of screen tearing. Both screen tearing and standstill are caused by the unequal value between frame rate and refresh rate. Regardless of refresh rate depending on hardware, the frame rate is controlled by Surface Manager [8].

Android graphic system is extremely complex and it penetrates the Android system architecture, illustrated in Figure 2. Each app may have one or more surfaces, which are the cache of screen and store the drawing data from graphic libraries such as Skia Graphic Library (SGL) and OpenGL ES [6]. Surface Manager (aka. SurfaceFlinger) is responsible for combining the rendered surfaces of apps and then updating the frames by writing data to frame buffer to draw them on screen through EGL (see Figure 3).

Rendering frames need several steps such as draw, process, execute, etc. The number of steps is not fixed (e.g. three or four). It depends on version of the Android system.

Current Android platform since Jelly Bean [4] has already reached a new height of displaying graphic user interface with making progress of Open GL ES from 2.0 to 3.0 and employing triple buffering rather than double buffering, etc.

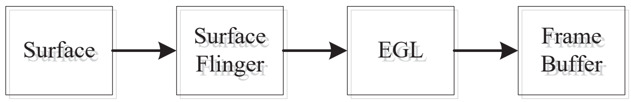


**Figure 3. The process Android drawing to screen.**

## 2.2 Metrics

The performance of apps can be measured in a number of ways, including execution time, memory usage or battery consumption that typically yield useful values for performance assessment [9]. Execution time actually refers to processing overhead of CPU. Compared to PCs, smart phones have limited computing resources, and smart phones applications are more prone to have performance problems [25]. In terms of CPU, memory, battery belonging to the resources category, the less resources the app consumes, the better performance it should have. In addition, metrics also include network traffic, frame rate, 16-ms-per-frame benchmark and so on. Network traffic is the amount of data transferred through network; frame rate is the rate at which consecutive images drawing to the screen, and the 16-ms-per-frame benchmark describes the maximum time of rendering a frame of a UI window.

Among those metrics measuring the performance of apps, both the frame rate and the 16-ms-per-benchmark are popular to evaluate and measure the performance of Android application GUIs. In Android, it is required that the frame rate of an app should reach 60 fps to run fluently on devices. Hence, each frame should be rendered in no more than 16.67ms. Compared to 16.67ms, 16-ms-per-frame benchmark is stricter. In another word, if 16-ms-per-frame benchmark is satisfied, the app should run smoothly. Since the frame rate is closely related to 16-ms-per-frame, the less time rendering frames used, the larger the frame rate is.

## 2.3 Existing Tools and Improvement Space

A great many of tools are available and used by developers to detect performance-related issues. Linares-Vasquez et al. [1] listed out some of them, which are divided into six categories. There are also some tools provided by Google or third-party, which are to help profile and analyze the performance of Android application GUIs.

Systrace [12] is used to analyze the performance of rendering. It can collect and inspect timing information across the entire Android device. It inspects the frame rendering information of an Android app, and uses different colors to distinguish whether a frame exceeds the 16-ms-per-frame run-time limit. However, it needs source code of Android apps.

Profile GPU Rendering [13] exists in a mobile device running at least Android 4.1 with Developer Options enabled. It provides a visual representation of how much time it takes to render the frames of a UI window relative to the 16-ms-per-frame benchmark. It can inspect rendering frames info and record at most 128 frames when rendering frames of each UI across an entire Android device. It does not need the source code of an app. However, it cannot be employed in Android of versions prior to 4.1. At the same time, the representation can neither transform into accurate numerical data nor write data into a file.

FPS Meter [14] only measures the frame rate of the entire Android device rather than that of a specific application. It then shows FPS values on arbitrary corner of the screen in real time without storing them. It can be used to measure the frame rate of the app without its code on devices with the prerequisite that it requires root privileges.

GT (Great Tit) [16] is an open source project in Github. It is a portable debugging tool for bug hunting and performance tuning anytime and anywhere. It is designed for skillful developers to monitor their apps in devices just under the circumstances of developing the apps with GT project.

The four tools mentioned above have merits and demerits respectively. However, none of them can obtain frame rate or frames rendered info of a specific app. Open source Android developers primarily rely on manual testing and analysis of reviews for detecting performance bottlenecks using tools like the first two [1], while the other two tools can be used as reference tools to develop a suitable tool for automated testing platforms which arise with the advantage of compatibility testing, etc.

In addition, there are mainly two ways to get FPS. One is to instrument the app during developing. The other is to decompile the apk file and edit the source code. Either way needs to modify the code of an app. Mostly, apps are encrypted avoiding being cracked. When testing on ATP, it is rather problematic to obtain the frame rate. Moreover, timing info about rendering frames against 16-ms-per-frame is also of great importance.

Therefore, a tool should be developed to obtain frame rate and frames rendered info across system and multiple apps without modifying the source code so as to test and analyze the smooth performance of GUI in Android apps.

## 3. OUR TOOL

In this section, we introduce our tool --- ARFluency and elaborate on its mechanism.

## 3.1 ARFluency

We implement a tool named ARFluency, which can access the frame rate and frames rendered information of a specific app across system and apps without modifying their source code.

In Android, each app has its own process and one app usually cannot get information of another, for security consideration. Each app should request permissions to deal with tasks. For example, if an app asks to install, system may ask the user whether to install it and accordingly grant corresponding permissions. If the user chooses not to install, then the app could not run normally until it actually accesses these permissions.

The permissions in Android are mainly categorized into three kinds as follows, based on the user experience perspective:

1) Android owner permissions. Once an Android device is bought, the user has privileges to install apps.
2) Android root privileges. They are the highest permissions in Android system, roughly equaling to administrator privileges in Windows operating system.
3) The permissions requested by apps. Developers can develop various apps leveraging the SDK (Software Development Kit). The authority of an app attached to access the resources may request permissions. For different resources, the apps are supposed to request permissions respectively.

The information about frame rate and frames rendered of apps belongs to relatively independent processes. So among the tools mentioned above, FPS Meter and GT perform well with the prerequisite that the Android device should have root privileges. Therefore, ARFluency should be granted Android root privileges for the sake of operating normal on devices.

## 3.2 Mechanism

The mechanism of our tool is to find the SurfceFlinger process and inject it with a so file, which is a library used to monitor the process and obtain relevant information such as frame rate. The step makes use of code injection technique. Besides, ARFluency

**Table 1. Experimental data.**

| App name | Issued frames (%) | Average time per frame (ms) | frame rate (fps) | The most time-consuming step |
|---|---|---|---|---|
| Chrome | 1.14 | 2.05 | 34.75 | Process |
| Shadowsocks | 0.84 | 3.72 | 21.25 | Process |
| Google translate | 0.78 | 3.43 | 15.47 | Process |
| Google photos | 4.36 | 13.12 | 24.00 | Process |
| AnTuTu Benchmark | 16.33 | 11.77 | 26.25 | Draw |

adopts adb shell instruction (e.g. Dumpsys command) . It executes the instruction every certain time, and dumps the result data.

Our tool is implemented in C++ language and the NDK [26] tool is used to compile the codes into executable and static libraries. Our tool finds the target process and injects it using a prior static library if the process exists.

The code injection technique in our tool consists of two parts. One is an executable file called Inject. It is used to hook the process. The other is a static library named libsurfaceflinger.so. It consumes little resources of CPU and RAM. Once Inject is successfully executed, the so file exists in Android system to obtain information until it is unloaded by force.

There is only one UI in ARFluency, which consists of three parts. The first part is to show the device information including the version of Android system, instruction set, and so on. The second part is to select and list all package names of all apps installed on the devices. And the third part presents the run-time information of the app tested.

ARFluency is allowed to select the package name of an app in entire Android device. After choosing the app, the user may click "Start" button. Once clicked, the button changes itself to "Stop". Then we can just manipulate the app selected. To finish testing, just switch back to ARFluency and click the "Stop" button. Meanwhile, ARFluency can obtain and collect the frame rate and frames rendered information of the app between the moments of two clicks.

## 4. EXPERIMENT AND ANALYSIS

In this section, we give our experiment to validate the tool.

### 4.1 Research questions

The goal of the study is to obtain the frame rate and frames rendered information from realistic apps and analyze the performance of Android application GUIs. The following research questions are raised.

1) RQ1: Is there a strong negative correlation between the frame rate and time usage to render a frame?
2) RQ2: Does the frame rate describe the GUI running fluency in Android apps accurately?
3) RQ3: Which part consumes most resources during the process of rendering frames in GUI?

### 4.2 Experiment Design

To ensure ARFluency operate normally, the minimum version of Android system (Android 4.1) must be satisfied. Besides, according to the Android specification [28], if an app can be developed and run on Android 4.1, it can run on approximately 94.8% of active devices. Therefore, a Samsung galaxy note 10.1 device based on Android 4.1.2, API 16 is selected. To ensure the reliability of experimental data and effectiveness captured during the experiment, we create a pure, independent and secure environment on the test device. In order to reduce the impact of various factors influencing the performance of the apps (e.g. Out of Memory, Memory Leak, and other bugs), we use apps of good quality, from Google Play [15].

The experiment encompasses several tests, namely, (1) to study whether ARFluency is workable, (2) to evaluate the efficiency of ARFluency, and (3) to answer the research questions.

The above three tests are conducted almost at the same time. Five top selling apps are downloaded from Google Play, including Google Chrome, Google photos, Shadowsocks, Google translate and Qihoo Security. Besides, AnTuTu benchmark used to evaluate and measure the performance of the entire Android device is also tested.

### 4.3 Experiment Results

When the apps artoe is tested, the same operation is executed. We first install the app on the device and run ARFluency. We select the package name of the app, click "Start" button, and then play with the app for about 3minites. Last, we come back to ARFluency, click the "Stop" button, and then uninstall it. We fail to obtain the frame rate and frames rendered information of Qihoo security among the six apps. The others perform successfully. Table 1 shows the experimental data.

In table1, *issued frames* refer to frames which their rendering time more than 16-ms-per-frame, and the step consuming time most is the step processing the most time during rendering frames process.

From table 1, we can observe that issued frames exist in all apps, which can be tested by ARFluency successfully. According to the above experimental data, a line chart is shown in Fig.4. Different colors are used to distinguish different data. The red line presents the average time rendering per frame, the blue one stands the frame rate, and the green is on behalf of the percentage of issued frames.

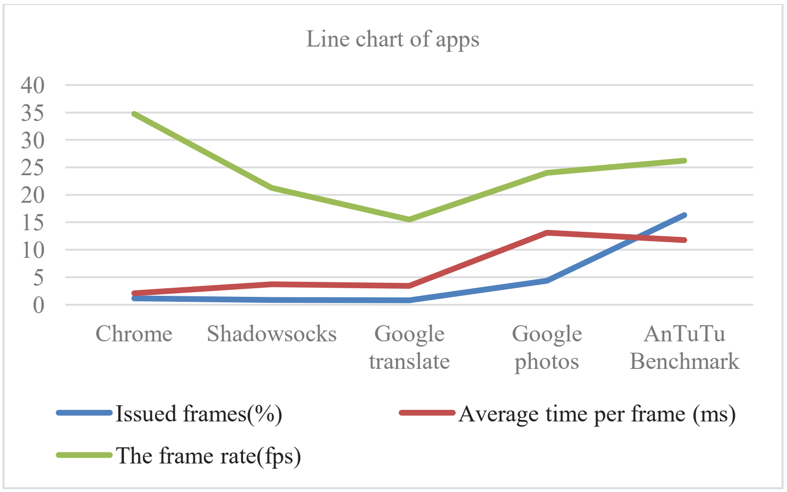
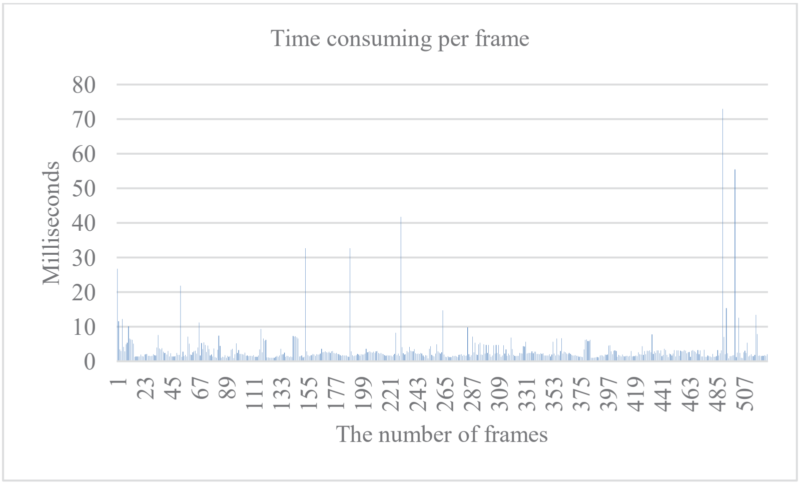
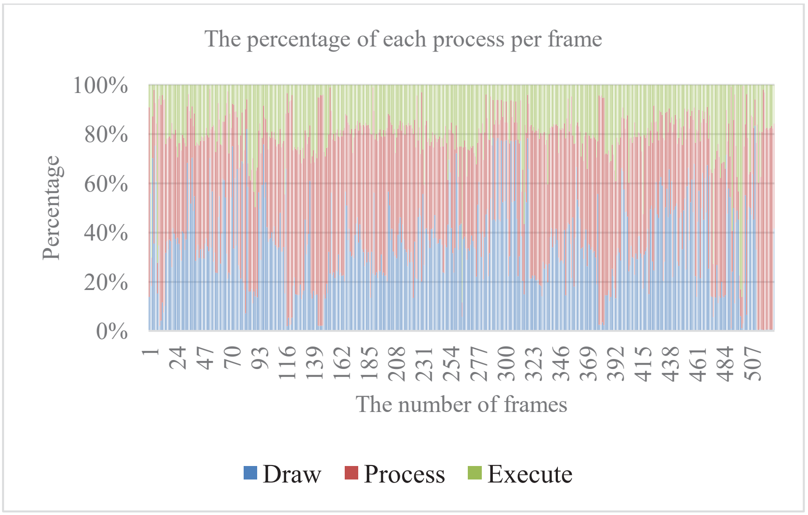


**Figure 4. Issued frames and time rendering.**

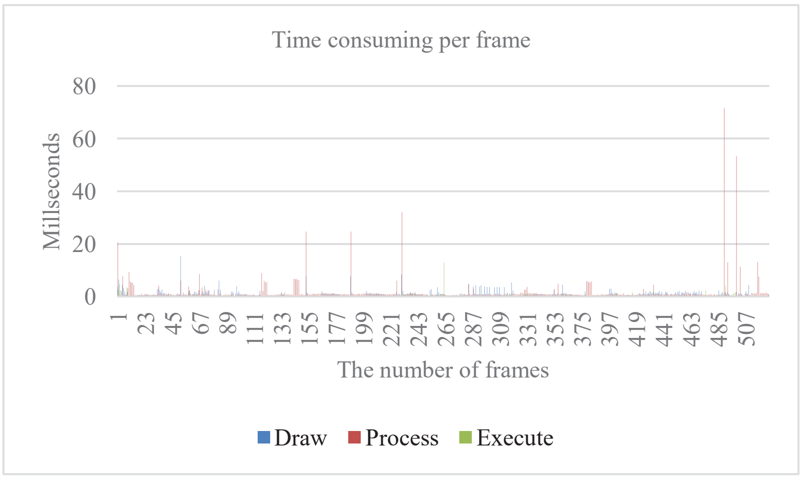


**a) Time consuming per frame**

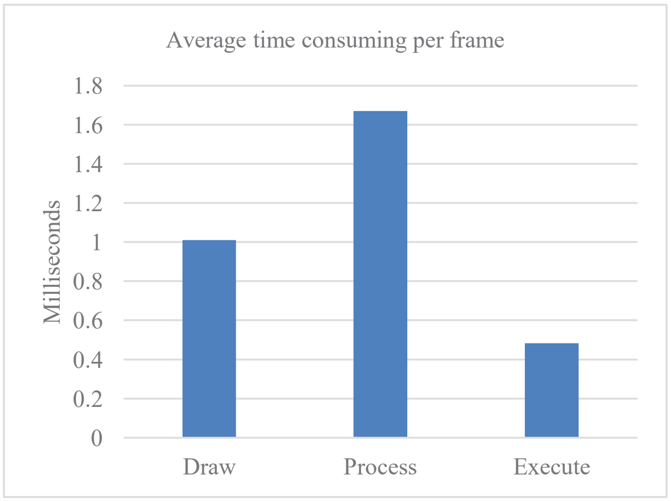


**b) The percentage of each process per frame**

**Figure 5. Information of execution per frame.**



**a) Time consuming of each step per frame**



**b) Average time consuming of each step per frame**

**Figure 6. Time consuming of each process.**

In addition, from Fig. 4, we observe that there is also no observable relationship between the frame rate and the percentage of issued frames. Therefore, the frame rate may have no strong correlation with both factors. Our observation is that though the frame rate may not measure the GUI running fluency in Android apps accurately, to some extend, it can qualitatively reflect the performance.

We can see that the Process step usually consumes the most time when rendering frames in table 1. The five results are rather similar in rendering frames. Take Chrome as an example. Information of rendering a UI window per frame is shown in Fig. 5. A majority of frames (approximately 98.86%) finish updating in less than 16.67ms, while a minimum part of issued frames operate over 20ms. Fig. 6 shows the time consuming of each process. We can see that to process is the most unstable and time-consuming step to deal with one frame on average, in about 1.67 milliseconds.

## 4.4 Answering the Research Questions

We answer the research questions as follows.

1) A1: There is **no** strong negative correlation between the frame rate and time rendering a frame.
2) A2: The frame rate does **not** describe the GUI running fluency in Android apps accurately.
3) A3: The step Process consumes most resources during the process of rendering frames in GUI.

## 4.5 Other Findings

The above five apps run fluently on the Samsung device in despite of different values of the frame rate (also Frames per Second, FPS).FPS is widely used to evaluate and measure the performance. Although it is closed related with the 16-ms-per-frame benchmark in theory, there is no strong correlation between them in the experiment. FPS is the outer reflection of running fluency from the user's perspective and the 16-ms-per-frame benchmark is also a measurement of rendering a UI window. In fact, there are various factors influencing the result. For example, different apps may belong to distinct categories, providing corresponding services, behaving different. As a result, in each category, the higher the FPS, the better the performance is.

## 5. RELATED WORK

Android phone cannot be counted as fast mobile device until Froyo was released. Froyo [5] introduced the Dalvik JIT compiler, which delivered up to 5x performance improvement in CPU-bound code. And it also brought in 2-3x improvement in JavaScript performance.

There are many studies on performance issues, many of which [1, 2, 3, 8, 9, 21, 25] focus on performance analysis. Among them, [2, 25] aim at helping developers characterize and detect performance bugs in Android apps while the work [1] pay paid attention on how developers detect and fix performance bottlenecks by surveying 485 developers. Yang et al. [3] proposed a systematic technique to discover and quantify common causes of poor responsiveness in apps. Kim et al. [8] proposed an effective scheme to reduce energy consumption without compromising user experience. Some other work [19, 22] also researched GUI automated testing. Related work such as analysis based on concolic testing [20] and multi-surface computing [18] were also carried on.

Although a plenty of work have focused on performance of Android, very few studies [3] have focus on performance evaluation and measurement of running fluency from user perspective. Qian et al. [17] put forward a new model or method in comparison with others using the frame rate as a metric, in which three programming models are been analyzed and FPS as a metric to measure the related performance.

## 6. CONCLUSION

While Android is a success on the basis of the number of available apps and market share, a plenty of performance issues do great harm to user experience.

In this paper, we focused on frame rate and made use of 16-ms-per-frame as metrics to evaluate and measure the performance. We implemented our tool ARFluency and conducted an experiment comparing it with existing tools. We found that although these apps run fluently, they do have problematic frames. Another observation is that though frame rate qualitatively reflects the running fluency, it cannot accurately measure the performance.

Future work is scheduled on using our tool in GUI optimization, reverse engineering of GUI models, and automated testing.

## 7. ACKNOWLEDGMENTS

This work was supported by the National Key Basic Research Program of China (project No. 2014CB340702) and the National Natural Science Foundation of China (project No. 61379045).

## 8. REFERENCES

[1] M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in Android apps. In *Proceedings of* ICSME, pages 352-361, 2015.

[2] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of* ICSE'14, pages 1013-1024, 2014.

[3] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In *Proceedings of* MOBS, pages 1-6, 2013.

[4] Jelly Bean. http://developer.android.com/about/versions/android-4.0.html.

[5] Android – History. www.android.com/history/.

[6] Y. Zhuang and Y. Li. Display Technique of Mobile Video Monitor on Android. Atlantis press, pages 2475-2482, 2015.

[7] M. Bulter. Android: Changing the Mobile Landscape. In *Proceedings of* IEEE Pervasive Computing, pages 4-7, 2011.

[8] D. Kim, N. Jung, and H. Cha. Content-centric display energy manager for mobile devices. In *Proceedings of* Design Automation Conference (DAC), ACM, 2014.

[9] L. Corral, A. Sillitti, and G. Succi. Mobile multiplatform development: An experiment for performance analysis. In *Proceedings of* the 9th International Conference on Mobile Web Information System (MobiWIS), Procedia Computer Science, pages 736-743, 2012.

[10] Keeping your app responsive. http://developer.android.com/training/articles/perf-anr.html.

[11] Loading large bitmaps efficiently. http://developer.android.com/training/displaying-bitmaps/load-bitmap.html.

[12] Analyzing UI performance with Systrace. http://developer.android.com/tools/debugging/systrace.html.

[13] Profiling GPU Rendering Walkthrough. http://developer.android.com/tools/performance/profile-gpu-rendering/index.html.

[14] FPS Meter. http://app.cnmo.com/android/188987/.

[15] Google Play. https://play.google.com/store.

[16] Github. https://github.com/TencentOpen/GT.

[17] X. Qian, G. Zhu, and X.-F. Li. Comparison and analysis of the three programming models in Google Android. In *Proceedings of* First Asia-Pacific Programming Languages and Compilers Workshop (APPLC), pages 1-9, 2012.

[18] A. Vant Hof, H. Jamjoom, J. Nieh, and D. Williams. Flux: multi-surface computing in Android. In *Proceedings of* EuroSys'15, pages 1-17, 2015.

[19] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of* ASE, pages 258-261, 2012.

[20] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In FSE, pages 1-11, 2012.

[21] M. Jovic and M. Hauswirth. Listener latency profiling: Measuring the perceptible performance of interactive Java applications. Science of Computer Programming, 76(11):1054-1072, 2011.

[22] X. Yuan and A.M Memon. Using GUI run-time state as a feedback to generate test cases. In *Proceedings of* ICSE, pages 396-405, 2007.

[23] D. Gavalas and D. Economou. Development platforms for mobile applications: status and trends. IEEE Software, pages 77-86, 2011.

[24] Best practices for performance. http://developer.android.com/training/best-performance.html.

[25] A. Nistor, L. Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *Proceedings of* ISSTA'14, pages 282-292, 2014.

[26] Android NDK. http://developer.android.com/tools/sdk/ndk/index.html.

[27] Y. Li, J. Fang, M. Liu, and S. Wu. Study on the application of Dalvik injection technique for the detection of malicious programs in Android. In *Proceedings of* Electronics Information and Emergency Communication (ICEIEC) Conference (Beijing, China, May 14-16, 2015), pages 309-312, 2015.

[28] Android Studio. https://developer.android.com/.