This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

Aging or Glitching? What Leads to Poor Android Responsiveness and What Can We Do About It?

Hao Lin, *Student Member, IEEE/ACM*, Cai Liu, Zhenhua Li, *Senior Member, IEEE/ACM*,
Feng Qian, *Member, IEEE/ACM*, Mingliang Li, Ping Xiong, and Yunhao Liu, *Fellow, IEEE/ACM*

Abstract—Almost all Android users have ever experienced poor responsiveness, including the common frame dropping events—slow rendering (SR) and frozen frames (FF), as well as the uncommon Application Not Responding (ANR) and System Not Responding (SNR) that directly disrupt user experience. This work takes two complementary approaches, *controlled benchmarking* and *in-the-wild crowdsourcing*, to comprehensively understand their prevalence, characteristics, and root causes, which turn out to be significantly different from common understandings and prior studies. We find that SR, FF, ANR, and SNR all occur prevalently on all the studied hardware models of Android phones, and better hardware does not seem to relieve ANR/SNR. Most surprisingly, they are oftentimes ascribed to defective software design that incurs substantial resource overuse—lightweight apps can experience severe SR/FF events due to *redundant UI rendering*, and the most ANR/SNR events stem from Android's aggressive implementation of *write amplification mitigation*. In fact, the former can be effectively overcome by simplifying the apps' UI hierarchy, and we design a practical approach to address almost all (>99%) of the latter while only decreasing 3% of the data write speed with large-scale deployment. We have released our measurement code/data to the research community.

Index Terms—Android; responsiveness; slow rendering (SR); frozen frames (FF); Application Not Responding (ANR); System Not Responding (SNR); redundant UI rendering (RUIR); write amplification mitigation (WAM).

1 Introduction

Responsiveness is a key metric that impacts smartphone user experience. Poor responsiveness would impair the productivity, satisfaction, and engagement of users. Specifically on Android, if a UI (graphic) frame takes more than 16.67 milliseconds (ms) to render, it is deemed as a *slow rendering* (SR) event; moreover, if the rendering time exceeds 700 ms, it is a *frozen frame* (FF) event [1]. Together SR and FF are also known as *frame dropping* events. Worse still, if a foreground app does not respond to user input or system broadcast for 5 seconds, or a background app does not respond to system broadcast for 10 seconds, an *Application Not Responding* (ANR) event will be triggered and a system dialog will be displayed [2]. The dialog asks users to either continue wait or kill the app, neither of which leads to pleasant user experience. Further, when a critical system thread (*e.g.*, I/O and UI) does not respond (*i.e.*, is blocked) for one minute, a restart of the system will be forced [3], which we call a *System Not Responding* (SNR) event.

Over the years, tremendous efforts have been made to optimize the responsiveness of Android at both the system's thread model and apps' programming model. For the former, dedicated render threads [4] are introduced to decouple GPU rendering tasks from common CPU tasks, while V-Sync [5] is leveraged to coordinate the GPU and CPU's execution. For the latter, Android currently requires all UI modification operations to be pushed to the main thread [6] for prioritized processing. Despite these efforts, SR, FF, ANR, and even SNR are still prevalent on Android [7], [8]. SR and FF occur frequently in daily usage, while ANR and SNR are relatively uncommon but can directly impact user experience. Unfortunately, little have we understood regarding their respective prevalence, characteristics, and root causes, due to the lack of measurement and analysis of different types of poor responsiveness. Such a lack of understandings, insights, and datasets significantly hinders practical solutions to address the problem.

- *H. Lin, Z. Li, M. Li, and Y. Liu are with Tsinghua University (e-mail: {linhaomails, lizhenhua1983, limingliang0527, yunhaoliu}@gmail.com).*
- *C. Liu and P. Xiong is with Xiaomi Technology Co. LTD, China ({liucai, xiongping1}@xiaomi.com).*
- *F. Qian is with University of Minnesota, MN, USA (fengqian@umn.edu).*

Study Methodology. Conducting a comprehensive and in-depth study on the poor responsiveness events is challenging. First, capturing fine-grained system status is crucial to root cause analysis, but is not sufficiently supported by existing mobile systems (as will be discussed in §2). Second, the remarkable distinctions between SR/FF and ANR/SNR (in terms of occurrence frequencies and the underlying mechanisms) require dedicated and considerate approaches to effectively collect data for all types of events.

To address the challenges, we devise a continuous system tracing framework for collecting detailed in-situ system-level data during poor responsiveness events, which combines the monitoring of common system status indicators (including CPU usage, memory consumption and I/O activity) with the instrumentation of critical system services to complete the puzzle. We modify existing Android tools and/or systems to realize this framework.

Specifically, for SR and FF, we develop a lightweight (in terms of computation) kernel tracing tool by customizing `atrace` [9], the debugging tool of Android, so as to efficiently trace critical system functions concerning frame rendering in real time in a *non-intrusive* manner (*i.e.*, no system modification but only toolchain customization). Un-

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

fortunately, for ANR and SNR, similar debugging tools and the built-in monitoring facilities of Android cannot provide sufficient diagnostic information regarding several important system services even with root privileges. We thus have to customize the vanilla Android Framework layer to record these system services' end-to-end call stacks.

Next, we design complementary approaches to measure the poor responsiveness of Android smartphones, involving 15 hardware models equipped with different Android versions. For SR and FF, our tracing framework incurs trivial computation/memory overhead, but requires debug (`adb`) privilege and sometimes nontrivial network traffic cost, therefore making large-scale measurements hard to conduct. We thus resort to *controlled benchmarking* by synthetically generating representative workloads. We automate a series of popular apps on the 15 experimental smartphones, and collect fine-grained data regarding SR/FF in the meantime.

On the other hand, ANR and SNR are not often observed in a common smartphone's daily usage, so small-scale measurements can easily lead to biased or even incorrect results. Fortunately, in collaboration with a major Android phone vendor called Xiaomi, we obtain a large-scale in-the-wild measurement opportunity for ANR/SNR. We invited the active users in Xiaomi's smartphone community to participate in our measurement study; 30,000+ users opted in and collected data for us for three weeks, involving 15 different models of Android phones. All data are collected with informed consent of opt-in users, and no personally identifiable information was collected.

Prevalence and Characteristics. Our measurements reveal that all types of poor responsiveness occur prevalently on every one of our studied hardware models. In particular, as many as 4.9%–18% of the rendered frames are subject to SR or FF under typical workloads. On average, 1.5 ANR events and 0.04 SNR events occur on an Android system during the 3-week measurement, and the maximum number of ANR (SNR) events reaches 37 (18) on an Android system.

In detail, the time interval of consecutive SR/FF events is $\sim$0.5 seconds, indicating that such events have considerable temporal locality; on the contrary, ANR and SNR are highly correlated in terms of occurrence probability but weakly correlated in terms of occurrence time (*i.e.,* an SNR event is usually *not* caused by an ANR event, and vice versa). While better hardware significantly reduces SR and FF, it surprisingly does not seem to relieve ANR/SNR—among the 15 hardware models, the six oldest and the six latest experience almost the same number of ANR events per phone; the six oldest models experience even 50% fewer SNR events per phone than the six latest models. In addition, as Android evolves from version 7.0 to 9.0 where considerable performance optimizations have been added, there are 74% fewer ANR events but 33% more SNR events.

Root Cause Analysis. To uncover the root causes of poor responsiveness, we develop automatic analysis pipelines for different types of event data. For SR and FF, we analyze the hierarchical tracing data logged by us to obtain the critical function call path of each rendering stage for a UI frame and its correlation with essential application and system events. We locate the most time-consuming stage in most SR/FF events as the measure/layout step, where the major workload lies in the calculation of UI components' locations and sizes. Most surprisingly, we observe that some seemingly lightweight apps with fairly simple UI functionalities, such as Gmail, can experience even more severe SR/FF events compared to video streaming and gaming apps. Through careful examination of our collected traces, we attribute this counter-intuitive phenomenon to the *redundant UI rendering* (RUIR) [10] problem of such apps, which stems from a subtle defect of Android's *painter's algorithm* [11] that renders UI components in a bottom-up manner.

For ANR and SNR, our pipeline processes the crowd-sourced logs by first extracting the blocked threads, and then generating their *wait-for graph* [12] to figure out the critical thread that leads to ANR/SNR. Based on this, we classify each ANR/SNR event into a root-cause cluster using *similar-stack analysis* [13], and manually analyze the root cause of unbiased samples in each dominant cluster. The correctness of our analysis is validated using a different set of unbiased samples. Eventually, we discover four major root causes of ANR/SNR, among which the largest one comes from Android's aggressive implementation of *write amplification mitigation* (WAM) [14], an I/O mechanism which was supposed to improve the user experience.

Mitigation Practice. Although there is no silver bullet for all the bugs and defects in Android software design, we notice that the critical root causes for SR/FF (*i.e.,* RUIR) and ANR/SNR (*i.e.,* WAM), can both be effectively addressed.

For RUIR, our suggestion is leveraging the *dynamic layout inspector* tool offered by Android SDK [15] to examine an app's UI hierarchy. Developers can then easily locate and remove overlapped UI components, redundant backgrounds, and problematic alpha settings to simplify the UI hierarchy. To demonstrate the practical efficacy, our optimizations on popular apps with RUIR problems have already yielded promising results, reducing SR/FF by an average of 27%.

As for Android's aggressive WAM strategy, a straightforward fix is to *batch* WAM. However, Android's batched WAM implementation is rather ineffective. First, its lazy nature (at most once a day) cannot mitigate write amplification in time. Second, it cannot be interrupted once started, leading to heavy I/O. Third, if killed by users, the process will restart from the head. To address the issues, we design a *practical* WAM by making batched WAM *fine-grained* and *non-intrusive*. It records the deleted data amount (S_d), and uses a data-driven approach to decide a proper threshold to trigger batched WAM on demand, achieving both timely mitigation while amortizing the cost. We also make our batched WAM interruptible and resumable. After rolling out our design on part of the 30,000+ opt-in users' phones, it reduces almost all (>99%) ANR/SNR events caused by WAM. Our design has been further adopted by five stock Android systems since May 2019, benefiting $\sim$20M users.

Contribution & Data/Code Availability. The above efforts measure and tackle complementary aspects of Android responsiveness, including the frequent SR/FF events and the disruptive ANR/SNR events, thereby forming a holistic landscape of Android responsiveness problems and their practical solutions. Our data and code are released in part at https://Android-Poor-Respond.github.io with detailed guides to benefit the community.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

2 METHODOLOGY

We conduct complementary controlled benchmarking and large-scale in-the-wild measurements on all types of poor responsiveness events to comprehensively understand the problems. This is enabled by our continuous system tracing framework for collecting find-grained system-level data, and our automatic pipelines for root cause analysis.

2.1 Continuous System Tracing Framework

To help app and system developers address the poor responsiveness problems, Android provides several debugging tools and built-in facilities to record diagnostic information regarding SR, FF, ANR and SNR events, which, however, are insufficient to uncover the root causes of the problems. Specifically, for SR and FF that manifest as rendering performance issues, Android only monitors and reports the events' occurrences (which can be acquired through the `dumpsys` system utility) as such events occur frequently in practice and thus traditional debugging method of call stack logging could incur considerable time overhead.

On the other hand, for ANR and SNR which are both response timeout events happening to an app process or a system thread, Android further records a series of additional diagnostic information including call stack of the target app process (only for ANR), call stacks of a predefined set of system service processes such as `SystemServer` and `MediaServer`, and the blocked threads. Unfortunately, we find that the above information is still insufficient for root cause analysis in practice due to missing the call stacks of several important system service processes, such as the `Vold` service (Android's storage volume daemon). This is because we constantly observe that the target app processes interact with these system services and we intend to obtain the visibility into those services that are not included in Android's diagnostic information.

To address the challenges, we modify existing Android tools and/or systems to build a continuous system tracing framework for efficiently collecting detailed system-level data during poor responsiveness events. To this end, our framework pieces together the in-situ system panorama by combining the monitoring of common system status indicators (including CPU/memory usages and I/O activity) with the instrumentation of critical system services, which are carefully selected to collect concerned information while avoiding excessive computation and memory overhead.

Tracing Framework for SR/FF. To select key system service instrumentation points for SR/FF, our insight is that the function call paths of frame rendering in Android mostly follow a fixed pattern, because almost all the apps' rendering tasks are realized in a four-stage fashion—1) *measuring* that calculates each UI component's size (*e.g.,* height and width), 2) *layout* that decides the relative positions of different UI components, 3) *drawing* that renders the UI components on a canvas based on their measured sizes and layout, and 4) *composition* that merges the app's canvas with those of other processes (*e.g.,* the system status bar and navigation buttons) to produce the final display for users. In practice, the first two stages are accomplished by the app's main thread, the drawing stage is done by a dedicated render thread, while the composition stage is realized in a system service called `SurfaceFlinger`, which manages all the other processes' rendering canvases.

With this insight, we propose to selectively instrument (a total of 28) concerned system functions involved in the above four rendering stages and trace their calls throughout the lifecycle of a target app. This enables us to efficiently collect fine-grained system-level data regarding SR and FF. To realize this, we customize the `atrace` utility of Android to implement the instrumentation of the 28 concerned system functions in a non-intrusive manner, which does not require root privileges or system modifications, but only debug privileges accessible to common app developers. During an app's running, our customized `atrace` will record the begin and end timestamps of the instrumented functions in a kernel ring buffer. We also activate other critical trace points already provided by `atrace`, including those of Binder transaction, I/O event, and CPU scheduling to facilitate analyzing the problems.

We implement this tracing framework as a debugging tool running on common PCs. When collecting data for SR/FF on a phone, the tool would load and initiate our customized `atrace` into the phone through `adb` commands (since the tracing requires debug privilege), and then pull data from the kernel ring buffer to the PC it runs on through network connections between the phone and the PC. The data pulling is necessary because the `atrace` ring buffer can only hold 15-second tracing data in practice. Consequently, while our benchmark toolchain incurs only ∼1% CPU overhead and ∼10 MB memory overhead for a common Android device, the network traffic is nontrivial—around 40 MB per minute.

Tracing Framework for ANR/SNR. As discussed above, for ANR and SNR, we are interested in the call stacks of several critical system services which frequently interact with apps in practice. Unfortunately, tracing tools similar to `atrace` and the built-in monitoring facilities of Android cannot provide such diagnostic information even with root privileges. As a consequence, we are unable to build our tracing framework without modifying the Android Framework layer. Therefore, we develop a customized Android system, called Android-MOD, to collect additional information essential for our analysis by modifying the code of vanilla Android versions 7.0, 8.0 and 9.0.

Our data collection requires an Android device to install (or upgrade to) Android-MOD. However, once it is installed, our data collection is lightweight and incurs negligible runtime overhead. Note that our modifications only include logging additional lightweight system-level information and the logging is triggered only upon the occurrences of ANR and SNR events. Eventually, we observe only KB-level overhead for storage and negligible overhead for CPU and memory, compared to Android's original mechanism.

2.2 Complementary Measurements

With the devised system tracing framework, we next design complementary approaches to measure the poor responsiveness of Android smartphones, so as to accommodate the distinctions between SR/FF and ANR/SNR in terms of occurrence frequencies and the underlying mechanisms.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

Our measurements involve 15 phone models with different hardware and software configurations as listed in Table 1 to collect in-depth data. The phone models cover low-end (*i.e.*, the 5 models equipped with the SDM 450, SDM 625 and SDM 636 CPUs), middle-end (*i.e.*, the 5 models equipped with the SDM 660 CPU) and high-end models (*i.e.*, the 5 models equipped with the SDM 835 and SDM 845 CPUs) of a major Android phone vendor, Xiaomi, with which we collaborate to conduct the study (as to be detailed soon). Note that although our studied models are from a single vendor (Xiaomi), we believe our findings are also applicable to other vendors' Android systems. This is because different vendors (including Xiaomi) typically adopt the same set of core Android components [16]–[19]. Also, vendors' system customizations are required to pass Google's CTS tests [20] to ensure that they have consistent functionalities and thus do not break compatibilities with existing apps.

Controlled Benchmarking for SR/FF. As introduced above, for SR and FF, our tracing framework incurs trivial computation/memory overhead, but requires debug (`adb`) privilege and sometimes nontrivial network traffic cost, therefore making large-scale measurements hard to conduct. We thus resort to *controlled benchmarking*, which synthetically generates representative workloads on devices and monitors SR and FF occurrences in the meantime, while leveraging the system tracing framework to efficiently capture in-situ traces of the concerned function calls involved in the four-stage rendering during the benchmark process for subsequent analysis.

In more detail, we use the top-10 most downloaded apps from Google Play as of Nov. 21st in 2021 for benchmarking, covering seven major app categories. Table 2 lists the 10 apps and their corresponding app categories. To synthesize the benchmark workloads, we define a series of interactions with each app based on its functionalities and UI layouts, which are extracted by traversing the `Activities` of the app using *UI Automator* [21], the UI testing framework of Android. The workloads are defined on a per `Activity` basis, as each `Activity` contains a different UI layout.

Specifically, for social, email and messaging apps, we find that their main UI layouts are typically composed of a scrollable list (*i.e.*, the message list) and several clickable items (*i.e.*, message items). We thus define the corresponding workloads as scrolling the list and clicking the items to mimic users' viewing and checking the messages. For music and video apps, we define the workloads mainly as viewing and playing the multimedia contents which are the apps' primary functions. For web browsers, we access Alexa Top-10 websites [22] and scroll to view them. In particular, for a scrollable UI component, we would leverage the `flingToEnd` API of *UI Automator* to scroll to the component's end at the default rate, which performs a center-to-top (or center-to-left/right depending on the scrollable directions) swipe in 25 milliseconds for each action and repeats the action until the component is scrolled to the end. Also, we would wait for the previous action to finish and the UI to be idle (which can be achieved through the `waitForIdle` API) before initiating the next action. For gaming apps, however, their UI components are usually not traditional Android components and thus cannot be identified by *UI Automator*. For them, we set the workloads as a series of actions to finish a game set by manually identifying the components. The detailed actions and setups of our benchmark workloads for each app can be found at https://Android-Poor-Respond.github.io.

Table 1: Hardware and OS configurations of our measured phone models, ordered by hardware configurations. In particular, all the models' CPUs are octa-core Qualcomm Snapdragon Mobile (SDM) CPUs.

Model	CPU	Memory	Storage	Android
1. Redmi 5	SDM 450@1.8 GHz	3 GB	32 GB	7.0
2. Mi A1	SDM 625@2.0 GHz	4 GB	64 GB	7.0
3. Mi Max 2	SDM 625@2.0 GHz	4 GB	64 GB	7.0
4. Mi Max 3	SDM 636@1.8 GHz	6 GB	64 GB	9.0
5. Redmi Note 5 Pro	SDM 636@1.8 GHz	6 GB	64 GB	7.0
6. Mi A2	SDM 660@2.2 GHz	6 GB	64 GB	8.0
7. Mi 8 Lite	SDM 660@2.2 GHz	6 GB	64 GB	9.0
8. Redmi Note 7	SDM 660@2.2 GHz	6 GB	64 GB	7.0
9. Redmi Pro 2	SDM 660@2.2 GHz	6 GB	64 GB	8.0
10. Mi Note 3	SDM 660@2.2 GHz	6 GB	64 GB	7.0
11. Mi 6	SDM 835@2.3 GHz	6 GB	64 GB	8.0
12. Mi Mix 2S	SDM 845@2.8 GHz	6 GB	128 GB	8.0
13. Mi 8 Pro	SDM 845@2.8 GHz	8 GB	128 GB	9.0
14. Mi Mix 3	SDM 845@2.8 GHz	8 GB	128 GB	9.0
15. Black Shark	SDM 845@2.8 GHz	8 GB	128 GB	9.0

Table 2: All 10 apps used for benchmark ordered by the occurrence rate of SR/FF events.

Application	SR/FF Occurrence Rate	Category
Facebook	18.65%	Social
Gmail	16.02%	Email
Subway Surf	15.28%	Gaming
Clash of Clans	13.34%	Gaming
Chrome	12.08%	Web Browser
Instagram	11.96%	Social
Spotify	11.66%	Music
YouTube	9.65%	Video
Messenger	6.76%	Instant Messaging
WhatsApp	3.44%	Instant Messaging

To run the benchmarks, we execute the workloads on Android phones through *UI Automator* [21]; typically, each app will be run for one minute, which is sufficient for us to cover all the interactable `Activities` of an app in practice. We then run the controlled benchmarking on the 15 studied Android phones to collect data.

Large-Scale Measurement for ANR/SNR. On the other hand, ANR and SNR are not often observed in a common smartphone's daily usage, so small-scale measurements can easily lead to biased or even incorrect results. Fortunately, in collaboration with a major Android phone vendor called Xiaomi, we obtain a large-scale in-the-wild measurement opportunity for ANR/SNR. In Oct. 2018, we invited the active users in Xiaomi's smartphone community through email to participate in our measurement study by upgrading to Android-MOD, our customized Android system that realizes the continuous system tracing framework for ANR/SNR, on their phones. Eventually, more than 30,000 users opted in. We explicitly informed the opt-in users that Android-MOD is a lightweight update that will not affect their apps, data, OS version, or system performance. The recorded ANR/SNR data were uploaded to our data server when there is WiFi connectivity. The measurement lasted for three weeks from Nov. 1st to Nov. 21st in 2018.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

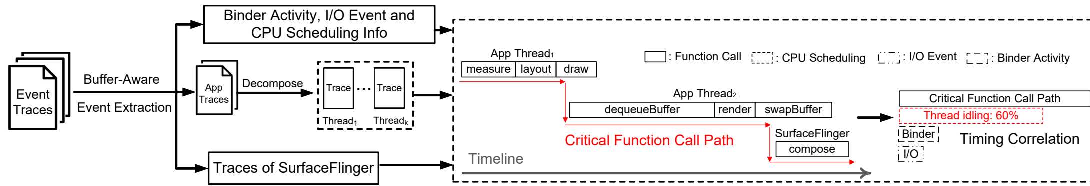


Figure 1: Workflow of our automatic pipeline for analyzing the root causes of SR and FF events.

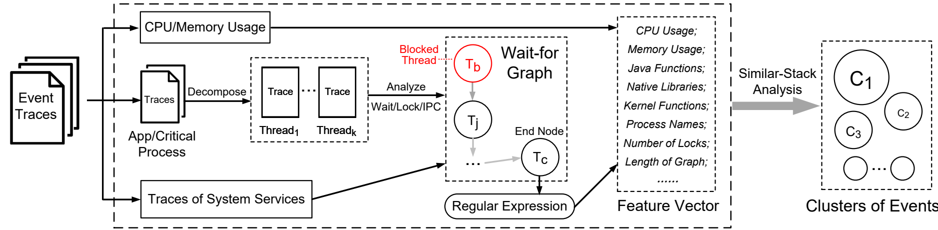


Figure 2: Workflow of our automatic pipeline for analyzing the root causes of ANR and SNR events.

2.3 Root Cause Analysis Pipelines

To figure out the root cause of a single poor responsiveness event, app or system developers usually analyze its corresponding log by hand. However, such manual analysis does not scale. Therefore, we devise automatic analysis pipelines for extracting root causes from the collected tracing data.

Analysis Pipeline for SR/FF. For SR and FF, our pipeline is based on our devised *buffer-aware hierarchical timing correlation* root cause analysis method, which first extracts true SR/FF events from the data by paying special attention to Android's frame buffering mechanism to rule out false positives, and then exploits the hierarchical nature of the kernel tracing data to locate the critical rendering function call path and its highly correlated system/app events based on their consumed time. Figure 1 shows the basic workflow of our automated pipeline.

Specifically, to extract SR/FF events from the tracing data collected in §2.2, we can calculate the time consumed for rendering a frame using the timestamps of the functions called during the frame's different rendering stages. Generally, if a frame takes more than 16.67 (700) milliseconds to render, we know that an SR (FF) event has occurred based on Android's definition [1]. However, we find that this simple calculation suggested by Android introduces many false positives in practice.

Delving deep, we find that such false positives stem from Android's *triple buffering* mechanism. Recall that in the composition stage of frame rendering, an app would send its canvas (which is a memory buffer) to the `SurfaceFlinger` service to composite the final display. Meanwhile, `SurfaceFlinger` should give back its previously composited canvas buffer to the app so that the app can draw the next frame on it, *i.e.,* the app and `SurfaceFlinger` "swap" their buffers. To reduce the app's waiting `SurfaceFlinger` in the swapping process, Android introduces triple buffering, where the app holds two canvas buffers (denoted as $Buffer_1$ and $Buffer_2$) and `SurfaceFlinger` holds one buffer. In this way, even if `SurfaceFlinger` is busy dealing with its buffer and cannot swap it with $Buffer_1$, the app can directly draw on $Buffer_2$ without waiting for swapping, therefore improving the responsiveness of Android.

When that happens, $Buffer_2$ becomes a "redundant" canvas buffer. As a result, even if the next frame takes more than 16.67 milliseconds to render, $Buffer_2$ can be swapped into `SurfaceFlinger`, thus avoiding SR/FF. Unfortunately, simple frame time calculation would still determine this case as SR/FF since the frame rendering time is long, leading to false positives. To cope with this, we take the number of ready buffers of the app into account (which can be known from its Binder queue)—we determine a frame with long rendering time as a true SR/FF event only if there is zero ready buffer.

Upon detecting an SR/FF event, we further attempt to pinpoint its root causes based on the fine-grained system tracing data we collect. To this end, we find that the tracing data bear inherent hierarchy, *i.e.,* the calling relations between functions—if a function F_A's begin/end time wraps that of another function F_B, we know that F_A calls F_B. Given this, we can extract the most time-consuming rendering stage based on different stages' execution time. However, in practice we notice that the rendering of a frame is usually parallelly performed in multiple threads or even processes (as discussed in §2.1) for optimal performance. Therefore, even if a rendering stage consumes a long time, it may not be on the *critical function call path* that decides the final rendering time. As exemplified in Figure 1, we thus further extract the critical path by identifying the longest call path in relevant rendering threads' tracing data.

After uncovering the critical function call path, we next correlate it with the in-situ Binder transaction, I/O event and CPU scheduling information we collect to pinpoint the actual causes in the system. Specifically, for Binder transaction and I/O event, we calculate the correlation value as the proportion of time used to accomplish them in the critical path. For CPU scheduling, we use the proportion of thread idling, waiting or blocking time as the correlation value. With these correlation values, we compare them with their corresponding average correlation values during normal frames to identify abnormality. For example, as shown in Figure 1, the thread idling time takes up 70% of the time

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

spent in the critical call path, which is significantly larger than the average value in normal frames (around 3%). Our analysis pipeline would then highlight the issue as the most probable root cause. If multiple issues are identified, they are all reported to facilitate analysis. Also, we pay special attention to the core scheduling of typical ARM CPUs with the `big.LITTLE` architecture [23], which couples power-efficient (`LITTLE`) cores with performant (`big`) cores to strike a balance between power efficiency and performance. If the app's main or rendering thread is scheduled to the `LITTLE` cores (the real-time scheduling information is recorded by the kernel and captured in our data) and the cores are fully utilized during rendering (*i.e.,* 100% CPU utilization), it is highly likely that its long rendering time is caused by the worse performance of the `LITTLE` cores.

However, if the critical call path's correlations with these crucial system factors are trivial (*i.e.,* less than the average correlations during normal frames), it is more likely that the root causes lie in the apps' own designs (*e.g.,* complicate UI). For them, we locate the corresponding UI components with long rendering time to facilitate the root cause analysis.

Analysis Pipeline for ANR/SNR. For ANR/SNR, we develope the pipeline based on the observation that ANR/SNR events with the same root cause tend to have similar symptoms in terms of call stack patterns and lock contention status. Recall that, for an ANR event, we collect call stacks of the app process and system service processes, as well as the blocked threads of the recorded processes. As shown in Figure 2, we first decompose the call stacks of the app process into several ones corresponding to each thread of the process. Note that among the multiple threads of the app process, there is only one *blocked thread* that is recorded as `Blocked` by Android. Nevertheless, this blocked thread (T_b) may not be the *critical thread* (T_c) that is expected to be the most relevant to the root cause of the ANR event, because the blocking of T_b might be in fact caused by other threads of the process or even threads of system services due to inter-process communication (IPC).

To identify T_c, we construct a *wait-for graph* [12] for the app's process, based on the wait, lock, and IPC information we recognize in each thread's call stack, as shown in Figure 2. In the wait-for graph, a node stands for a thread and an edge going from thread T_i to T_j indicates that T_i is currently blocked by T_j. Thus, we can trace from T_b until we find the last thread [1] that has no successor, which is T_c.

Having found the critical thread T_c, we remove irrelevant information (*e.g.,* line number, memory address, and thread ID) from the call stacks using regular expressions [2]. The regular expressions are diverse in terms of their lengths and complexities, *e.g.,* some are as simple as numbers while others involve more complex patterns. We also determine the appropriate order of applying them to avoid false removals. The remainder of the call stacks, which contains considerable "feature" information, is then reorganized into a feature vector. As depicted in Figure 2, a typical feature vector mainly consists of eight components that represent CPU usage, memory usage, Java functions, native libraries, kernel functions, process names, the number of locks, and the length of the wait-for graph.

Based on the above processing, we can classify an ANR event into the corresponding *root-cause cluster* using *similar-stack analysis* [13]. If the feature vector (V_i) of an ANR event i is similar to that (V_j) of another ANR event j , i and j will be classified into the same root-cause cluster. When measuring the similarity between V_i and V_j, instead of directly applying off-the-shelf similarity metrics, we customize the similarity metric by taking into account the high heterogeneity across the features' semantics, formats, and generality. Specifically, we take the following "split-and-merge" approach: we first separate all the features of each vector V into two feature sets: F_p and F_c given their heterogeneity; we then calculate the similarity values for F_p and F_c separately (denoted as $S_p(i,j)$ and $S_c(i,j)$ respectively between V_i and V_j); finally, we combine them to the overall similarity denoted as $S(i,j)$.

In our design, F_p contains CPU usage, memory consumption, the instruction set, the app fatal signal, and the app failure code, *etc.* These features tend to be "generic" in that similar measures may also be observed during the course of normal OS/app operations. To avoid over-fitting, we compute $S_p(i,j)$ using the *Jaccard Index* [24], a simple metric that measures the set similarity:

$$S_p(i,j) = J(F_{p,i}, F_{p,j}) = \frac{|F_{p,i} \cap F_{p,j}|}{|F_{p,i}| + |F_{p,j}| - |F_{p,i} \cap F_{p,j}|}, \quad (1)$$

where $J(...)$ is the Jaccard Index function. In contrast, F_c contains Java functions, native libraries, kernel functions, the number of locks, the length of the wait-for graph and process names, *etc.* that are more specific to ANR/SNR events compared to F_p. We therefore calculate $S_c(i,j)$ using the *term vector space model* [25] and *cosine similarity* [26], which provide fine-grained, dimension-by-dimension comparison between two feature vectors:

$$S_c(i,j) = cos\langle \boldsymbol{F_{c,i}}, \boldsymbol{F_{c,j}} \rangle = \frac{\boldsymbol{F_{c,i}} \cdot \boldsymbol{F_{c,j}}}{\|\boldsymbol{F_{c,i}}\| \, \|\boldsymbol{F_{c,j}}\|}, \quad (2)$$

The final similarity $S(i,j)$ is derived as the weighted average between $S_p(i,j)$ and $S_c(i,j)$ where the weights are the respective cardinalities of the set F_p and F_c. V_i and V_j will be classified into the same root-cause cluster if $S(i,j)$ is above a threshold, which is empirically set to 0.95 based on our inspection of representative ANR samples.

The similar-stack analysis can generate thousands of root-cause clusters. However, we observe there are only several *dominant clusters* that include the majority of ANR events. We manually analyze the dominant clusters to validate our analysis pipeline. Specifically, for each cluster, we first manually analyze the traces of the K (empirically set to 100) samples *nearest* to the cluster centroid to find out their root cause(s). In practice, we notice that usually the vast majority of the samples share exactly the same call stack due to our high similarity threshold (0.95), whose root cause is then most likely the cluster's root cause. We thus first analyze their root cause mainly by examining their critical threads' related system components, functionalities, and in-situ system status based on our experiences and domain knowledge. For example, when the samples' call stacks indicate that the critical thread experiences timeouts during Java

1. In a very small portion (<1%) of cases, *e.g.,* when a cycle is detected in the wait-for graph, we can find multiple critical threads for an ANR event. Then, each critical thread will be processed separately and the ANR event can simultaneously belong to multiple root-cause clusters.

2. The full list is at https://Android-Poor-Respond.github.io

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

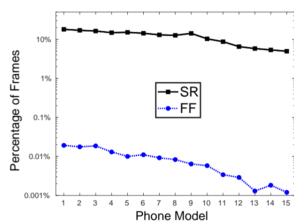


Figure 3: Occurrence rates of SR/FF per phone model.

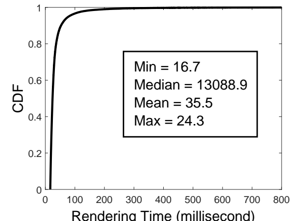


Figure 4: Frame rendering time of SR and FF events.

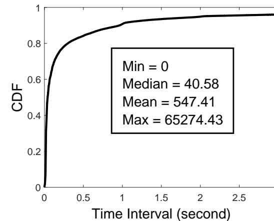


Figure 5: Time interval for consecutive SR/FF→SR/FF.

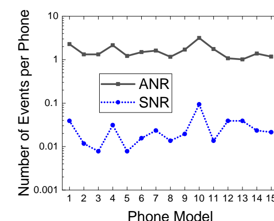


Figure 6: Avg. ANR/SNR event number per model.

VM's (related system component) garbage collection (functionality) when the available memory is low (in-situ system status), we attribute the root cause to insufficient memory. The above root cause analysis results is also validated by Xiaomi's internal testing procedure, which involves the independent manual examination of the corresponding traces by 3∼5 system experts from the OS development team of Xiaomi, so as to ensure that subsequent problem fixings are not affected by false positives. Next, for the other event samples with different yet similar call stacks in the same cluster, we also analyze their root causes through the above process and check whether they are consistent with the extracted root cause; if not, we determine that the events are falsely classified.

We then also apply the above analysis to the K samples *furthest* from the centroid, comparing their root cause(s) with those nearest to the centroid to check whether they are still consistent. The examination result shows that all the inspected cases are perfectly categorized with no false positives. This is mainly because our high similarity threshold (0.95) is not easily biased by high similarity in subdimensions of the call stacks, *e.g.*, high similarity in Java functions yet low similarity in native libraries, which usually implies different event root causes in practice according to our experiences.

For an SNR event, our collected log contains the call stacks of multiple system service processes, where only one is flagged by Android as the *critical process* that leads to SNR. Then, we figure out the critical thread from this process in a similar way as in the case of ANR; the subsequent processing and classification are similar to those of ANR.

3 Measurement Results

Based on our measurement and automatic analysis, we have multifold findings on Android poor responsiveness in terms of its prevalence and characteristics, as well as in-depth understandings of their root causes.

Prevalence of Poor Responsiveness. Our measurement reveals that all types of poor responsiveness occur prevalently on all the 15 studied phone models. As shown in Figure 3, as many as 4.9%–18% of the rendered frames during benchmark experiments are subject to SR and FF. Further, as depicted in Figure 4, among the captured SR and FF events, we notice that >99% of the frames are SR events with 70% of them having less than 32 ms frame rendering time, which translates to >30 frames rendered per second and thus are usually unnoticeable in practice. However, the <1% FF events can take up to several seconds to render the frames, which have severe impacts on user experiences.

Similar skewed distributions can also be observed for ANR/SNR. As shown in Figure 6, an average of 1.5 ANR events and 0.04 SNR events occur on an Android phone during the three-week measurement. However, for ANRs, around a half (51%) of phones do not experience ANR, while the maximum number of ANR events occurred on an Android phone is 37. For SNRs, most (97%) phones do not experience SNR, while the maximum number of SNR events occurred on one phone is 18. On average, 29% devices encountered at least an ANR or SNR event every ten days.

Correlations between Events. Although SR events are significantly more frequent (over 1000×) than FF events, we notice that their occurrences are in fact highly correlated as shown in Figure 3 across different phone models. The *sample correlation coefficient* [27] between their occurrence is as high as 0.93. Similarly for ANR and SNR events, the sample correlation coefficient between their occurrences can reach 0.73, which also suggests that they are highly relevant.

To understand the high correlations between events, we examine the time interval between two neighboring SR/FF events. As depicted in Figure 5, we discover that the median time interval is 40.58 ms, indicating that the occurrences of SR/FF events tend to be temporally localized. We further analyze the time between an FF event and its most recently preceding SR event ("FF→SR"), and find that the median time interval is 653.98 ms, which suggests that when an FF event occurs, it is highly likely that SR events will follow. In contrast, an SR event is usually not followed by FF events. Closer examination reveals that these patterns stem from SR and FF events' relations with system resource provisioning. We find that when FF events occur, the resource consumption of the system is extremely high (*e.g.*, the CPU utilization is around 100%), therefore can easily trigger SR events as well. On the other hand, during SR events the CPU utilization is ∼60%, which is higher than the average level but may not lead to FF events.

On the exact contrary, we find that the median time interval between every SNR event and its most recently preceding ANR event ("ANR→SNR") is as long as 0.95 day and the average is 2.19 days, as shown in Figure 7. Therefore, an SNR event is usually *not* caused by an ANR event. Additionally, we examine the time interval between every ANR event and its preceding SNR event ("SNR→ANR"), and find that ANR is *not* caused by SNR, either (Figure 7). The high probability correlation and weak time correlation suggest that ANR and SNR tend to be caused at the system level. There is no causality between ANR and SNR events.

Hardware Configurations. As affected by vendors' propaganda of "better hardware helps improve software respon-

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

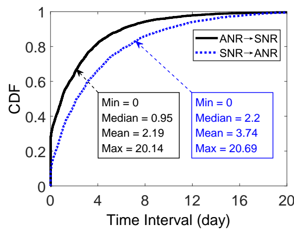


Figure 7: Time interval for consecutive ANR→SNR and SNR→ANR.

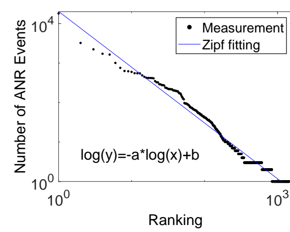


Figure 8: Ranking of apps by ANR event numbers. Here $a = 1.41$ and $b = 4.31$.

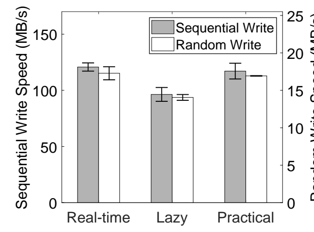


Figure 9: Random and sequential write speeds of different WAM mechanisms.

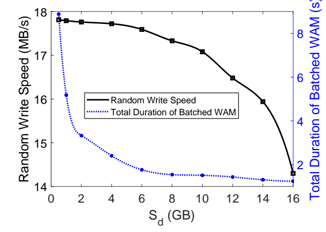


Figure 10: Duration of batched WAM and random write speed for different S_d.

siveness" [28]–[30], non-professional users might intuitively believe that a phone with more advanced hardware experiences fewer poor responsiveness events. Our measurement finds that this may be true for SR and FF events. As shown in Figure 3, with better hardware, a tested device is much more likely to encounter fewer SR and FF events during the measurement, which is due to their close relations with system resource provisioning as discussed above. Surprisingly perhaps, we can see from Figure 6 that this is not the case for ANR and SNR—hardware configurations have no correlations with the prevalence of ANR. Specifically, among the 15 models of phones we study, the six oldest models (Model 1–6, released between Dec. 2017 and Apr. 2018) and the six latest models (Model 10–15, released between May. 2018 and Oct. 2018) experience almost the same number of ANR events per phone (when the Android versions are the same). Detailed hardware configurations of the 15 phone models can be found in Table 1. Further, we notice that better hardware even appears to aggravate SNR—the six oldest models experience 50% fewer SNR events than the six latest models per phone. The above results clearly illustrate that ANR and SNR are *not* a hardware issue.

Android Versions. As Android evolves, considerable performance optimizations have been added to the Android framework and the OS kernel [31], [32]. In particular, we are interested in the occurrences of ANR and SNR events across different Android versions, which are less affected by hardware configurations as compared to SR and FF. Naturally, we expect ANRs and SNRs in recent Android versions to be substantially reduced. Compared with Android 7.0, there are 74% fewer ANR events but 33% more SNR events happening on Android 9.0 (per phone). This indicates that the performance optimizations have taken effect in improving the responsiveness of apps.

However, we find that the system-level responsiveness (*i.e.,* the situation of SNR) gets worse, probably because the more recent Android 9.0 (released in Sep. 2019) is not as stable and robust, despite bearing higher performance. In comparison, Android 8.0 (released in Aug. 2017) has the best system-level responsiveness, probably owing to its moderate performance and sound stability/robustness.

Mobile Apps. In our benchmark experiments, we run 10 apps on each tested device and examine the occurrence rates of SR and FF of each app. As a result, we show that Facebook (a social app), Gmail (an email app), and Subway Surf (a 3D game) are the top-3 apps that experience the highest SR/FF occurrence rates, which are 18.65%, 16.02% and 15.28%, respectively. For Facebook and Subway Surf, we attribute this to their high workloads, including frequent video streaming and 3D scene rendering.

For Gmail, the result may be somewhat surprising as email apps are fairly simple in terms of their UI functionalities. Further examination finds that this issue stems from the severe *redundant UI rendering* (RUIR) problem of Gmail's UI components. In Android, an app's UI components are usually organized in a hierarchical fashion, where each UI component resides in a certain UI layer. Using the painter's algorithm [11], Android draws an app's UI layers in a bottom-up manner. This algorithm can ensure that the overlapped UI components with different alpha (transparency) settings are properly blended. However, if the upper layers' alpha values are 1 (*i.e.,* opaque), lower layers are drawn but are in fact invisible, leading to unnecessary resource consumption. Therefore, if an app's UI layout is improperly designed, *e.g.,* having too many redundant layers and alpha settings, the RUIR problem could be rather severe. In fact, we confirm that the UI layout of Gmail incurs heavy RUIR, resulting in a large portion of the UI being redrawn for over four times (less than one time is ideal) during rendering. In Gmail, we find that every item in the email list is nested with 3∼4 UI layers, leading to severe RUIR problem.

On the other hand, for ANR and SNR, our large-scale measurement captures a total of 50,147 ANR events, involving a total of 1,446 Android apps. As depicted in Figure 8, when ranking these apps by their corresponding number of ANR events (in descending order), we observe a nearly-Zipf [33] skewed distribution, where an app's ranking (denoted as ANR_R) and its number of ANR events (denoted as ANR_N) should fit the following distribution:

$$log(ANR_N) = -alog(ANR_R) + b. \tag{3}$$

To validate this, we fit the data by first taking the logarithm of ANR_R and ANR_N, and then using linear regression to fit the negative linear relation shown in the above equation. Our validation shows that when $a = 1.41$ and $b = 4.31$, the distribution would fit the data with a 0.92 *coefficient of determination* [34] (R^2, ranging from 0 to 1), which is fairly close to 1 (the perfect fitting). Among the 50,147 ANR events occurring to 1,446 apps, 30,489 (60%) are attributed to only the top-10 (0.7%) apps, while the remaining (40%) belong to the vast majority (99.3%) of apps in the "long tail". The reason is straightforward: the top-10 apps are all extremely popular in users' daily life, thus bearing the highest probabilities of ANR.

Root Cause Analysis. To pinpoint the root causes of SR and FF, we leverage buffer-aware hierarchical timing corre-

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

lation (*cf.* §2.3) to analyze the fine-grained kernel tracing data collected in benchmark experiments on the studied phone models. As a result, we uncover three major root causes: 1) complex UI components and high rendering workloads of apps (61%), 2) slow I/O and Binder transactions (28%), and 3) long CPU scheduling delay (11%).

In detail, for the first root cause, we find that the critical function call paths in related events tend to be that of the measuring and layout rendering stages, where CPUs need to compute the size and position of each UI component. Particularly, we find that a considerable portion (73%) of such events occur on apps with the RUIR problem. For the latter two root causes, they suggest that the system is most probably experiencing resource underprovisioning or contention. More surprisingly, in 34% of the cases we notice that the `big` cores remain idle when the `LITTLE` cores are fully occupied, leading to SR/FF events. Delving deep we uncover that this is because `SurfaceFlinger`, *i.e.,* the system service that composites the final frame, is always scheduled to the `LITTLE` cores (their `cpuset` configurations are fixed to the `LITTLE` cores) to aggressively conserve battery power in vanilla Android. This works well on devices with more powerful CPUs, but tend to incur SR/FF on low-end devices. In fact, Xiaomi have recently adjusted their scheduling policy by allowing `SurfaceFlinger` to run on `big` cores for low-end devices when `LITTLE` cores are drained, which almost fully addresses the issue in practice. We thus suggest that vendors should adapt their resource scheduling policies to the specific hardware configurations.

For ANR and SNR, we leverage the automatic pipeline (cf. §2.3), to acquire 1,814 root-cause clusters, among which three dominant clusters include the majority (74%) of ANR/SNR logs. Then, we manually analyze the root causes and discover them as 1) inefficient Write Amplification Mitigation or WAM (35%), 2) lock contention among system services (21%), and 3) insufficient memory (18%). Finally, we merge all the other clusters into a single large cluster, whose root cause is regarded as 4) app-specific defects (26%).

Among the aforementioned root causes of SR/FF and ANR/SNR, the second (*i.e.,* slow I/O and Binder transactions for SR/FF and lock contention for ANR/SR) and the third (*i.e.,* long CPU scheduling delay for SR/FF and insufficient memory for ANR/SNR) can hardly be addressed since resource contention and underprovisioning are classic OS challenges; app-specific defects are even more challenging, given that there is no silver bullet for bugs and defects in software engineering. On the other hand, we find that the the critical root causes for SR/FF (*i.e.,* RUIR) and ANR/SNR (*i.e.,* WAM), can both be effectively addressed.

4 Mitigation Practices

In this section, we first present our practices of overcoming RUIR of apps in §4.1. We next describe the internals of the largest root cause (*i.e.,* the WAM issue) of ANR/SNR in §4.2, and then design a practical approach to effectively eliminating the root cause with negligible overhead in §4.3.

4.1 Overcoming RUIR with UI Layout Trimming

Recall that in Android, an app's UI components are organized in a hierarchical (layered) manner. Android renders different UI layers in a bottom-up fashion following the painter's algorithm, which, however, leads to the RUIR problem if the app's UI layout is not carefully designed, *e.g.,* the layout contains many unnecessary backgrounds which will be covered by upper-layer components and thus invisible to users, but are still rendered by the system. As discussed in §3, as many as 43% of SR/FF events occur on apps with the RUIR problem, indicating that it is a major root cause of SR and FF events.

Fortunately, the RUIR problem can be effectively overcome by optimizing a target app's UI hierarchy. To this end, we devise a dynamic layout trimming approach. In detail, we first leverage our system tracing tool (*cf.* §2.1) to capture fine-grained data at a target app's runtime, and then pick out UI components with long rendering time, which are most probably the components with the RUIR problem. With this, we extract the inner UI hierarchy of the above UI components using Android's dynamic layout inspector tool [15], based on which we can quickly pinpoint redundant UI components in the hierarchy. Meanwhile, we pay special attention to transparent components which may overlap with others but do not actually cover lower-layer components due to its transparency. Having uncovered the redundant UI components, we list their corresponding (Java/Kotlin) Class names in order to enable developers' quickly locating them at source code.

To evaluate the effectiveness of our proposed approach of mitigating RUIR, we apply it to five popular open-source Android apps: Wikipedia (a utility app), FairEmail (an email app), K-9 Mail (an email app), Amaze (a file explorer app) and Feeder (a news feed app). We first measure the apps' SR/FF occurrence rates on the 15 studied phone models, and then locate and remove redundant UI components using our proposed dynamic layout trimming method. For example, our dynamic layout trimming method uncovers that K-9 Mail uses the `<include>` tag in its UI layout file, which directly nests another layout file and introduces an unnecessary layer of hierarchy, leading to RUIR. To resolve this, we directly merge the two layouts with the `<merge>` tag to avoid redundant nesting, which preserves the functionality of `<include>` while being able to remove redundant hierarchy when including another layout file. We further validate that the trimming does not violate the original app functions by examining the `Activities` related to the trimmed UI components. This is achieved by applying our UI automation method in §2.2 to interact with all the interactable UI components in the `Activities` and checking whether runtime exceptions are raised that indicate function violation. We also extract and manually examine the code related to the trimmed components to ensure that there is no side effect introduced by our trimming.

Finally, we measure the apps' SR/FF occurrence rates on the 15 devices again to evaluate the effectiveness of our method. As a result, we find that the SR/FF occurrence rates are reduced by 11%∼32%, averaging at 27% for the apps. In particular, we observe that severe SR events with >400 ms frame rendering time and FF events have been reduced by 46% and 53%, respectively, which can all noticeably impact user experiences according to prior work [35]. Specifically, since FairEmail's RUIR problem is the least severe, its responsiveness improvement by our method is less significant

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

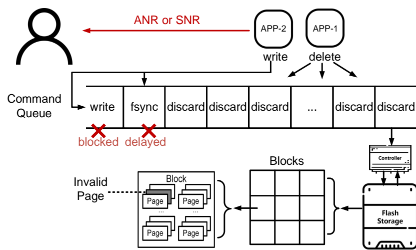


Figure 11: Android's write amplification mitigation for flash storage can lead to ANR or SNR events.

than that of others. On the other hand, Wikipedia, which is subject to severe RUIR, benefits the most.

4.2 Understanding Android's WAM

Android's Implementation of WAM. As the storage medium of almost all mobile phones, flash storage comes with two unique characteristics. On one side, *reading* a page (typically of 4 KB), which is the basic data access unit in flash storage, is direct and fast compared to that in traditional rotating-disk storage. On the other side, a block-level erase operation is required before *writing* data into a page, where a block consists of multiple (*e.g.,* 128 or 256) pages, resulting in an undesirable effect known as *write amplification* [36] which can significantly degrade the data write speed. Consequently, a write amplification mitigation (WAM) mechanism [37] is introduced into Android: once a page's stored data has been *logically* deleted in the file system, WAM marks it as *invalid* using the `discard` command. Thus, before the next write, the flash storage can trim a block containing *invalid* pages by moving valid pages in the block to other blocks. In this way, the flash storage can later (*e.g.,* when performing a write) directly erase the block with only invalid pages, leading to improved write performance.

In Android, two types of WAM are provided. By default, WAM is executed in a *real-time* manner. Many common operations (*e.g.,* screen unlock, app start, and app install/uninstall) in daily use could incur a number of file deletions. Upon a file deletion, a sequence of `discard` commands are sent to the storage controller, as demonstrated in Figure 11. In addition, when the mobile phone is idle at 3 a.m. and under charge, Android executes WAM in a batched manner (we call *lazy* WAM), which marks all the invalid pages in flash storage at a single run.

Benefits of WAMs. WAM (in particular real-time WAM) is useful and effective. For data write speed, we conduct benchmark experiments to measure the *random write speed* and the *sequential write speed* of each experimental phone. The former represents the worst-case data write speed while the latter represents the best-case. The benchmark results are listed in Figure 9, which shows that on our studied phone models (cf. Table 1), real-time WAM can increase the random (sequential) write speed by an average of 23% (26.6%) compared to the lazy WAM.

The Inefficiency of Android's WAM. Despite benefiting the data write speed, real-time WAM comes with an unexpected defect which can oftentimes lead to ANR or SNR. Specifically, from our collected logs of WAM-incurred ANR/SNR events, we observe a very common scenario as shown in Figure 11. Suppose APP-1 is issuing a `delete` command while APP-2 is issuing a `write` command. In principle, the `write` command (of APP-2) should not be affected by the `discard` commands (of APP-1), since the former is synchronous while the latter are asynchronous (so the former should be executed with a high priority). In practice, however, a special synchronous command, `fsync`, is often issued before `write` or `read` [38] to ensure the data consistency between memory and storage. The specialty of `fsync` lies in that its execution requires the completion of all the preceding `discard`s. Hence, due to `fsync`, `discard` has in fact become a *quasi-asynchronous* [39] command that could block its succeeding `write` command, thus leading to the ANR of APP-2 or SNR of Android.

To mitigate the defect of real-time WAM, an intuitive approach is to adopt "lazy" WAM instead of real-time WAM. Nevertheless, we find this lazy WAM mechanism can hardly meet our goal for three reasons. First, it is performed in a too "lazy" manner (at most once per day) and thus cannot mitigate write amplification in time. Second, once started, it cannot be interrupted; during the entire process (which is computation-intensive and time-consuming), if the screen is unlocked the user may well experience poor responsiveness. Third, if it is terminated (*e.g.,* the user kills the process) during the run, it will always make a "fresh" restart from the head when executed again.

4.3 Practical WAM

To mitigate write amplification in Android without bringing ANR or SNR, we design a practical WAM mechanism by making batched WAM *fine-grained* and *non-intrusive*.

Data-driven WAM. We take a data-driven approach to determine when to trigger the execution of batched WAM on demand. We use the analysis in benchmark experiments described in §4.2 which contain two-fold information: a) random write speed and b) total duration of batched WAM (how long it takes to fulfill all rounds of batched WAM in a whole day). As shown in Figure 10, when a smaller threshold is used for S_d, write amplification can be better addressed and the random write speed is expected to increase, but the total duration of batched WAM will increase since more rounds of batched WAM need to be executed for the same total amount of deleted data (given that each round of batched WAM involves non-trivial startup time and system overhead). We notice that S_d=6 GB tends to balance the above tradeoff. We also find that the 6 GB threshold works well under real workload based on our small-scale test deployment.

Support for Pausing and Resuming. A shortcoming of Android's batched WAM mechanism is that it cannot be interrupted once started. We thus adjust the execution logic of Android's batched WAM so that it can be paused and resumed to provide a better user experience. Specifically, we make two improvements. First, we register a broadcast receiver for the system's screen lock/unlock event, so that once the screen is unlocked, the receiver will get notified and then send a signal to pause the execution of the batched WAM. Second, we modify the batched WAM thread, which comprises a loop of trimming page groups (a page group

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

typically consists of 32K pages) to mitigate write amplification. In our modification, the batched WAM thread responds to the pause signal by recording the number of page groups that have already been trimmed and other necessary states before interrupting the execution. This allows the job to be resumed later when the screen is locked. In this way, the phone's perceived responsiveness in the presence of batched WAM is significantly improved.

Large-Scale Evaluation and Deployment. In order to understand the real-world impact of our design, we patched our proposed WAM mechanism to Android-MOD and sent invitations to the original 30,000 opt-in users to participate in our performance evaluation. This time, nearly 14,000 users opted in by installing the patched Android-MOD. The performance evaluation also lasted for three weeks (March 1st-21st, 2019). We observe that our design reduces 32% of the ANR events and 47% of the SNR events per phone. Furthermore, we use the automated analysis procedure described in §2.3 to analyze the collected logs of the ANR and SNR events after our patch is deployed. We find that almost all (>99%) of the ANR and SNR events caused by WAM have been avoided.

We also evaluate the effect on data write speed through benchmarks (as described in §4.2). As shown in Figure 9, with practical WAM, the random (sequential) write speed decreases by an average of merely 2% (3%). Given its effectiveness, our design has been incorporated into five stock Android builds by Xiaomi since May 2019. It is now benefiting ∼20M Android users every day. Other vendors (*e.g.*, Huawei and Honor) have also adopted the approach to benefit their users since the release of the patched Android-MOD. We are also working with Google to integrate the design into vanilla Android.

5 Related Work

Diagnosing Poor Responsiveness of Mobile Apps. Prior work has proposed approaches to detect and mitigate performance issues of mobile apps. First, some work utilizes dynamic approaches such as test amplification [40] and resource amplification [41] to study the runtime behavior of mobile apps. Second, researchers have employed static code analysis to pinpoint buggy code patterns such as a lack of timeout handling [42] and blocking operations in UI threads [43]. Compared to the above work, our study conducts controlled benchmarking and large-scale measurement of Android poor responsiveness. We reveal that, for example, the top reason of SNR/ANR is the inefficient WAM design in Android.

I/O Optimization for Mobile Storage. A number of I/O optimizations have been proposed for mobile storage [44]–[46]. For example, Jeong *et al.* [46] propose a number of I/O stack optimizations specialized for smartphone storage. Our work, instead, strives to address the shortcoming of Android's WAM implementation in a compatible and practical manner. Therefore, we choose to improve Android's existing batched WAM instead of completely replacing file system components. Our solution only requires small changes to the current Android OS, and has been well adopted by multiple stock Android systems.

6 Conclusion

This paper presents our experiences in understanding and combating poor responsiveness events including SR/FF and ANR/SNR in Android-based smartphone systems. Despite their disruptions to mobile user experiences, these events are not well measured and analyzed. Our study fills the above critical gap by complementarily combining controlled benchmarking on diverse devices and large-scale crowd-sourced measurement with around 30,000 opt-in users. We utilize lightweight kernel tracing and continuous monitoring infrastructure to collect fine-grained system-level data that capture every poor responsiveness event on studied devices. We then build automatic analysis schemes to infer the root causes of the observed events. The measurement and analysis help us understand poor responsiveness "in the wild". Most importantly, we develop practical solutions to mitigate the critical root causes of both SR/FF and ANR/SNR, which have yielded real-world impacts.

References

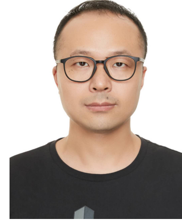
[1] Android.org, "The Slow Rendering of Android," Nov. 2019, https://developer.android.com/topic/performance/vitals/render.
[2] ——, "Keeping Your Android App Responsive," Nov. 2019, https://developer.android.com/training/articles/perf-anr.
[3] ——, "The Source Code of Android Watchdog," Nov. 2019, https://android.googlesource.com/platform/frameworks/base.git/+/android-4.3_r2.1/services/java/com/android/server/Watchdog.java.
[4] ——, "Dedicated RenderThread for UI Rendering Tasks," Nov. 2022, https://developer.android.com/about/versions/lollipop.html#Material.
[5] ——, "V-Sync Timing for Synchronizing CPU and GPU Tasks," Nov. 2019, https://developer.android.com/about/versions/jelly-bean#android-4.1.
[6] ——, "Executing UI Changes on Main Thread," Nov. 2022, https://developer.android.com/reference/android/app/Activity#runOnUiThread(java.lang.Runnable).
[7] Q. Yang, Z. Li, Y. Liu, H. B. Long, Y. A. Huang, J. He, T. Xu, and E. Zhai, "Mobile Gaming on Personal Computers with Direct Android Emulation," in *Proceedings of ACM MobiCom*, 2019, pp. 1–15.
[8] S. Yang, D. Yan, and A. Rountev, "Testing for Poor Responsiveness in Android Applications," in *Proceedings of IEEE MOBS*, 2013, pp. 1–6.
[9] Android.org, "Android ftrace/atrace," Nov. 2019, https://source.android.com/devices/tech/debug/ftrace.
[10] ——, "Reduce Overdraw," Nov. 2019, https://developer.android.com/topic/performance/rendering/overdraw.
[11] J. D. Foley, F. D. Van, A. Van Dam, S. K. Feiner, J. F. Hughes, E. Angel, and J. Hughes, *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1996, vol. 12110.
[12] E. G. Coffman, M. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, 1971.
[13] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale Analysis of Framework-specific Exceptions in Android Apps," in *Proceedings of ACM/IEEE ICSE*, 2018, pp. 408–419.
[14] Y. Lu, J. Shu, and W. Zheng, "Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems," in *Proceedings of USENIX FAST*, 2013, pp. 257–270.
[15] Android.org, "Android Layout Inspector," Nov. 2019, https://developer.android.com/studio/debug/layout-inspector.
[16] Xiaomi.com, "Xiaomi MIUI," Nov. 2019, https://en.miui.com/.
[17] Samsung.com, "Samsung One UI 2.0," Nov. 2019, https://www.samsung.com/global/galaxy/apps/one-ui/.
[18] Oneplus.com, "Oneplus OxygenOS," Nov. 2019, https://www.oneplus.com/oxygenos.
[19] Motorola.com, "Motorola Android System," Nov. 2019, https://www.motorola.com/us/software-and-apps/android.
[20] Android.org, "Android Compatibility Test Suite," Nov. 2019, https://source.android.com/docs/compatibility/cts.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2023.3237716

[21] ——, "UI Automator," Nov. 2019, https://developer.android.com/training/testing/ui-automator.
[22] Alexa.com, "Alexa Traffic Ranking for Websites," 2019, https://www.alexa.com/.
[23] P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, vol. 17, 2011.
[24] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.
[25] G. Salton, A. Wong, and C.-S. Yang, "A Vector Space Model for Automatic Indexing," *ACM Communications*, vol. 18, no. 11, pp. 613–620, 1975.
[26] A. Singhal *et al.*, "Modern Information Retrieval: A Brief Overview," *IEEE Data Eng. Bull.*, vol. 24, no. 4, pp. 35–43, 2001.
[27] J. Lee Rodgers and W. A. Nicewander, "Thirteen Ways to Look at The Correlation Coefficient," *ASA The American Statistician*, vol. 42, pp. 59–66, 1988.
[28] Samsung.com, "Performance of Samsung Galaxy S10," Nov. 2019, https://www.samsung.com/us/mobile/galaxy-s10/performance/.
[29] Oneplus.com, "Overview of OnePlus 6T," Nov. 2019, https://www.oneplus.com/6t?from=head.
[30] Oppo.com, "Overview of OPPO Reno Z," Nov. 2019, https://www.oppo.com/ae/smartphone-reno-z/.
[31] Android.org, "Help Optimize Both Memory Use and Power Consumption by Background Optimizations," Nov. 2019, https://developer.android.com/topic/performance/background-optimization.
[32] ——, "Improving App Performance with ART Optimizing Profiles in The Cloud," Nov. 2019, https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art.html.
[33] D. M. Powers, "Applications and Explanations of Zipf's Law," in *Proceedings of ACL NeMLaP3/CoNLL*, 1998, pp. 151–160.
[34] S. Wright, "Correlation And Causation," 1921.
[35] Y. Luo, K. Rodrigues, C. Li, F. Zhang, L. Jiang, B. Xia, D. Lion, and D. Yuan, "Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android," in *USENIX OSDI*, 2022, pp. 787–803.
[36] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *Proceedings of ACM SYSTOR*, 2009, p. 10.
[37] Redhat.org, "Write Amplification Mitigation," Nov. 2019, https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Storage_Administration_Guide/ch02s04.html.
[38] K. Lee and Y. Won, "Smart Layers and Dumb Result: IO Characterization of An Android-based Smartphone," in *Proceedings of ACM EMSOFT*, 2012, pp. 23–32.
[39] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting Quasi-asynchronous I/O for Better Responsiveness in Mobile Devices," in *Proceedings of USENIX FAST*, 2015, pp. 191–202.
[40] P. Zhang and S. Elbaum, "Amplifying Tests to Validate Exception Handling Code," in *Proceedings of IEEE ICSE*, 2012, pp. 595–605.
[41] Y. Wang and A. Rountev, "Profiling The Responsiveness of Android Applications via Automated Resource Amplification," in *Proceedings of IEEE/ACM MOBILESoft*, 2016, pp. 48–58.
[42] X. Jin, P. Huang, T. Xu, and Y. Zhou, "NChecker: Saving Mobile App Developers from Network Disruptions," in *Proceedings of ACM EuroSys*, 2016, p. 22.
[43] T. Ongkosit and S. Takada, "Responsiveness Analysis Tool for Android Application," in *Proceedings of ACM DeMobile*, 2014, pp. 1–4.
[44] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler." in *Proceedings of USENIX FAST*, 2012, pp. 13–13.
[45] D. T. Nguyen, "Improving Smartphone Responsiveness Through I/O Optimizations," in *Proceedings of ACM UbiComp*, 2014, pp. 337–342.
[46] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O Stack Optimization for Smartphones," in *Proceedings of USENIX ATC*, 2013, pp. 309–320.

Hao Lin (Student Member, IEEE/ACM) received the BS degree from the School of Software, Tsinghua University in 2020. He is working towards the PhD degree also at the School of Software, Tsinghua University, Beijing, China. His research areas mainly include operating systems and mobile networks.



Cai Liu received the BS degree from the School of Computer and Information, Hohai University in 2005. He is a Senior System Optimization Engineer in Xiaomi Technology Co. LTD. Prior to joining Xiaomi, he worked at Motorola, ZTE, and Samsung. He is expert at operating system.

Zhenhua Li (Senior Member, IEEE/ACM) received the BSc and MSc degrees from Nanjing University, in 2005 and 2008 respectively, and the PhD degree from Peking University, in 2013, all in computer science and technology. He is an Associate Professor with the School of Software, Tsinghua University. His research areas cover network measurement, mobile networking/emulation, and cloud computing/storage.

Feng Qian (Member, IEEE/ACM) received the BS degree from Shanghai Jiao Tong University, and the Ph.D. degree from the University of Michigan. He is currently an Associate Professor in the Computer Science and Engineering Department at University of Minnesota - Twin Cities. Prior to joining UMN, he worked at AT&T Labs and Indiana University. His research interests cover mobile systems, AR/VR, mobile networking, wearable computing, real-world system measurements, and system security.

Mingliang Li received the BS degree in computer science from Nanjing University. He is working towards the MEng degree at the School of Software, Tsinghua University, Beijing, China. Prior to joining THU in 2019, he worked at Xiaomi Technology Co. LTD. for system optimization, which is also his current research interest.

Ping Xiong received the BS and MS degrees from Wuhan University. He is now a Senior Software Engineer at Xiaomi Technology Co. LTD. He works in mobile software development and storage system optimization.

Yunhao Liu (Fellow, IEEE/ACM) received the BS degree in Automation Department from Tsinghua University, an MS and a Ph.D. degree in Computer Science and Engineering from Michigan State University. He is now a Full Professor at and the Dean of Global Innovation Exchange (GIX), Tsinghua University. His research interests include sensor network and IoT, localization, RFID, distributed systems, and cloud computing.

© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Tsinghua University. Downloaded on January 25,2023 at 07:47:06 UTC from IEEE Xplore. Restrictions apply.