Experience: Aging or Glitching? Why Does Android Stop Responding and What Can We Do About It?

Mingliang Li[1,2*], Hao Lin[1*], Cai Liu[2*], Zhenhua Li[1]
Feng Qian[3], Yunhao Liu[1,4], Nian Sun[2], Tianyin Xu[5]
[1]Tsinghua University [2]Xiaomi Technology Co. LTD [3]University of Minnesota, Twin Cities
[4]Michigan State University [5]University of Illinois Urbana-Champaign

ABSTRACT

Almost every Android user has unsatisfying experiences regarding responsiveness, in particular Application Not Responding (ANR) and System Not Responding (SNR) that directly disrupt user experience. Unfortunately, the community have limited understanding of the prevalence, characteristics, and root causes of unresponsiveness. In this paper, we make an in-depth study of ANR and SNR at scale based on fine-grained system-level traces crowdsourced from 30,000 Android systems. We find that ANR and SNR occur prevalently on all the studied 15 hardware models, and better hardware does not seem to relieve the problem. Moreover, as Android evolves from version 7.0 to 9.0, there are fewer ANR events but more SNR events. Most importantly, we uncover multifold root causes of ANR and SNR and pinpoint the largest inefficiency which roots in Android's flawed implementation of Write Amplification Mitigation (WAM). We design a practical approach to eliminating this largest root cause; after large-scale deployment, it reduces almost all (>99%) ANR and SNR caused by WAM while only decreasing 3% of the data write speed. In addition, we document important lessons we have learned from this study, and have also released our measurement code/data to the research community.

CCS CONCEPTS

• **Human-centered computing → Mobile phones**; *Ubiquitous and mobile computing systems and tools*; • **Software and its engineering → File systems management**; **Software testing and debugging**; *Software performance.*

KEYWORDS

Android; Responsiveness; Application Not Responding (ANR); System Not Responding (SNR); Write Amplification Mitigation (WAM).

ACM Reference Format:
Mingliang Li, Hao Lin, Cai Liu, Zhenhua Li, Feng Qian, Yunhao Liu, Nian Sun, Tianyin Xu. 2020. Experience: Aging or Glitching? Why Does Android Stop Responding and What Can We Do About It?. In *The 26th Annual International Conference on Mobile Computing and Networking (MobiCom '20), September 21–25, 2020, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3372224.3380897

[*] Co-primary authors. Zhenhua Li is the corresponding author.

MobiCom '20, September 21–25, 2020, London, United Kingdom

ACM ISBN 978-1-4503-7085-1/20/09…$15.00
https://doi.org/10.1145/3372224.3380897

1 INTRODUCTION

Responsiveness is among the basic and key metrics that determine smartphone user experience. Poor responsiveness, such as slow rendering and frozen frames [13], would impair the productivity, satisfaction, and engagement of users. Further on Android, 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 [11] [1]. The dialog asks users to either continue wait or kill the app, neither of which leads to pleasant user experience. Worse still, if 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 [14], which we call a System Not Responding (SNR) event.

Over the years, tremendous efforts have been made to optimize the responsiveness of Android systems. One example is Project Butter [8]. It introduces *triple buffering* that buffers an extra graphic frame in the GPU's memory to improve the stability of UI frame rate, and *V-Sync* that synchronizes the CPU and GPU's parallel processing of UI frames. Despite these efforts, slow rendering, frozen frames, ANR, and even SNR are still prevalent on Android [60, 61]. Although users can usually endure slow rendering and frozen frames, they can hardly put up with ANR and SNR which have a direct, disrupting impact on user experience. Unfortunately, little have we understood regarding the prevalence, characteristics, and root causes of ANR and SNR, due to the lack of large-scale measurement and analysis on real-world smartphone usage. Such lack of understandings, insights, and datasets significantly hinders practical solutions to address the problem and improve user experience.

1.1 Understanding ANR and SNR at Scale

To measure and analyze ANR and SNR at scale, we build a continuous monitor infrastructure based on a customized Android system called Android-MOD. Android-MOD records detailed traces upon the occurrence of any ANR or SNR event, including timestamp, CPU and memory usage, end-to-end call stacks of related processes (including both the app and the system services), and the blocked threads recorded in Android event logs. We invited the active users in Xiaomi's smartphone community to participate in our measurement study by installing Android-MOD on their phones. Over 30,000 users opted in and collected data for us for

[1]The two timeout thresholds for defining ANR/SNR events are supported by HCI studies on typical users' tolerance of delayed UI response [15, 17, 37, 53].

three weeks, involving 15 different models of Android phones. All the data are collected with informed consent of opt-in users, and no personally identifiable information (PII) was collected.

Our measurement reveals that ANR and SNR occur prevalently on all our studied 15 hardware models that run Android. On average, 1.5 ANR events and 0.04 SNR events occur on an Android system during the measurement, and the maximum number of ANR (SNR) events reaches 37 (18) on an Android system. Also, we notice that 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). Surprisingly perhaps, we observe that better hardware does not seem to relieve the problem. 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. Moreover, 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. In addition, video apps and 3D interactive games are more subject to ANR. We will delve deeper into the above findings in §3.

To uncover the root causes of ANR and SNR, we build an automatic pipeline to process and analyze the logs recorded by Android-MOD from the measured phones. For each log, we first extract the blocked threads, and then generate their *wait-for graph* [21] to figure out the critical thread that leads to ANR or SNR. Based on the above processing, we classify each ANR/SNR event into the corresponding root-cause cluster using *similar-stack analysis* [23], and manually analyze the root cause of unbiased ANR/SNR 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 and SNR events: the inefficiency of Write Amplification Mitigation [36] or WAM (35%), lock contention among system services (21%), insufficient memory (18%), and app-specific defects (26%). While resource contention and under-provisioning are classic operating system challenges and there is no silver bullet for bugs and defects in app software engineering, we surprisingly find that the largest root cause, *i.e.,* WAM in Android, comes from a flawed design and can be fundamentally eliminated with a clean and complete fix.

1.2 Eliminating the Largest Root Cause

WAM is an effective optimization to speedup writes to flash storage where writing a page needs to first erase a whole data block. It marks invalid pages (brought by file deletions) in the flash storage using `discard` commands to mitigate *write amplification* [25]. In Android, WAM is done at *real-time*, given that many common operations (*e.g.,* screen unlock) could incur a number of file deletions. This, however, comes with an unexpected effect that could lead to ANR/SNR, as shown in Figure 1. Suppose APP-1 is issuing a `delete` command while APP-2 is issuing a `write` command. In principle, `write` should not be affected by the `delete`-triggered `discard` commands, since the former is synchronous while the latter are asynchronous. But in practice, `write` often comes after `fsync` which requires the completion of all preceding `discards` [32].

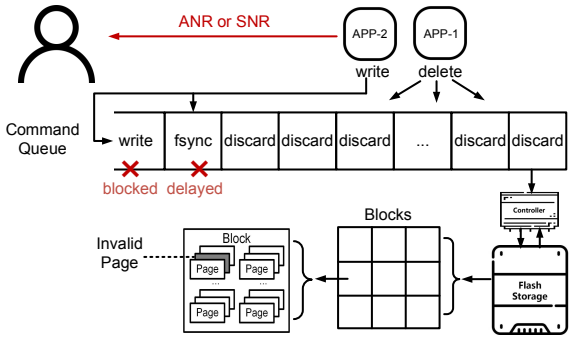

Figure 1: Android's write amplification mitigation for flash storage can lead to ANR or SNR events.

Consequently, `discard` in fact becomes *quasi-asynchronous* [28] that blocks its succeeding `write` and leads to ANR/SNR.

A straightforward fix is to *batch* WAM instead of real-time WAM. Android implements the batched WAM by marking all invalid pages in a single run, which we find is rather ineffective. First, its lazy nature (at most once a day) cannot mitigate write amplification in time. Second, once started, it cannot be interrupted and the I/O heavy process will make the phone unresponsive. Third, if the user kills the process, the process will restart from the head.

To address the issue, we design a *practical* WAM by making batched WAM *fine-grained* and *non-intrusive*. It records the amount of deleted data (S_d), and uses a data-driven approach to decide a proper threshold for S_d to trigger the execution of batched WAM on demand. This not only achieves timely mitigation but also amortizes the mitigation cost. We also make our batched WAM interruptible and resumable to cost-effectively guarantee responsiveness.

After rolling out our patched Android-MOD on part of the 30,000 opt-in users' phones, our design reduces almost all (>99%) ANR and SNR events caused by WAM. Meanwhile, the data write speed is decreased by only 3% on average. Our design has been further adopted by five stock Android systems since May 2019, benefiting ~20M Android users.

1.3 Summary of Contributions

- We conduct the first large-scale and in-depth measurement study of the unresponsiveness (ANR and SNR) of Android in the wild, and confirm their prevalence for various models of phones. We also discover that ANR and SNR are more of a software issue than a hardware issue.
- We present our end-to-end data collection and analysis pipeline for deeply understanding ANR and SNR. Our collection is lightweight and does not affect the performance of Android systems. Our analysis pipeline can automatically pinpoint the root causes of ANR and SNR.
- We carefully diagnose and practically address the largest root cause of ANR and SNR. After real-world deployment, our solution reduces 32% ANR and 47% SNR events while only decreasing 3% of the data write speed.

Our measurement code and data have been released in part at https://Android-Not-Respond.github.io to benefit the community.

2 METHODOLOGY

In this section, we describe our monitoring infrastructure that continuously captures detailed data of ANR/SNR at scale (§2.1), and our automatic pipeline for root cause analysis of ANR/SNR (§2.2).

2.1 Monitoring Infrastructure

As mentioned at the beginning of §1, ANR and SNR are both response timeout events happening to an app process or a system thread. Once an ANR or SNR event occurs, Android automatically records a series of diagnostic information [2, 5–7] including:

- Timestamp of the occurrence;
- CPU and memory usage;
- Call stack of the app process (only for ANR);
- Call stacks of a predefined set of system service processes, such as `SystemServer` and `MediaServer`;
- Blocked threads (recorded in Android's event log).

Unfortunately, we find that the above information is insufficient for our study 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. As a consequence, we are unable to build our monitoring infrastructure without modifying the Android framework (even with root privileges). 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 opt-in user 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 by patching merely 200 lines of code 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 diagnostic mechanism.

To study ANR and SNR at scale with our monitoring infrastructure, 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 on their phones. Eventually, more than 30,000 users opted in, most of whom are geek users willing to test experimental functionalities or systems. We explicitly informed the opt-in users in the email that Android-MOD is a lightweight update that will not affect their installed apps, relevant data, OS version, or system performance. The recorded ANR/SNR data was uploaded to our data server when there is WiFi connectivity.

In detail, the measurement lasted for three weeks from Nov. 1st to Nov. 21st in 2018, involving a wide range of phones across 15 different models as listed in Table 1 (all their CPUs have eight cores).

2.2 Root Cause Analysis Pipeline

To figure out the root cause of a single ANR or SNR event, app or system developers usually analyze its corresponding log by hand.

Table 1: Hardware and OS configurations of our measured phone models, manually ordered by performance.

Model	CPU	Memory	Storage	Android Version
1	1.8 GHz	3 GB	32 GB	7.0
2	2 GHz	4 GB	64 GB	7.0
3	2 GHz	4 GB	64 GB	7.0
4	1.8 GHz	6 GB	64 GB	9.0
5	1.8 GHz	6 GB	64 GB	7.0
6	2.2 GHz	4 GB	64 GB	8.0
7	2.2 GHz	4 GB	64 GB	9.0
8	2.2 GHz	6 GB	64 GB	7.0
9	2.2 GHz	6 GB	64 GB	8.0
10	2.2 GHz	6 GB	64 GB	7.0
11	2.3 GHz	6 GB	64 GB	8.0
12	2.8 GHz	6 GB	128 GB	8.0
13	2.8 GHz	8 GB	128 GB	9.0
14	2.84 GHz	8 GB	128 GB	9.0
15	2.84 GHz	8 GB	128 GB	9.0

However, such manual analysis does not scale. Therefore, we developed an automated analysis 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. Our analysis pipeline processes and analyzes the collected logs as illustrated in Figure 2.

Analysis Pipeline for ANR Events. 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. 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* [21] for the application'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 [2] 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 that are exemplified in Figure 3 [3]. The regular expressions are diverse in terms of their lengths and complexities, *e.g.,* some are as simple as numbers while others

[2] 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.

[3] The complete list can be found at https://Android-Not-Respond.github.io

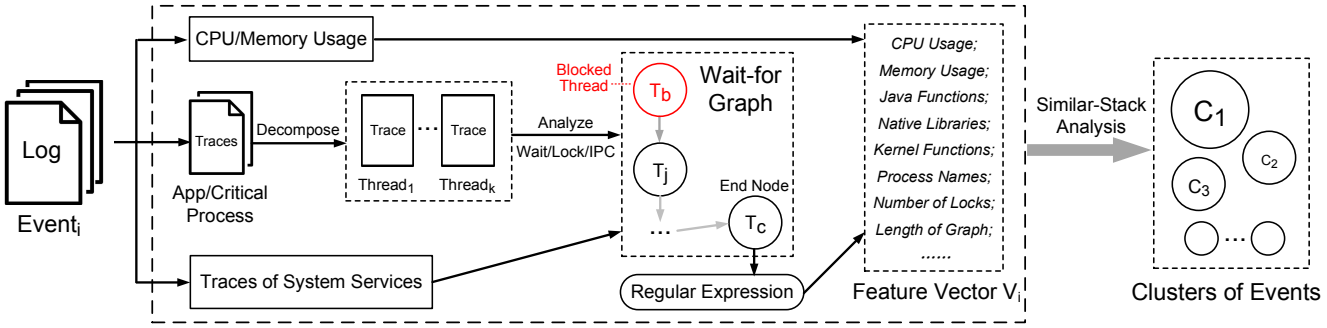

Figure 2: Work flow of our developed pipeline for automatically analyzing the root causes of ANR and SNR events.

```
// The regular expressions follow the Java style

// Kernel memory address
String REGEX_KER_ADDR =
"(\\+)?0x[0-9a-fA-F]{1,10}(/0x)?[0-9a-fA-F]{1,10}";
// Native memory address
String REGEX_NATIVE_ADDR = " *#\\d+ pc [0-f]{1,16} ";
// Native function memory offset
String REGEX_OFFSET = "offset 0x[0-f]{1,16}";
// Thread ID
String REGEX_TID = "tid=\\d+";
// Java line number
String REGEX_LINE_NUMBER = ":\\d+)";
```

Figure 3: Examples of regular expressions used to remove irrelevant information from the call stacks.

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* [23]. 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* [27], 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 $J(\ldots)$ 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* [48] and *cosine similarity* [54], 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.

$$S(i,j) = \frac{(\sum_{n=i,j} |F_{p,n}|) \cdot S_p(i,j) + (\sum_{n=i,j} |F_{c,n}|) \cdot S_c(i,j)}{\sum_{m=p,c} \sum_{n=i,j} |F_{m,n}|}. \quad (3)$$

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 manual inspection of representative ANR samples.

Validation. The similar-stack analysis can generate thousands of root-cause clusters. However, we observe there are only several *dominant clusters* with the largest sizes that include the majority of ANR events. We manually analyze the dominant clusters to validate our automated analysis pipeline. Specifically, for each cluster, we first examine the K (empirically set to 100) samples nearest to the cluster centroid to find out their root cause(s). We then analyze the K samples furthest from the centroid, comparing their root cause(s) with those nearest to the centroid. Our manual examination shows that all the inspected cases are perfectly categorized with no false positives, mainly attributed to our high similarity threshold (0.95).

Analysis Pipeline for SNR Events. For an SNR event, our collected log contains the call stacks of multiple system service processes, where only one process is flagged by Android as the *critical*

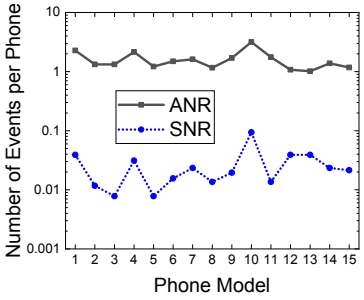

Figure 4: Avg. number of ANR/SNR events per phone for each model.

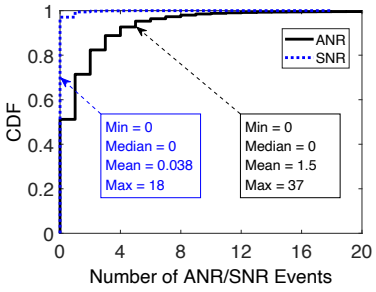

Figure 5: Number of ANR/SNR events happening to a single phone.

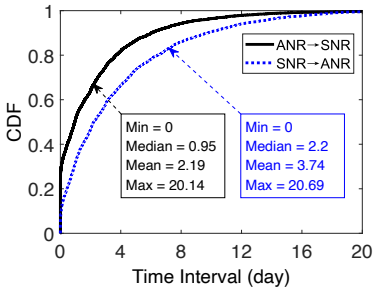

Figure 6: Time interval for consecutive ANR→SNR and SNR→ANR.

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 large-scale data collection and automatic analysis pipeline, we have multifold findings on Android ANR and SNR in terms of their prevalence and characteristics, as well as in-depth understandings of their root causes. Although our reported results 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 while only customizing the UI elements of the vanilla Android system [38, 40, 50, 59].

Prevalence of Unresponsiveness. Our measurement reveals that both ANR and SNR occur prevalently on all the studied 15 phone models, as shown in Figure 4. On average, 1.5 ANR events and 0.04 SNR events occur on an Android phone during the three-week measurement. The distributions of both ANRs and SNRs are skewed, as shown in Figure 5. For ANRs, around a half (51%) of Android phones do not experience ANR, while the maximum number of ANR events occurred on an Android phone is 37. For SNRs, most (97%) Android phones do not experience SNR, while the maximum number of SNR events occurred on one phone is 18. On average, 29% Android systems encountered at least an ANR or SNR event every ten days.

Correlations between ANRs and SNRs. Although ANRs are significantly more prevalent (40×) than SNRs, we notice that ANR and SNR events are highly correlated in terms of occurrence probability for a given phone model, as shown in Figure 4. The *sample correlation coefficient* [33] between their occurrence probabilities is as high as 0.73.

At a first glance, the high correlation indicates that an SNR event is likely to be caused by an ANR event. However, our timing analysis refutes the hypothesis. We examine the time interval between every SNR event and its most recently preceding ANR event ("ANR→SNR"). Figure 6 shows that the median time interval is as long as 0.95 day and the average is 2.19 days. Therefore, an SNR event is usually *not* caused by an ANR event. Additionally, we examine the time interval between every ANR event and its most preceding SNR event ("SNR→ANR"), and find that an ANR event is *not* caused by an SNR event, either, as shown in Figure 6.

The high correlation of ANRs and SNRs in the occurrence probability and weak correlation in the occurrence time 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 responsiveness" [41, 43, 49], non-professional users might intuitively believe that a phone with more advanced hardware experiences fewer ANR/SNR events. Surprisingly perhaps, we can see from Figure 4 that this is not true – 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. Detailed hardware configurations of the 15 phone models can be found in Table 1. On the other hand, 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 from version 7.0 to 9.0, considerable performance optimizations have been added to the Android framework and the OS kernel [9, 10, 12]. Therefore, we expect ANRs and SNRs in recent Android versions to be substantially reduced. As shown in Figure 7, 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 aforementioned performance optimizations have taken effect in improving the responsiveness of common apps.

However, we find that the system-level responsiveness (*i.e.,* the situation of SNR) gets worse, probably because the very new Android 9.0 (released in Aug. 2018) is not quite 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 as well as sound stability and robustness.

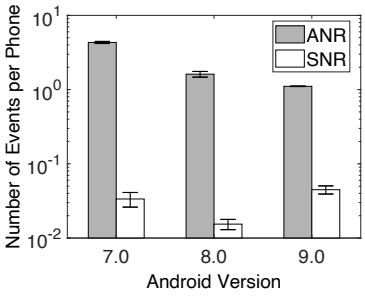

Figure 7: Average number of ANR/SNR events per phone for each Android version.

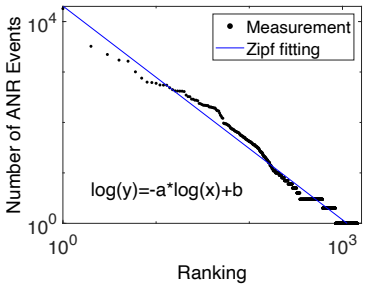

Figure 8: Ranking of apps by their number of ANR events. Here $a = 1.41$ and $b = 4.31$.

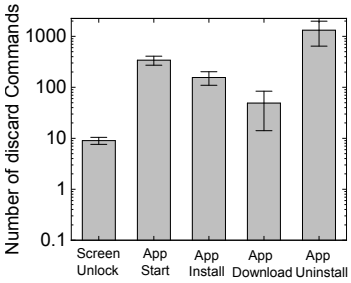

Figure 9: Number of `discard` commands incurred by a common operation in daily use.

Table 2: Top-10 apps ordered by number of ANR events.

Application	# ANR Events	Category
WeChat	18060	Instant Messaging
Arena of Valor	3234	Game
Kwai	2226	Video
Mobile QQ	1722	Instant Messaging
Alipay	1638	Mobile Payment
Youku	1008	Video
Xigua Video	756	Video
Bilibili	630	Video
iQIYI	618	Video
Toutiao	597	News Browsing

Mobile Apps. Our measurement captures a total of 50,147 ANR events, involving a total of 1,446 Android apps. When ranking these apps by their corresponding number of ANR events (in descending order), we observe a nearly-Zipf [45] skewed distribution as depicted in Figure 8. Among the 50,147 ANR events occurring to 1,446 apps, 30,489 (60%) are attributed to only the top-10 (0.7%) apps, as listed in Table 2, while the remaining (40%) belong to the vast majority (99.3%) of apps that lie 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. We further analyze the top-10 apps in Table 2 in more detail. Among the 10 apps, five are used for video streaming, two for instant messaging, and the remaining three are for 3D interactive gaming, mobile payment, and news browsing. Overall, the results indicate that ANR can occur to a wide range of diverse apps. It is easy to understand that video streaming and 3D interactive gaming are more likely to encounter ANR, since they are both computation-intensive. Note that although the specific app ranking will vary in different geographic regions where the frequently-used apps usually differ, the top app types are expected to remain largely consistent across different markets [51, 55, 56]. We therefore believe that our high-level findings where computationally-intensive apps are more likely to incur ANR will hold for other Android markets or in other regions.

Root Cause Analysis. In order to uncover the root causes of ANR and SNR, we collect 50,147 logs for ANR and 1,271 logs for SNR. Leveraging the automatic pipeline (cf. §2.2) to process and analyze the 51,418 logs, we 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 of 100 unbiased samples in each dominant cluster, and discover three major root causes as 1) inefficient Write Amplification Mitigation or WAM (35%), 2) lock contention among system services (21%), and 3) insufficient memory (18%). We also validate the correctness of our manual analysis using the method described in §2.2, and find the analysis accuracy to be 100%. Finally, we merge all the non-dominant clusters into a single large cluster, whose root cause is regarded as 4) app-specific defects (26%).

We closely examine the logs of the aforementioned root causes to unravel some of their typical scenarios. We notice that lock contention often arises when one system service is waiting for the release of a resource lock preempted by another system service involving heavy I/O tasks or having a low execution priority. Regarding insufficient memory, we observe that most of the related ANR/SNR events result from a usually time-consuming mechanism triggered by low available memory – *garbage collection* [3] of Android's runtimes, which aims to free and recycle unused memory. Unfortunately, we find these two root causes are hard to fundamentally address since resource contention and under-provisioning are classic operating system challenges. However, some cases of these problems are practically fixable such as deadlocks brought by inappropriate resource contention. In particular, we discover a deadlock caused by the mutual IPC between the network management process `netd` and the system service `SystemServer`. Specifically, as shown in Figure 10, Process-1 and Process-2 first leverage IPC to invoke an API of `netd` and an API of `SystemServer`, respectively (①, ②). Then, these two APIs would attempt to establish IPC with `netd` and `SystemServer`, respectively (③, ④). However, neither attempt will be successful since the IPC thread pools of both `netd` and `SystemServer` have been drained by previous IPCs of Process-1 and Process-2 (for simplicity, in Figure 10 we assume only one IPC thread is allowed in a thread pool). This finding was then reported to Google, who quickly acknowledged it and collaborated with us to develop a patch for the latest version of Android [4]. In this patch,

the API of `netd` no longer establishes IPC with `SystemSever` upon being invoked (*i.e.,* ④ is removed) to resolve the deadlock.

Further, 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 largest root cause, *i.e.,* WAM in Android, comes from a flawed design and can be fundamentally eliminated with a clean and complete fix as to be detailed next.

4 ADDRESSING THE INEFFICIENT WAM

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

4.1 Analysis and Measurement

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* [25] which can significantly degrade the data write speed. Consequently, a write amplification mitigation (WAM) mechanism [47] 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 that contains 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 (via a command queue in the Linux kernel), as demonstrated in Figure 1. Each `discard` command is meant to mark specific pages as invalid, so that the corresponding block can be trimmed when the flash storage is idle. 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, to further mitigate the write amplification problem.

The Issue with Discards. We find that the `discard` operations have a significant impact on the performance of flash storage. To understand that, we first conduct a user study to measure the number of `discard` commands in daily usage and the data write speed under different WAM mechanisms. Note that continuously monitoring the occurrence of `discard` or data write speed on the 30,000 opt-in users' phones could bring non-trivial overheads. Thus, we resort to small-scale measurements of 15 experimental phones corresponding to the 15 models in Table 1. For each experimental phone, we first use it for a whole day in a normal manner, and then conduct our measurements.

To count `discard` commands, we adopt the `ftrace` tool to record every invocation of the kernel function `ext4_free_blocks` that issues the `discard` command. Figure 9 lists the numbers of `discard` commands incurred by five common operations, from which we see that screen unlock incurs the fewest (9 on average) `discard` commands [4] while app uninstall incurs the most (1,317 on average) `discard` commands. Due to the numerous file deletions and the accompanying `discard` commands, a large amount of data (20 GB on average) is deleted on a common Android phone every day.

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 11, 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 1. 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` [32] 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* [28] 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.

4.2 Our Practical Solution

To effectively mitigate write amplification in Android without bringing ANR or SNR, we design a practical WAM mechanism by making

[4] To understand why screen unlocks require file deletions, we traced the end-to-end workflow of screen unlock. When unlocking the screen, the state of the Android system would be changed from Screen Locked to Screen Unlocked. The state change requires modifications to a few configuration files implemented using the `AtomicFile` class [1]. `AtomicFile` creates temporary, shadow files that will later be removed after the files are successfully modified.

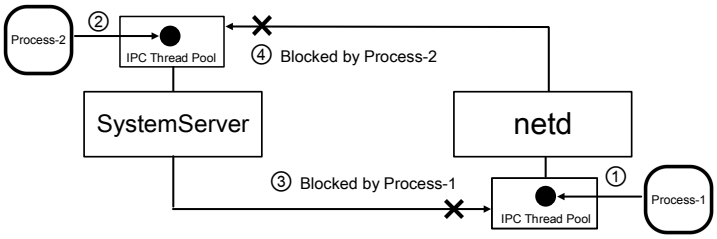

Figure 10: A deadlock caused by the mutual IPC between `netd` and `SystemServer`. Here an arrow represents an IPC.

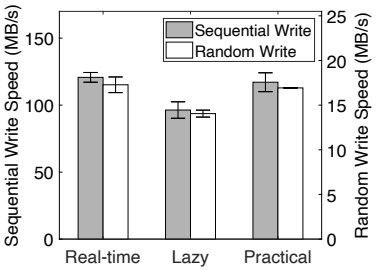

Figure 11: Random and sequential data write speeds using different WAM mechanisms.

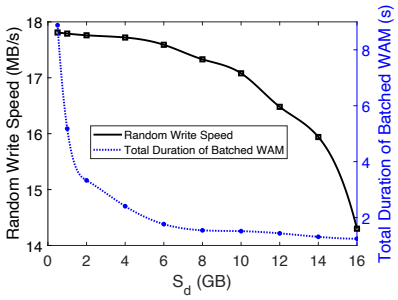

Figure 12: Total duration of batched WAM and random write speed for different thresholds (S_d).

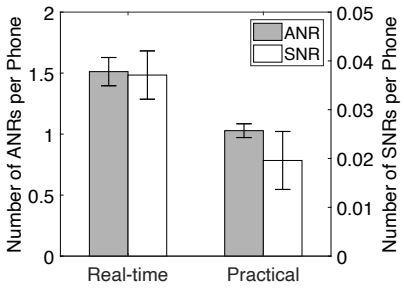

Figure 13: Number of ANR/SNR events per phone using real-time WAM and practical WAM.

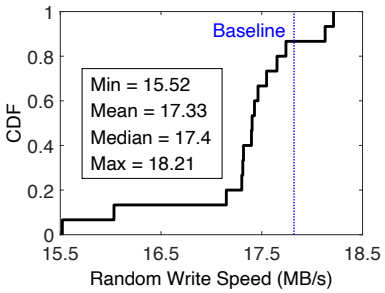

Figure 14: Random write speed of our tested 15 devices when our proposed WAM is applied using the 6 GB threshold.

batched WAM *fine-grained* and *non-intrusive*. The basic idea is to strike a balance between the real-time and the lazy approach by performing WAM when the amount of deleted data reaches a threshold. Also, we deploy it on part of the ~30,000 opt-in users' mobile phones for performance evaluation, and further port it to multiple stock Android systems.

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 §4.1 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 12, 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.

In detail, we take a lightweight approach to record the total amount of deleted data, by monitoring the invocation of every `discard` command in the file system (*e.g.,* EXT4) of Android. Specifically, every `discard` command is issued through the kernel function `ext4_free_blocks(..., unsigned long count, ...)`, where `count` denotes the number of deleted pages. Thus, the corresponding amount of deleted data is calculated as `count`×4 KB, and S_d is the accumulated amount for all `discard` commands.

To understand the effectiveness of our 6 GB threshold in real-world scenarios, we carry out a small-scale test on 15 devices, each corresponding to a phone model in Table 1. We perform the benchmark analysis similar to that in §4.1 to measure random write speed of the 15 devices under real workloads. To obtain the performance baseline that represents the best-case result, we first execute a full-pass WAM to trim the entire storage on all devices, and then measure their average random write speed to be 17.82 MB/s. We then ask real users to use the 15 devices while sampling their random write speed during the course of normal operations. As depicted in Figure 14, for the majority (80%) of the devices, the average random write speed distributes narrowly between 17.15 MB/s and 18.21 MB/s, which are quite close to the baseline performance.

Support for Pausing and Resuming. A shortcoming of Android's batched WAM mechanism is that it cannot be interrupted once started and thus may lead to poor responsiveness upon screen unlock. 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 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.

Our patch consists of only around 150 lines of code. The logic of detecting and responding to the pause signal as well as monitoring S_d is implemented in the kernel; the remaining logic is implemented in the user space.

4.3 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 from Figure 13 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 §2.2 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 of our design on data write speed through benchmark experiments (as described in §4.1). As shown in Figure 11, after our practical WAM is applied, the random (sequential) write speed decreases by an average of merely 2% (3%), compared to real-time WAM. 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.

5 RELATED WORK

We discuss related work in three topics: diagnosing responsiveness issues, optimizing storage I/O, and analyzing mobile OS logs.

Diagnosing Unresponsiveness of Mobile Applications. Prior work has proposed approaches to detect and mitigate defects in mobile apps that lead to unresponsiveness and other relevant performance issues. First, some work utilizes dynamic approaches such as test amplification [24, 61, 62] and resource amplification [58] to study the runtime behavior of mobile apps in response to blocking or computation-intensive operations. Second, researchers have employed static code analysis to pinpoint buggy code patterns such as a lack of timeout handling [30], blocking operations in UI threads [42], and other performance issues [34]. Moreover, the research community has built diagnostic tools to identify the root causes of unresponsiveness or other performance issues in mobile systems. For example, Ravindranath *et al.* [46] developed *AppInsight*, which instruments mobile app binaries to automatically identify the critical path (fundamentally determining the user-perceived latency) in user interactions; Brocanelli *et al.* [18] designed *Hang Doctor*, an automated analysis tool that unveils the root causes of apps' timeout events by locating the most frequent timeout operations. Compared to the above work, our study conducts a large-scale root cause analysis of real-world SNR and ANR events. We reveal that, for example, the top reason of SNR/ANR is the inefficient WAM design in Android system.

I/O Optimization for Mobile Storage. A number of I/O optimizations have been proposed for mobile storage [29, 35, 39, 44, 52]. For example, Jeong *et al.* [29] propose a number of I/O stack optimizations specialized for smartphone storage. Nguyen *et al.* [39] measure the I/O delays in Android at scale; they propose to improve the responsiveness by prioritizing `read` operations over `write` operations. In particular, the drawbacks of Android's WAM implementation have recently been discussed. Jeong *et al.* [28] attribute the quasi-asynchronous I/O (QASIO) operations as the root cause of Android's WAM issues, and propose to address it by prioritizing the QASIO operations. They have also pointed out that the inefficiency of WAM can be effectively eliminated if WAM is executed in a batch manner at the device's convenience, as what Android's native "lazy" WAM does. However, they do not adopt this out of concern that I/O performance could be gravely degraded when the `discard` command is not issued in time. Lee *et al.* [31] further propose a new file system called F2FS which is optimized for flash storage. In F2FS, a checkpoint mechanism is introduced to overcome the shortcoming of real-time WAM by only performing QASIO operations when checkpointing is triggered.

Compared to the above studies, our work instead strives to address the limitations of Android's WAM design in a practical manner. Therefore, we choose to improve Android's existing batched WAM mechanism instead of developing a new file system component from scratch. Our solution only requires small changes to the current Android OS and has been commercially adopted by multiple stock Android systems. With regard to the I/O performance, we observe an average of only 3% decrease in data write speed brought by our proposed solution.

Log Analysis. The Android framework supports capturing a rich set of logs and traces for monitoring and diagnostic purposes [2, 5–7], such as event logs, call stacks of the Android native layer, and call stacks of the Android kernel. The affluent information recorded in these logs and traces can be analyzed in depth to troubleshoot various system-level issues. To this end, we apply the *Exception Buckets* and the similar-stack analysis technique proposed in [23] to cluster the root causes of SNR and ANR events collected from a large number of mobile devices. We also use the wait-for graph [21] to identify the critical thread that leads to ANR/SNR.

6 LESSONS LEARNED

We summarize several important lessons we learned from this study.

Performing Large-scale Measurement at the OS level is Feasible. Large-scale, crowdsourced ANR/SNR event collection is a prerequisite for this study. Despite the rich logs provided by Android, ANR/SNR events cannot be directly captured from the user space. We thus resort to OS modification by following three philosophies: (1) collecting only the necessary information (*i.e.,* the call stacks of several important system services), (2) making our changes transparent to applications and their data, and (3) engineering-wise,

minimizing the modifications and paying attention to the code performance. Our experiences indicate that by properly following the above principles, it is entirely feasible to launch large-scale measurement studies in the wild using a modified mobile OS.

Applying a Principled Method to Identify Sources of Performance Issues is Helpful. When we start this project, we envision that ANR/SNR may possibly be attributed to hardware (slow phones), OS (inefficient OS design), or applications (buggy application implementation). It is therefore beneficial to identify the source(s) of the problem to facilitate subsequent in-depth analyses. Our experiences dictate that investigating a wide range of phones, OSes, and applications, despite being laborious, turns to be critical. To this end, we study 15 models of diverse phones with 3 mainstream Android versions (Table 1), as well as capture system-wide ANR/SNR events for *all* applications. Leveraging such rich data and through rigorous data mining, we are able to locate the sources of the majority of our collected ANR/SNR events. Such a principled approach eases our analysis and makes it more accountable.

Considering Inter-app and App-OS Interactions is Important. Traditional software engineering methods for code testing and debugging typically focus on a single application. Our findings regarding the inefficient WAM suggest that when troubleshooting bugs exhibited on a single app, it is also important to further consider interplays among applications as well as those between application and the OS, as demonstrated in Figure 1. Broadly speaking, cross-layer, cross-app, and even cross-device issues are typically more challenging to troubleshoot compared to their single layer/app/device counterparts. This is in particular the case in the mobile context with a rather complex ecosystem. We believe more research is needed in this direction through the synergy among OS, mobile computing, and software engineering, to name a few, to facilitate such cross-entity troubleshooting.

Simple System Tuning can Yield Great Performance Benefits. Our solution to the inefficient WAM problem balances the two extreme strategies: real-time WAM and daily "lazy" WAM. Our patch, which has registered commercial deployment, only consists of 150 lines of code. Despite a simple method, its result is encouragingly positive: it completely eliminates WAM-incurred ANR/SNR with negligible I/O performance degradation. At a high level, although mobile systems are becoming increasingly sophisticated with complex tradeoffs, the fundamental dimensions that the tradeoffs involve remain largely unchanged, such as the I/O performance vs. user-perceived latency in the case of WAM. It is thus important to develop principled, developer-friendly approaches to systematically examine these tradeoffs and judiciously balance them by considering real workload and users' quality-of-experience (QoE).

Mobile Device Contextual Information can Hint System Optimization. Regarding improving the perceived QoE, our WAM solution employs the screen status as a hint: the WAM operation is paused and resumed when the screen is unlocked and locked, respectively. In this way, WAM will not interfere with users' typical foreground activities [26]. Generally speaking, modern mobile devices provide a rich set of "contexts" such as sensor reading, battery status, user engagement level, wireless network performance, and cellular billing status, to name a few, which have registered a broad range of real-world application [16, 19]. The mobile OS can leverage such contextual information to guide system optimizations – an opportunity that has been somewhat overlooked by the community.

Incorporating Domain Knowledge into Off-the-shelf Machine Learning Algorithms Helps Improve the Inference Accuracy. When clustering ANR/SNR events, we initially construct a single feature vector for each ANR/SNR event sample and directly feed all feature vectors into the off-the-shelf clustering algorithm [20, 22, 57]. We find that this leads to very poor clustering results because the semantics of the features (*e.g.,* the instruction set vs. memory usage) are highly heterogeneous and it is difficult to directly "normalize" or "reconcile" the features. To overcome this limitation, we apply our domain knowledge to enhance the machine learning (ML) algorithm performance. Specifically, we separate all the features into two sets (F_p and F_c) based on their characteristics as elaborated in §2.2. Then to accommodate their heterogeneity, we employ different similarity metrics for each set: we use the *Jaccard Index* [27] for F_p to avoid over-fitting, while we leverage the *cosine similarity* [48, 54] for F_c to thoroughly capture their semantics in the similar-stack analysis, before strategically merging both similarity metrics. The above process helps significantly improve the clustering accuracy and allow us to acquire the three major root causes of ANR/SNR in an automated fashion with no false positives. Overall, our experiences indicate that domain knowledge could be properly applied to enhance the quality for feature selection, representation, and measurement, which are much more important compared to selecting the ML algorithm itself. We believe this is applicable to applying ML to mobile system data in general.

7 CONCLUSION

This paper presents our experiences in understanding and combating ANR and SNR events (known as unresponsiveness issues) in Android-based smartphone systems. Despite their disruptions to mobile user experiences, ANR and SNR events are not well measured and analyzed at scale. Our study fills the above critical gap by conducting a large-scale crowd-sourced measurement with around 30,000 opt-in users. Collaborating with a major Android phone vendor, we were able to deploy our continuous monitoring infrastructure to collect detailed logs that capture every ANR or SNR event on users' Android devices. We then build an automated analysis pipeline to extract relevant information and to infer the root causes of the observed ANR and SNR events. The measurement and analysis help us understand ANR and SNR events "in the wild". Most importantly, we develop a practical solution to eliminate the largest category of ANR/SNR events that are caused by the suboptimal WAM design in Android. Being commercially deployed, our solution is already benefiting nearly 20 million users.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their insightful and detailed comments, as well as the shepherd for guiding us through the revision process. This work is supported in part by the National Key R&D Program of China under grant 2018YFB1004700, the National Natural Science Foundation of China (NSFC) under grants 61822205, 61632020 and 61632013, and the Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] Android.org. 2019. Android AtomicFile. https://developer.android.com/reference/android/support/v4/util/AtomicFile.

[2] Android.org. 2019. Android Event Log. https://developer.android.com/reference/android/util/EventLog#writeEvent(int,%20java.lang.String).

[3] Android.org. 2019. Android Memory Management. https://developer.android.com/topic/performance/memory-overview.

[4] Android.org. 2019. AOSP Patch #866871. https://android-review.googlesource.com/c/platform/system/netd/+/866871.

[5] Android.org. 2019. Collecting Framework Call Stacks. https://android.googlesource.com/platform/frameworks/base/+/4f868ed/services/core/java/com/android/server/am/ActivityManagerService.java#4920.

[6] Android.org. 2019. Collecting Kernel Call Stacks. https://android.googlesource.com/platform/frameworks/base.git/+/android-4.2.2_r1/core/jni/android_server_Watchdog.cpp#55.

[7] Android.org. 2019. Collecting Native Call Stacks. https://android.googlesource.com/platform/frameworks/base/+/56a2301/core/jni/android_os_Debug.cpp#544.

[8] Android.org. 2019. Features of Android 4.3 Jelly Bean. https://developer.android.com/about/versions/jelly-bean.

[9] Android.org. 2019. Help Optimize Both Memory Use and Power Consumption by Background Optimizations. https://developer.android.com/topic/performance/background-optimization.

[10] Android.org. 2019. Improving App Performance with ART Optimizing Profiles in The Cloud. https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art.html.

[11] Android.org. 2019. Keeping Your Android App Responsive. https://developer.android.com/training/articles/perf-anr.

[12] Android.org. 2019. The Neural Networks API Provides Accelerated Computation and Inference for Machine Learning Frameworks. https://developer.android.com/about/versions/oreo/android-8.1#nnapi.

[13] Android.org. 2019. The Slow Rendering of Android. https://developer.android.com/topic/performance/vitals/render.

[14] Android.org. 2019. The Source Code of Android Watchdog. https://android.googlesource.com/platform/frameworks/base.git/+/android-4.3_r2.1/services/java/com/android/server/Watchdog.java.

[15] Android.org. 2019. Why Performance Matters? https://developers.google.com/web/fundamentals/performance/why-performance-matters.

[16] Ravi Bhandari, Akshay Uttama Nambi, Venkata N Padmanabhan, and Bhaskaran Raman. 2020. Driving Lane Detection on Smartphones using Deep Neural Networks. *ACM Transactions on Sensor Networks* 16, 1 (2020), 1–22.

[17] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. 2000. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In *Proceedings of ACM CHI*. 297–304.

[18] Marco Brocanelli and Xiaorui Wang. 2018. Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps. In *Proceedings of ACM EuroSys*. 6.

[19] Nam Bui, Anh Nguyen, Phuc Nguyen, Hoang Truong, Ashwin Ashok, Thang Dinh, Robin Deterding, and Tam Vu. 2020. Smartphone-Based SpO2 Measurement by Exploiting Wavelengths Separation and Chromophore Compensation. *ACM Transactions on Sensor Networks* 16, 1 (2020), 1–30.

[20] Yizong Cheng. 1995. Mean Shift, Mode Seeking, and Clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17, 8 (1995), 790–799.

[21] Edward G Coffman, Melanie Elphick, and Arie Shoshani. 1971. System Deadlocks. *Comput. Surveys* 3, 2 (1971), 67–78.

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of ACM SIGKDD*. 226–231.

[23] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale Analysis of Framework-specific Exceptions in Android Apps. In *Proceedings of ACM/IEEE ICSE*. 408–419.

[24] Lu Fang, Liang Dou, and Guoqing Xu. 2015. PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification. In *Proceedings of ECOOP*. 296–320.

[25] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write Amplification Analysis in Flash-based Solid State Drives. In *Proceedings of ACM SYSTOR*. 10.

[26] Junxian Huang, Feng Qian, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. Screen-off Traffic Characterization and Optimization in 3G/4G Networks. In *Proceedings of ACM IMC*. 357–364.

[27] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New phytologist* 11, 2 (1912), 37–50.

[28] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. 2015. Boosting Quasi-asynchronous I/O for Better Responsiveness in Mobile Devices. In *Proceedings of USENIX FAST*. 191–202.

[29] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Proceedings of USENIX ATC*. 309–320.

[30] Xinxin Jin, Peng Huang, Tianyin Xu, and Yuanyuan Zhou. 2016. NChecker: Saving Mobile App Developers from Network Disruptions. In *Proceedings of ACM EuroSys*. 22.

[31] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of USENIX FAST*. 273–286.

[32] Kisung Lee and Youjip Won. 2012. Smart Layers and Dumb Result: IO Characterization of An Android-based Smartphone. In *Proceedings of ACM EMSOFT*. 23–32.

[33] Joseph Lee Rodgers and W Alan Nicewander. 1988. Thirteen Ways to Look at The Correlation Coefficient. *ASA The American Statistician* 42 (1988), 59–66.

[34] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of ACM/IEEE ICSE*. 1013–1024.

[35] Youyou Lu, Jiwu Shu, and Wei Wang. 2014. ReconFS: A Reconstructable File System on Flash Storage. In *Proceedings of USENIX FAST*. 75–88.

[36] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Proceedings of USENIX FAST*. 257–270.

[37] Gale L Martin and Kenneth G Corl. 1986. System Response Time Effects on User Productivity. *Behaviour & Information Technology* 5, 1 (1986), 3–13.

[38] Motorola.com. 2019. Motorola Android System. https://www.motorola.com/us/software-and-apps/android.

[39] David T Nguyen. 2014. Improving Smartphone Responsiveness Through I/O Optimizations. In *Proceedings of ACM UbiComp*. 337–342.

[40] Oneplus.com. 2019. Oneplus OxygenOS. https://www.oneplus.com/oxygenos.

[41] Oneplus.com. 2019. Overview of OnePlus 6T. https://www.oneplus.com/6t?from=head.

[42] Thanaporn Ongkosit and Shingo Takada. 2014. Responsiveness Analysis Tool for Android Application. In *Proceedings of ACM DeMobile*. 1–4.

[43] Oppo.com. 2019. Overview of OPPO Reno Z. https://www.oppo.com/ae/smartphone-reno-z/.

[44] Stan Park and Kai Shen. 2012. FIOS: A Fair, Efficient Flash I/O Scheduler.. In *Proceedings of USENIX FAST*. 13–13.

[45] David MW Powers. 1998. Applications and Explanations of Zipf's Law. In *Proceedings of ACL NeMLaP3/CoNLL*. 151–160.

[46] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in The Wild. In *Proceedings of USENIX OSDI*. 107–120.

[47] Redhat.org. 2019. Write Amplification Mitigation. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Storage_Administration_Guide/ch02s04.html.

[48] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A Vector Space Model for Automatic Indexing. *ACM Communications* 18, 11 (1975), 613–620.

[49] Samsung.com. 2019. Performance of Samsung Galaxy S10. https://www.samsung.com/us/mobile/galaxy-s10/performance/.

[50] Samsung.com. 2019. Samsung One UI 2.0. https://www.samsung.com/global/galaxy/apps/one-ui/.

[51] SensorTower.com. 2019. Top Apps Worldwide for Q1 2019. https://sensortower.com/blog/top-apps-worldwide-q1-2019-downloads.

[52] Kai Shen and Stan Park. 2013. Flashfq: A Fair Queueing I/O Scheduler for Flash-based SSDs. In *Proceedings of USENIX ATC*. 67–78.

[53] Ben Shneiderman. 1984. Response Time and Display Rate in Human Performance with Computers. *Comput. Surveys* 16, 3 (1984), 265–285.

[54] Amit Singhal et al. 2001. Modern Information Retrieval: A Brief Overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[55] Statista.com. 2019. Leading Android App Categories in the United Kingdom 2017. https://www.statista.com/statistics/516297/android-app-categories-uk/.

[56] Statista.com. 2019. Time Spent on Mobile App Categories in the U.S. 2019. https://www.statista.com/statistics/248343/distribution-of-time-spent-ios-and-android-apps-by-category/.

[57] Kiri Wagstaff, Claire Cardie, Seth Rogers, Stefan Schrödl, et al. 2001. Constrained K-means Clustering with Background Knowledge. In *Proceedings of ACM ICML*. 577–584.

[58] Yan Wang and Atanas Rountev. 2016. Profiling The Responsiveness of Android Applications via Automated Resource Amplification. In *Proceedings of IEEE/ACM MOBILESoft*. 48–58.

[59] Xiaomi.com. 2019. Xiaomi MIUI. https://en.miui.com/.

[60] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Bo Long, Yuanchao A. Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. 2019. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proceedings of ACM MobiCom*. 1–15.

[61] Shengqian Yang, Dacong Yan, and Atanas Rountev. 2013. Testing for Poor Responsiveness in Android Applications. In *Proceedings of IEEE MOBS*. 1–6.

[62] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying Tests to Validate Exception Handling Code. In *Proceedings of IEEE ICSE*. 595–605.