How Developers Detect and Fix Performance Bottlenecks in Android Apps

Mario Linares-Vásquez, Christopher Vendome, Qi Luo, Denys Poshyvanyk
The College of William and Mary, Williamsburg, VA, USA
{mlinarev, cvendome, qluo, denys}@cs.wm.edu

***Abstract*—Performance of rapidly evolving mobile apps is one of the top concerns for users and developers nowadays. Despite the efforts of researchers and mobile API designers to provide developers with guidelines and best practices for improving the performance of mobile apps, performance bottlenecks are still a significant and frequent complaint that impacts the ratings and apps' chances for success. However, little research has been done into understanding actual developers' practices for detecting and fixing performance bottlenecks in mobile apps. In this paper, we present the results of an empirical study aimed at studying and understanding these practices by surveying 485 open source Android app and library developers, and manually analyzing performance bugs and fixes in their app repositories hosted on GitHub. The paper categorizes actual practices and tools used by real developers while dealing with performance issues. In general, our findings indicate that developers heavily rely on user reviews and manual execution of the apps for detecting performance bugs. While developers also use available tools to detect performance bottlenecks, these tools are mostly for profiling and do not help in detecting and fixing performance issues automatically.**

***Index Terms*—Performance, Bottlenecks, Developers, Android**

I. Introduction

Nowadays, the performance of mobile apps, which are rapidly evolving, is one of the top concerns for users and developers. The official markets of the top three mobile platforms (i.e., Android, iOS, and Windows Phone) hosted 1.43M [32], 1.4M [31], and 300K [33] active apps in 2014, respectively. While the apps from Google Play and the Apple Store have been downloaded more than 125 billion times, performance bottlenecks are still among significant and frequent complaints that impact apps' ratings and chances for success. For example, a recent study by Liu *et al.* [59] reported that 11K+ out of 60K Android apps have suffered or are suffering from performance bugs. Also, Khalid *et al.* [55] analyzed 6K+ user reviews of iOS apps and found that unresponsiveness and heavy resource usage are among the major reasons for the negative user reviews. Despite the efforts of researchers [51], [59], [61] and mobile API designers in providing developers with guidelines and best practices [2], [35], [34] for improving the performance of mobile apps, there is still a significant gap between research and practice in terms of dealing with performance issues by developers in the wild [59], [55].

Our work is empirical in nature and is aimed at filling this important gap and at increasing the understanding of current performance related practices by developers of Android apps. While previous empirical studies [51], [59], [61] focused on understanding and categorizing performance bugs, in this paper, we analyze *real practices* that are followed and *actual tools* that are used by developers to fix performance related bugs. In the context of our study, we surveyed 485 contributors of open source Android apps and libraries hosted on GitHub, inquiring about their practices and tools for detecting and fixing performance bottlenecks; then, we manually analyzed the repositories (i.e., bugs from the issue trackers and commits from the code change histories) of their apps to study real performance bottlenecks and fix-inducing commits to investigate actual strategies followed by developers to deal with performance related issues.

This paper makes the following noteworthy contributions:

- To the best of our knowledge, this is the first study aimed at analyzing *real practices* of open source developers for detecting and fixing performance issues in Android apps;
- The study provides an overview and key insights into types of performance related issues faced by developers as well as prevalent practices and tools used to deal with performance bugs and bottlenecks. The results also reveal current performance related needs of developers that can be used to drive future efforts of researchers;
- Our findings complement previous studies of performance bottlenecks in Android apps by providing the viewpoint of real developers via analysis of their practices;
- We provide extensive online appendix that includes the anonymized answers collected from the survey, the list of tools reported by the participants, and the examples of real performance bottlenecks in Android apps [10].

In summary, our findings demonstrate that the preferred practices of open source Android developers for detecting performance bottlenecks are those related to manual execution of apps and analysis of both user reviews and bug reports. Also, manual execution of apps is usually accompanied by using specific tools (e.g., Traceview, Eclipse MAT) or add-hoc practices (e.g., adding debugging statements with time measurements) that support observation-driven analysis of performance problems. Surprisingly, the results point to the fact that developers do not rely on existing static analysis tools for detecting performance bottlenecks. Although a large number of tools for dynamic analysis are available, these tools neither support automatic detection of performance bottlenecks nor provide developers with suggestions about how to fix the bottlenecks. Regarding the practices for fixing performance bottlenecks, the most frequent practices are the usage of multi-threading to avoid lengthy operations in the main thread, GUI optimizations for reducing the complexity of the UI, and caching to avoid redundant, blocking, or time consuming resource accesses.

II. Design of the Study

The *goal* of the study is to identify current practices of Android developers for detecting and fixing performance bottlenecks in Android apps. In particular, we selected open source Android apps hosted on GitHub and the developers contributing to those apps as the *context* for this study. Our decision to focus on open source apps and developers is based on the fact that we wanted to identify the state of practice and have access to the revision history of their Android apps for further analysis. The choice of GitHub is motivated by the fact that this is the largest repository of Android apps. Consequently, we aim at answering the following RQs:

- **RQ$_1$:** *What practices are used by Android developers to detect performance bottlenecks?* This question aims at identifying current practices, strategies, and sources used by developers for identifying performance bottlenecks;
- **RQ$_2$:** *What tools are used by Android developers to detect performance bottlenecks?* This question focuses on studying the tools used by developers in practice;
- **RQ$_3$:** *What practices are used by Android developers to fix performance bottlenecks?* This question aims at building a taxonomy of practices and investigating specific practices for fixing performance bottlenecks.

The goal of this study goes beyond identifying the practices and tools. We also categorize the practices in a taxonomy and present them in such a way that (i) researchers can analyze the state-of-the-practice to propose new approaches and tools, and (ii) practitioners can use these findings as a reference guide.

A. Data Collection

To answer the research questions, we designed an online survey with the questions (SQs) listed in Table I. SQ$_1$ to SQ$_3$ are for collecting demographic background, filtering participants with short or over claimed experience in Android development, and understanding the diversity of our sample. SQ$_4$ to SQ$_7$ were designed to answer our RQs. In particular, SQ$_4$ and SQ$_5$ were designed to answer **RQ$_1$**; SQ$_6$ is related to **RQ$_2$**; and SQ$_7$ serves to answer **RQ$_3$**.

The survey was sent to GitHub contributors of open source Android apps. To select the potential participants, we first extracted all of the Android projects on GitHub by first identifying all Java projects (381,161) through GitHub's API and locally cloning the projects. We searched the local repositories for the presence of an *AndroidManifest.xml* file, required by Android apps. We refined this list to the projects where the manifest file was at the top-level. To select active projects and to prevent duplicate projects, we filtered projects with at least one fork, star, or watcher and that were not a fork.

After filtering, we found 16,331 potential repositories from which we extracted the contributors. We identified the developers that contributed changes to the apps, and we merged email addresses that occurred across multiple projects to avoid duplicates. Subsequently, we filtered invalid email addresses and removed accounts of Developers at Google to avoid developers that work on Android's core operating system, which is beyond the scope of our study. Then, we emailed the survey to 24,340 email addresses from which we got 5K+ messages reporting "undelivered message" and 628 survey responses. From those responses, we filtered the invalid ones and those ones from the participants who indicated less than one and more than ten years of experience in Android development (note that the first two internal alpha releases of Android were available in late 2005 - early 2006). In the end, we obtained 485 valid responses. On average, the participants had 12 years of programming and three years of Android development experience (Fig. 1 (a)). Concerning the academic level, 45.57% of the participants had only a B.S. degree and 38.76% had a M.S. degree (Fig. 1 (b)).

Table I
Survey questions

Id	Question (Type)
SQ$_1$	For how many years have you been programming? (Numeric)
SQ$_2$	For how many years have you been developing Android apps? (Numeric)
SQ$_3$	What is your academic level? (Single-choice: *High school* | *Bachelor* | *Master* | *PhD* | *PostDoc*)
SQ$_4$	How do you detect performance bottlenecks in mobile Apps? (Multiple-choice: *App reviews/issue reports* | *Manual testing* | *Other strategies* | *Tools*)
SQ$_5$	What other strategies (if any) do you use for detecting performance bottlenecks? (Open)
SQ$_6$	What tools (if any) do you use to detect performance bottlenecks? (Open)
SQ$_7$	What strategies/practices do you use for improving performance bottlenecks in mobile apps? (Open)

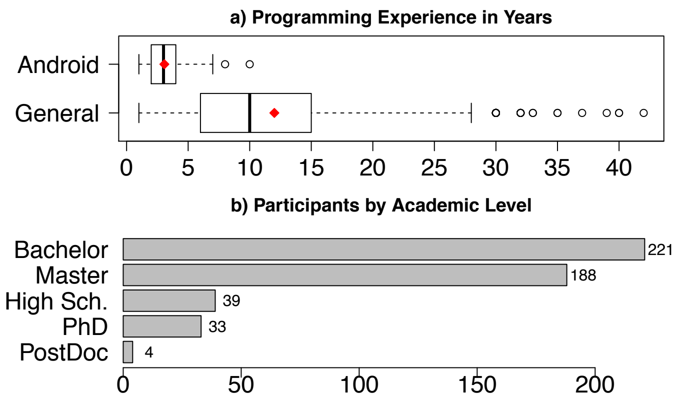

Figure 1. Programming experience of survey participants

B. Analysis Method

The answers to SQ$_1$ to SQ$_4$ were analyzed using descriptive statistics. In the case of the open answers (SQ$_5$-SQ$_7$), we categorized the answers manually by following a grounded theory-based approach [37]. Three of the authors went through all of the free-text answers and performed one round of open coding by independently creating categories for the answers. After the round of open-coding, the codes were standardized. In the cases of no-agreement between the three coders, corresponding answers were marked as "Unclear". Then, we used the categories to build a taxonomy of practices for improving performance bottlenecks in Android apps. Our findings for the

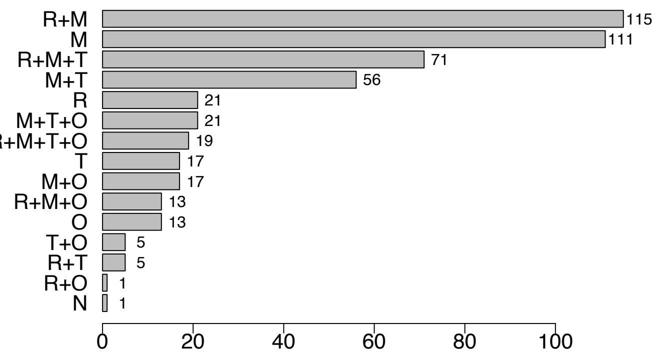

Figure 2. How developers detect performance bottlenecks in mobile Apps (**R**eviews/issue reports | **M**anual testing | **T**ools | **O**ther strategies | **N**one)

practices (**RQ**$_1$) and tools (**RQ**$_2$) for detecting performance bottlenecks are presented in Sections III and IV, respectively. The practices for improving performance bottlenecks (**RQ**$_3$) are discussed in Section V.

In SQ$_4$ of the survey, we asked the participants to voluntarily type the URLs of the GitHub repositories to which they previously contributed. We manually inspected the change histories in these repositories by looking for commit messages and issue reports related to performance bottlenecks. Then, we inspected the commits related to fixing performance issues in order to identify the concrete strategies used by developers. The results (i.e., examples and detected practices) of this qualitative analysis are presented also in Section V.

C. Threats to Validity

The results of our study cannot be generalized to developers of all mobile platforms. Our focus is only on Android developers. Another threat to the *external* validity is that the sample of surveyed developers may not be statistically representative of the Android developers community. However, as showed in Fig. 1, our sample of 485 participants is diverse in terms of programming experience. Additionally, we focused on open source Android apps hosted on GitHub. Thus, we cannot generalize our findings to all Android developers, since the findings may not necessarily apply to developers working on commercial apps. It might be possible that developers working for commercial apps use different tools and follow best practices. To avoid selection bias, we only filtered clearly invalid email addresses and Google developers, which would be working on the operating system and are out of scope. In terms of threats to *construct* validity (i.e., the relationship between theory and observation), they are mainly related to the measurements that we performed in our study. To avoid potential issues with construct validity, we filtered answers provided by developers with less than one year of experience and more than ten years of experience in Android programming (as explained in Section II-C) to avoid invalid and non-informative answers. Also, because our survey included free-text answers, we coded the answers using Grounded Theory (Section II-B). In particular, we discarded the answers when agreement was not achieved between the three coders.

III. RESULTS FOR **RQ**$_1$: PREFERRED PRACTICES FOR DETECTING BOTTLENECKS

Fig. 2 shows the distribution of practices followed by the surveyed developers. The figure caption explains the labels used for the bars, e.g., R+M means *Reviews and Manual testing*. In general, responses indicate that developers rely on a diverse set of practices and tools for detecting performance bottlenecks in Android apps. However, we also observed that when using tools, they were inclined toward dynamic analysis based ones. For instance, the combination of manual testing with analysis of user reviews is the most preferred practice by 23.66% of the respondents. Manual testing as a standalone practice was the top-2 practice (22.85%); and the combination of analysis of reviews, manual testing, and relying on existing tools was the top-3 practice (14.61%). The next preferred practice was the combination of manual testing and existing tools (11.52%). The tools reported by the participants are analyzed in Section IV.

Manual testing was mentioned in all top four combinations of practices and it accounts for 353 answers (72.63%). The analysis of the reviews appears in all top five practices accounting for 207 answers (42.59%). When looking into the results while considering each practice separately, the results suggest the same order of preferences: manual testing (87.04%), analysis of user reviews (50.41%), existing tools (39.92%), and other strategies (18.32%).

89 answers to the SQ$_4$ "Other strategies" were expanded upon with textual descriptions in SQ$_5$ (in a free response field). Among these answers, we discarded 15 responses that were ambiguous with respect to the strategies for detecting bottlenecks (e.g., "'Reasoning","If it's slow, I improve it"). We also omitted the answers where respondents claimed that they do not care about improving performance. In the remaining valid answers (74 answers), the most prominent strategy was the one relying on instrumenting the code with debugging statements and analyzing the logs (31 answers); 14 out of those 31 answers mention logging time measurements. The second most used strategy was code inspections (11 answers) followed by checking apps on different devices with low specifications and different OS versions. The application of best practices and good design principles are also claimed, but as a prevention (instead of detection) strategy in six answers. Different types of testing, mostly automated, account for only nine answers including unit testing, mock-based testing, stress testing, and end-user testing. Observation-based practices such as profiling, debugging, and monitoring were mentioned in ten responses; however, these answers overlap with the answers for the tools that were used. Algorithmic analysis was mentioned only in two cases and GUI overdraw checking was mentioned only once.

Answer to RQ$_1$. Open source Android developers primarily rely on *manual testing* and *analysis of the reviews* for detecting performance bottlenecks. Manual execution of apps is also accompanied by tools that help measure and visualize performance-related measurements (e.g., execution time

Table II
TOOLS USED BY ANDROID DEVELOPERS TO DETECT PERFORMANCE-RELATED ISSUES

Purpose	**Tool / Suite**
Profiling-Monitoring	MAT, TraceView, DDMS, Android Device Monitor, ADT, ADB, Strict Mode API, OpenGL Tracer, ADB profiler, Android Debug API, Heapdump, Surface Flinger, Hprof, Dumpsys, Dmtracedump, Eclipse profiler, Systrace, Trepn, Valgring/Calgrind, Intel Studio, AT&T ARO, Emmagee, Procrank, Chrome trace tool, NewRelic, Adreno profiler, IntelliJ profiler, LittleEye, Netbeans Profiler, Nvidia Nsight, Perf, VisualVM, High Performance C++ profiler, Top, Linux process monitor, Acra, Charles Network Analyzer, Crashlytics
Testing	Roboelectric, JUnit, Appium, Robotium, Xdotool+Geniemotion, Robospock, Monkey, MonkeyRunner, Cloud-services
Debugging	Eclipse debugger, GDB, JDB, SQLLITE Explain Query Plan, Nvidia Nsight
Static Analysis	Sonar, Lint, FindBugs, PMD
GUI-based analysis	Hierarchy Viewer, Android overdraw visualizer, Surface flinger, OpenGL tracer
Analytics	GoogleAnalytics, MintSplunk, Crashlytics, NewRelic
Logging	Logcat, Custom logging systems
Other	Jenkins, Git Bisec, Android Battery Stats, Caliper

and memory). Despite the availability of tools for profiling performance-related measurements, a number of practitioners still rely on manually inserting statements in the code to measure and print execution time and memory consumption.

IV. RESULTS FOR $\mathbf{RQ_2}$: TOOLS FOR DETECTING BOTTLENECKS

194 participants selected the option "Tools" in SQ_4 and expanded the answer with a textual description for SQ_6. We categorized the tools according to their purpose as well as whether these tools are provided and/or supported by Google or third-party vendors. Table II lists the tools grouped by their purpose. 156 participants reported tools from the Android ecosystem and 54 participants reported third-party tools (note that some of the respondents reported both Android and third-party tools in the same answer). However, the reported tools were mostly used for observation-based analysis, which encompasses tasks such as profiling, monitoring, testing, and debugging. Only five participants mentioned using static analysis tools (i.e., Lint, SONAR, FindBugs, and PMD).

Fig 3 depicts Android tools used by the developers (see frequencies in appendix [10]). The participants referred to specific tools (e.g., Traceview [45] and Memory Analyzer Tool, MAT [4]) as well as tool suites such as Android Device Monitor [41] or Android Debugger Bridge [38]. A total of 30 tools and suites from Android ecosystem were mentioned. For instance, Traceview is the top used tool (34 answers) followed by MAT (20 answers). Next top three Android tools are DDMS [49] (17 answers), Systrace [48] (16 answers), and Logcat [43] (11 answers). In addition to the survey answers, one respondent replied with an email providing a listing of the tools that she uses for improving apps performance [68].

Fig 4 depicts the third-party tools used by the participants. In 54 answers mentioning third-party tools, we found 47 different tools mostly used for profiling, monitoring, debugging, and testing purposes. However, the set of tools in responses

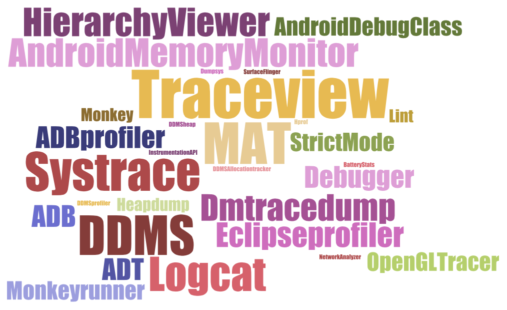

Figure 3. Word cloud of Android tools

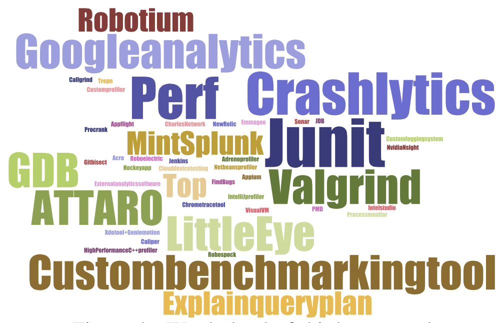

Figure 4. Word cloud of third-party tools

was diverse enough in that 34 participants mentioned a unique tool that was not utilized by any other participant. The top most prevalent third-party tools are JUnit (4), Perf (4), Crashlytics (4), Valgrind/Callgrind (4), and LittleEye (3).

Answer to $\mathbf{RQ_2}$. Although there is a diverse set of available tools, Android developers mostly rely on tools for performance profiling and debugging of their apps; most of the tools are from Google, as expected. There is a preference towards tools supporting observation-based analysis, and most of the tools do not support automatic detection of bottlenecks. Only a few tools support automatic detection of a limited number of types of performance bottlenecks (Google's StrictMode Android API and PerfChecker tool [59]), yet these tools are not widely used. Static analysis tools are used less frequently and only in a few cases for detecting performance optimizations.

V. RESULTS FOR $\mathbf{RQ_3}$: PRACTICES FOR IMPROVING PERFORMANCE BOTTLENECKS

For SQ_7, we obtained 410 free-text answers. We discarded 90 answers coded as "Unclear" and 25 answers in which the developers stated they do not need or care about improving performance. In the end, we obtained 72 codes identifying individual practices for improving performance bottlenecks. These 72 codes were further categorized into 16 groups by considering the type of practice (e.g., threading) or the goal (e.g., memory management). Finally, these 16 groups were linked to the following types of high-level categories: *GUI lagging*, *Memory bloat*, *Energy leak*, *General-purpose*, *Unclear-benefit*. The first three categories represent performance bugs defined by Liu *et al.* [59]; thus, the groups linked to these categories represent practices aimed at solving corresponding bug types (e.g., GUI optimization aims at reducing "Application Not Responding-ANR" errors and GUI lagging). The last two categories are for general-purpose cases (e.g., profiling) and

Table III
TAXONOMY OF PRACTICES FOR IMPROVING PERFORMANCE BOTTLENECKS: CATEGORIES, GROUPS, AND ANSWER FREQUENCIES

Category	Group (frequency)
GUI lagging	Threads (74), GUI optimization (47), Caching (40), Memory management (32), Network resources(17), Avoid blocking operations (5), Delegation(cloud) (4), On-demand resources(2), Others (3)
Memory bloat	GUI optimization (45), Memory mgmt. (30), Network resources(5), On-demand resources(2), Etc(2)
Energy leak	Delegation (4), Reduce GPS calls (1), Wakelock (1)
General	Design(74), Optimizations (68), Observation-driven analysis (61), Testing (24), Native code (13), APIs (10), DB-optimizations (5)
Unclear benefit	No reflection(1), Depend. injection(1), Recursion(1)

when the perceived benefit is unclear. The complete taxonomy (codes, groups, and categories) is in our appendix [10].

It is worth noting that some answers mentioned more than one practice; in those cases, the coders assigned more than one code to the answer: For example, the following answer was coded with the practices *Wakelock*, *GC*, *Separate-thread*, *Data-structures*, *Logic*, *Profiling*:

"WakeLock usage, since energy is very important. Reducing garbage collection. Android Debug class and Eclipse MAT could be useful. One should not block main UI thread but it happens... In case of large data, IPC (even AIDL) causes too many garbage collection, which lags UI. Proper data structure choice: e.g., just plain array instead of LinkedList<T> or ArrayList<T> whenever possible. Proper algorithm choice: e.g., most of the time, just plain binary search should do... complicated algorithms work best for complicated problems. We should truly understand an algorithm before actually using it. Being mindful about dynamically allocated objects while coding helps"

Another answer noted combination of practices not only for improving responsiveness but also memory consumption:

"Background threads for CPU/IO intensive operations; memory allocations outside of loops; caching onerous computations/io accesses; recycling of views (esp. Android listviews)"

As the answers illustrate, the set of practices used by developers is diverse. Table III summarizes the taxonomy of practices. In particular, the table lists the high-level categories, the groups of practices linked to the categories, and the number of answers including a practice in the groups oriented to the corresponding category. For example, 32 answers mention memory management practices aimed at reducing *GUI lagging*, and 30 answers mention memory management practices oriented to avoid *Memory bloat*. There is also the case of practices that are double-purpose, such as recycling/compressing resources that are used to reduce *GUI lagging* and *Memory bloats*. In the following, we describe the most representative practices grouped by the high-level categories in our taxonomy.

A. Improving GUI lagging

Lengthy operations can impact app responsiveness and smoothness and result in GUI lagging or ANR (Application Not Responding) crashes. Both scenarios are usually triggered by blocking operations that run in the UI thread. In fact, threading-related practices are the most frequent way used by developers to deal with GUI lagging and ANRs. The following answer illustrates this case:

"As far as Android is concerned, improving performance is generally easiest to do by way of reducing memory consumption rather than raw algorithmic performance. GC runs are expensive, and avoiding them is key to smooth performance. Also, many inexperienced Android devs simply do too much work on the main thread; most app sluggishness isn't caused by CPU spikes at all, but rather, the fact that these spikes occur on the main thread instead of a background thread. Using Handlers to shift work off the main thread generally results in much better app performance, and having sane a threading model and smart memory allocation generally does much more to increase performance than "tweaking" algorithms for speed. "

In general, "threading" practices were mentioned 74 times, and the main purpose is to separate lengthy or blocking operations from the UI thread by implementing worker threads. Those worker threads are implemented by the participants using background services [3], asynchronous tasks [39], or threads carefully managed using a `Handler` [42].

An example of the threading practice can be found in the repository of the *Sgtpuzzle* app [22], which is an app containing 38 single-player logic games. One project contributor created an issue based on an old report concerning particular game taking an exceptional amount of time to cancel the generation, and identified the problem as *"The cancel button sets a flag which the game generation loop checks, because it is not possible to safely stop a thread. Pearls just needs a few more checks of this flag adding to its generator,"* [27]. The developer indicated that the single-threaded design created difficulty to reduce the cancellation time by stating, "*Generating a game is a long and complex operation that has to have regular cancel-checks because it can't be easily killed without killing a whole process. So let's move it to a separate process that can be killed if a cancel is requested*" [27]. In particular, the developer modified the class `SGTPuzzles` by adding several methods: `killGenProcess`, `startGameGenProcess`, `startNewGame`, `startGameThread`, `waitForProcess` and `generateGame`; additionally, the author had to modify native code and all the changes spanned four commits [25], [24], [23], [26]. The fix is not simply a thread creation, but managing the threads (i.e., waiting for the threads to prevent a memory leak in memory, storing process IDs, ensuring successful thread exit) and ensuring that the underlying C code accommodated the new environment. The developer confirmed that the multi-threaded implementation was working and closed the ticket [28].

Another frequent group of practices (47 answers) are GUI optimizations, which aim at reducing complexity of the GUI and optimizing rendering (especially for games and animations). Two of the participants described the practices as "*simplifying the view and decreasing overdraw*", and "*Is the* *problem poor display performance? Analyze the view stack and find ways to simplify or eliminate views and view layers*". In fact, the most used GUI optimizations are simplifying views layouts, minimizing the number of views used in the screens, and reducing overdraw [50]. Surprisingly, although reducing overdraw has been promoted by Google as one of the best practices for optimizing GUIs and performance, only 7 answers mentioned the practice. Having flat UIs and reducing/reusing views (i.e., GUI components extending the `View` class) is a desired practice because of several reasons such as: (i) the `Activity.findViewById` method has been described by a Google Engineer as an expensive function [11], and the method was detected as an energy greedy API [58]; (ii) overdrawing pixels in the screen is expensive because the GUI components in Android apps are redrawn frequently. Other GUI optimizations are more specific for apps requiring rendering; in this case, the reported practices include reusing bitmaps, reducing texture pages and frame sizes, applying sprite batching, and reducing screen updates (e.g., *"lower draw latency to keep 60 fps"*). Concerning tools for detecting bottlenecks related to the GUI, some participants reported the usage of HierarchyViewer, the "Show GPU overdraw" setting in Android devices, and the `adb shell dumpsys gfxinfo` command for profiling GPU rendering.

One example of GUI lagging generated by screen update frequency can be found in the *PresureNet* app, which automatically collects pressure measurements using barometers in Android devices and provides weather forecast based on the collected data [20]. A user reported a GUI lagging: "*When zoomed out and panning around, pressureNET lags due to the large number of measurements contained in the view. Improve the performance and make all map movement snappy*" [19]. To solve the issue, the developer changed the code of the method `BarometerNetworkActivity.addDataToMap` to reduce the refreshing frequency of the map [18] and later commented that the fix significantly improved the performance.

Developers also avoid rendering every pixel on the screen because this is expensive for apps relying on rendering. One specific mechanism provided by OpenGL is called face culling, which ignores the back face of a shape during the rendering to save time and memory [17]. The following response mention that practice:

> *"If the bottleneck involves graphical performance problems perhaps the app could make use of hardware accelerated functionality or change the algorithm to "cheat", i.e. you don't need to draw every single pixel in an animation. Maybe just skip every second update value and the performance will improve and the user will never notice."*

> *"Optimizing the rendering (culling, batching, caching). Use threads for computationally intensive operations"*

Caching techniques are also useful to avoid GUI lagging and ANRs. 40 answers mentioned this type of practice that aims at avoiding redundant blocking/time consuming operations related to resources access. Caching is not only used for local resources (e.g., images or database queries), but also for resources obtained from external providers like in a network response as expressed in the following answers: *"caching onerous computations/io accesses"; "Caching network responses and images are normally the best thing I can do for performance";"Optimise using caches and/or inline functions which are executed more often";"Also, requesting data only when needed (and use a local cache with timestamps) reduces the number of "big" data requests"*. Concerning caching specific mechanisms, developers use their own-solutions based on memory data structures and Android API mechanisms [40], [46] or specialized caching such as "sprite caches" [12] for 2D images/animations in games.

B. *Improving Memory bloats*

Excessive memory consumption in mobile apps can cause "Out Of Memory" (OOM) errors [59] or frequent Garbage Collection (GC) events (i.e., automatic invocations) that are triggered when the allocated memory goes beyond a predefined threshold. The latter can also impact responsiveness, because GC events can stop apps from executing in order to free memory. Developers improve memory-related issues by relying on several practices including GUI optimization, memory management, source code optimizations (e.g., avoid instantiating objects in loops), delegating operations to remote services, and invoking resources on demand.

The respondents confirmed that GUI optimizations are not only useful for avoiding GUI lagging but also for reducing the memory allocated by the GUI:

> *"Most of the apps I have developed have memory related issues. I usually solve them by image compression techniques, reusing bitmaps and caching."*

We found an example of how to avoid memory bloats in the repository of the GitHub app for Android [9]. The issue 513 [7] reported a performance problem that was exhibited when displaying the avatar image (bitmap). A developer reviewed the source code of the class `AvatarLoader` and found that the bitmaps were loaded in the app with their original sizes (i.e., the same used in the web version of GitHub). Hence, some images were large enough to generate memory bloats. A hot-fix was committed aimed at saving memory by resizing the image before rendering it in the GUI [6]. The developer later confirmed that the commit fixed the issue [8]. Recycling images is another practice reported by developers that helps to reduce memory consumption. Recycling is mostly done in Android apps by calling the `Bitmap.recycle` method [44].

In general, memory-management practices reported by the developers aim at reducing memory allocations and avoiding GC events as described in these responses:

> *"Due the bottleneck be the rendering, and the creation of new objects(memory allocation), we improved the performance using an object pool so that we could reuse objects"*

> *"But there is possible few solution patterns : [...] background loading of heavy data or memory allocation improving (reduce count of new objects, e.g., using different*

object pools, especially for big objects like bitmaps, to reduce load on GC) [...]"

Thus, good programming practices such as reusing objects, avoiding object instantiations in loops, and using appropriate data structures, are common practices reported by the participants since excessive memory allocations trigger GC events that can drastically impact performance of Android apps.

C. Energy leaks

Although the survey did not ask specifically about energy leaks, two participants mentioned that wakelocks and GPS calls impact the memory consumption. Both are pretty well-known in the research community as threats to increases energy consumption and have been previously analyzed by researchers [62], [67]. In addition to reducing wakelocks and GPS calls, we included service delegation as a practice for reducing energy consumption (see Table III). Executing expensive computations on remote servers reduces execution time and resource consumption, which can also lead to reduction in the energy drawn from the battery.

D. General-purpose practices

We used this category for answers describing (i) high-level methodologies or observation-driven practices (e.g., *"Profiling performance and memory using Android and IDE tools."*), and (ii) practices that can improve performance without focusing on specific performance bugs (e.g., APK optimization, improving algorithm implementation). In 74 cases, participants mentioned "design", "best practices", "guidelines", "design patterns", "refactoring", and crowd-based documentation (e.g., Stack Overflow) as a method for preventing performance bottlenecks or finding a solution when a bottleneck is detected. In 61 cases, observation-based practices were described or mentioned; these practices include profiling, code reviews, debugging, cross-platform checks, algorithm analysis.

Source code optimizations were reported 68 times. This group of practices includes improving implementation of the algorithms (22 answers), using micro-optimizations at statement level (22), using relevant data-structures (16), and using static analysis tools for detecting optimization opportunities (5). Two example responses in the "Optimizations" group are:

"Optimizing floating point operations, for loops optimizations, remove unnecessary allocations, use primitive types instead of enums."

"Look at the profile, check where time is spent. E.g. if there's much garbage collection going on with Bigints inside calculation-intensive loop then try to use vanilla unboxed ints or doubles instead of proper rational numbers. Another example is to optimize actual algorithm, replace linear lookups with a hash table or tree-based map (or binary search inside of array, depending on the particulars of the problem at hand). Lastly it's simple tricks to keep in mind, e.g. avoid linked lists if you don't need constant inserts/deletions and use ArrayList instead."

The issue 244 from *Android-Universal-Image-Loader* library shows how micro-optimizations can improve performance [1]. The issue describes a situation when after profiling, a developer using the library found a method `MemoryCacheUtil.generateKey` that appears to be a bottleneck; this method could lag the GUI when it is called from the main thread on each image displayed by the library. In particular, the issue is reported when calling the `String.format` in the `MemoryCacheUtil.generateKey` method. Additionally, the reporter provided a link to a Stackoverflow question that discusses the low performance of `String.format` as compared to `StringBuilder` [29]. As a response to the issue, the repository owner committed a fix replacing the `String.format` calls with `StringBuilder`.

Two more answers suggest that optimizations at compile/build time can improve the performance of apps:

"Optimizing apks, eliminating code not needed in the apks. Reducing the timing. Odexing while realigning the apks."

"look for cache/branch prediction misses, missed loop vectorization.unrolling/peeling. then investigate why the compiler generated poor code and either 1) adjust GCC optimizations to fix the issue, 2) file a bug with GCC and possibly supply a patch, or 3) tune the source code to fix"

The remaining groups in these category describe (i) testing practices as a way to detect bottlenecks, (ii) the usage of native code over Java to improve the performance, (iii) the usage of APIs under the assumption that they are optimized (e.g., *"[...] I keep looking for libraries able to provide a given feature with a good performance."*), and (iv) optimizing SQL queries (e.g., *"If I do any sqlite query, I do split them into small segments."*).

Answer to RQ$_3$. The most frequent practices used by the surveyed developers for fixing performance bottlenecks include the usage of multi-treading to avoid lengthy operations in the main thread, GUI optimizations for reducing the complexity of the UI, caching to avoid redundant or blocking/time consuming resource accesses, memory management to avoid GC events and OOM errors, and source code optimizations.

VI. Lessons Learned

In this study, we analyzed the current practices of open source developers for detecting and fixing performance bottlenecks in Android apps. The results are described as a taxonomy of practices along with specific answers and examples of real bugs and their fixes that we manually analyzed. In addition to the results, we identified the factors that need to be considered carefully by developers, API designers, and researchers. We describe these findings as *lessons learned* and highlight the implications for the community.

Tools. Despite the diverse offering of tools, developers rely mostly on user reviews and manual testing to detect performance bottlenecks. Developers tend to prefer crowdsourcing to detection, because current tools are not enough to detect performance bugs, and the time devoted to testing mobile apps is reduced as reported in [53], [56]. In the case of participants using tools, they mostly rely on tools that provide profiling, monitoring, and debugging capabilities. We suppose that using existing dynamic-analysis based tools may impose significant taxes on developers' time in order to get useful information. For instance, the following response suggests that additional effort is required when using profiling tools:

"I use the core Android tools such as tracedump, the ddms profiler activated in-code, traceview etc to find hotspots, narrowing down the search by visual identification of the trace graph."

Another developer also reported on the issues of the tools:

"Didn't really cover it in the survey, but android's standard profiling tools are absolutely useless and most people I've spoken to about it in the industry resort to println and System.get SystemTimeNanos in a lot of cases!"

While there are tools that help generate performance-related warnings for Android apps (e.g., Lint), only two participants used such tools for detecting potential performance bottlenecks. Another Android-specific mechanism is StrictMode API that allows developers to declare policies in the source code and provides mechanisms for reporting policy violations for in-field app executions [30]. However, only four participants mentioned using such API. Finally, the *RenderScript* framework helps improve app performance through parallelization of computationally expensive tasks across all the available processors in the device [47]. Yet, only one participant mentioned using *RenderScript*. This observation could be an artifact of the sampled participants we had or due to the fact that developers had particular reasons for not using those tools. The main message here is that more research needs to be done to evaluate effectiveness of current tools given the fact that the existing tools are not able to automatically detect hotspots or performance bottlenecks, and more effort from the community should be devoted to develop useful tools that automatically detect the bottlenecks.

Diversity of practices. The identified practices are diverse because performance bugs can manifest themselves in different ways; however, this diversity is also due to the wide adoption of mobile apps in several app domain categories. For instance, some techniques are specific to GUI-related practices for games and apps that involve heavy computations; other techniques are aimed at common issues such as asynchronous tasks, GUI overdraw, executing long operations in the main thread, memory management, among others. Therefore, our observation is that developers in specialized domains (e.g., games) are more familiar with practices for quality and performance assurance of their apps. Consequently, less experienced developers need to pay attention to the best practices promoted by the API designers [2], [35], [34]. Finally, mobile apps are more prone to performance bottlenecks, because of the GUI-centered philosophy and limited resources in the devices [51], and the default single-thread policy of Android apps that inexperienced developers often do not take care of. When the guidelines are not followed carefully, it is rather easy to "induce" certain types of performance bottlenecks such as lengthy operations in the UI thread or memory bloats due to naive mistakes when using images.

Tradeoff between quality attributes. Excessive caching can generate memory bloats or frequent GC events. Also, operations aimed at reducing the size of resources can be lengthy and then impact app's responsiveness. We found a real case illustrating this tradeoff. There is a reported issue in the *Novoda image-loader* library for asynchronous image loading and caching in Android [15]. In the issue, a user suggested to include a setting for disabling bitmap resizing, since resizing bitmaps was expensive [16]. Bitmap resizing usually samples the pixels from an original bitmap to generate a new one with a different size, which requires extra computation and memory. Therefore, the user suggested to disable the resizing when it was not necessary. A developer agreed and added a variable `alwaysUseOrignalSize` to `ConcurrentLoader.load` as a flag to disable resizing. Therefore, some practices for improving performance bottlenecks need to be implemented carefully by developers, because a bug fix can introduce a different performance bug. One opportunity here for research is to provide developers with approaches and tools that automatically analyze the impact of a refactoring or change in performance-related quality attributes.

Suboptimal API usages. Performance bottlenecks can be also introduced by suboptimal usages of the APIs. We found a GUI-related performance issue in the *Mupen64Plus-ae* app, where a user complained that the frame per second (FPS) was very slow in the Sony Xperia Z or newer Android phones while playing any games [14]. To solve this issue, the developers first updated the SDL plug-in [21] to the most recent version (v2.0), but they observed that the latency problem persisted. One user (or developer) offered to help and utilized Logcat, and another developer recommended DDMS as well. Subsequently, they removed the `EGL10.eglWaitNative` and the `EGL10.eglWaitGL` method calls from the `GameSurface.flipBuffers` method [13]. The `GameSurface` class is described in the class header as "[...] a graphical area of memory that can be drawn to" in the source code [63]. The `flipbuffers` method is responsible for swapping the display and the surface buffers. This swap would physically display the drawn surface. However, the two wait calls were used prior to ensuring that both the native rendering calls and the OpenGL rendering calls finished executing; it means that the calls were done before swapping the display [5]. Although, we did not confirm this with the developers, it is likely that these two waits were unnecessary to ensure that the native and OpenGL processes were completed, since the buffers swapped are private fields and only initialized to store the objects (i.e., they are not modified after initialization). After the developers updated the emulator, two users confirmed that there was a noticeable performance improvement in the application.

VII. Related Work

To the best of our knowledge, this is the first paper that analyzes current practices of Android developers to detect and fix performance bottlenecks. However, previous work has been done on understanding and detecting performance bugs in mobile apps and non-mobile apps.

A. Performance Bugs in Mobile Applications

Guo *et al.* [51] developed a static analysis tool, Relda, for detecting resource leaks in Android apps. Relda detects energy and memory leaks as well as exclusive resources (i.e., an infinite wait due to a resource never being released). The approach builds a functional call graph and utilizes a depth-first search analysis to evaluate resource summaries. Then, it evaluates whether resources are released after use or at the function exit points to determine the presence of resource leaks. The paper also provides a classification of resource leaks by causes. Nistor and Ravindranath [61] present an approach, SunCat, which analyzes sequences of calls to `String` getter methods, to understand the impact of larger inputs on the user's perception in Windows Phone apps. The authors identified nine instances of performance problems from 29 scenarios across five Windows Phone apps. Lin *et al.* [57] found, in a study on 104 open source Android apps, that long running operations were extracted from the UI thread into asynchronous tasks only in 46% cases. Then, they propose an approach, ASYNCHRONIZER, to automatically refactor long-running operations into asynchronous tasks.

User reviews and app repositories have been also used as a resource to understand performance bugs in mobile apps. For instance, Khalid *et al.* [55] investigated the user reviews for iOS applications and presented a taxonomy of complaint types. The authors identified 12 different categories of complaints through a manual categorization. Two of the identified categories are specific cases of performance bugs: heavy resources and unresponsive app. Liu *et al.* [59] analyzed 70 real performance bugs (i.e., bug reports, patches, commit logs, and patch reviews) from eight Android applications. As a result, the authors defined the three categories of performance bugs that we used in the taxonomy of practices for improving performance bottlenecks (i.e., GUI lagging, memory bloat, energy leak). Additionally, the authors proposed an approach based on static analysis, namely PerfChecker, to identify two types of performance bugs: lengthy operations in the UI thread and violations of the view holder pattern.

Our work differs from previous work in that we propose a taxonomy of practices and tools for detecting an fixing performance bottlenecks based on the survey with 485 developers. While we manually analyzed bugs extracted from repositories, similarly to [51], [59], our goal was to identify real practices reported by the survey participants. Moreover, the combination of the survey analysis and manual investigation into performance bugs helped us derive actionable insights (Sect. VI).

B. Performance Bugs in Other Types of Applications

The differences between performance and non-performance bugs in other types of apps have been investigated before [66], [65], [60]. Zarman *et al.* [66] analyzed security, performance, and other types of bugs in Firefox; in particular, their work investigates the amount of time it takes to fix the bugs, the number of developers assigned to the bugs, and the characteristics of bug fixes. Afterward, the authors investigated the impact that the bugs had on the end-user, the context in which the bug appeared, the solution to the bug, and validation of the bug fix [65]; that study used a qualitative study on both performance and non-performance bugs in two systems, Firefox and Google Chrome [65]. Nistor *et al.* [60] investigated the difference in the fixing effort for performance and non-performance bugs, the likelihood that fixing these two types of bugs will introduce new functional bugs, and the way in which developers identify performance bugs in JDT, SWT, and Mozilla.

Other work presented approaches for automatic detection of performance bugs. Jovic *et al.* designed an approach for identifying perceptible performance bugs (i.e., lags that can be perceived by users) [54]. Jin *et al.* proposed a rule checker to automatically identify performance bugs after investigating 109 bugs from five systems [52]. Syer *et al.* proposed an approach to diagnose memory leaks by using performance counters and execution logs in order to link execution events to the memory usage at particular time intervals [64]. In situ visualization has also been proposed to support performance bottlenecks understanding in Java apps [36].

Nistor *et al.* found that few performance bugs are detected by using profiling in desktop apps [60]. They also found that many performance bugs are mostly identified by relying on code reasoning instead of direct observation. Our findings highlight that in the case of Android apps, people heavily rely on user reviews, manual execution, and direct observation; also, profiling in Android apps is a frequent practice.

VIII. Conclusion and Future Work

Our key findings can be summarized as the following: (i) Android developers primarily rely on manual testing and analysis of the reviews for detecting performance bottlenecks; (ii) Android developers prefer tools for performance profiling and debugging their apps; (iii) the most frequent practices used by the Android developers for fixing performance bottlenecks include the usage of multi-threading to avoid lengthy operations in the main thread, GUI optimizations for reducing the complexity of the UI, caching to avoid redundant or blocking/time consuming resource accesses, memory management to avoid GC events and OOM errors. The following lessons are emphasized based on our analysis: (i) there is a need for tools that automatically detect performance bottlenecks; (ii) developers need to follow the guidelines and best practices, because mobile apps are more prone to performance problems; (iii) developers need to be careful when following existing practices for fixing the bottlenecks, because there may be a tradeoff between the targeted quality attributes (e.g., responsiveness vs. memory management); and (iv) developers need to use APIs carefully, because suboptimal usage of the APIs can be a source of performance bottlenecks. These findings and lessons represent the main input for our future research agenda designing tools to support optimization and performance improvement of mobile apps.

References

[1] Android-universal-image-loader issue 244. https://github.com/nostra13/Android-Universal-Image-Loader/issues/244.

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

[3] Creating a background service. https://developer.android.com/training/run-background-service/create-service.html.

[4] Eclipse memory analyzer (mat). https://eclipse.org/mat/.

[5] EGL Documentation https://www.khronos.org/registry/egl/sdk/docs/man/html/indexflat.php.

[6] Gh android commit. https://github.com/Blei/android/commit/492c698418cfe71a7971bc3e272816102764d405.

[7] Gh android issue 513. https://github.com/github/android/issues/513.

[8] Gh android pull request 535. https://github.com/github/android/pull/535.

[9] Gh android repository. https://github.com/github/android.

[10] How developers fix performance bottlenecks in android apps – online appendix. http://www.cs.wm.edu/semeru/data/ICSME15-Android-bottlenecks.

[11] How expensive findViewById ??. https://groups.google.com/forum/?fromgroups=#!topic/android-developers/_22Z90dshoM.

[12] Libgdx: Class sprite cache. http://libgdx.badlogicgames.com/nightlies/docs/api/com/badlogic/gdx/graphics/g2d/SpriteCache.html.

[13] Mupen64plus-ae commit. https://github.com/mupen64plus-ae/mupen64plus-ae/commit/262be6de409202c86c70afa971da3a01863320c0.

[14] Mupen64plus-ae issue 80. https://github.com/mupen64plus-ae/mupen64plus-ae/issues/80.

[15] Novoda image-loader. https://github.com/novoda/image-loader.

[16] Novoda issue 26. https://github.com/novoda/image-loader/issues/26.

[17] OpenGL ES. http://developer.android.com/guide/topics/graphics/opengl.html.

[18] Pressurenet commit. https://github.com/Cbsoftware/PressureNet/commit/f5ae0db48a679a2d6598724f28c369bf269faeae.

[19] Pressurenet issue. https://github.com/Cbsoftware/PressureNet/issues/30.

[20] Pressurenet repository. https://github.com/Cbsoftware/PressureNet.

[21] Sdl. https://www.libsdl.org/.

[22] Sgtpuzzle. https://chris.boyle.name/projects/android-puzzles/.

[23] Sgtpuzzles commit. https://github.com/chrisboyle/sgtpuzzles/commit/12862f1cd95262b482a69f6d1a9ca383d256f13e.

[24] Sgtpuzzles commit. https://github.com/chrisboyle/sgtpuzzles/commit/2d7c71eef75f4bfb840b8b0091aaecc4cfa1e68b.

[25] Sgtpuzzles commit. https://github.com/chrisboyle/sgtpuzzles/commit/6409bebcda22cefff63339cfe44a39d7460f3a27.

[26] Sgtpuzzles commit. https://github.com/chrisboyle/sgtpuzzles/commit/7421dc82a53fa652c1ae66a26ac438204628219c.

[27] Sgtpuzzles issue 70. https://github.com/chrisboyle/sgtpuzzles/issues/70.

[28] Sgtpuzzles issue 80. https://github.com/chrisboyle/sgtpuzzles/issues/80.

[29] Stackoverflow question: Should i use java's string.format() if performance is important?. http://stackoverflow.com/questions/513600.

[30] Strictmode api. http://developer.android.com/reference/android/os/StrictMode.html.

[31] Wikipedia: App store. https://en.wikipedia.org/wiki/App_Store_(iOS).

[32] Wikipedia: Google play. http://en.wikipedia.org/wiki/Google_Play.

[33] Wikipedia: Windows phone store. http://en.wikipedia.org/wiki/Windows_Phone_Store.

[34] Android Developers. Android performance patterns. https://www.youtube.com/playlist?list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE.

[35] Apple. ios performance tips. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/PerformanceTips/PerformanceTips.html.

[36] F. Beck, O. Moseler, S. Diehl, and G. Rey. In situ understanding of performance bottlenecks through visually augmented code. In *ICPC'13*, pages 63–72, May 2013.

[37] J. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, 1990.

[38] Google. Android debugger bridge. http://developer.android.com/tools/help/adb.html.

[39] Google. Asynctask. http://developer.android.com/reference/android/os/AsyncTask.html.

[40] Google. Caching bitmaps. http://developer.android.com/training/displaying-bitmaps/cache-bitmap.html.

[41] Google. Device monitor. http://developer.android.com/tools/help/monitor.html.

[42] Google. Handler. http://developer.android.com/reference/android/os/Handler.html.

[43] Google. Logcat. http://developer.android.com/tools/help/logcat.html.

[44] Google. Managing bitmap memory. https://developer.android.com/training/displaying-bitmaps/manage-memory.html.

[45] Google. Profiling with traceview and dmtracedump. http://developer.android.com/tools/debugging/debugging-tracing.html.

[46] Google. Redundant downloads are redundant .http://developer.android.com/training/efficient-downloads/redundant_redundant.html.

[47] Google. Renderscript. http://developer.android.com/guide/topics/renderscript/compute.html.

[48] Google. Systrace. http://developer.android.com/tools/help/systrace.html.

[49] Google. Using ddms. http://developer.android.com/tools/debugging/ddms.html.

[50] Google Developers. Android performance patterns: Understanding overdraw. https://www.youtube.com/watch?v=T52v50r-JfE.

[51] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in android applications. In *ASE'13*, pages 389–398, 2013.

[52] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI '12*, pages 77–88, 2012.

[53] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile apps. In *ESEM'13*, pages 15–24, 2013.

[54] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *OOPSLA'11*, pages 155–170, 2011.

[55] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan. What do mobile app users complain about? a study on free ios apps. *IEEE Software*, 99(PrePrints):1, 2014.

[56] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *ICST'15*, pages 1–10, 2015.

[57] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for android applications through refactoring. In *FSE'14*, pages 341–352, 2014.

[58] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *MSR'14*, pages 2–11.

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

[60] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *MSR'13*, pages 237–246, 2013.

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

[62] A. Pathak, Y. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Hotnets'11*, page Article No 5, 2011.

[63] l. Paul Lamb. GameSurface.java https://github.com/mupen64plus-ae/mupen64plus-ae/blob/262be6de409202c86c70afa971da3a01863320c0/src/paulscode/android/mupen64plusae/GameSurface.java#L35.

[64] M. Syer, Z. M. Jiang, M. Nagappan, A. Hassan, M. Nasser, and P. Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *ICSM'13*, pages 110–119, 2013.

[65] S. Zaman, B. Adams, and A. Hassan. A qualitative study on performance bugs. In *MSR'12*, pages 199–208, 2012.

[66] S. Zaman, B. Adams, and A. E. Hassan. Security versus performance bugs: A case study on firefox. In *MSR'11*, pages 93–102, 2011.

[67] J. Zang, A. Musa, and W. Le. A comparison of energy bugs for smartphone platforms. In *MOBS'13*, 2013.

[68] M. Zechner. Of bytes, cycles and battery life. http://www.slideshare.net/mariozechner5/of-bytes-cycles-and-battery-life.