Characterizing and Detecting Performance Bugs for Smartphone Applications

Yepang Liu
Dept. of Comp. Sci. and Engr.
The Hong Kong Univ. of Sci. and Tech.
Hong Kong, China
andrewust@cse.ust.hk

Chang Xu*
State Key Lab for Novel Soft. Tech.
Dept. of Comp. Sci. and Tech.
Nanjing University, Nanjing, China
changxu@nju.edu.cn

Shing-Chi Cheung
Dept. of Comp. Sci. and Engr.
The Hong Kong Univ. of Sci. and Tech.
Hong Kong, China
scc@cse.ust.hk

ABSTRACT
Smartphone applications' performance has a vital impact on user experience. However, many smartphone applications suffer from bugs that cause significant performance degradation, thereby losing their competitive edge. Unfortunately, people have little understanding of these performance bugs. They also lack effective techniques to fight with such bugs. To bridge this gap, we conducted a study of 70 real-world performance bugs collected from eight large-scale and popular Android applications. We studied the characteristics (e.g., bug types and how they manifested) of these bugs and identified their common patterns. These findings can support follow-up research on performance bug avoidance, testing, debugging and analysis for smartphone applications. To demonstrate the usefulness of our findings, we implemented a static code analyzer, PerfChecker, to detect our identified performance bug patterns. We experimentally evaluated PerfChecker by applying it to 29 popular Android applications, which comprise 1.1 million lines of Java code. PerfChecker successfully detected 126 matching instances of our performance bug patterns. Among them, 68 were quickly confirmed by developers as previously-unknown issues that affect application performance, and 20 were fixed soon afterwards by following our optimization suggestions.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms
Experimentation, Measurement, Performance.

Keywords
Empirical study, performance bug, testing, static analysis.

1. INTRODUCTION
The smartphone application market is expanding rapidly. Until July 2013, the one million Android applications in the Google Play store received 50 billion downloads [20]. With more emerging applications of similar functionalities (e.g., various web browsers), performance and user experience has gradually become a dominant factor that affects user loyalty in application selection.

However, our inspection of 60,000 Android applications randomly sampled from Google Play store using a web crawler [11] revealed an alarming fact: 11,108 of them have suffered or are suffering from performance bugs of varying severity, as judged from their release logs or user reviews. These bugs can significantly slow down applications or cause them to consume excessive resources (e.g., memory or battery power). The pervasiveness of such performance bugs is attributable to two major reasons. First, smartphones are resource-constrained as compared to PCs, but their applications often have to conduct non-trivial tasks like web browsing and graphics rendering. Thus, poorly implemented applications can easily exhibit unsatisfactory performance. Second, many smartphone applications are developed by small teams without dedicated quality assurance. These developers lack viable techniques to help analyze the performance of their applications [23]. As such, it is hard for them to exercise due diligence in assuring application performance, especially when they have to push applications to market in a short time.

Research questions (RQ1-4)	**Combating performance bugs**
1. Bug types and impacts	• Bug avoidance techniques (RQ2, RQ4)
2. Bug manifestation	• Effective testing methodologies (RQ1, RQ2)
3. Debugging and fixing effort	• Debugging assistance tools (RQ3)
4. Common bug patterns	• Bug detectors (RQ1, RQ4)

Figure 1. Potential benefits of our empirical findings

Existing studies have focused on performance bugs in PC or server-side applications, and proposed several interesting testing and analysis techniques [27][28][35][52]. Yet, smartphone platforms are relatively new, and we have limited understanding of their applications' performance bugs. Whether existing techniques are applicable to smartphone applications is an open question. Therefore, in this paper, we aim to bridge this gap by conducting an empirical study. This study focuses on performance bugs from real-world smartphone applications. We restrict our scope to Android applications due to their popularity and code availability.

Our study covered 70 real-world performance bugs collected from eight large-scale and popular Android applications (e.g., Firefox) across five different categories. The study aims to answer the following four research questions:

- **RQ1 (Bug types and impacts):** *What are common types of performance bugs in Android applications? What impacts do they have on user experience?*
- **RQ2 (Bug manifestation):** *How do performance bugs manifest themselves? Does their manifestation need special inputs?*
- **RQ3 (Debugging and bug-fixing effort):** *Are performance bugs more difficult to debug and fix than non-performance bugs? What information or tools can help with this?*

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568229

* Corresponding author.

- **RQ4 (Common bug patterns):** *Are there common causes of performance bugs? Can we distill common bug patterns to facilitate performance analysis and bug detection?*

We studied related work [27][54] and formulated the above research questions. Through answering them, we aim to better understand characteristics of performance bugs in smartphone applications. For example, we found that smartphone applications are more susceptible to performance bugs than PC or server-side applications. Besides, many smartphone performance bugs do not need sophisticated data inputs (e.g., a database with hundreds of entries) to manifest, but instead their manifestation needs special user interactions (e.g., certain user interaction sequences). Some of our findings differ largely from those for performance bugs in PC or server-side applications [27][54], as we will explain later. Besides facilitating bug understanding, our findings can also support follow-up research on performance bug avoidance, testing, debugging and detection for smartphone applications, as illustrated in Figure 1. For instance, with our identified common bug patterns, one can propose guidelines for avoiding certain performance bugs in application development. One can also design and implement bug detection tools for identifying performance optimization opportunities in application testing and maintenance.

To evaluate the usefulness of our empirical findings, we further conducted a case study with 29 large-scale and popular Android applications. In the study, we implemented and tested a static code analyzer PerfChecker, which is built on Soot, a widely-used Java program analysis framework [47]. PerfChecker supports the detection of two performance bug patterns identified in our empirical study. We applied it to analyze the latest version of the 29 Android applications, which comprise 1.1 million lines of Java code. PerfChecker successfully detected 126 matching instances of the two bug patterns in 18 of these applications. We reported them to the corresponding developers, and 68 instances have been confirmed as real issues affecting application performance while others are pending. Among the confirmed instances, developers have quickly fixed 20 of them by following our bug-fixing suggestions. They also expressed great interest in our PerfChecker.

To summarize, we make two major contributions in this paper:

- To the best of our knowledge, we conducted the first empirical study of real-world performance bugs in smartphone applications. Our findings can help understand characteristics of performance bugs in smartphone applications, and provide guidance to related research (e.g., performance testing and analysis).
- We implemented a static code analyzer, PerfChecker, to detect our identified performance bug patterns in Android applications. PerfChecker successfully identified performance optimization opportunities in 18 popular Android applications. This inspiringly validated the usefulness of our findings.

The rest of this paper is organized as follows. Section 2 briefs Android application basics. Section 3 presents our empirical study of real-world performance bugs in Android applications. Section 4 presents our case study, in which we leverage our empirical findings to find performance optimization opportunities in Android applications. Section 5 discusses related work from recent years, and finally Section 6 concludes this paper.

2. BACKGROUND

Android is an open-source Linux-based operating system. It is now one of the most widely adopted smartphone platforms. Many equipment and device manufacturers (e.g., Samsung) customize their own Android variants by modifying the Android software stack (e.g., kernel and libraries). Applications running on the Android platform are mostly written in Java language. For performance considerations, developers may write critical parts of their applications using native-code languages such as C and C++.

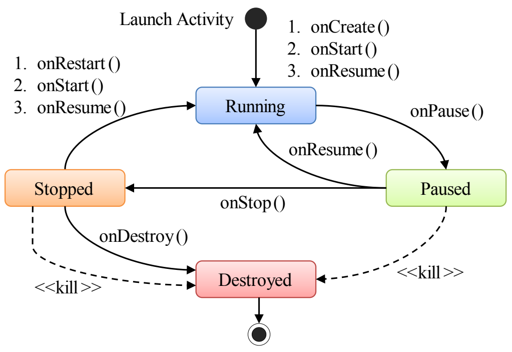


Figure 2. Lifecycle of an activity

Application component and lifecycle. Conceptually, an Android application consists of four types of components: *activity*, *service*, *broadcast receiver* and *content provider*. For example, an application's graphical user interfaces (GUIs) are defined by activities. Each application component is required to follow a prescribed lifecycle that defines how this component is created, used, and finally destroyed. For example, Figure 2 gives the lifecycle of an activity. It starts with a call to onCreate() handler and ends with a call to onDestroy() handler. An activity's foreground lifetime (i.e., the "Running" state) starts after calling onResume() handler, and lasts until onPause() handler is called, when another activity comes to foreground. An activity can interact with its user only when it is at foreground. When it goes to background and becomes invisible (i.e., the "Stopped" state), its onStop() handler would be called. When its user navigates back to a paused or stopped activity, the activity's onResume() or onRestart() handler would be called, and the activity would come to foreground again. In exceptional cases, a paused or stopped activity may be killed for releasing memory to other applications with higher priorities.

Single thread policy. When an Android application starts, Android OS creates a "main thread" (also known as an "UI thread") to instantiate this application's components. This thread dispatches system calls to responsible application components, and user events to appropriate UI widgets (e.g., buttons). After dispatching, the corresponding components' lifecycle handlers and UI widgets' GUI event handlers will run in the main thread to handle the system calls or user events. This is known as the "single thread policy" [1]. The policy requires developers to control workloads of their applications' main threads (e.g., not overwhelming a main thread with intensive work). Otherwise, applications can easily exhibit poor responsiveness.

3. EMPIRICAL STUDY

In this section, we present our empirical study of real-world performance bugs from Android applications. The study aims to answer our earlier four research questions RQ1–4. In the following, we first describe our application subjects and their reported performance bugs, and then discuss our empirical findings.

We selected open-source Android applications as our subjects for studying questions RQ1–4 because the study requires application bug reports and corresponding code revisions. 29 candidate applications that satisfy the following three criteria were randomly selected from four popular open-source software hosting platforms, namely, Google Code [18], GitHub [17], SourceForge [46], and Mozilla repositories [33]. First, a candidate should have achieved more than 10,000 downloads (popularity). Second, it

Table 1. Subjects and selected bugs

Application name	Category	Size (LOC)	Programming language	Downloads	Availability	# selected bugs
Firefox[3]	Communication[1]	122.9K[2]	Java, C++, C	10M[2] ~ 50M	Mozilla Repositories	34
Chrome[3]	Communication	77.3K	Java, C++, Python	50M ~ 100M	Google Code	19
AnkiDroid	Education	44.8K	Java	500K ~ 1M	Google Code	4
K-9 Mail	Communication	76.2K	Java	1M ~ 5M	Google Code	3
My Tracks	Health & Fitness	27.1K	Java	10M ~ 50M	Google Code	3
c:geo	Entertainment	44.7K	Java	1M ~ 5M	GitHub	3
Open GPS Tracker	Travel & Local	18.1K	Java	100K ~ 500K	Google Code	2
Zmanim	Books & Reference	5.0K	Java	10K ~ 50K	Google Code	2

[1]: The application category information is obtained from Google Play store [19]; [2]: 1K = 1,000 & 1M = 1,000,000
[3]: For Firefox and Chrome, we counted only their lines of code specific to Android.

should own a public bug tracking system (traceability). Third, it should have at least hundreds of code revisions (maintainability). The three criteria provide a good indicator of popular and mature applications. For these 29 candidates, we tried to identify performance bugs in their bug tracking systems. Due to different management practices, some application developers explicitly labeled performance bugs using special tags (e.g., "*perf*"), while others did not maintain a clear categorization of reported bugs. To ensure that we study real performance bugs, we refined our application selection by keeping only those containing clearly labeled performance bugs for our study. As a result, eight applications were finally selected as our subjects (all 29 applications were still used in our later case study for validating the usefulness of our empirical findings). From them, we obtained a total of 70 performance bugs, which were clearly labeled (confirmed) and later fixed.

Our selection process could miss some performance bugs (e.g., those not performance-labeled). Some related studies selected performance bugs by searching keywords like "slow" or "latency" in bug reports [27][54]. We found that such searches resulted in more than 2,800 candidate performance bugs in all 29 applications. We randomly sampled and manually analyzed 140 of these candidate bugs (5%), and found that most of them are inappropriate for our study. This is because more than 70% of these candidates are either not related to performance (i.e., their bug reports accidentally contain such keywords) or are actually complex bugs that contain both performance and functional issues (e.g., low performance as a side effect of wrongly implemented functionality). To avoid introducing such threats or uncontrollable issues to our empirical study, we refrained from keyword search, while focusing on the 70 explicitly labeled performance bugs.

Table 1 lists basic information of our eight selected Android applications. They are large-scale (up to 122.9K LOC), popularly-downloaded (up to 100 million downloads), and cover five different application categories. In the following we analyze 70 performance bugs collected from these applications and report our findings. The whole empirical study took about 180 person-days, involving three students (two postgraduate and one final-year undergraduate) for data collection, analysis and cross-checking.

3.1 RQ1: Bug Types and Impacts

We studied the bug reports and related discussions (e.g., comments and patch reviews) of the 70 performance bugs, and assigned them to different categories according to their major consequences. If a bug has multiple major consequences, we assigned it to multiple categories (so accumulated percentages can exceed 100%). We observed three common types of performance bugs in Android applications:

GUI lagging. Most performance bugs (53 / 70 = 75.7%) are *GUI lagging*. They can significantly reduce responsiveness or smoothness of the concerned applications' GUIs and prevent user events from being handled in a timely way. For example, in Firefox browser, tab switching could take up to ten seconds in certain scenarios (Firefox bug 719493[2]). This may trigger the infamous "Application Not Responding (ANR)" error and cause an application to be no longer runnable, because Android OS would force its user to close the application in such circumstances.

Energy leak. The second common type of performance bugs (10 / 70 = 14.3%) is *energy leak*. With such bugs, concerned applications could quickly consume excessive battery power with certain tasks, which actually bring almost no benefit to users. For example, the energy leak in Zmanim (bug 50) made the application render invisible GUI widgets in certain scenarios, and this useless computation simply wasted valuable battery power. If an Android application contains serious energy leaks, its user's smartphone battery could be drained in just a few hours. For instance, My Tracks has received such complaints (bug 520):

> *"I just installed My Tracks on my Galaxy Note 2 and it is a massive battery drain. My battery lost 10% in standby just 20 minutes after a full charge."*
>
> *"This app is destroying my battery. I will have to uninstall it if there isn't a fix soon."*

Energy leaks in smartphone applications can cause great inconvenience to users. Users definitely do not want their smartphones to power off due to low battery, especially when they need to make important phone calls. As shown in the above comments, if an application drains battery quickly, users may switch to other applications that offer similar functionalities but are more energy-efficient. Such a "switch" can be common since nowadays users have many choices in selecting smartphone applications.

Memory bloat. The third common type of performance bugs (8 / 70 = 11.4%) is *memory bloat*, which can incur unnecessarily high memory consumption (e.g., Firefox bug 723077 and Chrome bug 245782). Such bugs can cause "Out of Memory (OOM)" errors and application crashes. Even if a concerned application does not crash immediately (i.e., mild memory bloat), its performance can become unstable as Dalvik garbage collection would be frequently invoked, leading to degraded application performance.

These three performance bug types have occupied a majority of our studied 70 performance bugs (94.7%; some bugs belong to more than one type as aforementioned). There are also other types of bugs (e.g., those causing high disk consumption or low network throughput), but we observed them only once for each type in our dataset. Thus, we consider them not common.

[2] All bugs can be retrieved in their applications' bug tracking systems using our provided bug IDs. We omit detailed URLs due to page limit.

GUI lagging, energy leak and memory bloat are three dominant performance bug types in our studied Android applications. Research effort can first be devoted into designing effective techniques to combat them.

3.2 RQ2: Bug Manifestation

Understanding how performance bugs manifest in Android applications can provide useful implications on how to effectively test performance bugs. Our study reveals some observations, which demonstrate unique challenges in such performance testing.

Small-scale inputs suffice to manifest performance bugs. Existing studies reported that two thirds of performance bugs in PC applications need large-scale inputs to manifest [27]. However, in our study, we observed only 11 performance bugs (out of 70) that require large-scale inputs to manifest. Here, we consider a database with 100 data entries already large-scale (e.g., Firefox bug 725914). Other bugs can easily manifest with small-scale inputs. For example, Firefox bugs 719493 and 747419 only need one user to open several browser tabs to manifest. Manifested bugs would significantly slow down Firefox and make its GUI less responsive. We give some comments from their bug reports below:

> *"I installed the nightly version and found tab switching is so slow that it makes using more than one tab very hard."*
>
> *"Firefox should correctly use view holder patterns. Otherwise, it will just have pretty bad scrolling performance when you have more than a couple of tabs."*

These comments suggest that Android applications can be susceptible to performance bugs. If an application has issues affecting its performance, users can often have an uncomfortable experience when conducting simple daily operations like adding a browser tab (Firefox bug 719493). A few such operations can quickly cause performance degradation. Due to this reason, cautious developers should try their best to optimize the performance of their code. For example, c:geo developers always try to avoid creating short-term objects (c:geo bug 222), because Android documentation states that less object creation (even an array of Integers) means less garbage collection [2].

Special user interactions needed to manifest performance bugs. More than one third (25 out of 70) of performance bugs require special user interactions to manifest. For example, Zmanim's energy leak needs the following four steps to manifest: (1) switching on GPS, (2) configuring Zmanim to use current location, (3) starting its main activity, and (4) hitting the "Home" button when GPS is acquiring a location. Such bugs are common, but can easily escape traditional testing. They can only manifest after a certain sequence of user interactions happen to the concerned application, but traditional code-based testing adequacy criteria (e.g., statement or branch coverage) do not really consider sequences of user interactions. A recent study also shows that existing testing techniques often fail to reach certain parts of Android application code [25]. Hence, our findings suggest two challenges and corresponding research directions in testing performance bugs for smartphone applications:

- Effectively testing performance bugs requires coverage criteria that explicitly consider sequences of user interactions in assessing the testing adequacy. Since the validity of user interaction sequences is essentially defined by an application's GUI structure, existing research on GUI testing coverage criteria [32] may help in addressing this challenge.
- Test input generation should construct effective user interaction sequences to systematically explore an application's state space. Since such sequences can be infinite, research effort should focus on effective techniques that can identify equivalence among constructed user interaction sequences, avoiding redundant sequences and wasted test efforts.

Automated performance oracle needed. Performance bugs can gradually degrade an application's performance. For example, Firefox becomes progressively slower when its database's size grows (bug 785945). Such bugs rarely cause fail-stop consequences like application crashes, thus it is challenging to decide whether an application is suffering any performance bug. Yet, our study found three common judgment criteria that have been used in real world to detect performance bugs in Android applications:

- **Human oracle.** More than half of the judgments were made manually by developers or users in our investigated Android applications. People simply made judgments according to their own experiences.
- **Product comparison.** Many developers compared different products of similar functionalities to judge whether a particular product contains any performance bugs (e.g., checking whether conducting an operation in one product is remarkably slower than in other products). We observed ten such cases in our study. For example, upon receiving user complaints about performance, K9 Mail developers checked whether their application's performance was comparable to other email clients and then decided what to do next (K9 Mail bugs 14 and 23).
- **Developers' consensus.** Developers also have some implicit consensus for judging performance bugs. For instance, Google developers consider an application sluggish (i.e., GUI lagging) if a user event cannot be handled within 200ms [53]. Mozilla developers assume that Firefox's graphics-rendering units should be able to produce 60 frames per second to make smooth animations (Firefox bugs 767980 and 670930).

Although these judgment criteria have been used in practice, they either require non-trivial manual effort (thus not scalable) or are not generally defined (thus not widely used). To facilitate performance testing and analysis, automated oracles are thus desirable. Even if general oracles may not be possible, application or bug specific oracles can still be helpful. Encouragingly, there have been initial attempts toward this end [41][55]. Besides, our previous work [30] also proposed a cost-benefit analysis to detect energy leaks caused by improper or ineffective uses of smartphone sensors. Still, more effort on general automated oracles for performance bugs is needed to further advance related research.

Performance bugs can be platform-dependent. We also observed that a non-negligible proportion (6 out of 70) of performance bugs require specific software or hardware platforms to manifest. For example, Chrome's caching scheme would hurt performance on ARM-based devices, but not on x86-based devices (Chrome bugs 170344 and 245782). Firefox's animation works more smoothly on Android 4.0 than older systems (Firefox bug 767980). This suggests that developers should consider device variety during performance testing, since Android OS can run on different hardware platforms and has so many customized variants. This feature differs largely from performance bugs in PC applications, which are not so platform-dependent [27][54].

Effective performance testing needs: (1) new coverage criteria to assess testing adequacy, (2) effective techniques for generating user interaction sequences to manifest performance bugs, and (3) automated oracles to judge performance degradation.

3.3 RQ3: Debugging and Bug-fixing Effort

To understand the effort required for performance debugging and bug-fixing for Android applications, we analyzed 60 of our 70

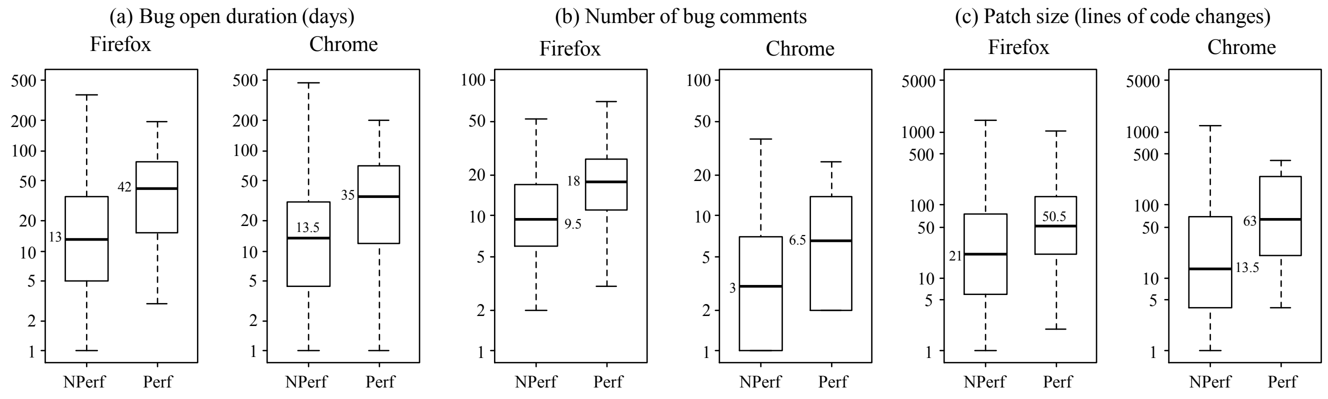


Figure 3. Comparison of debugging and bug-fixing effort ("Perf" = "performance bug", "NPerf" = "non-performance bug")

Table 2. Performance bug debugging and fixing effort

Metric	Min.	Median	Max.	Mean
Bug open duration (days)	1	47	378	59.2
Number of bug comments	1	14	71	16.7
Patch size (LOC)	2	72	2,104	182.3

Table 3. *p*-values of Mann-Whitney U-tests

Subject	*p*-value		
	Bug open duration	# bug comments	Patch size
Firefox	0.0008	0.0002	0.0206
Chrome	0.0378	0.0186	0.0119

performance bugs. We excluded 10 remaining bugs because we failed to recover links between their bug reports and code revisions[3]. To quantify debugging and bug-fixing effort for each of these bugs, we measured three metrics that were also adopted in related studies [54]: (1) *bug open duration*, which is the amount of time from a bug report is opened to the concerned bug is fixed; (2) *number of bug comments*, which counts discussions among developers and users for a bug during its debugging and bug-fixing period; (3) *patch size*, which is the lines of code changed for fixing a bug. Intuitively, if a bug is difficult to debug and fix, its report would be open for a long time, developers tend to discuss it more, and its patch could cover more lines of code changes.

Table 2 reports our measurement results. We observe that on average, it takes developers about two months to debug and fix a performance bug in an Android application. During this period, they can have tens of rounds of discussions, resulting in many bug comments (up to 71). Besides, on average, bug-fixing patches can cover more than 182 lines of code changes, indicating non-trivial bug-fixing effort. For comparison, we also randomly selected 200 non-performance bugs (bugs without performance labels) from the bug database of Firefox and Chrome (we selected 100 bugs for each). We did not select non-performance bugs from other application subjects for comparison, because each of these subjects contains only a few performance bugs (about two to four). Such small sample sizes may lead to unreliable comparison results, leaving a weak foundation for further research on related topics [12]. On the other hand, the vast majority of our studied performance bugs come from Firefox and Chrome, and therefore we selected non-performance bugs from these two subjects for comparison. The severity levels of our selected 200 non-performance bugs are comparable to those of performance bugs in Firefox and Chrome. Figure 3 compares these two kinds of bugs by boxplots. The results consistently show that performance debugging and bug-fixing require more effort than their non-performance counterparts. For example, in Firefox, the median bug open duration is 42 workdays for performance bugs, but only 13 workdays for non-performance bugs. To understand the significance of the differences between these two kinds of bugs, we conducted a Mann-Whitney U-test [31] with the following three null hypotheses:

- Performance debugging and bug-fixing do not take a significantly longer time than their non-performance counterparts.
- Performance debugging and bug-fixing do not need significantly more discussion than their non-performance counterparts.
- Patches for fixing performance bugs are not significantly larger than those for fixing non-performance bugs.

Table 3 gives our Mann-Whitney U-test results (*p*-values). The results rejected the above three null hypotheses all with a confidence level over 0.95 (i.e., *p*-values are all less than 0.05). Thus, we conclude that debugging and fixing performance bugs indeed requires more effort than debugging and fixing non-performance bugs. This result can help developers better understand and prioritize bugs for fixing in a cost-effective way, as well as estimating possible manual effort required for fixing certain bugs.

We further looked into bug comments and bug-fixing patches to understand: (1) why it is difficult to debug and fix performance bugs in Android applications, and (2) what support is expected in debugging and bug-fixing. We found that quite a few (22 / 70 = 31.4%) performance bugs involve multiple threads or processes, which may have complicated the debugging and bug-fixing tasks. In addition, these performance bugs rarely caused fail-stop consequences such as application crashes. Due to this reason, traditional debugging information (e.g., stack trace) can offer little help in performance debugging. We analyzed all 70 performance bugs, and found that only four bugs have had their debugging and fixing tasks receiving some help from such traditional information, as judged from their bug discussions (e.g., c:geo bug 949 and Firefox bug 721216). On the other hand, we found that debugging information from two kinds of tools has received more attention:

Profiling tools. Profiling tools (or profilers) monitor an application's execution, record its runtime information (e.g., execution time of a code unit), and trace details of its resource consumption (e.g., memory). For example, Firefox and Chrome developers often take three steps in performance debugging: (1) reproducing a performance bug with the information provided in its bug report if any, (2) running the application of concern for a long while to generate a profile using their own profilers [9][14], and (3) performing offline profile analysis to identify performance bottle-

[3] Our manual analysis of commit logs around bug-fixing dates also failed to find corresponding code revisions.

```
 1.      public void refreshThumbnails() {
 2.        //generate a thumbnail for each browser tab
 3.  -     Iterator<Tab> iter = tabs.values().iterator();
 4.  -     while (iter.hasNext())
 5.  -       GeckoApp.mAppContext.genThumbnailForTab(iter.next());
 6.  +     GeckoAppShell.getHandler().post(new Runnable() {
 7.  +       public void run() {
 8.  +         Iterator<Tab> iter = tabs.values().iterator();
 9.  +         while (iter.hasNext())
10.  +           GeckoApp.mAppContext.genThumbnailForTab(iter.next());
11.  +       }
12.  +     });
13.      }
     Note: the method genThumbnailForTab() compresses a bitmap to
     produce a thumbnail for a browser tab.
```

Figure 4. Firefox bug 721216 (simplified)

necks/bugs if possible. However, profile analysis can be very time-consuming and painful, because current tools (e.g., those from Android SDK) can record tons of runtime information, but which runtime information can actually help performance debugging is still an open question. Firefox developers have designed some visualization tools (e.g., Cleopatra [14]) to save manual effort in profile analysis, but these tools are not accessible to other developers or applicable to other applications. Researchers and practitioners are thus encouraged to design new general techniques and tools for analyzing, aggregating, simplifying and visualizing profiling data to facilitate performance debugging.

Performance measurement tools. Performance measurement tools can also ease performance debugging. They can directly report performance for a selected code unit in an application. For example, Firefox's frame rate meter [15] measures the number of frames a graphics-rendering unit can produce per second (e.g., when debugging Firefox bug 670930). This information can help developers in two ways. First, it prioritizes the code units that need performance optimization. Second, it suggests whether a code unit has been adequately optimized. For example, Firefox developers could stop further optimizing a graphics-rendering unit if the frame rate meter reports a score of 60 frames per second (e.g., when fixing Firefox bug 767980). Chrome developers also use similar tools (e.g., using smoothness measurement tools for debugging Chrome bug 242976). Such tools are useful and welcomed by Android developers. We show some comments about Firefox's frame rate meter from the developers' mailing list:

> *"I found it very useful for finding performance issues in Firefox UI, and web devs should find it useful too."*
>
> *"This is fantastic stuff. It's a must-have for people hacking on front end UI. Also for devs tracking animation perf."*

Besides understanding the challenges of performance debugging, we also looked for reasons from bug-fixing patches why fixing performance bugs is so difficult. We found that such patches are often complex and have to conduct: (1) algorithmic changes (e.g., Firefox bug 767980), (2) design pattern reimplementation (e.g., Firefox bug 735636), or (3) data structure or caching scheme redesign (e.g., Chrome bug 245782). Such bug-fixing tasks are usually complex. This explains why fixing performance bugs took a longer time and incurred much larger patch sizes than fixing non-performance bugs, as illustrated in Figure 3.

Debugging and fixing performance bugs are generally more difficult than debugging and fixing non-performance bugs. Information provided by profilers and performance measurement tools are more helpful for debugging than traditional information like stack trace. Existing profilers expect improvement for automatically analyzing, aggregating, simplifying and visualizing collected runtime profiles.

3.4 RQ4: Common Bug Patterns

To learn the root causes of our 70 performance bugs, we studied their bug reports, patches, commit logs and patch reviews. We managed to figure out root causes for 52 of these bugs. For the remaining 18 bugs, we failed due to the lack of informative materials (e.g., related bug discussions).

Performance bugs in Android applications can have complex or application-specific root causes. For example, Firefox's "slow tab closing" bug was caused by heavy message communications between its native code and Java code (Firefox bug 719494), while AnkiDroid suffered GUI lagging because its database library was inefficient (AnkiDroid bug 876). Despite such variety, we still identified three common causes for 21 out of the 52 performance bugs (40.4%). We explain them with concrete examples below.

Lengthy operations in main threads. As mentioned earlier, Android applications should not block their main threads with heavy tasks [1]. However, when applications become increasingly more complex, developers tend to leave lengthy operations in main threads. We observed quite a few occurrences of such bugs (11 / 52 = 21.2%). Figure 4 gives a simplified version of Firefox bug 721216 and its bug-fixing patch. This bug caused Firefox to suffer GUI lagging when its "tab strip" button was clicked. The bug occurred because the button's click event handler transitively called a "refreshThumbnails" method, which produced a thumbnail for each browser tab by iteratively calling heavy-weight Bitmap compression APIs (Lines 3–5). Later to fix this bug, developers moved such heavy operations to a background thread (Lines 6–12), which can asynchronously update Firefox's GUI.

Wasted computation for invisible GUI. When an Android application switches to background, it may still keep updating its invisible GUI. This brings almost no perceptible benefit to its user, and thus the performed computation (e.g., collecting data and updating GUI) simply wastes resources (e.g., battery power). Such bugs also form a common pattern, which covers 6 of the 52 performance bugs (6 / 52 = 11.5%). For instance, Figure 5 lists the concerned code and corresponding bug-fixing patch for our aforementioned energy leak in Zmanim (bug 50). When ZmanimActivity launches, it registers a location listener to receive location changes for updating its GUI (Lines 5–14). The location listener is normally unregistered when the activity is destroyed (Line 27). However, if a user launches Zmanim and then switches it to background (Android OS will call onPause() and onStop() handlers accordingly, but not onDestroy()), the application will keep receiving location changes to update its GUI, which is, however, invisible. The location sensing and GUI refreshing are then useless, but still drain battery power. This can be common for many smartphone applications, because users often perform multiple tasks at the same time (e.g., playing Facebook and Twitter while listening to music) and frequently switch between them. To fix such bugs, developers have to carefully monitor application states and disable unnecessary tasks when an application goes to background. For example, Firefox developers suggested disabling timers, animations, DOM events, audio, video, flash plugins, and sensors when Firefox went to background (Firefox bug 736311). Similarly, as Figure 5 shows, Zmanim developers disabled location sensing by unregistering the location listener in ZmanimActivity's onPause() handler (Line 23), and enabled it again in onResume() handler when necessary (Lines 17–19).

Frequently invoked heavy-weight callbacks. Four out of the 52 performance bugs (4 / 52 = 7.7%) concern frequently invoked callbacks. These callbacks need to be light-weight since they are frequently invoked by Android OS. However, many such callbacks in real-world applications are ill-implemented. They are

```
   public class ZmanimActivity extends Activity {
     private ZmanimLocationManager lm;
     private ZmanimLocationManager.Listener locListener;
     public void onCreate() {
       //get a reference to system location manager
       lm = new ZmanimLocationManager(ZmanimActivity.this);
       locListener = new ZmanimLocationManager.Listener() {
         public void onLocationChanged(ZmanimLocation newLoc) {
           //build UI using obtained location in a new thread
           rebuildUI(newLoc);
         }
       };
       //register location listener
       lm.requestLocationUpdates(GPS, 0, 0, locListener);
     }
     public void onResume() {
+      //register location listener if UI still needs update
+      if(buildingUINotFinished)
+        lm.requestLocationUpdates(GPS, 0, 0, locListener);
     }
     public void onPause() {
+      //unregister location listener
+      lm.removeListener(locListener);
     }
     public void onDestory() {
-      //unregister location listener
-      lm.removeListener(locListener);
     }
   }
```

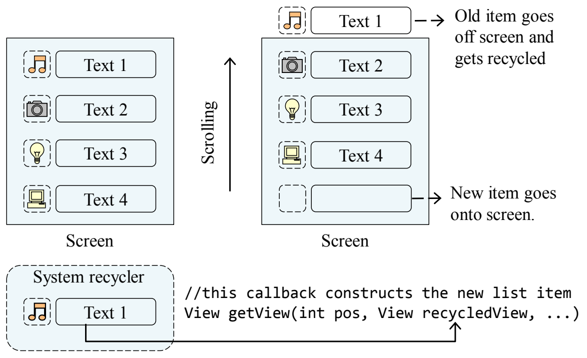
Figure 5. Zmanim bug 50 (simplified)

```
    //inefficient version
1.  public View getView(int pos, View recycledView, ViewGroup parent) {
2.    //list item layout inflation
3.    View item = mInflater.inflate(R.layout.listItem, null);
4.    //find inner views
5.    TextView txtView = (TextView) item.findViewById(R.id.text);
6.    ImageView imgView = (ImageView) item.findViewById(R.id.icon);
7.    //update inner views
8.    txtView.setText(DATA[pos]);
9.    imgView.setImageBitmap((pos % 2) == 1 ? mIcon1 : mIcon2);
10.   return item;
11. }

12. //apply view holder pattern
13. public View getView(int pos, View recycledView, ViewGroup parent) {
14.   ViewHolder holder;
15.   if(recycledView == null) { //no recycled view to reuse
16.     //list item layout inflation
17.     recycledView = mInflater.inflate(R.layout.listItem, null);
18.     holder = new ViewHolder();
19.     //find inner views and cache their references
20.     holder.text = (TextView) recycledView.findViewById(R.id.text);
21.     holder.icon = (ImageView) recycledView.findViewById(R.id.icon);
22.     recycledView.setTag(holder);
23.   } else {
24.     //reuse the recycled view, retrieve the inner view references
25.     holder = (ViewHolder) recycledView.getTag();
26.   }
27.   //update inner view contents
28.   holder.text.setText(DATA[pos]);
29.   holder.icon.setImageBitmap((pos % 2) == 1 ? mIcon1 : mIcon2);
30.   return recycledView;
31. }

32. //view holder class for caching inner view references
33. public class ViewHolder {
34.   TextView text;
35.   ImageView icon;
36. }
```

Figure 7. View holder pattern

heavy-weight and can significantly slow down concerned applications. We illustrate with a list view callback example below.

A list view displays a list of scrollable items and is widely used in Android applications. Figure 6 gives one example, where each listed item contains two elements (i.e., two inner views of the list item): an icon and a text label. When a user scrolls up a list view, some existing items will go off the top of the screen while some new items will be added to the bottom. To use a list view, developers need to write an adapter class and define its getView() callback (see Figure 7 for example). At runtime, when a new item needs to go onto the screen, Android OS will invoke the getView() callback to construct and show this item. This callback conducts two operations: (1) parsing the new item's layout XML file and constructing a tree of its elements (a.k.a., list item layout inflation), and (2) traversing the tree to retrieve specific elements for updating (a.k.a., inner view retrieval and update). However, XML parsing and tree traversing can be time-consuming when a list item's layout is complex (e.g., containing many elements or having hierarchical structures as Android applications typically do). Screen scrolling can thus slow down if such operations are commonly performed. For performance concerns, Android OS recycles each item that goes off the screen while users scroll a list view. The recycled items can be reused to construct new items that need to appear later. Such "recycle and reuse" can be done as list items often have identical layouts.



```
//this callback constructs the new list item
View getView(int pos, View recycledView, ...)
```

Figure 6. List view example

We give two versions of getView() implementation in Figure 7. The first inefficient version conducts two aforementioned operations (Lines 2–9) every time the callback is invoked. The second version applies a "view holder" design pattern suggested by Android documentation [3]. The basic idea is to reuse previously recycled list items. It avoids list item layout inflation when there are recycled items for reuse (Lines 24–25). Besides, when a list item is constructed for the first time, the references to its inner view objects are identified and stored in a special data structure (Lines 18–22; data structure defined at Lines 32–36). Later, when reusing recycled items, these stored references can be used directly for updating content (Lines 27–29), avoiding inner view retrieval operations. By doing so, the view holder pattern can save both computation for list item layout inflation and inner view retrieval, and memory for constructing new list items. Frequently invoked callbacks should adopt such efficient designs.

Our study identified three common performance bug patterns: (1) lengthy operations in main threads, (2) wasted computation for invisible GUI, and (3) frequently-invoked heavy-weight callbacks. Researchers and practitioners should design effective techniques to prevent and detect such performance bugs.

3.5 Discussions

Our findings of performance bugs in smartphone applications exhibit some unique features, as compared with those from PC or server-side applications. First, smartphone application platforms are new and quickly evolving. For example, the current Android platform is not fully optimized and developers keep improving its performance for better user experience. Smartphone applications are thus susceptible to performance bugs due to such platform instability. As shown earlier, Android users can easily manifest

Table 4. Subjects and the detected bug pattern instances in them

Application name	Application category	Revision no.	Size (LOC)	Downloads	Availability	Bug pattern instances		Bug ID(s)
						VH (69)	LM (57)	
Ushahidi	Communication	59fbb533d0	43.3K	10K ~ 50K	GitHub [48]	9*	2	146, 159
c:geo	Entertainment	6e4a8d4ba8	37.7K	1M ~ 5M	GitHub [8]	0	5	3054
Omnidroid	Productivity	865	12.4K	1K ~ 5K	Google Code [38]	9	8	182, 183
Open GPS Tracker	Travel & Local	14ef48c15d	18.1K	100K ~ 500K	Google Code [39]	1	0	390
Geohash Droid	Entertainment	65bfe32755	7.0K	10K ~ 50K	Google Code [16]	0	1	48
Android Wifi Tether	Communication	570	9.2K	1M ~ 5M	Google Code [4]	1	3	1829, 1856
Osmand	Travel & Local	8a25c617b1	77.4K	500K ~ 1M	Google Code [40]	18	17	1977, 2025
My Tracks	Health & Fitness	e6b9c6652f	27.1K	10M ~ 50M	Google Code [34]	2*	0	1327
WebSMS	Communication	1f596fbd29	7.9K	100K ~ 500K	Google Code [49]	0	1	801
XBMC Remote	Media & Video	594e4e5c98	53.3K	1M ~ 5M	Google Code [50]	1	0	714
ConnectBot	Communication	716cdaa484	33.7K	1M ~ 5M	Google Code [10]	0	6	658
Firefox	Communication	895a9905dd	122.9K	10M ~ 50M	Mozilla Repositories [33]	1	0	899416
APG	Communication	a6a371024b	98.2K	50K ~ 100K	Google Code [6]	4	8	140, 144
FBReaderJ	Books & References	0f02d4e923	103.4K	5M ~ 10M	GitHub [13]	6*	6	148, 151
Bitcoin Wallet	Finance	12ca4c71ac	35.1K	100K ~ 500K	Google Code [7]	4	0	190
AnySoftKeyboard	Tools	04bf623ec1	26.0K	500K ~ 1M	GitHub [5]	2*	0	190
OI File Manager	Productivity	f513b4d0b6	7.8K	5M ~ 10M	GitHub [37]	1*	0	39
IMSDroid	Media & Video	553	21.9K	100K ~ 500K	Google Code [24]	10	0	457

1. "VH" means "Violation of the view Holder pattern", and "LM" means "Lengthy operations in Main threads".
2. Underlined bug pattern instances have been confirmed by developers as real performance issues, and "*" marked instances have been fixed by developers accordingly. For more details, readers can visit corresponding subject's source repositories and bug tracking systems by our provided links and bug IDs.

performance bugs by simple daily operations. Second, smartphone applications run on devices with small-sized touch screens. Users interact with these devices in a way that is very different from what they do with PCs. For example, users perform screen scrolling much more frequently on smartphones than on PCs. This makes GUI responsiveness and smoothness more crucial for smartphones. Our study reported that GUI-related bugs are pervasive and have become a dominant bug type that concerns Android applications' performance. Third, smartphone applications run on devices with small-capacity batteries, but can access energy-consuming components like GPS sensor and accelerometer, which are usually not available on PCs. Cost-ineffective uses of such components can lead to high energy consumption. Indeed, we found energy leaks were clearly severe in our studied Android applications. These comparisons help researchers and practitioners understand how performance bugs occur in smartphone applications, as well as explaining why they differ from their counterparts in traditional PC or server-side applications.

The validity of our study results may be subject to several threats. The first is the representativeness of our selected Android applications. To minimize this threat, we selected eight large-scale and popularly-downloaded Android applications, which cover five different categories. We wish to generalize our findings to more applications, and we will show in Section 4 how our findings help detect performance bugs for a wider range of Android applications. The second threat is our manual inspection of the selected performance bugs. We understand that manual inspection can be error-prone. To reduce this threat, we asked participants to conduct manual inspection independently. We also re-examined and cross-validated all results for consistency.

4. DETECTING PERFORMANCE BUGS

In this section, we conduct a case study to investigate whether our empirical findings can help fight with performance bugs in real-world Android applications. This case study aims to answer the following research question:

- **RQ5:** *Can we leverage our empirical findings, e.g., common bug patterns, to help developers identify performance optimization opportunities in real-world Android applications?*

To answer research question RQ5, we built a static code analyzer, PerfChecker[4], on Soot, a widely-used Java program analysis framework [47]. We applied PerfChecker to 29 Android applications discussed earlier to check its effectiveness of detecting performance bugs. We first introduce PerfChecker's implementation in Section 4.1, and then report our study results in Section 4.2.

4.1 Performance Bug Checking Algorithms

PerfChecker takes as input an Android application's Java bytecode, and generates warnings if it detects any issues that may affect application performance. Its current implementation supports detecting two of our identified performance bug patterns: (1) lengthy operations in main threads, and (2) violation of the view holder pattern (as a concrete case of the "frequently invoked heavy-weight callbacks" bug pattern). We did not include the "wasted computation for invisible GUI" bug pattern in this study. This is because our previous work [30] has proposed a cost-benefit analysis to detect performance bugs of this pattern and demonstrated the effectiveness of such analysis. Therefore, we in this paper focus on the other two performance bug patterns.

Detecting lengthy operations in main threads. PerfChecker first conducts a class hierarchy analysis to identify a set of checkpoints. These checkpoints include those lifecycle handlers defined in application component classes (e.g., those extending the Activity class) and GUI event handlers defined in GUI widget classes. According to Android's single thread policy, these checkpoints are all executed in an Android application's main thread. Then PerfChecker constructs a call graph for each checkpoint, and traverses this graph to check whether the checkpoint transitively

[4] PerfChecker can be obtained at: http://sccpu2.cse.ust.hk/perfchecker

invokes any heavy APIs, e.g., networking, database query, file IO, or other expensive APIs like those for Bitmap resizing. PerfChecker would report a warning for any of such findings.

Detecting violation of the view holder pattern. Similarly, PerfChecker first conducts a class hierarchy analysis to identify a set of checkpoints including all getView() callbacks in concerned list views' adapter classes. PerfChecker then constructs a program dependency graph for each checkpoint, and traverses this graph to check whether the following rule is violated: *list item layout inflation and inner view retrieval operations should be conditionally conducted based on whether there are reusable list items*. PerfChecker would report a warning for any such of violations.

4.2 Study Results and Developer's Feedback

We applied PerfChecker to analyze the latest version of all 29 Android applications, which comprise more than 1.1 million lines of Java code. Encouragingly, PerfChecker finished analyzing each application within a few minutes and detected 126 matching instances of the two performance bug patterns in 18 of the 29 applications. Table 4 gives details of these applications. It reports for each application: (1) application name, (2) application category, (3) revision number, (4) program size, (5) number of downloads, (6) source code availability, (7) number of bug pattern instances detected, and (8) concerned bug ID(s). For example, Osmand is a large-scale (77.4K LOC) and popular (500K–1M downloads) Android application for navigation. PerfChecker detected 17 checkpoints (i.e., handlers) invoking file IO and database query APIs in Osmand's main thread, and 18 violations of the view holder pattern throughout Osmand. We reported our findings as well as optimization suggestions to corresponding developers and received prompt and enthusiastic feedback. Altogether, 68 detected bug pattern instances (54.0%; bold and underlined in Table 4) have been quickly (within a few workdays) confirmed by developers as real issues that affect application performance. These issues cover 9 of the 18 applications (50.0%). In addition, 20 of the 68 confirmed issues (29.4%; marked with "*" in Table 4) have been fixed in a timely fashion by following our suggestions. Some developers, although not immediately fixing their confirmed issues, promised to optimize their applications according to our suggestions (e.g., My Tracks bug 1327). Other reported issues are pending (their concerned applications may not be under active maintenance). We also communicated with developers via bug reports and obtained some interesting findings as discussed below.

First, developers showed great interest in our PerfChecker. For example, we received the following feedback:

> *"Thanks for reporting this. I'll take a look at it. Just curious, where is this static code checker? Anywhere I can play with it as well? Thanks."* -- Ushahidi bug 159

> *"Thanks for your report. The code is only a year old. That's probably the reason (why it's not well optimized). Your static analyzer sounds really interesting. I wonder if lint[5] can also check this."* -- OI File Manager bug 39

These comments suggest that developers welcome performance analysis tools to help optimize their Android applications. PerfChecker is helpful, especially for complex applications. For example, it detected some applications transitively calling heavy-weight APIs in their main threads, and the call chains can last for tens of method calls (e.g., c:geo bug 3054). Currently, there are few industrial-strength tools supporting smartphone performance analysis. Thus there is a strong need for effective tools to help developers fight with smartphone application performance bugs.

Second, some developers held conservative attitudes toward performance optimization. They concerned much, e.g., (1) whether optimization can bring significant performance gains, (2) whether optimization can be done easily, and (3) whether optimization helps toward an application's market success. They hesitated to conduct performance optimization when the optimization seem to require a lot of effort but bring no immediate benefits. For example, WebSMS and Firefox developers responded as follows:

> *"You are totally right. WebSMS is ported from J2ME and has a very old code base ... If I would write it from scratch, I'd do things differently. I once started refactoring, but gave up in the end. There were other things to do, and the SMS user base is shrinking globally … If you want to help, just fork it on GitHub and let me merge your changes. I'd be very thankful."* -- WebSMS bug 801

> *"Thanks for the report! This shouldn't be a big concern; that UI is not a high-volume part. We'll keep this bug open, and I'd accept a patch which improves the code, but it's not a high-priority work item."* -- Firefox bug 899416

Finally, some developers were cautious and willing to conduct code optimization to improve performance or maintainability for their applications. For example, c:geo developers responded:

> *"Such optimizations are 'micro optimizations', and they do not improve the user visible performance. Good developers however will still refactor code into the better version, mostly to make it more readable."* -- c:geo bug 222

c:geo developers quickly fixed this reported performance bug. This may explain why c:geo keeps being highly-rated and popularly-downloaded (1M–5M downloads) on the market.

Our static code analyzer detected quite a few new issues that affect performance in a wide range of real-world Android application. Developers showed great interest in such tools. This validates the usefulness of our empirical findings.

5. RELATED WORK

Our studies in this paper relate to a large body of existing work on testing, debugging, bug detection and understanding for application performance. We discuss some representative pieces of work in recent years. Some of them focus on smartphone application performance, while others are for PC or server-side applications.

Detecting performance bugs. Much research effort has been devoted to automating performance bug/issue[6] detection. For example, Xu et al. used cost-benefit analysis to detect high-cost data structures that bring little benefit to a program's output [52]. Such data structures can cause memory bloat. Xiao et al. used a predictive approach to detect workload-sensitive loops that contain heavy operations, which often cause performance bottlenecks [51]. Recent work Toddler by Nistor et al. detected repetitive computations that have similar memory-access patterns in loops. Such computations can be unnecessary and subject to optimization [35]. These pieces of work focused on performance issues in PC or server-side applications, while there are also other pieces of work particularly focusing on smartphone application performance. For example, Pathak et al. studied no-sleep energy bugs in

[5] Lint is a static checker from Android development tools. It can detect performance threats like using getters instead of direct field accesses within a class, but does not support our identified performance bug patterns. Details can be found at http://tools.android.com/tips/lint-checks.

[6] Some researchers prefer "performance issue" to "performance bug". We do not have a preference and use the two terms interchangeably.

Android applications and used reaching-definition dataflow analysis to detect such bugs (e.g., an application forgets to unregister a used sensor) [41]. Following in this direction, Guo et al. further proposed a technique to detect general resource leaks, which often cause performance degradation [22]. Similar to Xu et al.'s [52] and Zhang et al.'s work [55], we previously leveraged cost-benefit analysis to detect whether an Android application uses sensory data in a cost-ineffective way [30]. Potential energy leak bugs can be reported after cross-state data utilization comparisons.

Testing for application performance. Performance testing is challenging due to the lack of test oracles and effective test input generation techniques. Some ideas have been proposed to alleviate such challenges. For example, Jiang et al. used performance baselines extracted from historical test runs as tentative oracles for new test runs [26]. Grechanik et al. learned rules from existing test runs, e.g., what inputs have led to intensive computations. They used such rules to select new test inputs to expose performance issues [21]. These ideas work well for PC applications, but it is unclear whether they are effective for smartphone applications. Our empirical study discloses that many performance bugs in smartphone applications need certain user interaction sequences to manifest. Besides, smartphone applications also have some unique features, e.g., long GUI lagging can force an Android application to close. Such requirements and features should be considered in order to design effective techniques to test the performance of smartphone applications. We have observed initial attempts along this direction. For example, Yang et al. tried to crash an Android application by adding a long delay after each heavy API call to test GUI lagging issues [53]. Jensen et al. studied how to generate user interaction sequences to reach certain targets in an Android application [25]. These attempts support performance testing of Android applications, but how to assess the testing adequacy is still unclear. Our work thus motivates new attempts for performance testing adequacy criteria, as well as effective techniques to expose performance issues in smartphone applications.

Debugging and optimization for application performance. Existing work on debugging and optimization for smartphone application performance mainly falls into two categories. The first category estimates performance for smartphone applications to aid debugging and optimization tasks [23][29][42][57]. For example, Mantis [29] estimated the execution time for Android applications on given inputs. This helps identify problem-inducing inputs that can slow down an application, so that developers can conduct optimization accordingly. PowerTutor [57], Eprof [42] and eLens [23] estimated energy consumption for Android applications by different energy models. They can help debug energy leak issues. For example, eLens offered fine-grained energy consumption estimation at source code level (e.g., method and line level estimation) to help locate energy bottlenecks. The second category of existing work uses profiling to log performance-related information to aid debugging and optimization tasks [43][44][56]. For example, ARO [43] monitored cross-layer interactions (e.g., those between the application layer and the resource management layer) to help disclose inefficient resource usage, which commonly causes performance degradation to smartphone applications. AppInsight [44] instrumented application binaries to identify critical paths (e.g., slow execution paths) in handling user interaction requests, so as to disclose root causes for performance issues. Panappticon [56] shared the same goal as AppInsight, and further revealed performance issues from inefficient platform code or problematic application interactions. There are also performance debugging techniques [28][45] for PC applications. For example, LagHunter [28] detected user-perceivable latencies in interactive applications (e.g., Eclipse); Shen et al. constructed a system-wide I/O throughput model to guide performance debugging [45]. These techniques may not apply to multi-threaded and asynchronous smartphone applications, because LagHunter tracked only synchronous UI event handling and Shen et al.'s work required a system-level performance model, which may not be available.

Understanding performance bugs. Finally, understanding and learning characteristics of performance bugs is a very important step toward designing effective techniques to test and debug performance issues. Existing characteristic studies have mainly focused on PC or server-side applications [27][36][54]. For example, Zaman et al. [54] studied performance bug reports from Firefox and Chrome (for PCs), and gave recommendations on how to better conduct bug identification, tracking and fixing. Jin et al. [27] studied the root causes of performance bugs in several selected PC or server-side applications, and identified efficiency rules for their detection. The most recent work by Nistor et al. [36] studied lifecycles of performance bugs (e.g., bug discovery, reporting and fixing), and obtained some interesting findings. For instance, there is little evidence showing that fixing performance bugs has a high chance of introducing new bugs. This encourages developers to conduct performance optimization whenever possible. However, there is a lack of similar studies on performance bugs in smartphone applications. Our work fills this gap by studying 70 real-world performance bugs from large-scale and popularly-downloaded Android applications. Our study also reveals some interesting findings, which differ from those for performance bugs in PC or server-side applications. These findings can help researchers and practitioners to better understand performance bugs in smartphone applications, as well as proposing new techniques to fight with these bugs (e.g., as we did in our case study).

6. CONCLUSION AND FUTURE WORK

In this paper, we conducted an empirical study of 70 performance bugs from real-world Android applications. We reported our study results, which revealed some unique features of performance bugs in smartphone applications. We also identified some common bug patterns, which can support related research on bug detection, performance testing and debugging. To validate the usefulness of our empirical findings, we implemented a static code analyzer, PerfChecker, to detect two of our identified performance bug patterns. We applied it to 29 real-world Android applications. It detected 126 matching instances of the two bug patterns, and 68 of them were quickly confirmed by developers as previously-unknown performance issues. Besides, developers also fixed 20 of the confirmed issues accordingly. This encouragingly confirmed our empirical findings' and PerfChecker's usefulness in detecting performance bugs for smartphone applications.

In future, we plan to conduct more investigations of performance bugs in smartphone applications, aiming to identify more bug patterns and build bug taxonomies. We also plan to design effective techniques to detect these bugs and help developers conduct performance optimization in an easier way. We hope that our work together with related work can help improve the performance and user experience for smartphone applications, which will benefit millions of smartphone users.

7. ACKNOWLEDGMENTS

This work was partially funded by Research Grants Council (General Research Fund 611912) of Hong Kong, and by National High-tech R&D 863 Program (2012AA011205) and National Natural Science Foundation (61100038, 91318301, 61321491, 61361120097) of China. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

8. REFERENCES

[1] "Android processes and threads documentation." URL: http://developer.android.com/guide/components/processes-and-threads.html.

[2] "Android programming tips for performance." URL: http://developer.android.com/training/articles/perf-tips.html.

[3] "Android list view programming guidelines." URL: http://developer.android.com/training/improving-layouts/smooth-scrolling.html.

[4] "Android Wifi Tether repository." URL: https://code.google.com/p/android-wifi-tether/.

[5] "AnySoftKeyboard repository." URL: https://github.com/AnySoftKeyboard/AnySoftKeyboard.

[6] "APG repository." URL: https://code.google.com/p/android-privacy-guard/.

[7] "Bitcoin Wallet repository." URL: https://code.google.com/p/bitcoin-wallet/.

[8] "c:geo repository." URL: https://github.com/cgeo/cgeo.

[9] "Chrome testing tools." URL: https://sites.google.com/a/chromium.org/dev/developers/testing.

[10] "ConnectBot repository." URL: https://code.google.com/p/connectbot/.

[11] "Crawler4j website." URL: https://code.google.com/p/crawler4j/.

[12] Ellis, P. D. 2010. The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results. *Cambridge University Press*.

[13] "FBReaderJ repository." URL: https://github.com/geometer/FBReaderJ.

[14] "Firefox built-in profiler for performance analysis." URL: https://developer.mozilla.org/en-US/docs/Performance.

[15] "Firefox frame rate meter tool webpage." URL: http://blog.mozilla.org/devtools/tag/framerate-monitor/.

[16] "Geohash Droid repository." URL: https://code.google.com/p/geohashdroid/.

[17] "GitHub website." URL: https://github.com/.

[18] "Google Code website." URL: https://code.google.com/.

[19] "Google Play store." URL: https://play.google.com/store.

[20] "Google Play Wiki Page." URL: http://en.wikipedia.org/wiki/Google_Play.

[21] Grechanik, M., Fu, C., and Xie, Q. 2012. Automatically finding performance problems with feedback-directed learning software testing. In *Proc. 34th Int'l Conf. Soft. Engr.* ICSE '12. 156-166.

[22] Guo, C., Zhang, J., Yan, J., Zhang, Z., and Zhang, Y. 2013. Characterizing and detecting resource leaks in Android applications. In *Proc. ACM/IEEE Int'l Conf. Automated Soft. Engr*. ASE '13, 389-398.

[23] Hao, S., Li, D., Halfond, W.G.J., and Govindan, R. 2013. Estimating mobile application energy consumption using program analysis. In *Proc. 35th Int'l Conf. Soft. Engr*. ICSE '13. 92-101.

[24] "IMSDroid repository." URL: https://code.google.com/p/imsdroid/.

[25] Jensen, C. S., Prasad, M. R., and Møller, A. 2013. Automated testing with targeted event sequence generation. In *Proc. Int'l Symp. Software Testing and Analysis*. ISSTA '13. 67-77.

[26] Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. 2009. Automated performance analysis of load tests. In *Proc. Int'l Conf. Software Maintenance*. ICSM '09. 125-134.

[27] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu S. 2012. Understanding and detecting real-world performance bugs. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. PLDI '12. 77-88.

[28] Jovic, M., Adamoli, A., and Hauswirth, M. 2011. Catch me if you can: performance bug detection in the wild. In *Proc. ACM Int'l Conf. Object-Oriented Programming Systems Languages and Applications*. OOPSLA '11. 155-170.

[29] Kwon, Y., Lee, S., Yi, H., Kwon, D., Yang, S., Chun, B., Huang, L., Maniatis, P., Naik, M., and Paek, Y. 2013. Mantis: automatic performance prediction for smartphone applications. In *Proc. USENIX Annual Tech. Conf.* USENIX '13. 297-308.

[30] Liu, Y., Xu, C., and Cheung, S. C. 2013. Where has my battery gone? Finding sensor related energy black holes in smartphone applications. In *Proc. IEEE Int'l Conf. Pervasive Computing and Communications.* PERCOM '13. 2-10.

[31] Mann, H. B., and Whitney, D. R. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, vol. 18, 50-60.

[32] Memon, A. M., Soffa, M. L., and Pollack, M. E. 2001. Coverage criteria for GUI testing. In *Proc. 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '01. 256-267.

[33] "Mozilla Cross-References." URL: https://mxr.mozilla.org/.

[34] "My Tracks repository." URL: https://code.google.com/p/mytracks/.

[35] Nistor, A., Song, L., Marinov, D., and Lu, S. Toddler: detecting performance problems via memory-access patterns. 2013. In *Proc. Int'l Conf. Soft. Engr.* ICSE '13. 562-571.

[36] Nistor, A., Jiang, T., and Tan, L. 2013. Discovering, reporting, and fixing performance bugs. In *Proc. 10th Working Conf. Mining Software Repositories*. MSR '13. 237-246.

[37] "OI File Manager repository." URL: https://github.com/openintents/filemanager.

[38] "Omnidroid repository." URL: https://code.google.com/p/omnidroid/.

[39] "Open GPS Tracker repository." URL: https://code.google.com/p/open-gpstracker/.

[40] "Osmand repository." URL: https://code.google.com/p/osmand/.

[41] Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. 2012. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. 10th Int'l Conf. Mobile Systems, Applications, and Services*. MobiSys '12. 267-280.

[42] Pathak, A., Hu, Y. C., and Zhang, M. 2012. Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof. In *Proc. 7th ACM European Conference on Computer Systems*. EuroSys '12. 29-42.

[43] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. 2011. Profiling resource usage for mobile applications: a cross-layer approach. In *Proc. 9th Int'l Conf. Mobile Systems, App's, and Services*. MobiSys '11. 321-334.

[44] Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., and Shayandeh, S. 2012. AppInsight: mobile app performance monitoring in the wild. In *Proc. 10th USENIX Conf. Operating Systems Design and Implementation*. OSDI '12. 107-120.

[45] Shen, K., Zhong, M., and Li, C. 2005. I/O system performance debugging using model-driven anomaly characterization. In *Proc. 4th USENIX Conf. File and Storage Tech*. FAST '05. 309-322.

[46] "SourceForge website." URL: http://sourceforge.net/.

[47] "Soot: a Java Program Optimization Framework." URL: http://www.sable.mcgill.ca/soot/.

[48] "Ushahidi repository." URL: https://github.com/ushahidi/Ushahidi_Android/.

[49] "WebSMS repository." URL: https://code.google.com/p/websmsdroid/.

[50] "XBMC Remote repository." URL: https://code.google.com/p/android-xbmcremote/.

[51] Xiao, X., Han, S., Zhang, D., and Xie, T. 2013. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proc. Int'l Symp. Software Testing and Analysis*. ISSTA '13. 90-100.

[52] Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., and Sevitsky, G. 2010. Finding low-utility data structures. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. PLDI '10. 174-186.

[53] Yang, S., Yan, D., and Routev, A. 2013. Testing for poor responsiveness in Android applications. In *Proc. International Workshop on the Engineering of Mobile-Enabled Systems*. MOBS '2013. 10-20.

[54] Zaman, S., Adams, B., and Hassan, A. E. 2012. A qualitative study on performance bugs. In *Proc. Working Conference on Mining Software Repository*. MSR '12. 199-208.

[55] Zhang, L., Gordon, M. S., Dick, R. P., Mao, Z., Dinda, P. A., and Yang, L. 2012. ADEL: an automated detector of energy leaks for smartphone applications. In *Proc. 10th International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '12. 363-372.

[56] Zhang, L., Bild, D. R., Dick, R. P., Mao, Z. M., and Dinda, P. 2013. Panappticon: event-based tracing to measure mobile application and platform performance. In *Proc. 11th International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '13. 1-10.

[57] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R., Mao, Z., and Yang, L. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. the 8th Int'l Conf. Hardware/Software Codesign and System Synthesis*. CODES+ISSS '10. 105-114.