2016 IEEE 27th International Symposium on Software Reliability Engineering

Experience Report: Detecting Poor-Responsive UI in Android Applications

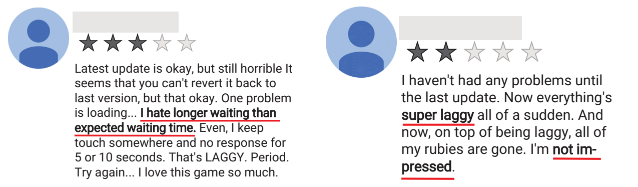
Yu Kang[†‡], Yangfan Zhou[*¶], Min Gao[*¶], Yixia Sun[§], Michael R. Lyu[†‡]
[*]School of Computer Science, Fudan University, Shanghai, China
[†]Computer Sci. & Eng. Dept., The Chinese University of Hong Kong, China
[‡]Shenzhen Research Institute, The Chinese University of Hong Kong, China
[§]School of Management, Zhejiang University, Hangzhou, Zhejiang, China
[¶]Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China
{ykang, lyu}@cse.cuhk.edu.hk, {zyf, 14210240071}@fudan.edu.cn, sunyixia@zju.edu.cn

***Abstract*—Good user interface (UI) design is key to successful mobile apps. UI latency, which can be considered as the time between the commencement of a UI operation and its intended UI update, is a critical consideration for app developers. Current literature still lacks a comprehensive study on how much UI latency a user can tolerate or how to identify UI design defects that cause intolerably long UI latency. As a result, bad UI apps are still common in app markets, leading to extensive user complaints. This paper examines user expectations of UI latency, and develops a tool to pinpoint intolerable UI latency in Android apps. To this end, we design an app to conduct a user survey of app UI latency. Through the survey, we find the tendency between user patience and UI latency. Therefore a timely screen update (*e.g.*, loading animations) is critical to heavy-weighted UI operations (*i.e.* those that incur a long execution time before the final UI update is available). We then design a tool that, by monitoring the UI inputs and updates, can detect apps that do not follow this criterion. The survey and the tool are open-source released on-line. We also apply the tool to many real-world apps. The results demonstrate the effectiveness of the tool in combating app UI design defects.**

I. Introduction

Rapid user interface (UI) responsiveness is a critical factor in the software quality of mobile apps. Apps with poor UI responsiveness lead to many user complaints [1]. Such performance defects are threat to software reliability [2], [3], [4]. Figure 1 presents two examples of user complaints on Google Play, a popular Android app market. Users give the app a low rating due to its poor responsiveness. Users may also have different expectations about UI latency (*i.e.*, the time between the commencement of a user operation and the corresponding UI update) in different UI operations. As suggested by the comments shown in Figure 1a, users "hate longer waiting than expected waiting time." Achieving rapid UI responsiveness and designing better UIs to boost user patience have long been goals of both the academic and industrial communities [5].

The key to user satisfaction with UI responsiveness is to offer timely UI feedback (*e.g.*, showing an animation to indicate a background task is being conducted) on user operations [6]. It is widely accepted that mobile devices may not be able to immediately complete all of the tasks intended by a UI operation due to resource limitations. For example, operations that involve loading Internet resources or accessing



[redacted]
★★★☆☆
Latest update is okay, but still horrible It seems that you can't revert it back to last version, but that okay. One problem is loading... I hate longer waiting than expected waiting time. Even, I keep touch somewhere and no response for 5 or 10 seconds. That's LAGGY. Period. Try again... I love this game so much.

(a) Rates 3 vs. avg. rating 4.4

[redacted]
★★☆☆☆
I haven't had any problems until the last update. Now everything's super laggy all of a sudden. And now, on top of being laggy, all of my rubies are gone. I'm not impressed.

(b) Rating 2 vs. avg. rating 4.3

Fig. 1: Examples of user ratings and comments about bad UI responsiveness

remote databases are types of "heavy-weighted" operations that require long waits before they are complete. In such cases, it is necessary to provide quick UI feedback to the user to let her know that the operation is being processed.

However, designing such feedback for every possible heavy-weighted UI operations requires a daunting amount of development effort. Moreover, developers generally have no idea on the latency on each UI operation. Without a comprehensive performance test and a handy tool to record UI latency, developers may neglect potential heavy-weighted operations.

We consider operations that may require a long execution time without offering quick UI feedback as poor-responsive operations. Poor-responsive operations, as software design defects, should be detected before an app is released to limit their influence on user experience. However, there are currently no methods for detecting such defects. First, there is no comprehensive understanding of what degree of UI latency (*i.e.*, latency threshold) leads to poor-responsive operations. The Android framework expects Android apps to be responsive in 5 seconds, otherwise it produces an "Application Not Responding (ANR)" alert [7]. Therefore, 5 seconds can be viewed as a loose upper bound for the latency threshold of poor-responsive operations. Alternatively, Google suggests that 200 ms as a general threshold beyond which users will perceive slowness in an application [7]. Therefore, 200 ms can be viewed as a lower bound for the latency threshold of poor-responsive operations. However, how much UI latency a typical user can really tolerate remains unknown, this information is a prerequisite for detecting poor-responsive operations.

2332-6549/16 $31.00 © 2016 IEEE
DOI 10.1109/ISSRE.2016.16

Second, currently there is no tool that can detect poor-responsive operations during an app test run. The official tool StrictMode [8] or other tools like Asynchronizer [9] can detect operations that block the UI thread, but they cannot detect non-blocking operations that incur long UI latency. The official performance diagnosing tools Method Tracing (and TraceView) cannot correlate operations with UI updates [10]. Other performance diagnosis tools such as Panappticon [11] also cannot capture the performance of UI feedback while a UI operation is being processed.

Hence, a tool that can detect poor-responsive operations during an app test run is necessary to combat poor UI responsiveness. In this paper, we first provide a threshold for classifying poor-responsive operations based on a real-world user study. Moreover, we observe that Android apps generally use a unique pattern when conducting UI updates. Specifically, although Android has a complicated procedure for conducting UI updates that involves diverse components, we find that every app uses a similar communication pattern based on a specific system process of Android when it conducts UI updates. Therefore, we can design a tool to track such patterns and detect poor-responsive operations.

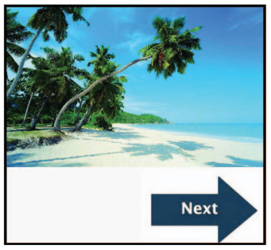
The contributions of this paper are as follows.

1) We conduct a real-world user study via a comprehensive survey. Where we find many insights on the user experiences and the UI latency. We thus find a reasonable threshold to detecting poor-responsive operations.
2) A handy tool, called `Pretect` (*P*oor-*Re*sponsive UI De*tect*ion in Android Applications), is implemented and open-source released [12]. The tool aims at detecting poor-responsive operations. We use the tool to find many UI design defects in many real-world Android apps.

The rest of this paper is organized as follows. Section II motivates the work by introducing an example of common program defect that causes poor-responsive operations. Section III presents our study on the relationship between user patience and operations delay levels, which also provides the motivation for developing our new tool. Sections IV and V illustrate the framework design and implementation of our tool. Section VI uses case studies to demonstrate the correctness and effectiveness of the tool. Section VII provides some discussions on the tool design considerations. Section VIII discusses related work. Finally, Section IX concludes this paper.

II. Motivation

Android has unique UI design patterns. The main thread of an app is the sole thread that handles the UI-related operations [13], such as processing user inputs and displaying UI components (*e.g.*, buttons and images). When a valid user input (*i.e.*, a UI event) comes, the main thread invokes the corresponding *UI event procedure*, *i.e.*, the codes that handle the UI event. However, some UI event procedures may be time-consuming; for example, a procedure may involve downloading a large file from the Internet. Android prevents such procedures from blocking the main thread, *i.e.*, unable to respond to other user operations, by introducing the Application Not Responding (ANR) [14] mechanism.



```
private class ImageDownloader extends AsyncTask<String, Void, Bitmap> {
    protected Bitmap doInBackground(String... urls) {
        return downloadBitmap(urls[0]);
    }

    protected void onPostExecute(Bitmap result) {
        imageView.setImageBitmap(result);
    }
}
```

Fig. 2: Screenshot and related source code of a simple gallery

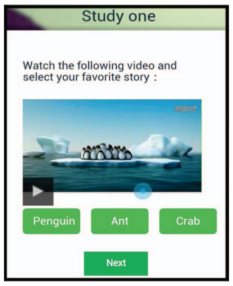
Commonly, Android apps conduct heavy tasks in an asynchronous manner so as not to block the UI thread. More specifically, when accepting a user operation, an app will start executing time-consuming tasks asynchronously in threads other than the main thread. Once the asynchronous task is finished, there is usually a callback to the main thread to update the UI accordingly [13].

The UI responsiveness is poor if the asynchronous task takes a long time to execute and there is no feedback (*e.g.*, no loading animation) during its execution. Providing no feedback to users is a common mistake of developers. Figure 2 shows a simple gallery design and the source code of its core functionality. The `ImageDownloader` task downloads the image with the given URL and then updates the imageView. Although such functionality is trivial, it contains a defect that may lead to slow UI responsiveness. When the image is large, or the Internet connection is poor (*e.g.*, on a cellular network), the user has to wait a long time before the image is loaded. The app provides no feedback during this waiting period. The user may be uncertain whether she has successfully touched the “next” sign shown in Figure 2. A better design for this case is showing the progress of loading with `onProgressUpdate` function or displaying an instance of `ProgressBar`/`ProgressDialog`.

Although Google has offered some responsive Android widgets such as `SwipeRefreshLayout`, in most cases, they only provide suggestions for improving responsive UI design (*e.g.*, showing the progress of background work using a ProgressBar) [14]. In next section, we present a user study that shows how poor-responsive UI design hurts an app.

III. User study

Unlike traditional PC apps, mobile apps are usually executed in an exclusive manner. Users generally do not switch their focus to other windows while an app is being used. Therefore, users tend to be more impatient with delays in mobile apps. This effect is not studied before, thus we design



Watch the following video and select your favorite story :

Fig. 3: Screen shot of survey app

a user study to reveal it. The study is available online [12]. Our study examines the relationship between UI latency and user patience. The results can guide the design of our tool.

We implement a mobile app with different delay levels by force sleeping and collect user feedback. The feedback reveals user tolerance under different delay levels. The app interacts with the users using common multimedia (*e.g.*, video, audio, microphone, etc.). The app requests user interaction (e.g., clicking buttons) before it continues to the next page (*i.e.*, an Android activity). A designed delay is attached to each user interaction. A sample page of the app is shown in Figure 3. After running the app with a designed procedure, we ask the subjects to rate 1-9 for UI responsiveness, their patience (*e.g.*, rate 1 for user impatient, 9 for user patient) and their free-text feedback on their experience of using the app. The statistical analysis of the ratings show the trends in user patience.

A. Test settings

In the study, we set three delay levels: 200 ms, 500 ms, and 2 seconds. As it is hard for a single user to rate fairly each separate operation with a different delay, we use a between-subject design [15], [16]. Thus, each user has one assigned delay level throughout the test and rates the app's overall performance; then the rates are compared between the sets of similar users.

The parameter settings are chosen based on the previous study of Johnson [6]; 200 ms is the editorial "window" for events that reach user consciousness, 500 ms is the attentional "blink" (inattentiveness to other objects) following recognition of an object, and 2 seconds is the maximum duration of a silent gap between turns in person-to-person conversation [6]. Google suggests 100 to 200 ms as the threshold at which users will perceive slowness in an application [7].

We take the following measures to address possible threats to the validity of our user study. 1) Subjects do not rate objectively if they know the purpose of the study. Therefore, we design more questions than required to hide our purpose. We ask the subjects to rate for about ten common questionnaire questions (*e.g.*, [17], [18], [19]), only two of which are related to user patience. We also ask the subjects their understanding on the purpose of the study. The results verify that the subjects are unaware of our real purpose. 2) The ability to learn a new app varies among subjects. To remove this effect, we conduct a practice before the study, but do not include the results of the practice in our analysis. 3) Internet delays are a common source of delay [2], [20], which can make subjects impatient. Therefore, to avoid introducing extra Internet delays, our tests are all conducted on local area networks.

TABLE I: Patience measurement under different delay levels

Delay level	Mean	Std. Deviation	N
200 ms	5.59	2.14	38
500 ms	4.68	1.80	37
2000 ms	4.24	2.43	41
Total	4.82	2.21	116

TABLE II: Pairwise patience measurement comparisons on different delay levels

			Dependent Variable				Dependent Variable
delay_level Simple Contrast			B2Items	delay_level Simple Contrast			B2Items
500 ms vs. 200 ms	Contrast Estimate		-0.92	2000 ms vs. 200 ms	Contrast Estimate		-1.35
	Hypothesized Value		0.00		Hypothesized Value		0.00
	Difference (Estimate -		-0.92		Difference (Estimate -		-1.35
	Std. Error		0.50		Std. Error		0.48
	Sig.		0.07		Sig.		0.01
	95% Confidence Interval for Difference	Lower Bound	-1.90		95% Confidence Interval for Difference	Lower Bound	-2.31
		Upper Bound	0.07			Upper Bound	-0.39

B. Results

We collect 116 valid replies. All of the subjects are college students between the ages of 20 and 27. We first collect the user feedback to get a general impression of user patience. Then we utilize `SPSS` tool for a detailed statistical analysis of the rating scores.

The written user feedback gives a general impression of the variation in user patience. Under the 200 ms delay condition, the subjects seldom complain about the responsiveness, whereas under the 500 ms delay condition, there are many complaints. For example, "very slow response", "Lagged response", "poor performance and UI", "very irresponsive", "It really responds slowly", etc. When the UI delay reaches 2 seconds, most subjects complain about the responsiveness, and some even become angry. For example, "slow as f**k", "too lag to use", "seriously, it is irresponsive", "extremely irrsponsive!", etc. We can already see a relationship between user patience and UI responsiveness. We resort to statistical analysis for a more detailed understanding of the relationship.

To increase the reliability of the test, we ask questions about both UI responsiveness and user patience. The statistical analysis demonstrates that the ratings of perceived responsiveness and participants' patience are highly correlated, with a .75 Pearson correlation value (significant at $p < .01$). Therefore, we average the two ratings as a single patience measurement.

The patience measurement shows a clear negative relationship with the delay level. The descending trend in the ratings with the delay levels can be instantly observed in the mean and standard deviation values in Table I: 200 ms ($M = 5.59, SD = 2.14$), 500 ms ($M = 4.68, SD = 1.80$), and 2000 ms ($M = 4.24, SD = 2.43$). The between-subjects test shows that the differences in the measurements are significant with $F(2, 113) = 4.00$ under significance level ≈ 0.021.

Further pairwise comparisons between different delay levels, shown in Table II, reveal that users perceive a great difference between 200 ms and 2000 ms delay (Mean diff $= 1.35$, SD Error $= 0.48$, Sig $= 0.01$). Users' impatience also increases between 200 ms and 500 ms delay with a marginal significance (Mean diff $= 0.92$, SD Error $= 0.50$, Sig $= 0.07$). However, the 500 ms and 2000 ms delays do not provoke significantly different levels of impatience (Sig > 1).

The results suggest that 1) mobile users are impatient and sensitive to delays and 2) the developers should be careful about operations with delays larger than 500 ms.

IV. Overall framework for poor-responsive UI detection

In this section, we first introduce the UI design problem of our interest. Then, we propose a workflow to solve the problem. Finally, we consider how our tool fits within the execution flow.

A. Problem Specification

As mentioned in Section II, the Android framework is specially tailored to suit the UI-driven requirements of Android apps. Developers try to keep apps responsive. A common practice when processing long-term operations is to provide a loading bar/circle animation as feedback to users. However, it requires daunting human efforts to design feedback for every possible heavy-weighted operation. In practice, developers only notice extremely long-term operations that may trigger ANR. However, our user study shows that a 500 ms delay is already long enough for users to perceive bad UI design.

To facilitate the following discussions, we define several terms related to UI design.

Definition 1 (Operation feedback)**.** A screen update that is triggered after receiving a user operation (i.e., a button click).

Definition 2 (Feedback delay)**.** The latency between a UI operation and the first UI update that it triggers.

Definition 3 (Poor-responsive operations)**.** A UI operation is *poor-responsive* if its feedback delay is not less than a threshold T.

In this work, we aim at detecting poor-responsive operations with feedback that takes longer than T. The UI feedback should be given as soon as the input event is accepted, but not until the event has finished processing. The feedback can reassure users that their input being processed by the app. Without timely feedback, users are unsure whether their operations have been accepted or the touch screen is insensible. As a result, they become impatient.

B. Proposed Execution Flow

We propose an execution flow, shown in Figure 4, for detecting poor-responsive operations. First, the system takes a user input I from either a testing tool (e.g., Monkey [21], MonkeyRunner [22]) or human input. Then an *event monitor module* records the input event without interfering with the

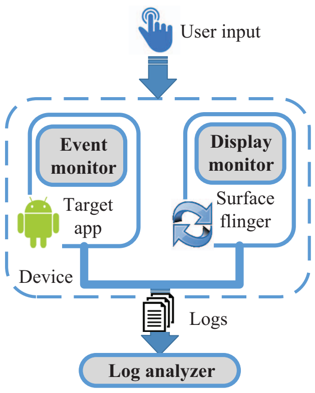


Fig. 4: Detecting poor-responsive operations

execution of the app. After the input event is processed, the display may or may not update within a preset time window. A *display monitor module* captures all of the display updates.

All of the related information is logged and analyzed offline. A *log analyzer module* analyzes the logs by calculating feedback delays for each input event. It then generates a report about poor-responsive operations. With the report, developers can easily detect the UI designs that should be improved.

C. Framework Design

We design a framework called `Pretect` (*P*oor-*re*sponsive UI De*tect*ion) for Android apps that realizes the proposed execution flow in Figure 4. The main modules are as follows.

1) Event Monitor: The event monitor module monitors the input events entered on the touchscreen. Whenever an input event (*e.g.*, touching a button) is performed, the event monitor module records the event information in the log including the type of event, the related UI component, and the time when the input occurs.

2) Display Monitor: The display monitor module monitors the screen updates. Whenever the screen refreshes, this module logs the UI update information including the source of the screen update (*i.e.*, the process that requests the UI update) and the update time.

3) Log Analyzer: The log analyzer module offline analyzes the logs of input events and screen updates. The purpose of this module is to identify the UI designs that could be improved. It reports poor-responsive operations. The supporting information it provides in the report includes the input event information, the feedback delay, and the related logs.

V. Implementation Details

We have implemented a poor-responsive UI detection tool based on the proposed `Pretect` framework. The implementation of the main modules in Section IV-C is described in detail in this section. Details could also be found from the released source codes [12].

It is worth noting that we rely on a dynamic instrumentation mechanism to keep `Pretect` compatible with most Android versions and devices. The mechanism requires no changes to the target app *per se*. It also does not require us to recompile the underlying OS and the Android framework. Moreover, the tool requires little human effort to install and apply.

We intercept Android framework methods in both Java and C. This approach is more light-weight and easier to implement than tracking the functions at the OS level, which typically requires heavy-weighted and sophisticated tools for kernel instrumentation. More importantly, we can rely on an Android-specific feature to conveniently track the relevant methods.

For Java method tracking, we note that, unlike general Linux processes, all Android app processes are created by duplicating a system process called `Zygote`. The framework binaries are loaded in `Zygote` before this duplication. Therefore, we can instrument the `Zygote` process and "hijack" the Java-based framework methods we are interested in before the app runs. When the app is running, the method invocations are inherently hijacked by `Pretect` via the forking of `Zygote`. Hence, we can easily track the methods. We implement this idea by adopting a tool called Xposed [23], which is usually used to improve the appearance of user interfaces [24]. It can substitute the original `Zygote` process with an instrumented one. We rely on its mechanism, and program our own codes to hijack the Java methods we are interested in.

For our C method interception, we note that the Android OS is based on the Linux kernel. A well-known Linux system tool named `ptrace`, which is commonly used in debugging tools (*e.g.*, gdb), is also available on Android. `Ptrace` makes it possible to inspect the child process of the parent process. `Ptrace` enables the parent process to read and replace the value of the register of the child process. We can utilize `ptrace` to attach code to a target process (with a known process ID `pid`). Then, we are able to take over the execution of the target process. By analyzing the elf-format library files of the target process, we can locate the memory addresses of the methods with the relevant names and invoke them accordingly. Therefore, it is feasible to invoke the `dlopen`, `dlsym` library-related system calls of the target process. We implement the idea by adopting a tool called LibInject [25].

A. Event Monitor

We implement the event monitor by instrumenting the related Android framework Java methods to obtain the input event information. In particular, we intercept several event dispatch methods of the `View` class; all touchable widgets such as Buttons, ImageView and ListView are subclasses of the `View` class. We carefully select a set of methods to cover all types of possible input events. The methods include `dispatchKeyEvent`, `dispatchTouchEvent` and some rarely used methods such as `dispatchKeyShortcutEvent`, `dispatchKeyEventPreIme` and `dispatchTrackballEvent`.

A sample log of an event is shown in line 1 of Figure 5. From this line we can see that the input event is a touch event, as indicated by the *Motion-Up* action. The touch event is conducted on a `ChildProportionLayout` with the ID *cutout_tab_artistic*. Besides the text ID, the unique

```
1.  … 2365 …: com.cyberlink.youperfect[Event]com.cyberlink.youperfect.widgetpool.common.
    ChildProportionLayout{425193c8V.E...C....P....270,0-540,67#7f0a051dapp:id/cutout_tab_art
    istic}_null-Motion-UP : 125696

2.  … 138 …: BIPC:****android.gui.SurfaceTexture****, sender_pid:2365, UptimeMilli: 127932
```

Fig. 5: Sample logs of `Pretect`

hexadecimal ID is also recorded. With the ID information, we can locate the component easily via the Hierarchy Viewer [26] tool published along with Android SDK. The event is performed 125696 ms after the system is booted. The highlighted information is important for the subsequent steps of the analysis.

B. Display Monitor

There are numerous functions that can update Android app displays (*e.g.*, `TextView.setText`, `ImageView.setImageBitmap`). Therefore, it is hard to list and instrument all of them. Moreover, updating UI display is a cross-layer procedure. Multiple layers created by the app, Android framework, kernel, and driver are involved in the procedure. Many components such as `SurfaceFlinger`, `OpenGL ES`, and `FrameBuffer` are included. The complicated nature of the UI display update mechanism makes it hard to trace. Luckily, we find that all of the UI updates are done via the `surfaceflinger` process provided by the Android OS. All of the UI update requests from the app are sent to the `surfaceflinger` process via `binder`, which is the standard inter-process communication (IPC) mechanism in Android.

Therefore, we can obtain the UI update information by intercepting the communication related functions of `binder`. More specifically, we intercept the `surfaceflinger` process on `ioctl` method of the shared library `libbinder.so`, which the binder mechanism is embedded in. The `ioctl` method is responsible for reading and writing the inter-process communication contents. We successfully intercept the invocations of `ioctl` to get detailed information about the UI update requests.

There are hundreds of `binder` communication messages per second; the UI updating messages are the one type of request that `SurfaceTexture` sends from the app under test (`pid` 2365 in this example). We show a UI update message in line 2 of Figure 5. The corresponding UI request time is 127932 ms after the system is booted.

C. Log Analyzer

The log analyzer extracts information from the logs collected during the offline tests. We implement it in Python.

The most important task of the log analyzer is to correlate the input events with their associated UI updates. The log analyzer first scans the logs to retrieve the input events. For each input event I, we check the following `binder` requests set ($\mathbb{R}$), until it reaches the next input event. With the information about the process ID (`pid`) of the sender process (*e.g.*, we record the sender's `pid`, as shown in the

TABLE III: List of applications

Type	Name	Operation
Synthetic	Worker Thread	load an image by worker thread
	AsyncTask	load an image by AsyncTask
	ThreadPool-Executor	load an image by ThreadPool-Executor
	HandlerThread	dump a table of a database
	IntentService	load an image by IntentService
Open source	K9Mail	refresh email Inbox
	ASqliteManager	dump a table of a database
	AFWall+	start network firewall
	Amaze	scan disk for all images
	ePUBator	convert a pdf file to epub type

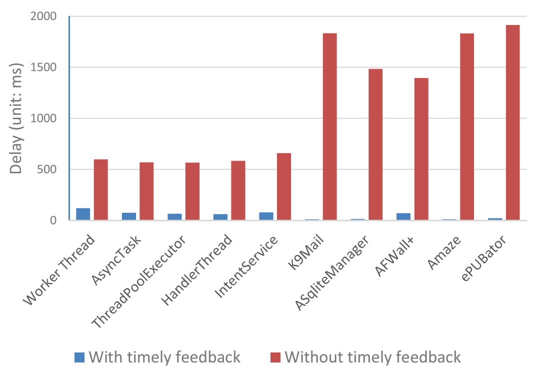


Fig. 6: Feedback delay of applications detected by `Pretect`

second line in Figure 5); the analyzer is able to distinguish the source of the `binder` request. We then search for the first `binder` request from set $\mathbb{R}$ such that 1) the sender's `pid` is the same as that of the app process (which we can determine from logs, for example, line 1 shown in Figure 5); and 2) the requested component is `SurfaceTexture`, which means it is a UI update request. We regard this UI update request as feedback of the input event I. Then it is not hard to calculate the feedback delay which is the time span between the input event and the feedback.

Worse than long feedback delay, some operations may have no feedback at all. For these operations, there are no following UI update logs. We filter such events by calculating the interval between the input event of interest and the following event. If the interval is too large, we regard the event as having no feedback. The log analyzer reports poor-responsive operations as these input events without feedback (*i.e.*, no following UI update request by the app) or without timely feedback (*i.e.*, feedback delay $\geq T$).

From lines 1-2 in Figure 5, we can infer that: 1) the two lines are an input event and its first UI update; and 2) the feedback delay of the input event is $127932 - 125696 = 2236$ ms.

VI. Experimental Study

In this section, we first conduct several experiments on synthetic apps and open source apps to show the effectiveness of our tool. Then we illustrate how our tool improves the UI designs by presenting several case studies.

A. Tool effectiveness validation

We examine the accuracy of `Pretect` by evaluating ten apps, including five synthetic benchmarks and five open source apps. These apps are selected to represent those with common heavy-weighted UI operations that may incur long UI latency. The apps and the selected operations are listed in Table III. The details of test configurations can be found on our website [12].

As mentioned in Section II, various types of asynchronous tasks and UI updates are common sources of poor-responsive operations. We implement all the five Android asynchronous mechanisms for this purpose, including Worker Thread, AsyncTask, ThreadPoolExecutor, HandlerThread, and Intent Service. The app loads an image after an asynchronous task finishes. We use sleep method to ensure the asynchronous tasks finish in about 500 ms. To validate `Pretect`'s ability to detect poor-responsive operations, we update the UI for each type of asynchronous mechanisms with two settings: separately with timely feedback and without timely feedback. More specifically, for the setting with timely feedback leading to responsive operations, we set a loading circle to appear while the asynchronous tasks are executing. For the setting without timely feedback, leading to poor-responsive operations, we do nothing when the asynchronous tasks are executing.

We validate our tool on open source projects. We select five open source projects from five different categories of F-Droid, an app market that hosts only free and open-source Android apps [27]. These open source projects have representative heavy-weighted UI operations that may incur long UI latency, as listed in Table III. These operations may incur UI delays from various sources such as network requests, database operations, system settings, disk scanning with querying content provider, and CPU-intensive computing. The original apps offer good timely UI feedback on these long-term operations. During the experiment, we manually switch off the UI feedback for comparison.

The results presented in Figure 6 show a notable difference between poor-responsive operations and responsive operations. `Pretect` can easily distinguish responsive operations from poor-responsive operations.

B. Overview of Experimental Results

We apply the tool to 115 popular Android apps covering 23 categories (including BooksReferences, Photography, Sports, etc.). We download apps on AndroidDrawer [28] from all of the categories except the library demo category. We randomly select five popular apps from each category. To test the compatibility of the tool, we conduct the experiments on two devices, Huawei G610-T11 with Android 4.2.2 and Lenovo K50-T5 with Android 5.1.

The overall statistics for the case where the threshold is 500 ms are shown in Table IV. We find that poor-responsive operations are common defects in UI designs. Of the 115 apps examined, 94 contain potential UI design defects. The maximum number of defects in an app is 23 and the minimum

TABLE IV: Feedback delay statistics

App Statistics		Issues Statistics	
# apps contain bugs	94	Total	327
Max bugs an app	23	Max	29189.0 (ms)
Min bugs an app	0	Min	504.0 (ms)
Avg. bugs per app	2.8	Avg	1503.9 (ms)
Median bugs per app	2	Stdv	2535.5

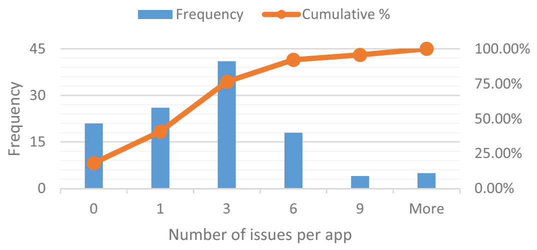


Fig. 7: Distribution of number of issues

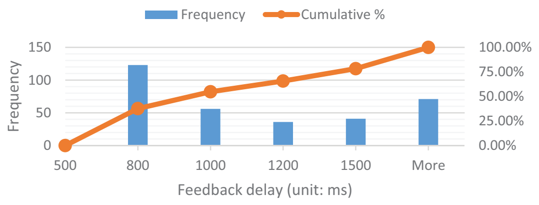


Fig. 8: Distribution of feedback delay

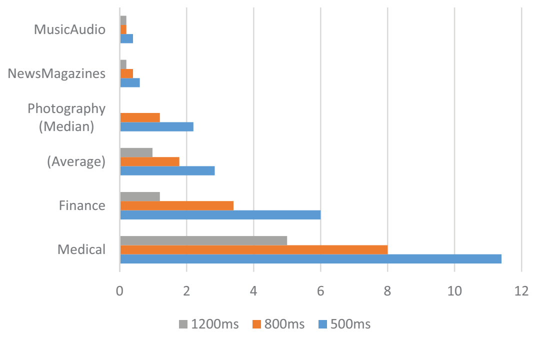


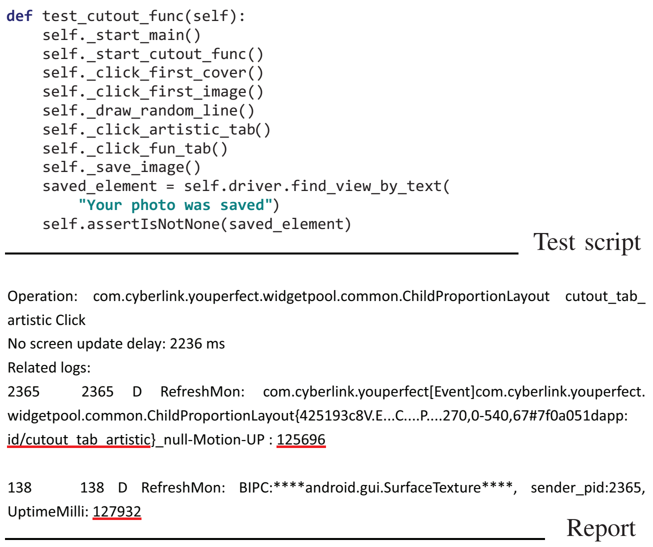
Fig. 9: Avg. number of cases per category by threshold

number is 0. On average, there are 2.8 (median 2) defects per an app. Long feedback delays are common. We find totally 327 independent components with feedback delays larger than 500 ms. The maximum delay is larger than 29 seconds. The distributions of the number of defects per app and UI feedback delays are shown in Figure 7 and 8. Less than 6 defects are reported for most apps. Therefore the report will not annoy developers even some of the defects are not of their interest.

Developers can define their preferred threshold for poor UI designs according to the category of their app. Figure 9 demonstrates the correlation of the number of bad components per category with the threshold cutoff. We also note the large differences between apps in different categories. For example, Medical apps contain the most number of bad UI design components (avg. 11.4 bad components), followed by

TABLE V: Top 10 components containing poor-responsive operations

Component	Issues found
android.widget.Button	78
android.widget.ListView	30
android.widget.ImageButton	22
android.widget.EditText	21
android.widget.ScrollView	20
android.widget.RelativeLayout	15
android.widget.ImageView	13
android.widget.TextView	10
android.widget.LinearLayout	10
android.support.v7.widget.Toolbar	7



```
def test_cutout_func(self):
    self._start_main()
    self._start_cutout_func()
    self._click_first_cover()
    self._click_first_image()
    self._draw_random_line()
    self._click_artistic_tab()
    self._click_fun_tab()
    self._save_image()
    saved_element = self.driver.find_view_by_text(
        "Your photo was saved")
    self.assertIsNotNone(saved_element)
```

Test script

```
Operation:   com.cyberlink.youperfect.widgetpool.common.ChildProportionLayout   cutout_tab_
artistic Click
No screen update delay: 2236 ms
Related logs:
2365     2365   D   RefreshMon:   com.cyberlink.youperfect[Event]com.cyberlink.youperfect.
widgetpool.common.ChildProportionLayout{425193c8V.E...C....P....270,0-540,67#7f0a051dapp:
id/cutout_tab_artistic}_null-Motion-UP : 125696

138     138  D  RefreshMon:  BIPC:****android.gui.SurfaceTexture****,  sender_pid:2365,
UptimeMilli: 127932
```

Report

Fig. 10: Selected report for YouCam Perfect

Finance apps (avg. 6). Music Audio contains the least number of bad UI design components (avg. 0.4), followed by News Magazines (avg. 0.6). Moreover, the threshold for feedback delay is a key factor for poor-responsive operations detection. According to the user study, 500 ms is a reasonable choice. Nevertheless, developers could choose their own threshold based on their own criteria. We have chosen 500 ms, 800 ms, 1200 ms respectively as the thresholds for our tests.

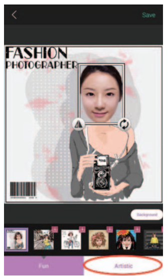
We also investigate the top-ranked components that commonly suffer from poor responsiveness. They are shown in Table V. When designing these UI components, developers must take special care to ensure their responsiveness.

Next, we select three representative cases to show how `Pretect` detects real-world poor-responsive operations.

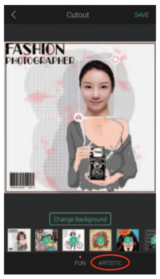
C. Case Study 1: YouCam

YouCam Perfect - Selfie Cam (http://www.perfectcorp.com/#ycp) is a popular selfie application. YouCam Perfect helps users to create the perfect image each and every time. The Android app has had more than 60 million downloads.

We apply `Pretect` to test Youcam Perfect, version 4.10.1. A representative testing scenario is shown in the "Test script"



(a) Ver. 4.10.1



(b) Ver. 5.4.1

Fig. 11: Screenshots of Youcam Perfect

section of Figure 10. A portion of the report generated by `Pretect` is shown in the lower section of Figure 10. We test on the *cutout* functionality of Youcam Perfect; this process cuts out a piece of an image and attaches it to a pre-defined template. As can be seen from the report, the `ChildProportionLayout` with ID “cutout_tab_artistic” is problematic. We detect a delay of 2236 ms without UI updates after the button is clicked. This decreases the quality of the software and hurts the user experience. This issue is the only reported issue, which we can immediately identify the defect. Via simply searching the component ID “cutout_tab_artistic” with Hierarchy Viewer [26], we successfully locate the problematic component, circled in Figure 11a. The component is the “Artistic” tab line 10 of the test scenario. By repeating the test scenario manually, we observe the latency that occurs without a feedback.

We have reported our findings to Perfect Corp. The leader of the YouCam Perfect development team has provided us with positive feedback on our results: “With your hint, we find that we have used a widget which tends to be slower. We will fix as soon as possible.” In a later version of Youcam Perfect 5.4.1, the UI design has been modified, as shown in Figure 11b. A test run with `Pretect` shows that this modification has reduced the feedback delay to below 100 ms.

D. Case Study 2: Bible Quotes

Bible Quotes (http://www.salemwebnetwork.com/) is a popular Bible app that provides verses from the Bible. Users can lookup verses in the Bible, save favorites, share quotes, etc. The app has more than a million downloads on Google Play.

We apply `Pretect` to test Bible Quotes, version 4.9. A representative testing scenario is shown in the “Test script” section of Figure 12. A portion of the report generated by `Pretect` is shown in the middle section of Figure 12. We test the functionality of the *donation* process of Bible Quotes. As from the report, the `Button` with ID “btnDonate01” is problematic. Note that this is the first issue in our generated report (other issues are about other donate buttons which are similar to this one). Via searching the component ID in Hierarchy Viewer, we immediately locate the button of the app. We find that there are no UI update-related logs generated between the time the “btnDonate01” button is clicked and

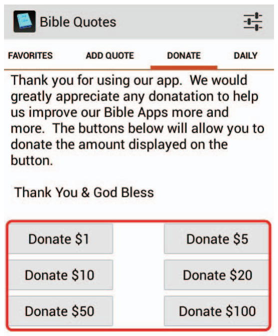
```
def test_donate_func(self):
    self._start_main()
    self._click_donate_tab()
    self._click_donate_button(1)
    self._click_donate_button(5)
    self._click_donate_button(10)
    self._click_donate_button(20)
    self._click_donate_button(50)
    self._click_donate_button(100)
    donate_view_element = self.driver.find_view_by_text(
        "Thank you",is_regular_expr=True)
    self.assertIsNotNone(donate_view_element)
```

Test script

```
Operation: Button btnDonate01 Click
No screen update delay: 2073 ms
Related logs:
2519    2519  D  RefreshMon:  com.dodsoneng.biblequotes[Event]android.widget.Button{41a3da
0VF.D..C.........0,300-225,372#7f09005aapp:id/btnDonate01}_"Donate $1"-Motion-UP : 78535

2519    2519 D RefreshMon: com.dodsoneng.biblequotes[Event]android.widget.Button{41b63330
VF.D..C.......315,300-540,372#7f09005bapp:id/btnDonate02}_"Donate $5"-Motion-DOWN : 80608
```

Report



Thank you for using our app. We would greatly appreciate any donation to help us improve our Bible Apps more and more. The buttons below will allow you to donate the amount displayed on the button.

Thank You & God Bless

Screen Shot

Fig. 12: Selected report for Bible Quotes

the time the “btnDonate02” button is clicked (corresponding to the test scenario steps 3-4). This means that the UI does not update for more than 2 seconds after the “btnDonate01” button is clicked. This clearly decreases the quality of the app. We further investigate the app to find the relevant information about the UI component. The logs reveal that the text on the button is “Donate $1”; the component is circled in the lower section of Figure 12. The component we are interested in is the “Donate $1” button on line 3 of the test scenario.

By repeating the test scenario manually, we observe that the operations provide no feedback. Similar issues can be found in all of the donation buttons. The implementation of donation functionality may have been omitted in this app. Therefore, these operations produce no feedback.

E. Case Study 3: Illustrate

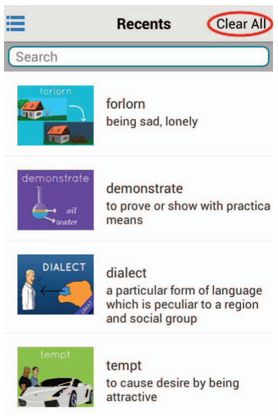
Illustrate - The Video Dictionary (http://www.mocept.com/illustrate/) is an effective app for teaching new words and their meanings. It provides videos with context and real-life examples allowing the learner to grasp definitions and usage with ease. ICAL TEFL, a leading provider of English language courses, says that it is fun and will help students. The University of Michigan Campus Life News recommend

```
def test_clear_all_func(self):
    self._start_main()
    self._click_first_word()
    for i in range(4):
        self._explore_more_word()
    self._click_back()
    self._click_menu()
    self._click_recents()
    self._click_clear_all()
    count_recent = self._count_recent()
    self.assertEquals(count_recent, 0)
```

Test script

```
Operation:    TextView clearall Click
No screen update delay:    2351 ms
Related logs:
2319    2319 D RefreshMon: com.mocept.illustrate[Event]android.widget.TextView{41c9c9f0V.ED..
C....P....0,17-100,54#7f09011fapp:id/clearall}_"Clear All"-Motion-UP : 181382
138      138  D  RefreshMon:  BIPC:****android.gui.SurfaceTexture**********,  sender_pid:2319,
UptimeMilli: 183733
```

Report


Recents
Clear All
Search
forlorn
being sad, lonely
demonstrate
to prove or show with practica means
dialect
a particular form of language which is peculiar to a region and social group
tempt
to cause desire by being attractive

Screen Shot

Fig. 13: Selected report for Illustrate

it as the best app in “7 Helpful Study Apps for GRE, LSAT, and GMAT Preparation”.

We apply `Pretect` to test Illustrate, version 1.2.7. A piece of the testing scenario is depicted in the upper section of Figure 13. Part of the report generated by `Pretect` is shown in the middle section of Figure 13. We test the *history clearing* functionality of Illustrate. As can be seen from the report, the `TextView` with ID “clearall” is problematic. Note that this is the only issue in our generated report. Via searching the ID in Hierarchy Viewer, we immediately locate the component. We detect a delay of 2351 ms without UI updates after the `TextView` is clicked. This decreases the quality of the software and tests users’ patience. We further investigate the app to find the relevant information about the UI component. An examination of the logs reveals that the text of the `TextView` is “Clear All”, which is circled in the lower section of Figure 13. The component of interest is the “Clear All” button in line 7 of the test scenario. By repeating the test scenario manually, we observe the latency without feedback.

VII. Discussions

`Pretect` is a tool for identifying poor responsiveness designs instead of obtaining precise delay. We have shown that `Pretect` can successfully distinguish poor responsiveness in long-processing situations in Section VI-A.

Our user study results show that more than 500 ms delay can cause the users become impatient. To conduct the survey, we choose most active mobile users (between 20 to 27 years old) and common usage scenarios (*e.g.*, video, audio, map). Users may tolerate 500 ms delay in other scenarios. Whereas, our focus is on revealing the relationship between user patience and UI latency, rather than obtaining the exact threshold of user impatience. We have shown in Section VI-B that the threshold is just a parameter of `Pretect`which can be easily set to suit the requirement of developers.

We write Appium [29] scripts to exercise apps in Section VI-C, VI-D and VI-E (complete scripts available on our website [12]). Apps are generally tested with scripts on various devices simultaneously to detect bugs in practice. Commonly functional exceptions are recorded while the performance of apps is not. It is hard to manually monitor the performance of apps on all devices. `Pretect` aims at automating UI-responsiveness test. With our tool, test scripts for functional tests can be utilized for performance tests.

Current approaches cannot properly detect poor-responsive operations. Approaches including StrictMode [8] and Asynchronizer [9] can detect operations that block the UI, but fail to detect asynchronous tasks that do not provide feedback. Other approaches (*e.g.*, Appinsight[30], Panappticon[11]) define a delay as the time interval between the initiation of an operation and the completion of all of the triggered tasks. These approaches aim to detect the abnormal execution of asynchronous tasks rather than feedback delay. In such cases, using `Pretect` to detect feedback delay is a better approach to improving the responsiveness of UI design. Moreover, these approaches may report some long-term background tasks (*e.g.*, download) as suspicious, as a result of focusing on the life cycle of the tasks. However, this kind of tasks do not update the display when finished. They are less relevant to the responsiveness. In comparison, `Pretect` pays attention to whether the app notifies the users that the operations that will trigger the background tasks have been accepted.

The screen refreshes even when the app UI does not update, as the action bar on the top of the screen also refreshes. However, the action bar is not our focus. Therefore, intercepting low level system functions related to full screen display refresh such as `eglSwapBuffers` in /system/lib/libsurfaceflinger.so is not a good choice. We further inspect the related screen update procedure and find `binder` used in the process. We then intercept `binder` and measure the screen update time of the app more precisely with the logged `pid`.

VIII. Related Work

A. Performance and GUI Design of Mobile Apps

Performance is a critical concern for mobile apps [2], [31]. Liu *et al.* [32] show that many performance issues are caused by blocking operations in the main thread. StrictMode [8] analyzes the main thread to find such blocking operations. Asynchronizer [9] provides an easy way to refactor specific blocking synchronous operations into standard `AsyncTask` implementations. AsyncDroid [33] further refactors `AsyncTask` to `IntentService` to eliminate the memory leakage problems. CLAPP [34] finds potential performance optimizations by analyzing loops. However, such static analysis-based tools cannot capture runtime execution dependency. Banerjee *et al.* [35], [36] design static analysis-driven testing for performance issues caused by anomalous cache behaviors. Tango [37], Outatime [38] and Cedos [39] optimize WiFi offloading mechanisms to keep a low-latency for app. SmartIO [40] reduces the app delay by reordering IO operations. Offloading tasks to remote servers can also reduce the delay [3], [4]. Resource leakage is a common source of performance issues and has been widely investigated [41]. These approaches solve specific performance issues.

User interface (UI) design is one of the key considerations in mobile app development [42]. Methods for diagnosing UI performance have captured much research attention recently. Method tracing [10] is an official tool that is often used to diagnose known performance issues due to its high overhead. QoE Doctor [43] bases its diagnosis on Android Activity Testing API [44], but can only handle pre-defined operations. Appinsight [30] is a tracing-based diagnosis tool for Windows phone apps. It traces all of the asynchronous executions from a UI event to its corresponding UI update, and identifies the critical paths that influence the performance. Panappticon [11] adopts a similar approach on Android. These studies focus on finding the anomalous task delays; in contrast, `Pretect` focus on identifying the delays that affect user experience.

Performance diagnosis often requires researches to execute the target app automatically. Script-based testing is widely used (*e.g.*, UIAutomator [45], Monkey runner [22], and Robotium [46]). The record-and-replay approaches (*e.g.*, MobiPlay [47], Reran [48] and SPAG-C [49]) record an event sequence during the manual exercising of the app, and generate replayable scripts. Complementary to these semi-automatic approaches, fuzz testing approaches (*e.g.*, Monkey [21], Dynodroid [50] and VanarSena [51]) generate random input sequences to exercise Android apps. Symbolic execution-based testing (*e.g.*, Jensen *et al.* [52], EvoDroid [53], A^3E [54], and SIG-Droid [55]) aims at exploring the app functions in a systematical way. Model-based testing (*e.g.*, Android Ripper [56], SwiftHand [57], and PBGT [58]) aims at generating a finite state machine model and event sequences to traverse the model. TestPatch [59] utilizes the GUI ripping methods for regression tests. Test case selection techniques (*e.g.*, [60], [61]) can also be adopted in exercising the target apps. App exercising mechanisms can work as plugin modules of `Pretect`, enabling the developers to exploit their merits under different circumstances.

B. Response Time and User-Tolerant UI Design

Tolerance for delays on mobile devices has been studied for long. Oulasvirta *et al.* [62] reveal that user attention spans vary from 4 to 16 seconds on mobile devices once the page loading has started. Anderson [5] suggests that user tolerance for touch screen latency below 580 ms. Jota *et al.* [63] and their follow-up work [64] showed that 1) the perceivable initial delay feedback ranges from 10 to 20 ms, and 2) the detectable threshold of direct and indirect operations ranging from 11 ms to 96 ms. Ng *et al.* [65] show that the user noticeable improvements below 10 ms. These results work as suggestions for touch screen UI design, while we move one step forward to detect the violations.

The above-mentioned research focuses on the absolute value of delays and does not consider how feedback affects user satisfaction with a delay. Visual and non-visual feedback is an important design element of delay-tolerable UI. Duis *et al.* [66] point out without experimental study that a system should let a user know immediately that her input has been received. Johnson [6] suggest showing a progress bar for long-term operations rather than nothing or only a busy bar. Roto *et al.* [67] show that multimodal feedback for delays of more than 4 seconds is required. Lee *et al.* [68] note that the absence of feedback affects user performance (*e.g.*, typing on flat piezo-electric buttons that have no tactile feedback significantly reduce expert typists performance). Kristoffersen *et al.* [69] suggest using audio feedback on mobiles. Ng *et al.* [65] recommend providing low-fidelity visual feedback immediately. Poupyrev *et al.* [70] find that tactile feedback is most effective when the GUI widgets need to be held down or dragged on the screen. Ripples [71] provide a special system on top of the screen that can give feedback about the successes and errors of the touch interactions. In contrast, we do not modify the current mobile UI framework. Our work focuses on detecting UI elements with long delays or with no feedback that may cause ambiguity. This information can be used by developers to avoid such problems.

IX. Conclusion

In this paper, we discuss the problem of responsive UI design in Android apps. We motivate the problem of detecting poor-responsive operations by conducting a user survey. The survey results show that users' patience is correlated with UI responsiveness. We design and implement a tool called `Pretect` that can detect poor-responsive operations. The tool is shown to work correctly on synthetic benchmarks and on open source apps. We further verify `Pretect` with real-world case studies. The results demonstrate the effectiveness of `Pretect`.

Acknowledgement

The work described in this paper was supported by the National Basic Research Program of China (973 Prj. No. 2014CB347701), the National Natural Science Foundation of China (Prj. No. 61332010), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14205214 of the General Research Fund), and 2015 Microsoft Research Asia Collaborative Research Program (Prj. No. FY16-RES-THEME-005). The main work was conducted when Yu Kang was a visiting student of Fudan University. Yangfan Zhou is the corresponding author.

References

[1] C. Gao, B. Wang, P. He, J. Zhu, Y. Zhou, and M. R. Lyu, "Paid: Prioritizing app issues for developers by tracking user reviews over versions," in *Proc. of the International Symposium on Software Reliability Engineering (ISSRE '15)*, 2015, pp. 35–45.

[2] C. Amrutkar, M. Hiltunen, T. Jim, K. Joshi, O. Spatscheck, P. Traynor, and S. Venkataraman, "Why is my smartphone slow? on the fly diagnosis of underperformance on the mobile internet," in *Proc. of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, 2013, pp. 1–8.

[3] Q. Wang, H. Wu, and K. Wolter, "Model-based performance analysis of local re-execution scheme in offloading system," in *Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, 2013, pp. 1–6.

[4] Q. Wang, M. G. Jorba, J. M. Ripoll, and K. Wolter, "Analysis of local re-execution in mobile offloading system," in *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '13)*, 2013, pp. 31–40.

[5] G. Anderson, R. Doherty, and S. Ganapathy, "User perception of touch screen latency," vol. 6769, pp. 195–202, 2011.

[6] J. Johnson, *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*, 2010.

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

[8] StrictMode. [Online]. Available: http://developer.android.com/reference/android/os/StrictMode.html

[9] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proc. of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '14)*, 2014, pp. 341–352.

[10] Profiling with Traceview and dmtracedump. [Online]. Available: http://developer.android.com/tools/debugging/debugging-tracing.html

[11] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda, "Panappticon: Event-based tracing to measure mobile application and platform performance," in *Proc. of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '13)*, 2013, pp. 1–10.

[12] Pretect: Poor responsiveness ui detection for android applications. [Online]. Available: http://www.cudroid.com/pretect

[13] Processes and Threads, http://developer.android.com/guide/components/processes-and-threads.html. [Online]. Available: http://developer.android.com/guide/components/processes-and-threads.html

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

[15] J. A. Hoxmeier and C. Dicesare, "System response time and user satisfaction: An experimental study of browser-based applications," in *Proc. of the Association of Information Systems Americas Conference (AMCIS '00)*, 2000, pp. 10–13.

[16] C. J. Goodwin, *Research in psychology : methods and design*, 1995.

[17] J. P. Chin, V. A. Diehl, and K. L. Norman, "Development of an instrument measuring user satisfaction of the human-computer interface," in *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '88)*, 1988, pp. 213–218.

[18] D. D. Suthers, R. Vatrapu, R. Medina, S. Joseph, and N. Dwyer, "Beyond threaded discussion: Representational guidance in asynchronous collaborative learning environments," *Computer & Education*, vol. 50, no. 4, pp. 1103–1127, May 2008.

[19] B. D. Harper, L. A. Slaughter, and K. L. Norman, "Questionnaire administration via the www: A validation & reliability study for a user satisfaction questionnaire." in *WebNet*, vol. 97, 1997, pp. 1–4.

[20] S. Srinivasan, F. Buonopane, S. Ramaswamy, and J. Vain, "Verifying response times in networked automation systems using jitter bounds," in *Proc. of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW '14)*, 2014, pp. 47–50.

[21] Ui/application exerciser monkey(monkey). [Online]. Available: http://developer.android.com/tools/help/monkey.html

[22] Monkeyrunner. [Online]. Available: http://developer.android.com/tools/help/monkeyrunner_concepts.html

[23] Xposed module repository. [Online]. Available: http://repo.xposed.info/

[24] Xposed module repository - module overview. [Online]. Available: http://repo.xposed.info/module-overview

[25] Libinject – c/c++ code injection library. [Online]. Available: http://blog.csdn.net/jinzhuojun/article/details/9900105

[26] Hierarchy viewer. [Online]. Available: http://developer.android.com/tools/help/hierarchy-viewer.html

[27] F-Droid. [Online]. Available: https://f-droid.org/

[28] Android drawer. [Online]. Available: http://www.androiddrawer.com/

[29] S. Shiwangi, R. Gadgil, and A. Chudgor, "Automated testing of mobile applications using scripting technique: A study on appium," *International Journal of Current Engineering and Technology (IJCET)*, vol. 4, no. 5, pp. 3627–3630, 2014.

[30] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild," in *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, 2012, pp. 107–120.

[31] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling user-perceived delays in server-based mobile applications," in *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013, pp. 85–100.

[32] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. of the International Conference on Software Engineering (ICSE '14)*, 2014, pp. 1013–1024.

[33] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming," in *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, 2015, pp. 224–235.

[34] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Clapp: Characterizing loops in android applications," in *Proc. of the Joint Meeting on Foundations of Software Engineering (FSE '15)*, 2015, pp. 687–697.

[35] A. Banerjee, "Static analysis driven performance and energy testing," in *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*, 2014, pp. 791–794.

[36] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "Static analysis driven cache performance testing," in *Proc. of the 2013 IEEE 34th Real-Time Systems Symposium (RTSS '13)*, pp. 319–329.

[37] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Accelerating mobile applications through flip-flop replication," in *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, 2015, pp. 137–150.

[38] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, 2015, pp. 151–165.

[39] Y. Moon, D. Kim, Y. Go, Y. Kim, Y. Yi, S. Chong, and K. Park, "Practicalizing delay-tolerant mobile apps with cedos," in *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, pp. 419–433.

[40] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang, "Reducing smartphone application delay through read/write isolation," in *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, 2015, pp. 287–300.

[41] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in android applications," in *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '13)*, 2013, pp. 411–420.

[42] M. E. Joorabchi, M. Ali, and A. Mesbah, "Detecting inconsistencies in multi-platform mobile apps," in *Proc. of the International Symposium on Software Reliability Engineering (ISSRE '15)*, 2015, pp. 450–460.

[43] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau, "Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis," in *Proc. of the Conference on Internet Measurement Conference (IMC '14)*, 2014, pp. 151–164.

[44] Automating User Interface Tests. [Online]. Available: http://developer.android.com/training/testing/ui-testing/index.html

[45] Android ui testing (uiautomator). [Online]. Available: http://developer.android.com/tools/testing/testing_ui.html/

[46] Robotium. [Online]. Available: https://code.google.com/p/robotium/

[47] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE '16)*, 2016.

[48] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *Proc. of the International Conference on Software Engineering (ICSE '13)*, 2013, pp. 72–81.

[49] Y.-D. Lin, J. Rojas, E.-H. Chu, and Y.-C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for android devices," *IEEE Transactions on Software Engineering (TSE)*, vol. 40, pp. 957–970, Oct 2014.

[50] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proc. of the Joint Meeting on Foundations of Software Engineering (FSE '13)*, 2013, pp. 224–234.

[51] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*, 2014, pp. 190–203.

[52] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA '13)*, 2013, pp. 67–77.

[53] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*, 2014, pp. 599–609.

[54] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proc. of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*, 2013, pp. 641–660.

[55] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '15)*, 2015, pp. 461–471.

[56] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, 2012, pp. 258–261.

[57] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proc. of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*, 2013, pp. 623–640.

[58] R. M. L. M. Moreira, A. C. R. Paiva, and A. Memon, "A pattern-based approach for gui modeling and testing," in *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '13)*, 2013, pp. 288–297.

[59] Z. Gao, C. Fang, and A. M. Memon, "Pushing the limits on automation in gui regression testing," in *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE '15)*, 2015.

[60] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proc. of the International Conference on Software Engineering (ICSE '15)*, 2015, pp. 483–493.

[61] V. Terragni, S.-C. Cheung, and C. Zhang, "Recontest: Effective regression testing of concurrent programs," in *Proc. of the International Conference on Software Engineering (ICSE '15)*, 2015, pp. 246–256.

[62] A. Oulasvirta, S. Tamminen, V. Roto, and J. Kuorelahti, "Interaction in 4-second bursts: The fragmented nature of attentional resources in mobile hci," in *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*, 2005, pp. 919–928.

[63] R. Jota, A. Ng, P. Dietz, and D. Wigdor, "How fast is fast enough? a study of the effects of latency in direct-touch pointing tasks," in *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*, 2013, pp. 2291–2300.

[64] J. Deber, R. Jota, C. Forlines, and D. Wigdor, "How much faster is fast enough? user perception of latency & latency improvements in direct and indirect touch," in *Proc. of the ACM Conference on Human Factors in Computing Systems (CHI '15)*, 2015, pp. 1827–1836.

[65] A. Ng, J. Lepinski, D. Wigdor, S. Sanders, and P. Dietz, "Designing for low-latency direct-touch input," in *Proc. of the ACM Symposium on User Interface Software and Technology (UIST '12)*, 2012, pp. 453–464.

[66] D. Duis and J. Johnson, "Improving user-interface responsiveness despite performance limitations," in *Proc. of the IEEE Computer Society International Conference (Compcon Spring '90)*, Feb 1990, pp. 380–386.

[67] V. Roto and A. Oulasvirta, "Need for non-visual feedback with long response times in mobile hci," in *Special Interest Tracks and Posters of the International Conference on World Wide Web (WWW '05)*, 2005, pp. 775–781.

[68] S. Lee and S. Zhai, "The performance of touch screen soft buttons," in *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*, 2009, pp. 309–318.

[69] S. Kristoffersen and F. Ljungberg, " "making place" to make it work: Empirical explorations of hci for mobile cscw," in *Proc. of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP '99)*, 1999, pp. 276–285.

[70] I. Poupyrev and S. Maruyama, "Tactile interfaces for small touch screens," in *Proc. of the ACM Symposium on User Interface Software and Technology (UIST '03)*, 2003, pp. 217–220.

[71] D. Wigdor, S. Williams, M. Cronin, R. Levy, K. White, M. Mazeev, and H. Benko, "Ripples: Utilizing per-contact visualizations to improve user interaction with touch displays," in *Proc. of the Symposium on User Interface Software and Technology (UIST '09)*, 2009, pp. 3–12.