DiagDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions

Yu Kang[†‡] Yangfan Zhou[*§] Hui Xu[†‡] Michael R. Lyu[†‡]
[*]School of Computer Science, Fudan University, China
[†]Shenzhen Research Institute, The Chinese University of Hong Kong, China
[‡]Department of Computer Science and Engineering, The Chinese University of Hong Kong, China
[§]Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China

ABSTRACT

Rapid UI responsiveness is a key consideration to Android app developers. However, the complicated concurrency model of Android makes it hard for developers to understand and further diagnose the UI performance. This paper presents `DiagDroid`, a tool specifically designed for Android UI performance diagnosis. The key notion of `DiagDroid` is that UI-triggered asynchronous executions contribute to the UI performance, and hence their performance and their runtime dependency should be properly captured to facilitate performance diagnosis. However, there are tremendous ways to start asynchronous executions, posing a great challenge to profiling such executions and their runtime dependency. To this end, we properly abstract five categories of asynchronous executions as the building basis. As a result, they can be tracked and profiled based on the specifics of each category with a dynamic instrumentation approach carefully tailored for Android. `DiagDroid` can then accordingly profile the asynchronous executions in a task granularity, equipping it with low-overhead and high compatibility merits. The tool is successfully applied in diagnosing 33 real-world open-source apps, and we find 14 of them contain 27 performance issues. It shows the effectiveness of our tool in Android UI performance diagnosis. The tool is open-source released online.

CCS Concepts

•Software and its engineering → Software performance;

Keywords

Android; Performance Diagnosis; UI Responsiveness

1. INTRODUCTION

As daily-use personal devices, smartphones are required to provide quick response to the user interface (UI). UI performance of a smartphone app is a critical factor to its user experience, and hence becomes a major concern to developers [38, 39]. Many recent research efforts have therefore been put on addressing the performance issues of Android apps (*e.g.*, Asynchronizer [36], Panappticon [72]). However, poor UI performance of Android apps remains a widely-complaint type of issues among users [38, 39]. App developers are still lacking a handy tool to help combat performance issues.

Android provides a non-blocking paradigm to process UI events (*i.e.*, user inputs) for its apps. The UI *main* thread dispatches valid UI events to their corresponding UI event procedures (*i.e.*, the UI event-handling logic). A UI event procedure generally runs in an asynchronous manner, so that the main thread can handle other UI events simultaneously. After the asynchronous part is done, the UI can be updated with a call-back mechanism. This paradigm will lead to complicated concurrent executions. The asynchronous execution processes may bear implicit dependency during their runtime. For example, two may be scheduled to run in the same thread by Android, and one may consequently wait for the other to complete. Such unexpected waiting may result in longer delay for a UI procedure, leading to UI performance issues. However, it is hard to predict such runtime dependency during the coding phase due to the complications of Android's asynchronous execution mechanisms [53]. Performance issues are hence inevitable.

Concurrency is a notorious source of bugs [41]. Current tools for diagnosing Android UI performance issues generally consider either the synchronous part of the UI event procedure [63], or the execution process of one UI event procedure *per se* [72]. They do not focus on the dependency of multiple asynchronous execution processes. Hence, they are still not enough to cope with the UI performance issues largely caused by such runtime dependency.

Long-term testing is a well-known, viable means to trigger bugs caused by concurrency [34]. Unfortunately, we lack an automatic mechanism to verify whether there exists a performance issue in the long-term testing. Manual inspection of the tremendous traces produced by current method tracing tools (*e.g.*, Traceview [54]) is extremely labor-intensive, if not infeasible, not to mention their huge overhead.

We find that unlike general concurrent programs [13, 14, 28, 32], an Android UI event procedure can be anatomized into a set of trackable *tasks*, which can then be properly profiled so as to facilitate the detection and localization of performance issues. Specifically, although Android supports tremendous ways to schedule asynchronous executions, we conclude that they can actually be abstracted as five categories. Executions of each category can be tracked and profiled in task granularity according to their specifics. UI performance can hence be modeled by the performance of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950316

```
public class MyActivity extends Activity {
    private class RetrieveDataTask extends AsyncTask<String, Void, String> {
        ...
        // doInBackground will be executed asynchronously in a worker thread
        protected String doInBackground(String... urls) {
            ...   // Retrieve content from Internet
            return content;
        }

        // onPostExecute will be invoked in the main thread after doInBackground
        // completes, which shows the downloaded content in the UI.
        protected void onPostExecute(String content){
            this.textView.setText(content);
        }
    }
    ...
    private class MyOnClickListener implements OnClickListener{
        protected void onClick(View v){
            RetrieveDataTask task1 = new RetrieveDataTask(textView1);
            // call execute method according to the example of official document
            task1.execute(url1);
        }
    }
}
```

Figure 1: An AsyncTask example

the tasks. We further tackle the complication of runtime dependency via examining the dependency of tasks, which can be solved by checking whether the tasks request the same execution unit (*e.g.*, a thread pool). Via modeling task performance by not only its execution time, but the time when it waits for execution (*i.e.*, the time between when it is scheduled and when it starts execution), we can model how a task is influenced by the others. Thus, performance issues due to asynchronous executions can be properly captured.

Hence, this paper proposes `DiagDroid` (Performance **Di**agnosis for An**droid**), a novel tool to exercise, profile, and analyze the UI performance of an Android app without modifying its codes. First, via a light-weight static analysis of the target app, `DiagDroid` obtains the necessary information for profiling. Then it employs a plugin testing approach (*e.g.*, Monkey [67]) to exercise the original app. The required runtime data are then captured during the testing run via its profiler. The data are then processed offline to generate a human readable report. The report can unveil potential performance bugs to developers and direct them to suspicious locations in the source codes. Human efforts can greatly be reduced in diagnosing UI performance issues. Finally, `DiagDroid` solves the compatibility and efficiency challenges generally faced by the dynamic analysis tools by slightly instrumenting only the general Android framework invocations with a dynamic instrumentation approach. Hence, it can be applied to most off-the-shelf smartphone models and apps.

We have implemented and released `DiagDroid` [19]. We show it is easy to apply it to real-world apps with light configurations. In the 33 open-source real-world apps we study, 27 performance defects in 14 apps are found, and we receive positive feedbacks from their developers. These defects are caused by the complicated dependency of asynchronous executions, which can hardly be located with current diagnosis practice. This shows the effectiveness of `DiagDroid`.

2. ANDROID APPLICATION SPECIFICS

2.1 UI Event Processing

Designed mainly for user-centric usage patterns, Android apps are typically UI oriented: An app will iteratively process user inputs, and accordingly update the display to show the intended contents. The `main` thread of an app is the sole thread that handles the UI-related operations [53], 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 can invoke its corresponding *UI event procedure*, *i.e.*, the codes that handle the UI event.

Some UI event procedures may be time-consuming, *e.g.*,

```
//Create a new thread and download in that thread
Thread thread = new Thread(new DownloadRunnable(url));                          Thread
thread.start();

//Download in one thread of a thread pool with capacity 10
ExecutorService threadPool = Executors.newFixedThreadPool(10);                  ThreadPool
threadPool.execute(new DownloadRunnable(url));                                  Executor

//Download with AysncTask
AsyncTask asyncTask = new DownloadTask(url);                                    AsyncTask
asyncTask.execute();

//Download in a HandlerThread by posting a task on the attached handler
HandlerThread handlerThread = new HandlerThread("DownloadHanderThread");       Handler
handlerThread.start();

//Download in a user-defined Service
Intent downloadIntent = new Intent(this, DownloadService.class);               Intent
downloadIntent.putExtra(DownloadService.URLKEY, url);                           Service

//Use standard DownloadManager Service, utilizing ThreadPoolExecutor implictly
DownloadManager dm = (DownloadManager) getSystemService (DOWNLOAD_SERVICE);     Download
DownloadManager.Request req = new DownloadManager.Request(Uri.parse(url));      Manager
dm.enqueue(req);
```

Figure 2: Asynchronous execution examples

one to download a file from the Internet. To avoid blocking the `main` thread, UI event procedures conduct heavy-weighted work in an asynchronous manner so that the `main` thread can handle other UI inputs simultaneously [53]. After such *asynchronous executions* are done, the UI can be updated in the `main` thread with a call-back mechanism.

Figure 1 shows the codes of an `Activity` (*i.e.*, a window container for the UI components to be displayed). It retrieves data from the Internet and displays the data in a `TextView` (*i.e.*, a UI component to display text) after a button is touched. The Internet access is done asynchronously in another thread while the `TextView` update is done in the `main` thread. More specifically, the `RetrieveDataTask` extends the `AsyncTask` class. It overrides the `doInBackground` method to allow accessing the Internet asynchronously in a worker thread. Its `onPostExecute` method is a call-back mechanism to allow the corresponding `TextView` object update in the `main` thread. These codes are abstracted from RestC [66] project which shows a common coding practice.

2.2 Asynchronous Executions

Android provides high flexibility to implement asynchronous executions. There are tremendous ways for an app to start asynchronous executions (*e.g.*, using `AsyncTask`, `ThreadPoolExecutor`, and `IntentService`). Actually we find hundreds of classes or methods in the Android framework that can start asynchronous executions.The implicit ways to start asynchronous executions include, for example, those via the customized classes that override the Android framework classes such as `AsyncTask` or `HandlerThread`.

Figure 2 shows various ways of conducting asynchronous executions. For a simple task to download Internet contents, we could name at least 6 ways (including examples shown in Figure 1 and Figure 2). Choosing which way generally depends on the developer's own preference.

No matter how an asynchronous execution starts, it is executed by the operating system (OS) via the thread mechanism so as to implement concurrency. However, Android may start a new thread or reuse a running thread for the asynchronous execution. As a result, different asynchronous executions may share the same thread and run sequentially. In other words, they may compete for the same execution unit. Unfortunately, it is hard for the developer to be aware of such dependency of asynchronous executions: She may not know exactly how the asynchronous executions run.

The complex ways of starting asynchronous executions, together with their complicated runtime dependency, make it difficult for developers to comprehend the performance of the UI event procedures they write. Performance issues are hence hard to be eliminated without a proper tool. Next, we will show a representative performance issue.

3. A MOTIVATING EXAMPLE

Synchronous part of a UI event procedure may cause laggy UI, if it contains time-consuming codes [36, 63]. However, addressing performance problems solely in the synchronous part is far from enough. When a user is suffering from laggy UI, she is actually experiencing a long period of time between her UI operation and its corresponding display update. Even if asynchronous executions are introduced to make the synchronous part completing quickly, she still experiences laggy if the asynchronous executions are slow, and consequently cause the slow intended display update.

As Android allows complex ways to start asynchronous executions, it may introduce various tricky performance issues. The issues may lie in simple, widely adopted and seemingly-correct codes. Next, we show a subtle performance issue caused by unexpected sequentialized `AsyncTask`s.

Suppose an event procedure will show Internet contents in three `TextView`s in an `Activity`. As the contents are independent, developers expect to download them in parallel. They may instantiate three `RetrieveDataTask` objects (defined in Figure 1) and invoke their `execute` methods which is given as a usage example in the official guide [7], as follows.

```
(new RetrieveInfoTask()).execute(url1);
(new RetrieveInfoTask()).execute(url2);
(new RetrieveInfoTask()).execute(url3);
```

They may believe every `RetrieveDataTask`s will be executed in separated threads, and hence calling their `execute` methods makes them run in parallel. However, the codes contain a subtle potential performance issue. In the recent versions of Android, the `execute` method defined in the super `AsyncTask` class will insert the corresponding tasks into a global thread pool with capacity one. Thus all `RetrieveDataTask`s will be executed in sequence in one thread instead of in parallel in multiple threads. This incurs more time to complete the download tasks and to accordingly update the UI. As a result, the user may experience a laggy UI.

Such a code defect can be resolved by calling `executeOnExecutor` to customize a larger thread pool instead. It is also worth noting that such sequential execution mechanism is introduced in Android 3.0 or above. In the earlier versions, the codes will run in parallel as expected. It is easy for developers to neglect such changes and introduce potential performance issues. Such defects are common. Our experimental study finds eight such cases in real-world apps.

The above performance issue actually can hardly be tackled by current tools. StrictMode [63] and Asynchronizer [36] consider only the synchronous part of a UI event procedure, which cannot locate the issues caused by the asynchronous executions. Other tools like Panappticon [72], Method Tracing [54] can track such executions. But they largely do not focus on the runtime dependency of asynchronous executions. It is hard to find out such dependency via examining the tremendous traces produced by these tools. Hence, they are still not enough to cope with UI performance issues. Fixing this gap is one major aim of `DiagDroid`.

4. UI PERFORMANCE DIAGNOSIS

We notice the key to pinpoint the above performance issue is to know not only the execution time of an `AsyncTask`, but the time between when it is scheduled and when it starts to execute (*i.e.*, the queuing time), as well as the other `AsyncTask`s that are in the same thread pool. Specifically, an unexpected long queuing time of an `AsyncTask` indicates a performance issue. We can know there are too many `AsyncTask`s in the pool. By examining which `AsyncTask`s are in the pool and the pool capacity, we can locate the cause.

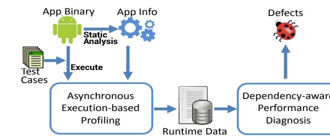

Figure 3: `DiagDroid` Overview

The above notion can also be applied to other mechanisms that start asynchronous executions. This is the basis of the `DiagDroid` design, which we overview in Figure 3. `DiagDroid` anatomizes the Android UI event procedures into a set of tasks and then quantize them so that data analysis can be conducted towards automating performance diagnosis.

Specifically, as shown in Figure 3, `DiagDroid` first performs a light static analysis of the target app and obtains some required information to assist runtime profiling. It then exercises the original app via a plugin testing approach which can involve random test cases (*e.g.*, Monkey [67]) or user-defined ones (*e.g.*, Robotium [59], UIAutomator [65]). During the testing run, the profiler can track asynchronous executions, so as to anatomize the UI event procedures into a set of tasks. The performance of the tasks, together with their runtime dependency, can then be captured. Based on the profiling data, `DiagDroid` detects performance issues and analyzes their causes. A report can finally be generated with an aim to direct the debugging process.

To this end, we need to address several critical considerations. Next, we will discuss the profiling granularity of `DiagDroid`, and the required runtime data for modeling asynchronous executions and their runtime dependency (in Section 4.1). Then, we illustrate how such data can facilitate UI performance diagnosis (in Section 4.2).

4.1 Modeling Tasks and Their Dependencies

As shown in Section 3, the subtle runtime dependency of asynchronous tasks can result in tricky performance issues. Analyzing such dependency is a key concern to `DiagDroid`. We analyze the app runtime in *task* level, defined as follows.

Definition 1. An *asynchronous task* (or task) is a segment of codes that run in sequence in a single thread. It defines a developer-intended asynchronous execution process.

`DiagDroid` profiles the app runtime in task granularity. The reasons are as follows. First, it is good enough for performance diagnosis to profile in such a granularity. A task is a short segment of codes that can also be well understood by its developer. If the developer can know which task is anomalous, she can instantly reason its cause by inspecting the task-related codes. Second, such a granularity will not incur too much profiling overhead, compared with the finer granularity (*e.g.*, in method level or in line level). Most importantly, profiling app runtime in task granularity can well capture the runtime dependency of two asynchronous tasks. As a result, UI performance issues caused by such dependency can be easily detected and located.

The task performance naturally reflects the performance of the entire UI event procedure: A slow task may result in a slow UI event procedure, leading to a laggy UI. Next, we discuss modeling task performance. We are aware that a task is possibly queued in an execution unit before it is executed. As a result, the execution time as well as the queuing time

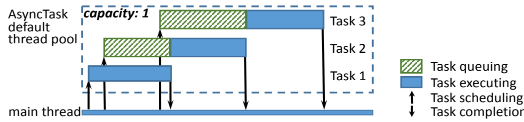

Figure 4: Asynchronous tasks runtime

in the unit should be considered in the performance model.

Definition 2. The *queuing time* of a task is the interval between when the task is scheduled (*e.g.*, when `execute` is called to start an `AsyncTask`) and when it starts to execute.

We propose using both the queuing time and the execution time to model the task performance. Note that this is generally different from the current diagnosis practice with tools like TraceView and dumpsys [54]. It is hard, if not infeasible, for these tools to obtain such data as they focus only on the execution time of individual methods.

The queuing time of a task is influenced by the other tasks that may compete for the same execution unit. We formally define task runtime dependency as follows.

Definition 3. Two tasks bear *execution dependency* if 1) they run in the same execution unit; and 2) one task is scheduled in during the other task's queuing time.

As shown in Section 3, task runtime dependency is a critical factor that influences the UI performance. We propose employing three queue-related features, the *queuing time*, the *pool capacity*, and the *queuing length* of a task, to model the task execution dependency.

Definition 4. The *pool capacity* of an execution unit is the maximum tasks that the unit can simultaneously execute.

For example, for a thread the pool capacity is 1; while for a thread pool the pool capacity is its size. Pool capacity is usually set once and remains unchanged during runtime. Suppose k tasks bear runtime dependency in an execution unit with capacity N and $k>N$. k-N tasks have to wait for execution in the unit. Then, when one task completes its execution, one of the waiting tasks can be executed.

Definition 5. The *queuing length* of a task is the total number of tasks waiting for execution in the execution unit after it is scheduled.

A queuing length L indicates that the task should wait for the completion of other L tasks before it can be executed.

Figure 4 illustrates how the three tasks discussed in Section 3 are executed. The thread pool capacity of `AsyncTask` is 1. When Task 2 is scheduled, it has to wait until Task 1 finishes its execution. So the queuing length of Task 2 is 1. Similarly, the queuing length of Task 3 is 2.

The *queuing time* of a task reflects how other tasks influence its performance. The *queuing length* and *pool capacity* indicate the cause of a bad-performance task. Such information can greatly help performance diagnosis. However, existing tools for performance diagnosis (*e.g.*, Panappticon [72], Method Tracing [54]) cannot provide such information. As a result, it is difficult for them to diagnose the subtle performance issues caused by execution dependency.

We will elaborate how `DiagDroid` collects these runtime data in Section 5. Next, we will first discuss how `DiagDroid` conducts performance diagnosis with the collected data.

4.2 Dependency-Aware Diagnosis

A laggy UI indicates that a UI event procedure requires longer time to complete. As discussed, this can be rooted in either the synchronous or the asynchronous executions. Although many tools (*e.g.*, [36, 63]) have addressed the former case, `DiagDroid` moves a step further by focusing on the latter case, a far more difficult task in addressing the subtle performance issues caused by the asynchronous executions.

If the asynchronous part is laggy, it means at least one of the asynchronous tasks requires more time to complete. Consequently, `DiagDroid` should detect performance anomaly by checking whether there are any anomalous asynchronous tasks. Human inspection of all the involved tasks is prohibitively labor-intensive. `DiagDroid` requires an automatic way to detect anomalous tasks. A possible approach is to group the tasks in such a way that we can assume the tasks in the same group have similar performance. Then we can perform anomaly detection in a group basis.

But easy as it looks, how to group the tasks is challenging. An instant way is to consider the *method call-stack* when a task is scheduled. We name such call-stack the *execution context* of the task. The execution context actually links to the source codes that define the task and how the codes are reached. Two tasks with the same execution context mean that they are corresponding to the same specific source code segment and execution sequence. Hence, they should naturally be grouped together.

But this simple consideration will result in extensive debugging efforts. A code defect may manifest in similar tasks with slightly-different execution contexts. Reporting all such tasks one by one based on their execution contexts is very tedious, and even makes the diagnosis difficult with such tedious information. For example, two UI event procedures of two buttons may invoke the same buggy asynchronous task (in the source codes). In these two cases, the two task invocations have different execution contexts since they are invoked by different event procedures. But we should group them together to reduce human efforts for code inspection.

`DiagDroid` addresses the challenge by putting *similar tasks* into a group with properly defined task *similarity*. By considering each method call as a symbol, an execution context can be encoded into a vector. Then the difference of two tasks is the *edit distance* of their execution contexts. We adopt such an edit distance as a similarity measure due to following considerations. First, it is suitable to model the differences of two tasks. Consider the above example, if two UI event procedures of two buttons invoke the same buggy asynchronous task, the edit distance of execution contexts of the two task invocations will be close. As a result, they can be grouped together. Second, it considers the invocation order information, where such order is important to describe the app runtime. Consequently, two close execution contexts indicate that the corresponding tasks are similar during runtime. With such a similarity measure, `DiagDroid` conducts the single-linkage clustering, a widely-adopted sequence clustering method, to form groups [24].

`DiagDroid` examines whether performance anomaly manifests in each group of tasks with the execution context, queuing time and execution time. `DiagDroid` considers both the maximum value of queuing time and the execution time of all tasks within each group A. The values are denoted by $M_q(A) = max(Q(a))$ and $M_e(A) = max(E(a))$ $(\forall a \in A)$, $Q(a)$ and $E(a)$ are queuing time and execution time for task a respectively. $M_q(A)$ and $M_e(A)$ are the performance metrics of group A, since either a long execution time or a long queuing time can result in anomalous performance.

`DiagDroid` considers a group is anomalous if one of its two performance metrics is larger than a threshold τ. It ranks

Table 1: Categories of asynchronous tasks

Category	Type	Representative classes
Reusing existing threads	Looper & Handler	HandlerThread
		IntentService
	Pool-based executor	ThreadPoolExecutor
		AsyncTask
Creating new threads	Thread	Thread

the anomalous groups according to their performance metrics as well as their execution contexts. Since each group is corresponding to a specific source code segment, the rank can direct the manual debugging efforts towards a suspicious code segment that may cause the performance anomaly. Moreover, a key consideration of `DiagDroid` is that the runtime dependency of tasks may also cause performance issues. In other words, the anomalous task *per se* is not always the root cause of its poor performance. Especially, when the queuing time of the task is too long, it is usually caused by other tasks that bear runtime dependency. Therefore, `DiagDroid` also employs the performance data (the queuing length and performance) of such tasks to locate the root cause. We will show in our experimental study that the localization approach can greatly save human efforts.

We consider maximum values instead of average values as performance metrics. A large average means that many tasks in the group perform poorly, hence is a good indicator of performance issues. However, a small average for cases that only a small portion of the tasks perform poorly, can still be an important symptom of performance issues [20]. Since for the above two situations the maximum remains large, we consider the maximum as performance metrics.

We have considered a performance threshold τ as an indicator of poor-performance tasks. τ is selected empirically based on the developers' consideration on laggy UI. Previous work (*e.g.*, [47, 50, 60]) has suggested user-tolerable waiting time in web browsing, mobile web browsing and mobile services, which ranges from two to several seconds. One second is considered as the limit for the user's flow of thought to stay uninterrupted [49]. We regard that mobile app users are more sensitive to UI response time. Thus, we use $500ms$ as the value of τ. We will show in our experimental study in Section 6 that such a value is an effective choice.

5. PROFILING ASYNCHRONOUS TASKS

`DiagDroid` requires to profile the queuing time, the execution time, and the queuing length of a task, as well as the pool capacity of an execution unit. Hence, `DiagDroid` must firstly track the life-cycle of a task, *i.e.*, when it is scheduled, when it is executed, and when it completes.

However, there is no sole entry/exit points for hundreds of ways to implicitly/explicitly start asynchronous tasks (mentioned in Section 2). It is difficult, if not infeasible, to design specific profiling mechanism for each. We attack this challenge with a separation-of-concerns approach. We first classify the tremendous ways to start tasks into five categories (Section 5.1). Then we can specifically track and profile the necessary runtime data for each category (Section 5.2).

5.1 Categorizing Asynchronous Tasks

We notice that the underlying mechanisms for Android to execute a task can be narrowed down into two approaches: 1) reusing existing threads created beforehand, and 2) creating a new thread. The former case can be further divided into two types: One directly schedules a task (*Pool-based Executor* mechanism) and the other requests the scheduling of a task by a delegate via sending a message (*Looper & Handler* mechanism). We list them in Table 1, together with their representative classes in Android.

Both `HandlerThread` and `IntentService` depend on the Looper & Handler mechanism to start tasks. They create a *worker thread* and wait for new tasks to the *looper* associated with the thread. The request of scheduling a task is sent via a *handler* attached to the looper. The requested task will then wait to be processed in the worker thread. Since there is only one worker thread, it can process one message at a time. Other requests should wait in a queue.

`ThreadPoolExecutor` and `AsyncTask` both use the Pool-based Executor mechanism. They maintain a pool of worker threads with its size not exceeding a preset capacity. A new coming task will be executed in one thread in the pool if there are available threads (*i.e.*, the number of busy threads is smaller than the capacity). Otherwise, the task has to wait for an available thread.

`Thread` mechanism starts a task immediately in a new thread. Its building basis, *i.e.*, the `Thread` class in Android, is the same as the traditional Java one.

The underlying mechanisms of numerous ways to implicitly or explicitly start asynchronous tasks are based on these five representative classes. For example, the `AsyncQueryHandler` class, which conveniently queries data from a content provider, is based on `HandlerThread`. The `CursorLoader` class, which acquires data from the database, is based on `AsyncTask`. Moreover, the `DownloadManager` mentioned in Section 2 employs the `ThreadPoolExecutor`.

Each of the five classes for conducting asynchronous tasks has its pros and cons. The `Thread` class is flexible which enables fully control on the threading mechanism, while more management efforts are required. Moreover, creating a new thread per task consumes system resources. The `HandlerThread` class requires many development efforts to customize both the background threads and the handler for the tasks. The `ThreadPoolExecutor` class, as a traditional JAVA class for multi-threading, is widely used to manage a pool of worker threads. However, comparing to the `AsyncTask` class, it is unsuitable for tasks updating UI since Android prohibits UI updating in worker threads. The `IntentService` class can start a background service independent of the activity life cycle. It is a relatively heavier container which requires more system resources on execution. Choosing which way depends on the specific programming requirements, as well as the developer's preference.

5.2 Profiling Asynchronous Tasks

`DiagDroid` tracks tasks with a dynamic instrumentation mechanism on Android framework methods. It requires no changes to the target app, or recompiling the underlying OS and the Android framework. This can guarantee the compatibility of `DiagDroid` with diverse Android versions and smartphone models. Moreover, it requires little human efforts in installing and applying the tool.

Specifically, unlike general Linux processes, Android processes of its apps are all created by duplicating a system process called `Zygote` [18]. Android framework functionalities have already been loaded in `Zygote` before such duplication. Therefore, we can instrument the `Zygote` process and "hijack" the Android framework methods of interest be-

forehand. Then when an app runs by forking `Zygote`, the method invocations are inherently hijacked by `DiagDroid`. Hence, we can easily track methods. We adopt such a mechanism implemented in the tool named Xposed [70], usually used for modifying UI [69]. We program our own codes to hijack the methods of our interest. Next we introduce how `DiagDroid` tracks tasks in each category.

1) `Thread`: An asynchronous task that implements as a thread always starts with the `start` method. Hence, we can instantly obtain the time when it is scheduled by tracking the `start` method. However, the task is executed in the overridden `run` method of a `Runnable` object, which is an abstract method that cannot be instrumented directly. Hence, we resort to static analysis to find the implementations of the abstract `run` method and instrument them instead.

The static analysis is performed via the tool `apktool` [6]. It decompiles the binary into well-structured Dalvik bytecode. They can be parsed to obtain the implementations of the abstract `run` method, which can direct our dynamic instrumentation approach to obtain the execution time. Note that `DiagDroid` only decompiles and discovers these methods, instead of modifying and recompiling the app.

2) `HandlerThread`: `HandlerThread` is a thread that provides a `Looper` object attached to it. A `Handler` is associated with the `Looper` object and handles messages for the `Looper`. Hence, we can obtain the request time of a task by tracking the time when a `Message` object is sent to the `Handler`. Since eventually `sendMessageAtTime` or `sendMessageAtFrontOfQueue` must be invoked to send a `Message`, we record the invocation time of these two framework methods as the time when a task is scheduled. `Handler` performs the task by processing its corresponding `Message`, we track task execution by instrumenting its `dispatchMessage` method.

3) `IntentService`: An `IntentService` task always starts by invoking the `startService` method of the framework class `ContextImpl`. Hence, the invocation time of this method is the task scheduling time. `IntentService` actually relies on the `ServiceHandler` class, which inherits from `Hanlder`, to process the task. Hence, we track task execution by tracking the `dispatchMessage` method of `Handler`.

4) `ThreadPoolExecutor`: `ThreadPoolExecutor` is a pool-based execution mechanism which has an elegant pattern. A task is always requested via invoking the `execute` method. Moreover, the task always starts immediately after the `beforeExecute` method, and is followed by the `afterExecute` method. Hence, we track tasks via these methods.

5) `AsyncTask`: A task can base its implementation on the complicated Java class inheritance of the basis `AsyncTask` class. However, it is always scheduled by the `execute` or `executeOnExecutor` methods eventually, regardless of class inheritance layers. Hence, the task scheduling time is the invocation time of the two methods. For both cases, `AsyncTask` actually relies on the `ThreadPoolExecutor`. Hence, we track task execution similar to that of `ThreadPoolExecutor`.

For the tasks in categories 2-5, they are put in a queue before executed. To model their task runtime dependency, we use the hash code of the execution unit (*e.g.*, `ThreadPoolExecutor` object) as the *queue identifier*. Such a hash code is easy to obtain during runtime according to Java specifics. Two tasks with the same queue identifier may bear runtime dependency. Finally, the pool capacity is obtained via checking some internal fields of the queue object (*e.g.*, the maximumPoolSize field of a `ThreadPoolExecutor` object).

Table 2: Representative performance issues found (Rank: ranking of the buggy issue / total # issues reported)

Category	Issue description	Class@App	Rank
Sequential execution	Not awaring AsyncTask.execute() method results in undesired seqential execution	LawListFragment@OpenLaw	1/7
	Loading tens of icons in sequence	AppListArrayAdapter@AFWall+	1/3
Forgetting canceling execution	Improper cancelation of asynchronous tasks	GetRouteFareTask@BartRunnerAndroid	1/4
	Not canceling obsolete queries when new query arrives	AsyncQueryTripsTask@Liberario	2/2
Improper thread pool	Failed to set optimal size of the thread pool	ZLAndroidImageLoader@FBReader	1/2
	Use the same pool for loading app list and app icons	MainActivty@AFWall+	2/3
Overloading message queue	Posting various types of tasks (e.g., update progress, store book) to the same backgroundHandler	ReadingFragment@PageTurner	3/9
	Executing Filter method of AutoComplete-TextView occupies the Handler of a public message queue	LocationAdapter@Liberario	1/2
Misusing third-party library	Not canceling the tasks implemented by third-party library, Android asynchronous http client - loopj	HeadlineComposerAdapter@OpenLaw	7/7
	Use the deprecated findall method of WebView class which causes blocking	MainActivity@Lucid Browser	1/5

6. EXPERIMENTAL STUDY

We have implemented `DiagDroid` and released the project online [19]. In our experimental study, we target on open-source apps since we need the source codes to verify the effectiveness. To this end, we download such apps from F-Droid, an app market that hosts only free and open-source Android apps [21]. It is also a popular app source for the research community [42]. We employ Monkey [67], the official random testing tool, to exercise our target apps. It is also known as the most efficient tool in terms of code coverage [17]. Among the apps we download, we exclude those that require a login account for convenience consideration (so that we do not bother to register new accounts). Note that `DiagDroid` can easily handle such apps by applying a login script, which is trivial and will not influence the effectiveness. We thus get 33 target apps covering diverse categories including *Reading*, *Multimedia*, *Science* & *Education*, *Navigation*, *Security* and *Internet*.

We verify the compatibility of `DiagDroid` on four smartphone models covering a wide range of device capacities: Samsung GT-I9105P (Android 4.1.2), Huawei G610-T11 (Android 4.2.2), Huawei U9508 (Android 4.2.2), and Lenovo K50-T5 (Android 5.1). Experiments are conducted on the four devices simultaneously to save time. We also conduct stress tests by injecting loads on CPU, memory, sdcard IO and network respectively with customized Android background services. We implement five background services occupying 80% CPU, five background services occupying 80% memory, five background services consuming 2 Internet downloading threads each, two background services each reading 1 file and writing 1 file on sdcard in separate threads. The parameters are chosen by common practice. Developers could configure with their own preferences. We design a system app to guarantee that these services would run persistantly (i.e., they will not be terminated by LowMemoryKiller [40]). Four devices with five configurations each (four with load injections and one without load injection) come up to 20 test configurations, each configuration is under Monkey testing for 30 minutes. We run 19,800-minute test in total for the 33 apps. `DiagDroid` reports overall 48 performance issues marked as highly suspicious for 14 apps, on average 3.4 issues per app. The reports are published online. Via inspecting the related source codes in the reports

```
1.  asynchronous tasks with context c1
2.      max queuing time: 1650ms
3.      pool capacity: 1
4.      cases with queuing time ≥ 500ms:
5.          avg. queue length: 1.00
6.          avg. execution time of the in-queue tasks: 1666.00ms
7.      runtime dependency: c2
Context c1
Class name:   de.jdsoft.law.data. UpdateLawList
Call-stack:   android.os.AsyncTask.executeOnExecutor (Native Method)
              android.os.AsyncTask.execute (AsyncTask.java:535)
              de.jdsoft.law.LawListFragment.onCreate (LawListFragment.java:91)
              •••
Context c2
Class name:   de.jdsoft.law.data.LawSectionList
Call-stack:   android.os.AsyncTask.executeOnExecutor (Native Method)
              android.os.AsyncTask.execute (AsyncTask.java:535)
              de.jdsoft.law.LawListFragment.onCreate (LawListFragment.java:87)
              •••
```
Report

```
public void onCreate(Bundle savedInstanceState) {
    •••
    // Load actual list
    final LawSectionList sectionDB = new LawSectionList(LawSectionList.TYPE_ALL);
    •••
    sectionDB.execute(adapter);
    // And parallel update the list from network
    UpdateLawList updater = new UpdateLawList();
    updater.execute(adapter);
    •••
}
```
Codes

Figure 5: Report and code segments of case 1

for several minutes per case and understanding the original project, we surprisingly find 14 of the target apps contain 27 performance issues. The bug cases are ranked highly in the report, (with an average rank of 1.7). Although unfamiliar with the target app design, we find it very convenient for us to pinpoint the root causes of the issues.

We categorize 27 detected issues into 5 categories. Ten representative issues are shown in Table 2, with their causes, defect locations and rankings in the reports. We have reported the issues to app developers, many of which have been confirmed and corrected accordingly. We have got positive feedbacks like "for faster search results", "I've modified and I see the performance improvements." after developers fix the issues. Next, we elaborate our experiences of performance diagnosis via five representative cases.

6.1 Case Studies

Case 1: Unwanted Sequential Executions

We provide our experiences on diagnosing `OpenLaw` [52], which provides access to over 6,000 laws and regulations. `DiagDroid` reports 7 highly suspicious issues. After half an hour inspecting of related source codes according to the report, we summarize 2 performance issues and localize the causes. One case is that the queuing time of the task group with context c_1 is longer than the threshold $500ms$. This case is found in 14 test configurations, mostly under those with heavy CPU or sdcard IO load, which indicates the case may be related to some IO intensive operations. We demonstrate the content of `DiagDroid` report in Figure 5 (Line numbers are added for discussion convenience).

We instantly find that such a long queuing time is because the task should wait in queue till the completion of other long executing tasks based on Lines 4-7. We know on average *one* task (Line 5) bears runtime execution dependency with an anomalous task, of which the context is c_2 (c_2 task for short) (Line 7). In other words, anomalous c_1 tasks are due

```
1.  asynchronous tasks with context c1:
2.      max queuing time: 31885ms
3.      pool capacity: 1
4.      cases of queuing time ≥ 500ms:
5.          avg. queue length: 19.50
6.          avg. execution time of the in-queue tasks: 1240.60ms
7.      runtime dependency: c1, c2
8.    execution time of c1    max: 19337.00ms    avg. 1419.95ms
9.    execution time of c2    max: 3446.00ms     avg. 786.69ms
```
Report

```
protected String doInBackground(Params... paramsArray) {
    Params params = paramsArray[0];
    if (!isCancelled()) {
        return getFareFromNetwork(params, 0);
    } else {
        return null;
    }
}
```
Codes

Figure 6: Report and code segments of case 2

to the heavy-weighted c_2 tasks. The pool capacity is only 1 (Line 3), which indicates the tasks have to run in sequence. Waiting for the completion of another heavy-weighted task should generally be avoided via proper scheduling.

The c_1 and c_2 contexts are included in the report (Figure 5). We can conveniently find the related source codes and how they are scheduled. The comment "parallel update" indicates developers intend to execute the two tasks in parallel. But we notice this is the same mistake in Section 3. The fix is to call `executeOnExecutor` with a larger pool instead.

Usually, developers wrongly assume the availability of the execution unit during task scheduling. Even worse, such *sequential* tasks may be defined and scheduled across several source files, making it harder to capture their dependency manually. In the example, `OpenLaw` handles tens of UI events. It is hence difficult to manually test and detect performance issues in all UI event procedures. Even if developers notice the laggy UI procedure (*e.g.*, loading `LawListActivity`), they can hardly pinpoint the defect by inspecting nearly 300 hundred lines of codes in several files, which even involves complicated third-party library invocations.

Existing tools [54, 72] focus on the execution time of methods. Generally, they lack the capability to model the queuing time of tasks and to identify the execution dependency. Therefore developers can hardly detect the subtle symptoms and reason the defect caused by task dependency. Note that the method tracing-based tools will generate a trace of thousands of methods for a UI event procedure. The performance diagnosis based on such data is like finding a needle in the hay stack, given the fact that Method Tracing will produce about 36GB of data for our 600-minute testing run per app.

In contrast, `DiagDroid` properly models the task execution dependency, and provides tidy but helpful information to guide the diagnosis process. We show that it can greatly reduce the human efforts by directing the developer to several lines of codes that cause the performance issue.

Case 2: Not Canceling Obsolete Tasks

The cancelation of a time-consuming task is necessary when the task is no longer required. For example, when a user performs activity-switching by a sliding operation, the previous content downloading task becomes obsolete since its associated activity is invisible. Obsolete tasks occupy resources (*e.g.*, Internet bandwidth), and therefore deteriorate the UI performance. Sometimes, they even block other tasks by occupying the thread pool. However, canceling obsolete tasks is not obligatory and hence often neglected by develop-

```
1.    asynchronous tasks with context c1:
2.          max queuing time: 514ms
3.          pool capacity: 3
4.          cases of queuing time ≥ 500ms:
5.                avg. queue length: 1.50
6.                avg. execution time of the in-queue tasks: 659.29ms
7.          runtime dependency: c1
```
Report

```
private static final int IMAGE_LOADING_THREADS_NUMBER = 3;//TODO: how many threads?
private final ExecutorService myPool = Executors.newFixedThreadPool(
      IMAGE_LOADING_THREADS_NUMBER, new MinPriorityThreadFactory());
...
void startImageLoading(...){
      final ExecutorService pool =
            image.sourceType() == ZLImageProxy.SourceType.FILE ? mySinglePool : myPool;
      pool.execute(...);
}
```
Codes

Figure 7: Report and code segments of case 3

ers. we reveal that many popular apps contain performance issues caused by not canceling obsolete tasks.

`DiagDroid` finds 5 performance issues for `BartRunnerAndroid`, a public transport app. We can conveniently locate 5 bugs in the source codes with the report. Specifically, we detect the anomalous c_1 tasks under all 20 test configurations. The corresponding content of the report is demonstrated in Figure 6. Similar to Case 1, we can instantly find the queuing effect caused by execution dependency (Lines 4 to 7).

We can simply apply a fix similar to Case 1 here, *i.e.*, by allowing tasks to run in separate execution units. However, we notice both c_1 and c_2 tasks can be executed for a long period (Lines 8-9), and they often block each other. By inspecting the codes, we find developers have already intended to cancel the obsolete tasks. c_1 tasks inherit from `GetRouteFareTask`, whose source codes for executing are shown in Figure 6. The cancelation checking is done before the task begins, thus the task will not be canceled during execution when it is obsolete. Similar analysis could be applied to c_2. In other words, the developers fail to conduct proper cancelation steps via cancelation checking.

The correct cancelation involves 2 steps: 1) invoke `cancel` method in `onStop` method of the container `Activity`, and 2) periodically check `isCancelled` method in `doInBackground` method, release the resource when `true`. Note that releasing the resource in `onCancel` method is also a common mistake.

Case 3: Improper Thread Pool Size

Improper thread pool size is a common cause of long task queuing since the pool is often busy. We show how to diagnose such defects on `FBReader` [22], a popular e-book reader.

`DiagDroid` reports only one issue for this app (Figure 7). We detect this case under all 20 test configurations. We pinpoint the issue in source codes in 20 minutes. Both the execution and queuing time of c_1 tasks are anomalous. We can quickly confirm that the image loading tasks reasonably execute for long. However, it is undesired that a c_1 task has to wait long for other c_1 tasks to complete (Lines 4-7). Unlike Case 2, c_1 tasks cannot be canceled since images are loading simultaneously. Hence, a quick fix is to set a larger pool size (5 may be good according to the report).

Properly setting a pool size is often hard during the coding phase. We find four such cases in our experiment. Developers are hard to predict the possible number of concurrent tasks in the same pool. For example, in this case study, developers are not sure about the proper pool size, and hence put down a to-do comment in the source (Figure 7).

Finally, note that the existing approaches [54, 72] can not identify issues caused by defects in Cases 2 and 3, since these approaches focus only on the execution time. Even if developers notice the bug symptoms, existing approaches lack a way to automatically analyze the runtime dependency of tasks. As a result, it requires daunting manual efforts to find that a task *sometimes* has to wait for other tasks by the inspection of tremendous runtime traces.

```
1.    asynchronous tasks with context c1:
2.          max queuing time: 6138ms
3.          pool capacity: 1
4.          cases of queuing time ≥ 500ms:
5.                avg. queue length: 1.38
6.                avg. execution time of the in-queue tasks: 3274.31ms
7.          runtime dependency: c1
```
Report

Figure 8: Report of case 4

```
1.    asynchronous tasks with context c1:
2.          max queuing time: 664ms
3.          pool capacity: 1
4.          cases of queuing time ≥ 500ms:
5.                avg. queue length: 3.00
6.                avg. execution time of the in-queue tasks: 323.00ms
7.          runtime dependency: c1
```
Report

Figure 9: Report of case 5

Case 4: Overloading Message Queue

Message handler is also a queue-based execution unit with only one thread. If messages come from separate UI operations (*i.e.*, continuous text inputs) too quickly, a message should wait for processing previous messages. This type of issues can also be easily pinpointed with `DiagDroid`. We find two such cases and take the `Transportr` (a public transport app) case as an example. `DiagDroid` reports two highly suspicious issues, from which we find two code defects in less than an hour. Developers fix the cases accordingly.

We can see from the report that c_1 tasks are with anomalous queuing time and execution time (Figure 8). This problem is detected under all 20 test configurations. A c_1 task has to wait long for other c_1 tasks to complete (Lines 4-7), while the long execution time indicates that a c_1 task may long occupy the handler. Via inspecting the source codes of c_1, we can easily find the reason. The c_1 task is a `Handler` for filtering input text, which invokes `performFiltering` directly. The method involves a time-consuming Internet query on requesting a list of suggested locations with an incomplete input. Consequently, the upcoming messages will have to queue in the message queue for previous inputs. Actually the old queries are no longer useful, and hence should be canceled when processing new messages.

Note we can know the long execution time with existing tool [54]. But we may falsely accept the time-consuming Internet query. Moreover, Panappticon [72], an event tracing tool to identify critical execution paths in user transactions, is unaware of the dependency between tasks invoked in different UI operations. In this case, tasks bearing execution dependency are invoked by independent text inputs in the `AutoCompleteTextView`. It is hard to know how a long execution time influences other tasks, without a tool like `DiagDroid` to model the dependency of the concurrent tasks.

Case 5: Misusing Third-Party Library

Misusing third-party libraries is also a source of performance issues. Without knowing the implementation details, developers misunderstand the usage of the third-party library and introduce performance issues. For example, unaware of the asynchronous tasks in a third-party library, the

	Context c_1	Context c_2	Context c_3
1.	de.jdsoft.law.data.UpdateLawList	de.jdsoft.law.data.UpdateLawList	de.jdsoft.law.data.UpdateLawList
2.	android.os.AsyncTask.executeOnExecutor(Native Method)	android.os.AsyncTask.executeOnExecutor(Native Method)	android.os.AsyncTask.executeOnExecutor(<Xposed>)
3.	android.os.AsyncTask.execute(AsyncTask.java:534)	android.os.AsyncTask.execute(AsyncTask.java:534)	android.os.AsyncTask.execute(AsyncTask.java:539)
4.	de.jdsoft.law.LawListFragment.onCreate(LawListFragment.java:91)	de.jdsoft.law.LawListFragment.onCreate(LawListFragment.java:91)	de.jdsoft.law.LawListFragment.onCreate(LawListFragment.java:91)
	…	…	…
9.	android.support.v4.app.FragmentActivity.onCreateView(FragmentActivity.java:285)	android.view.LayoutInflater.createViewFromTag(LayoutInflater.java:676)	android.view.LayoutInflater.createViewFromTag(LayoutInflater.java:727)
	…	…	…

Figure 10: Similar contexts without clustering

developer will neglect to cancel obsolete tasks. DiagDroid can also save the efforts in troubleshooting such defects. We detect three such cases and show the defect in Lucid Browser, a web browser app, as an example. We infer this issue from the 3 suspicious issues reported by DiagDroid .

We can know this is an unintended sequential execution case similar to Case 1 (Lines 3-7, Figure 9). Checking the source codes related to c_1, we can locate the invocation of the findAll method in the third-party library WebView. It finds the occurrences of a specific text in a webpage when the text is changed. Revisiting findAll, we can find that it is deprecated, and should be replaced by findAllAsync.

Note that the case is only detected on 3 devices other than Lenovo K50-T5 (Android 5.1). findAll does not introduce performance issues in Android versions above 4.4 because the Chromium-based WebView replace the Webkit-based WebView. Panappticon [72] requires to recompile the kernel, can work only on a small set of devices can not cope with such defects that do not persist in all Android versions [12].

6.2 Why Clustering

As mentioned in Section 4.2, to reduce the number of reporting suspicious cases, we cluster the execution contexts (*i.e.*, call-stacks) belonging to the same asynchronous task triggering by similar running sequences. First we extract two features via scanning through massive contexts: 1) similar call-stacks are similar in line level, and 2) similarity of call-stacks is transitive. Then we perform clustering accordingly. In this section, we show that the clustering is necessary and effective with a randomly selected example of app OpenLaw.

For OpenLaw, there are totally 1462 distinct contexts found under all 20 test configurations. As a result, 226 suspicious performance cases are reported. However, we find many of the reported cases are with similar contexts. We select three similar contexts as examples in Figure 10. Actually, the three contexts refer to the same asynchronous task UpdateLawList presented as context c_1 in Case study 1. They are the same until the 9th line of the call-stacks; more specifically, they have slight difference in low-level VM processing sequences. There is only one performance issue instead of three in developers' viewpoint. To lighten the workload of developers, the three contexts should be grouped into one.

Considering the aforementioned features, we cluster contexts with a customized edit distance (feature 1) plus single-linkage strategy (feature 2). After the clustering, we successfully reduce the amount of total contexts from 1462 to 75 (groups). Moreover, only 7 performance cases are reported without losing meaningful cases. This result indicates the effectiveness of our clustering mechanism.

6.3 Performance Enhancement

DiagDroid is able to present to developers with the performance enhancement after fixing performance issues. DiagDroid offers the distribution of the queuing & execution delays of *asynchronous executions*. Besides confirming the disappearing of the related case in the report of the fixed version, developers can ensure the performance gain via double-checking the delay distributions of related asynchronous executions before and after fixing the issue. Next, we illustrate how to visualize the performance enhancement via demonstrating two official fixes by developers. Notice the example distributions are simplified (yet good enough) to demonstrate the enhancement. Developers could tune the parameter to obtain finer distributions.

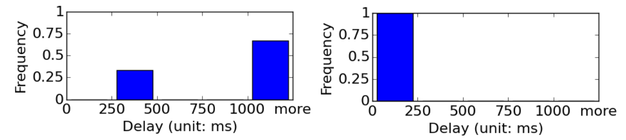

Figure 11: Message handler blocking delays before (left) and after (right) fix of Transportr

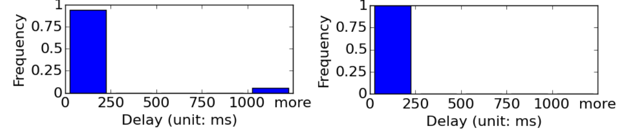

Figure 12: Queuing delay of showing apps before (left) and after (right) fix of AFWall+

Developers of Transportr fix an issue of message queue overloading with our report. They modify 11 files with 364 additions and 274 deletions. Since the message processing is network related, we show the performance enhancement on the Huawei G610-T11 with network load injected. The result is depicted in Figure 11. Similar patterns can be found in other test configurations. It could be seen that there is no more blocking problem for the new app version.

With our report, developers of AFWall+ fix the sequential loading issue on displaying the app list. They modify the source from (new GetAppList()).setContext(this).execute() to (new GetAppList()).setContext(this).executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR). We notice that the queuing effect is more obvious with CPU load injected on a poorer device. We illustrate in Figure 12 the distribution of queuing delay of the AsyncTask with testing configuration of Samsung GT-I9105P with CPU load injected. Similar patterns can be found in other test configurations. It could be seen that there is no more queuing problem in the new app version.

6.4 Discussions

Next, we discuss the threats to the validity of our experiments, and the measures we take to address them. First is the overhead of DiagDroid, *i.e.*, how it affects the test efficiency. We employ the time command to obtain the CPU time for conducting $10,000$ Monkey operations ($200ms$ interval between two consecutive operations) with DiagDroid on and off. The 0.8% overhead shows that DiagDroid does not have a considerable impact on testing efficiency.

Instead of relying solely on Monkey, test executor is a plug-in in DiagDroid. Incorporating script-based testing tools like UIAutomator [65] and Monkey runner [45] is allowed.

We test 30 minutes for each of 20 configurations per app to show the capability of DiagDroid in such a short-term test. The settings can be changed to explore the specific app more thoroughly. Note that DiagDroid also carefully addresses

its compatibility issues. Parallel testing in multiple device models is feasible as already shown.

Finally, `DiagDroid` resorts to dynamic analysis as we focus on the performance issues caused by complicated runtime dependency of asynchronous executions. It is hard for the static analysis approaches to deal with such issues. For example, determining the proper pool size based only on the source codes is hard, since it is impossible to predict the possible number of concurrent tasks. Moreover, it is hard to know when to cancel a task beforehand. The aim of `DiagDroid` is to save inevitable human efforts, rather than approaching the task of automatic code correction. As shown in our case studies, such efforts are light. `DiagDroid` is able to provide a small set of possible issues. The issues that really contain bugs are with high rankings. As shown in Table 2, the buggy cases rank 1.7 averagely. This indicates we can easily be directed to where the code defect lies. The debugging time of each case is generally less than an hour, even for us who are unfamiliar with the app implementations.

7. RELATED WORK

Performance diagnosis has long been studied in many systems. Much work is conducted on predicting performance of configurable systems [61, 73]. CARAMEL [51] detects unnecessary loop executions with static analysis. Yu *et al.* [71] propose a performance analysis approach based on real-world traces. But such traces are lacking in our problem. PPD [68] resorts to dynamic instrumentation for goal-oriented *known* performance issues search. Lag Hunting [31] aims at finding performance bugs in JAVA GUI applications in the wild without addressing concurrency issues. Existing work [2, 30] also considers "thread waiting time" (*i.e.*, the time when a thread waiting for other threads during its execution) as a metric to find performance bottlenecks. In contrast, we focus on queuing time, *i.e.*, the time when a task waiting before its execution, to model task dependency. Performance issues of Javascript programs are also studied [62]. SAHAND [1] visualizes a behavioral model of full-stack JavaScript apps' execution. But it does not take the contentions of asynchronous tasks into consideration.

Performance is critical to mobile apps [4, 58]. Blocking operations in the `main` thread are widely known as a cause of many performance issues [38]. StrictMode [63] aims at finding such operations. Asynchronizer [36] and AsyncDroid [35] provide a way to refactor specific blocking operations into standard `AsyncTask` and `IntentService` further to eliminate the memory leakage problems. CLAPP [23] finds potential performance optimizations via loop analysis. However, such static analysis-based tools cannot capture runtime execution dependency. Banerjee *et al.* [10, 11] design static analysis driven testing for performance issues caused by anomalous cache behaviors. Tango [26], Outatime [33] and Cedos [46] optimize WiFi offloading mechanism to keep low-latency of app. SmartIO [48] reduces the app delay via reordering IO operations. These approaches solve specific performance issues. `DiagDroid` in contrast aims at solving general UI performance issues caused by runtime task dependency.

UI performance diagnosis captures much research attention. Method tracing [54] is an official tool used to diagnose known performance issues due to its high overhead. QoE Doctor [15] bases its diagnosis on Android Activity Testing API [8] which can only handle pre-defined operations. Appinsight [57] is a tracing-based diagnosis tool for Windows phone apps. It traces all asynchronous executions from a UI event to its corresponding UI update, and identifies the critical paths that influence the performance. Panappticon [72] adopts a similar approach on Android. But these approaches generally neglect the runtime dependency between tasks, especially tasks executed for different UI operations. Moreover, it suffers from low compatibility since they largely require to recompile the Android framework and OS kernel.

Performance diagnosis often requires to exercise the target app automatically. Script-based testing is widely used (*e.g.*, UIAutomator [65], Monkey runner [45] and Robotium [59]). MobiPlay [55], Reran [25] and SPAG-C [37] are record-and-replay approaches which record the event sequence during the manual exercising, and generate replayable scripts. Complementary to these semi-automatic testing, fuzz testing approaches, for example, Monkey [67], Dynodroid [42] and VanarSena [56], generate random input sequences to exercise Android apps. Symbolic execution-base testings (*e.g.*, Mirzaei *et al.* [44], ACTEve [5], Jensen *et al.* [29], EvoDroid [43] and A^3E [9]) aim at exploring the app functions systematically. Model-based testings (*e.g.*, Android Ripper [3] and SwiftHand [16]) aim at generating a finite state machine model and event sequences to traverse the model. Test case selection techniques (*e.g.*, [27, 64]) can also be adopted to exercise the apps. App exercising mechanisms work as plugin modules of `DiagDroid` , enabling the developers to exploit their merits under different circumstances.

8. CONCLUSION

In this paper, we design `DiagDroid` for diagnosing Android performance issues. To make `DiagDroid` a practical, handy tool, we carefully consider the system design requirements like *compatibility*, *usability*, *flexibility* and *low overhead*. Specifically, `DiagDroid` relies solely on the general features of Android. Hence it works for most mainstream Android devices, depending on no manufacturer specifics. Moreover, `DiagDroid` is convenient to use via a simple installation. It requires no efforts to recompile the OS kernel and the Android framework. With a plugin mechanism, `DiagDroid` provides the flexibility in selecting the test executor to exercise a target app. Finally, `DiagDroid` keeps low overhead by instrumenting slightly on the framework.

To conclude, this paper focuses on an important type of Android performance issues caused by task execution dependency. We carefully model the performance of the asynchronous tasks and their dependency. `DiagDroid` is implemented accordingly for task-level performance diagnosis. It is equipped with a set of sophisticated task profiling approaches based on the Android multithreading mechanisms. We show `DiagDroid` can effectively reduce human efforts in detecting and locating performance issues by applying the tool successfully in finding bugs in tens of real-world apps.

9. ACKNOWLEDGEMENTS

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.

10. REFERENCES

[1] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proc. of ICSE '16*, 2016.

[2] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proc. of OOPSLA '10*, pages 739–753, 2010.

[3] 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 ASE '12*, pages 258–261, 2012.

[4] 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 DSN '13*, pages 1–8, 2013.

[5] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. of FSE '12*, pages 1–11, 2012.

[6] Apktool: A tool for reverse engineering Android apk files. http://ibotpeaches.github.io/Apktool/.

[7] AsyncTask. http://developer.android.com/reference/android/os/AsyncTask.html.

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

[9] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proc. of OOPLSA '13*, pages 641–660, 2013.

[10] A. Banerjee. Static analysis driven performance and energy testing. In *Proc. of FSE '14*, pages 791–794, 2014.

[11] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. Static analysis driven cache performance testing. In *Proc. of RTSS '13*, pages 319–329, 2013.

[12] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, 41(4):384–407, Apr 2015.

[13] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *Proc. of FSE '09*, pages 3–12, 2009.

[14] Y. Cai and L. Cao. Effective and precise dynamic detection of hidden races for java programs. In *Proc. of FSE '15*, pages 450–461, 2015.

[15] 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 IMC '14*, pages 151–164, 2014.

[16] W. Choi, G. Necula, and K. Sen. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proc. of OOPSLA '13*, pages 623–640, 2013.

[17] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *Proc. of ASE '15*, pages 429–440, 2015.

[18] Configuring ART. https://source.android.com/devices/tech/dalvik/configure.html.

[19] DiagDroid - Android Performance Diagnosis via Anatomizing Asynchronous Executions. http://www.cudroid.com/DiagDroid.

[20] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of SOSP '01*, pages 57–72, 2001.

[21] F-Droid. https://f-droid.org/.

[22] FBReader - Favorite Book Reader. https://fbreader.org/.

[23] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. Clapp: Characterizing loops in android applications. In *Proc. of FSE '15*, pages 687–697, 2015.

[24] G. Gan, C. Ma, and J. Wu. *Data Clustering: Theory, Algorithms, and Applications (ASA-SIAM Series on Statistics and Applied Probability)*. 2007.

[25] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proc. of ICSE '13*, pages 72–81, 2013.

[26] 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 MobiSys '15*, pages 137–150, 2015.

[27] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proc. of ICSE '15*, pages 483–493, 2015.

[28] J. Huang and L. Rauchwerger. Finding schedule-sensitive branches. In *Proc. of FSE '15*, pages 439–449, 2015.

[29] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proc. of ISSTA '13*, pages 67–77, 2013.

[30] M. Ji, E. W. Felten, and K. Li. Performance measurements for multithreaded programs. *SIGMETRICS Perform. Eval. Rev.*, 26(1):161–170, June 1998.

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

[32] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proc. of FSE '09*, pages 13–22, 2009.

[33] 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 MobiSys '15*, pages 151–165, 2015.

[34] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32(6):382–403, June 2006.

[35] Y. Lin, S. Okur, and D. Dig. Study and refactoring of android asynchronous programming. In *Proc. of ASE '15*, pages 224–235, 2015.

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

[37] 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. *Software Engineering, IEEE Transactions on*, 40(10):957–970, Oct 2014.

[38] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and

detecting performance bugs for smartphone applications. In *Proc. of ICSE '14*, pages 1013–1024, 2014.

[39] Y. Liu, C. Xu, and S.-C. Cheung. Diagnosing energy efficiency and performance for mobile internetware applications. *Software, IEEE*, 32(1):67–75, Jan 2015.

[40] Low RAM Configuration. https://source.android.com/devices/tech/config/low-ram.html.

[41] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. of ASPLOS '08*, pages 329–339, 2008.

[42] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proc. of FSE '13*, pages 224–234, 2013.

[43] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proc. of FSE '14*, pages 599–609, 2014.

[44] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, Nov. 2012.

[45] Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[46] 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 MobiSys '15*, pages 419–433, 2015.

[47] F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[48] 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 Mobisys '15*, pages 287–300, 2015.

[49] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[50] S. Niida, S. Uemura, and H. Nakamura. Mobile services. *IEEE Vehicular Technology Magazine*, 5(3):61–67, Sep 2010.

[51] A. Nistor, P. C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proc. of ICSE '15*, 2015.

[52] OpenLaw - Die Gesetze App. https://openlaw.jdsoft.de/.

[53] Processes and Threads. http://developer.android.com/guide/components/processes-and-threads.html.

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

[55] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proc. of ICSE '16*, 2016.

[56] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proc. of Mobisys '14*, pages 190–203, 2014.

[57] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proc. of OSDI '12*, pages 107–120, 2012.

[58] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proc. of SOSP '13*, pages 85–100, 2013.

[59] RobotiumTech. Robotium: User scenario testing for Android. http://www.robotium.org.

[60] V. Roto and A. Oulasvirta. Need for non-visual feedback with long response times in mobile hci. In *Proc. of WWW '05*, pages 775–781, 2005.

[61] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems. In *Proc. of ASE '15*, pages 342–352, 2015.

[62] M. Selakovic and M. Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *Proc. of ICSE '16*, 2016.

[63] StrictMode. http://developer.android.com/reference/android/os/StrictMode.html.

[64] V. Terragni, S.-C. Cheung, and C. Zhang. Recontest: Effective regression testing of concurrent programs. In *Proc. of ICSE '15*, pages 246–256, 2015.

[65] Testing Support Library - UI Automator. https://developer.android.com/tools/testing-support-library/index.html#UIAutomator.

[66] R. A. to search for countries based on several parameters. https://github.com/abhi2rai/RestC.

[67] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[68] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proc. of ICSE '13*, pages 552–561, 2013.

[69] Xposed Module overview. http://repo.xposed.info/module-overview.

[70] Xposed Module Repository. http://repo.xposed.info.

[71] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. *SIGPLAN Not.*, 49(4):193–206, Feb. 2014.

[72] 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 CODES+ISSS '13*, pages 1–10, 2013.

[73] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance prediction of configurable software systems by fourier learning. In *Proc. of ASE '15*, pages 365–373, 2015.