Responsiveness Analysis Tool for Android Application

Thanaporn Ongkosit
Keio University
Yokohama, Japan
koy@doi.ics.keio.ac.jp

Shingo Takada
Keio University
Yokohama, Japan
michigan@ics.keio.ac.jp

ABSTRACT

Responsiveness is an important type of quality factor in Android application because it directly affects user experience. When the user interface thread performs lengthy operations, the user may feel that the application has become sluggish or frozen. This may lead to a negative user experience, poor review, and loss in market success. This paper proposes a static responsiveness analysis tool for Android applications to find potentially poor responsiveness defects which are difficult to detect by conventional testing methods as they are sensitive to the user environment. This tool finds responsiveness defects by discovering operations invoked in the user interface thread that may block the execution of other operations. We collect these operations according to Android developer guideline and previous related work. The proposed tool successfully found 45 potential responsiveness defects in seven open source Android applications.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance Analysis

Keywords

Android App, Static Analysis, Responsiveness

1. INTRODUCTION

The popularity of smartphones has continued to increase rapidly especially for Android which is now the leading platform in the smartphone market [4]. The number of users using smartphones in their daily lives has risen dramatically every year. Mobile applications can be seen in almost every industry such as education, banking, news and gaming. As with any applications, a successful mobile application requires effort to improve and assure software quality before release. Otherwise, the user may choose to discontinue its use.

One report [5] showed that the top reasons that cause users to uninstall an application, submit low rating and/or give negative review on an app market include freezes and slow response, both of which are related to responsiveness. Thus, developers need to be careful not to include defects in the code that will cause responsiveness issues. Although there are many testing techniques and performance tools [8][9][10], very few focus on responsiveness [11].

Poor responsiveness is an important defect that affects the user's perception. Google guideline states that "100 to 200ms is the threshold beyond which users will perceive slowness in an application" [1][6]. An application that frequently responds slowly is one of the top reasons for negative user experience [5].

There are many possible causes for unresponsiveness, such as deadlock, infinite loop and incorrect termination condition, all of which are concerned with the correctness of the program. Another possible cause is operations that need to execute for a long time. This paper focuses only on the last cause, i.e., poor responsiveness caused by lengthy operations.

Testing responsiveness defects is very challenging. First, responsiveness of an application may differ depending on the environment. The same application may perform differently in a 3G network and a stable wireless network. As a result, it is difficult to both expose and reproduce responsiveness failures. Second, there is no tool to help developers uncover responsiveness issue automatically. These reasons motivate our work.

We propose a responsiveness analysis tool for Android applications to help developers identify potential causes of unresponsive problem focusing on long running operations. Our tool also generates a report regarding the identified problem. The main contributions of this work are as follows:

1. We define a list of potentially long running operations based on four categories of API calls, specifically network operations, database operations, I/O operations and bitmap processing operations.

2. We present a technique to perform static analysis for Android applications to uncover responsiveness defects based on our defined potentially long running operations.

3. We implement a responsiveness static analysis tool, and perform an experiment on open source Android

DeMobile'14, November 17, 2014, Hong Kong, China
ACM 978-1-4503-3225-5/14/11
http://dx.doi.org/10.1145/2661694.2661695

applications. We found several responsiveness defects in each application.

The rest of this paper first discusses the responsiveness issue in Android. Section 3 describes our proposed responsiveness analysis tool. Section 4 shows the results of a preliminary evaluation. Section 5 makes concluding remarks.

2. RESPONSIVENESS ISSUE IN ANDROID

Poor responsiveness is a critical type of defect that should be taken into account to assure software quality. It is easy for an application to still be sluggish or freeze even if it has already passed extensive performance tests. A key point is that an Android application can run entirely on a single thread called "UI thread" which handles all user input events. When an operation is being executed on the UI thread, no other user input events will be processed. Thus, the probability becomes higher for an application run entirely on the UI thread to block the system from processing any other user input events because that application has some long running operation, such as network access, running on the UI thread.

In Android, the worst case scenario is the appearance of the "Application Not Responding"(ANR) dialog. This dialog is displayed when the application has stopped responding to user input for 5 seconds [1][6]. This error dialog directly affects the user experience in an undesirable way. Even if the ANR dialog does not appear, the user may still decide to uninstall an application if its operation frequently takes longer than 200 ms.

Thus, methods that run on the UI thread should perform as little work as possible. Long running operations should be spawned to other threads, so that the UI thread is free to process other incoming user inputs. If we can check for operations on the UI thread that may have long execution time, we can mitigate the responsiveness issue.

3. RESPONSIVENESS ANALYSIS TOOL

We propose a responsiveness analysis tool to statically analyse potentially poor responsiveness issues and generate reports regarding the identified issues. The overview of our architecture is shown in Figure 1. The **Call graph generator** first takes Android source code and generates context sensitive call graphs. The **Poor Responsiveness Analysis** component traverses the generated call graph, and discovers potential responsiveness defects using a list of **potentially blocking API's** as a guide. Finally, the found defects are output in the form of a report where the potential responsiveness defects may be ranked.

3.1 Potentially blocking operations

The starting point of our work is to define operations that may take too long to process, resulting in the application blocking other incoming user inputs from being processed. According to Android developer guidelines and previous work [2][11], there are four categories of API that are common causes of poor responsiveness: network access, storage access, on-device database access, and bitmap processing. However, they provided only categories of operation and some examples of library packages. We used these examples as a guideline to find potentially blocking APIs by studying related library packages and how each API in the package works. Thus, based on these four categories, we made a list

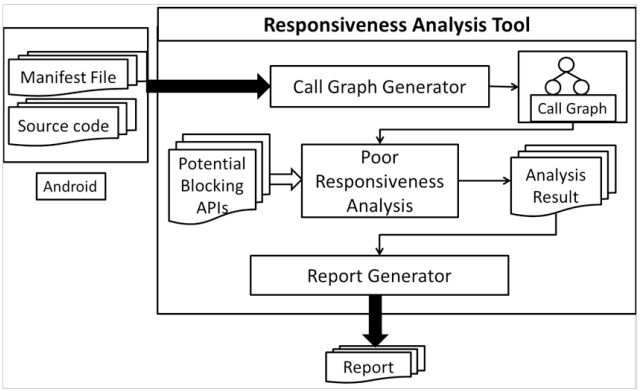

Figure 1: Overview of Architecture

of potentially blocking API's which we consider as the basic unit of operation that we would like to detect. Table 1 shows 12 out of 50 of the operations for each of the four categories.

3.2 Call Graph Generator

A call graph is a basic representation of programs. It is a directed graph that represents calling relationship between program's procedures. Each node represents a procedure while each edge represents calling relationship. For example, edge(f, g) indicates that procedure f calls procedure g. The precision of a call graph can vary depending on the type of call graph [7]. The most precise call graph is context-sensitive call graph which means that for each procedure, the graph contains a separate node for each call to the procedure. The least precise call graph is context-insensitive call graph which means that there is only one node for each procedure. Our work takes the context-sensitive approach.

We use Modisco [3], which is an Eclipse plugin, to first generate the Java model of an application. We then extract information of interest from the model such as variable declaration and method invocation as well as the entry point of the call graph to construct context-sensitive call graphs. The entry point is especially important because Android is event-driven, and instead of a single "main" method, an Android application may contain a large number of callback methods. As a result, there are multiple entry points to the call graph.

We can categorize callback methods into two types:

1. Callback methods which are triggered automatically when the application is in some certain state without user interaction. These include callback methods for each application component such as Activity.onCreate(). There are many other callback methods in the Android Framework. For example, the database onCreate() is called automatically by the framework, if the database is accessed but not yet created. The camera onPictureTaken() callback is called when image data is available after a picture is taken.

2. Callback methods triggered by user input events. We refer to these as event callbacks. Examples are methods triggered by Touch events (e.g. onClick, onLongClick, onTouch) and Keyboard events (e.g. onKeyUp).

Figure 2 shows an example of a generated call graph.

Table 1: Example of Potentially Blocking Operations

Category	API Call	
	Method	Description
Network	java.net.URL.openConnection()	Returns a new connection to the resource
	org.apache.http.client.HttpClient.execute (HttpUriRequest request)	Executes a request using the default context and returns the response to the request
	java.net.URLConnection.connect()	Opens a connection to the resource
	java.net.Socket.connect(SocketAddress address)	Connects this socket to the given remote host address and port
	java.net.URLConnection.getInputStream()	Returns an InputStream for reading data from the resource
Storage	java.io.OutputStream.write (byte[] byteArray)	Writes the byte array buffer to this stream
	java.io.InputStream.read(byte[] buffer)	Reads a single byte from this stream and returns it as an integer
	java.io.BufferedInputStream.read (byte[] buffer, int byteOffset, int byteCount)	Reads up to specified location from this stream and stores them in the byte array buffer
Database	android.database.sqlite.SQLiteDatabase.execSQL(String query)	Execute a single SQL statement
	android.database.sqlite.SQLiteDatabase.query (String table,...)	Query the given table and returns a Cursor over the result set
Bitmap	android.graphics.BitmapFactory.decodeFileDescriptor (FileDescriptor fd)	Decode a bitmap from the file descriptor.
	android.graphics.BitmapFactory.decodeStream (InputStream is)	Decode an input stream into a bitmap

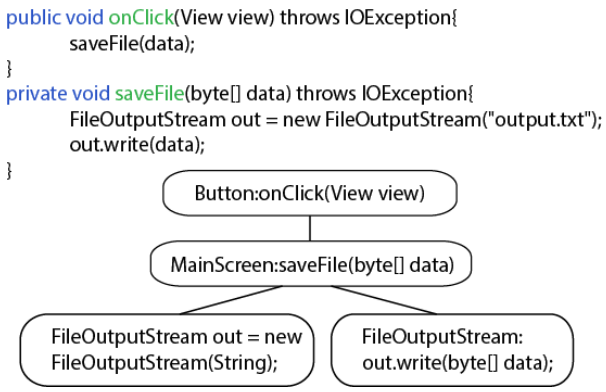

Figure 2: Generated Call Graph

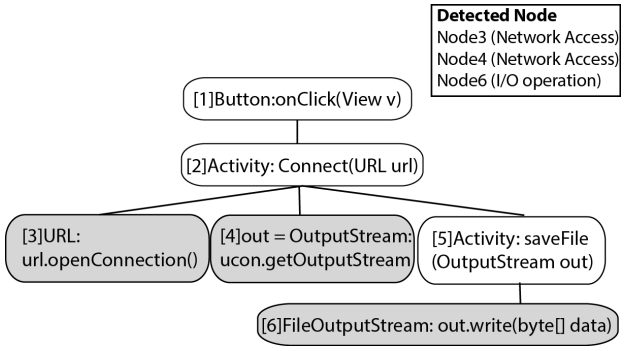

Figure 3: Poor Responsiveness Analysis

3.3 Poor Responsiveness Analysis

In our work, an application is responsive if all methods that run on the UI thread are not blocking operations. If a blocking operation is running on the UI thread, then that application may be unresponsive. As a result, the main goal of our analysis is to uncover potentially blocking operations running on the UI thread to help developers locate potential causes of poor responsiveness.

The analysis first traverses the context-sensitive call graph, and searches for problematic nodes, i.e., nodes that contain blocking operations, by using a list of defined potentially blocking operations. Whenever the tool finds a problematic node, it will keep three pieces of information which are the problematic node, the method that contains the detected node and type of operation, i.e. database access, network access, storage access or bitmap processing.

Figure 3 shows a call graph of operations that are executed on the UI thread. The analysis found that several nodes (nodes 3, 4 and 6) contain blocking operations. Nodes 3 and 4 are network access operations, whereas node 6 writes data to the server.

The tool focuses on the variable the operation uses to distinguish the type of operation. For example, in Figure 3, although out.write() is storage API, out is an outputStream from server. As a result, this operation should be considered as network access type. These information will be used later when generating the report.

3.4 Report Generator

The Report Generator's main function is to generate the report of the detected issues which provides rank, number of invocations, detected node, parent node and filename. Report Generator also ranks the issues (if there are more than one issue). The rank is based on the number of invocations to the detected node. The Report generator uses information kept in the analysis phase to generate a report as follows:

1. If there is only one defect, it generates a report that shows the defect and the file that contains the defect.

Table 2: Experiment Result

Application	App Size		API category			
	Activity	Class	Database	Bitmap	Network	Storage
AnkiDroid	19	298	16	2	-	-
Wallabag	4	10	6	-	-	-
AnyMemo	29	290	-	1	-	-
VLC	7	139	5	4	-	5
APV	4	28	2	-	-	1
ACV	12	167	1	-	-	-
FBReader	35	825	-	2	-	-

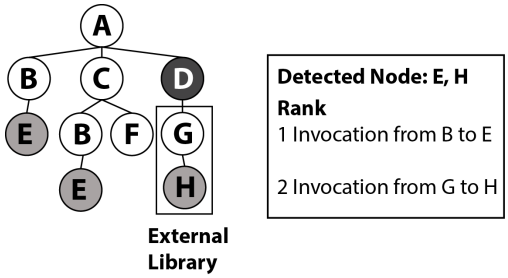

Figure 4: Report Generation

2. If there are more than one defect, the Report generator will count the number of invocations to the API. For example (Figure 4), there are two invocations from node B to node E whereas one invocation from node G to node H. The higher number of invocations has the higher rank.

3. If there is more than one defect with the same number of invocations, the node with 'network access' type is ranked higher.

If the application has a blocking operation that is in an external library, the report will show the node in the application code that leads to the defect, because the developer can only handle the application's code. So, in Figure 4, the tool will report node D as problematic because it calls the operation that leads to the actual problematic node.

4. EVALUATION

We performed a preliminary evaluation of the proposed responsiveness analysis tool on seven open-source Android applications. The experiment result is shown in Table 2. The table shows the list of applications, and each application's number of activities and number of classes. The last four columns show the number of potentially blocking operations in the UI thread detected by the proposed tool classified by categories.

The result shows that each application contains potential responsiveness issues, with the database category API's having the most. One interesting point is that there were no network-related responsiveness defects. This may indicate that most developers already realize that the network operation can directly lead to undesirable user experience.

5. CONCLUSION

Responsiveness is a crucial issue because it can directly affect the user experience and application success. We proposed a responsiveness analysis tool for Android applications to help developers discover potential responsiveness issues in the source code. A preliminary experiment shows that our tool can effectively uncover potential responsiveness defects.

For future work, the precision of call graph can be improved by defining more entry points. The precision of analysis can be improved with information from profiling. The tool can provide deeper analysis especially for database operation e.g. SQL queries.

6. REFERENCES

[1] Keeping your app responsive. http://developer.android.com/training/articles/perf-anr.html,(accessed:27-June-2014).

[2] Loading bitmaps efficiently http://developer.android.com/training/displaying-bitmaps/process-bitmap.html, (accessed:27-June-2014).

[3] Modisco. http://www.eclipse.org/gmt/modisco/, (accessed:27-June-2014).

[4] Smartphone platform market share. http://www.comscore.com/insights/press-releases/2014/3/comscore-reports-january-2014-us-smartphone-subscriber-market-share, (accessed:27-June-2014).

[5] Survey: Exploring the reasons users complain about apps. http://www.fiercedeveloper.com/story/survey-exploring-reasons-users-complain-about-apps/2012-11-09,(accessed:27-June-2014).

[6] B. Fitzpatrick. Writing zippy android apps. *Google I/O Developers Conference 2010*.

[7] D. Grove and C. Chambers. A framework for call graph construction algorithms. *TOPLAS 2001*, 23:685–746, November 2001.

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

[9] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. In *Newsletter ACM SIGMETRICS Performance Evaluation Review*, pages 26–31, September 2008.

[10] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Proling resource usage for mobile apps: a crosslayer approach. In *MobiSys 2011*, pages 321–334, 2011.

[11] S. Yang, D. Yan, and A. Rountev. Testing for responsiveness in android applications. In *MOBS 2013*, pages 1 – 6, May 2013.