CAPTURE AND REPLAY TECHNIQUE FOR REPRODUCING CRASH IN ANDROID APPLICATIONS

Ajay Kumar Jha, Woo Jin Lee
School of Computer Science and Engineering, Kyungpook National University
1370, Sankyuk-dong, Buk-gu, Daegu, Korea
ajaykjha123@yahoo.com, woojin@knu.ac.kr(Correspondence)

ABSTRACT
Software testing takes significant amount of time and cost. Many in-house software testing techniques are available but all the techniques can't be used due to time and budget constraints so limited testing efforts are applied most of the time. Due to this limited testing effort there is always chance of failing the software in the field. To make this worse, developers don't have any clue why deployed software failed or crashed. When the software crashes, stack trace is usually sent back to developer which in most of the cases does not provide enough information to pinpoint the cause of crash. We present a capture and replay technique which addresses this problem by recording the sequence of events in the field before crash and reproducing the sequence of events in-house after crash. Our approach is content-based which records events and data associated with those events during program execution. Our technique mainly focuses on the Android applications but similar approach can also be used for other object oriented applications.

KEY WORDS
Reproducing crash, field failure, capture, replay, debugging, android applications

1. Introduction

Software behaves differently under different circumstances which needs environment and input commands to run. During in-house testing it is not feasible to test the software under all the available environment and input conditions which introduces non-determinism in the software behavior. Even software itself can cause non-determinism due to highly advanced programming techniques like multi-threading. Debugging software containing non-determinism behavior is highly complex task.

Debugging is all about reproducing the execution and pin-pointing the bug. Cyclic debugging is still very popular among developers in which a program is executed repeatedly and the part of the program which causes the bug is narrowed down till the actual bug is found. But if program behaves differently in each execution then it is impossible to reproduce the execution and cyclic debugging is useless in this condition. So cyclic debugging is only useful in sequential, deterministic programs which produce the same execution in different run.

One way to solve this non-deterministic problem in debugging is to reproduce the execution and this can be achieved by recording the execution and replaying the recorded execution while debugging. Record and replay solves the problem of debugging non-deterministic programs but not without some expanses in the form of execution time and memory overhead. If record and replay technique is used for in-house debugging purpose then moderate overhead can be acceptable but if this technique is used for field failure debugging purpose then even moderate overhead is unacceptable.

The major obstacle is to reduce the time and memory overhead so that the technique can be used for debugging of deployed software. These overheads are mainly caused by recording huge volume of data during capture phase in the field. It is obvious that to reduce overhead less data should be recorded but recording less data has another drawback. Due to the lack of sufficient data execution may not be reproduced accurately. So to reproduce execution accurately while maintaining the acceptable level of time and memory overhead tradeoff is required between the volume of data recorded and time and memory efficiency.

Many capture and replay techniques have been proposed previously but they all have their own limitations. Content-based techniques [1, 2, 3, 4] record events and data associated with those events during capture phase and based on those recorded information execution is reproduced during replay phase. This approach generates huge volume of data causing huge time and space overhead. Later this technique was improved by recording only selective data. Another approach for capture and replay technique is order-based [6, 7, 11] in which only order of execution events are recorded. Later based on that ordered events, execution is reproduced in replay phase. Order-based approach is more efficient but it has also drawback since slight change in environment during replay phase can diverge the execution path.

Though there are many capture and replay techniques available for debugging of deployed software, they have been hardly used by developers. One of the main reason behind this, which we mentioned before, is they are not efficient enough to be used in the deployed software. Another reason is they are highly complex to implement. We present a capture and replay technique to reproduce crash in android applications. Our approach is content-based in which only selective execution events and data associated with those events which are necessary to reproduce the crash are captured. Our technique is very simple to implement and generate acceptable level of time overhead to be used in the deployed applications. Our technique efficiently reproduces crash in Android application and effectively captures and replays GUI related events. Preliminary experimental results show that the technique can be used in the deployed applications.

The rest of the paper is organized as follows. Section 2 introduces the background on android applications and overview on existing related capture and replay techniques. Section 3 presents the detailed procedure of our capture and replay technique with preliminary experimental results on section 4. Section 5 concludes the paper.

2. Background And Related Work

2.1 Android Fundamentals

Android [13] is a Linux based operating system primarily designed for mobile devices. Android applications are written in the Java programming language. Android Software Development Kit (SDK) offers the tools necessary to develop and debug applications on the Android platform. By default every application runs in its own Linux process and each process has its own Dalvik virtual machine. Android starts the process when any of the application's components need to be executed, then shuts down the process when it is no longer needed or when the system must recover memory for other applications.

Application components are the essential building blocks of an Android application. There are four different types of application components.

- **Activities:** An activity represents single screen with which user can interact. An application generally consists of several activities. Activities are independent of each other but they may interact with each other to complete a task. In an application one activity is specified as “main” activity which is presented to the user when the application is launched for the first time. Each activity can then start another activity to perform different tasks. Each activity in an android application is either a direct subclass of the Activity base class or a subclass of an Activity subclass. Activity’s lifecycle is managed by the application framework. An application that presents anything on the display must have at least one activity responsible for that display.

- **Services:** A service is an application component which performs long-running operations or works for remote processes in the background. It does not provide user interface. Another application component can start or bind a service. If a service is started then it can run indefinitely in the background and usually performs a single operation without returning result to the caller however if a service is bounded then it runs only as long as the service is bounded to component. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with inter process communication (IPC). Services are implemented as a subclass of Service class.

- **Content Providers:** A content provider manages a shared set of application data. It encapsulates data and provides that to application. Through content provider application can access the data from file, SQLite database, web, or any other persistent storage location. Content providers are also useful for manipulating data that is private to the application. A content provider is implemented as a subclass of ContentProvider.

- **Broadcast Receivers:** A broadcast receiver is a component that responds to system-wide broadcast announcements. It may originate from system (e.g. a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured). Applications can also initiate broadcast. A broadcast receiver is implemented as a subclass of BroadcastReceiver.

2.2 Related Work

Though our technique is specifically tailored for android applications, it is closely related to Java based capture and replay technique jRapture [1]. jRapture captures interactions between a Java application and the underlying system by using modified Java API classes. During replay phase, it presents each thread with exactly the same input sequence it recorded during capture. The technique used in capture phase in jRapture has some practical limitations. Although Steven et. al mention that the fields accessible to methods need to be captured, they are not captured by implementing their technique [12]. Also in [1] objects are captured by using serialization but for this to happen class from which they are instantiated must implement Serializable interface which is not always possible.

Another closely related technique is SCARPE [3]. It is also a Java based capture and replay technique. The technique identifies the boundaries of the observed set based on the user-provided list of observed classes and suitably modifies the application to capture interactions between the observed set and rest of the system. It overcomes the problem of object serialization by generating an object ID which uniquely identifies a class instance during capture phase and based on that object ID it retrieves or creates the object during replay phase. On average it imposes 30% - 50% time overhead which is still high to be used in the deployed application. Also this technique does not deal with GUI related events.

Other techniques [6, 7, 9, 10, 11] focus mainly on concurrent behavior of applications. Though similar in approach their main purpose is to reproduce failures caused by concurrency related events. In contrast to our technique, these techniques capture different sets of events. Table 1 shows the comparison of events captured by our technique and other related techniques. Symbol √ denotes captured events whereas × denotes events which are not captured.

Table 1. Comparison of events captured by our technique and other related techniques.

	jRapture	SCARPE	ReCrash	Our
Method Signature	√	√	√	√
Parameters	√	√	√	√
Returned Value	√	√	√	√
Used Fields	√	√	√	√
Exceptions	√	√	√	√
Objects	√	ID & Type	√	Type
GUI Events	√	×	×	√

3. Capture and Replay Technique

Though android applications are developed in Java programming language, their organization is quite different than other Java-based applications. Our technique is specifically designed for android applications, which has three major components Data Collector, Checkpoint Detector, and Crash Detector as shown in figure 1. Data Collector records the execution events, Checkpoint Detector implements the checkpoint technique, and Crash Detector detects the crash and generates the log file. These components are described in detail in section 3.2.

Our technique has three main phases: instrumentation, capture and replay. In instrumentation phase the application is modified by inserting probes into the source code before the application is deployed in the field. During capture phase selected data from execution of the deployed application is recorded and periodically stored into a log file while in replay phase data from the log file is provided as input to execution and the program is replayed for debugging of field failures.

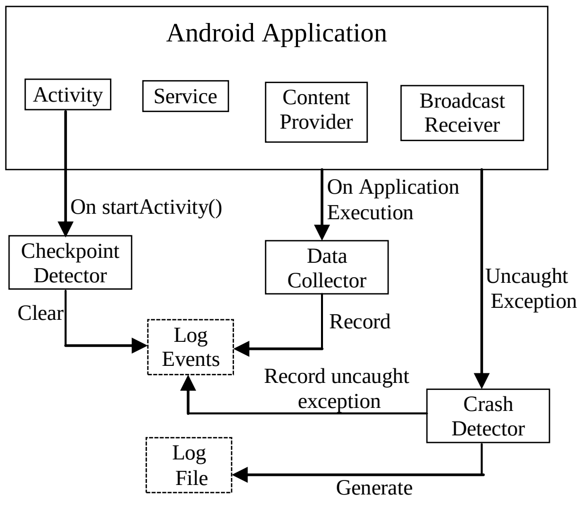


Figure 1. Overall structure of our system

3.1 Instrumentation Phase

This section describes the overview of the instrumentation technique. Our capture and replay technique uses AspectJ [16, 17] for instrumentation. AspectJ is an implementation of aspect-oriented programming for Java. Existing capture and replay techniques introduce probe by instrumenting directly in source code [2, 3, 7], modifying API [1] or virtual machine [8, 9], and making changes in host operating system [5, 14]. The instrumentation technique which we are using introduces probe into code but it separates the actual code from the instrumentation code and also the instrumentation code can be reused in another application. Code reusability is a huge advantage over existing instrumentation techniques. With AspectJ it is also possible to enable and disable the instrumentation code whenever required.

Aspect-oriented programming provides three main constructs which are join points, advice, and pointcuts. Join points are specific points within the application which developer would like to intercept for example join when a method is called. The purpose for which we are intercepting join points is defined in advice section for example record signature of a method when the method is called. The mechanism for declaring an interest in a join point to initiate a piece of advice is pointcut.

Pointcuts not only intercept join points but also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations. As our main goal of instrumentation is to capture those values exposed by pointcut, AspectJ serves the right purpose with additional advantage of code reusability.

3.2 Capture Phase

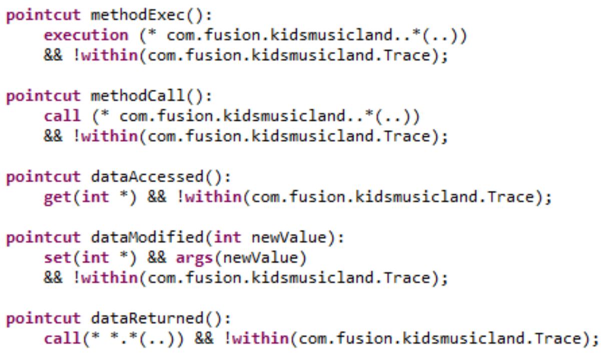
The capture phase takes place when the deployed application starts executing. The application must be instrumented before it is deployed in the field. When the application runs, the probes in the code suitably generates events. The events and the data associated with those events are stored in a list. During the execution if an un-handled exception is thrown then the un-handled exception along with the stored events of the list is flushed into a log file. The log file is then sent to the developer for replaying the execution.

3.2.1 Data Collector Component

For any application to crash it must change its state from normal to crash state and this state transition should be triggered by some events. Unless we know the behavior of normal state and events which caused the transition it's impossible to reproduce crash state of the application. Also it's impossible to know, in advance, when the application is going to crash so in our capture phase we record behavior of each state of the application and the events which triggered the transition.

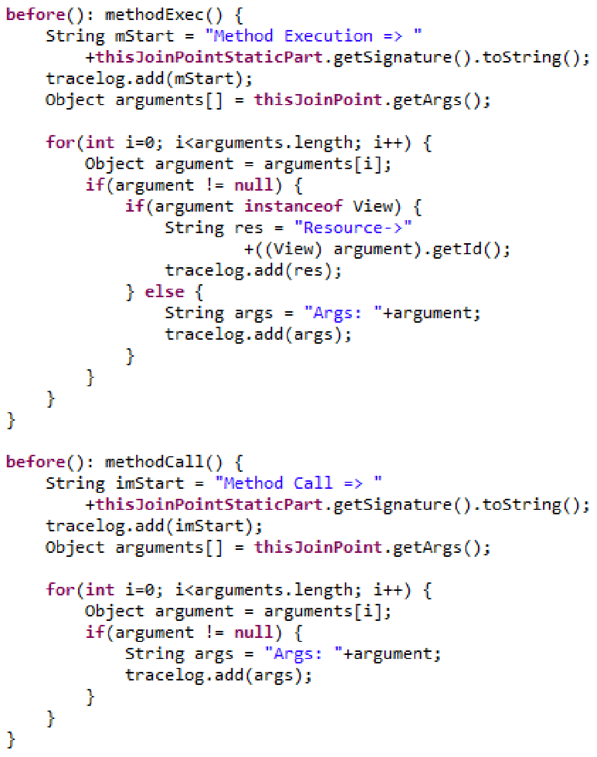
A method call can change the state of the application by changing the values of parameters, by changing the values of used fields, or by returning a value [1]. Our capture technique records method's signature, parameters, used fields, returned values, and any raised exceptions. In case of graphical user interface an additional value called resource ID is recorded. Figure 2 shows the AspectJ pointcuts which are used to capture these events.

Pointcuts methodExec and methodCall mentioned in figure 2 capture the method execution and method call respectively. The methodCall pointcut is invoked when the method is invoked and the context of the advice invocation is the calling class whereas methodExec pointcut is invoked once the method has been entered and the calling context is the method being executed. Detailed differences between these two types of poincuts are described in [16, 17]. Figure 3 shows the advices for methodCall and methodExec pointcuts.



```
pointcut methodExec():
    execution (* com.fusion.kidsmusicland..*(..))
    && !within(com.fusion.kidsmusicland.Trace);

pointcut methodCall():
    call (* com.fusion.kidsmusicland..*(..))
    && !within(com.fusion.kidsmusicland.Trace);

pointcut dataAccessed():
    get(int *) && !within(com.fusion.kidsmusicland.Trace);

pointcut dataModified(int newValue):
    set(int *) && args(newValue)
    && !within(com.fusion.kidsmusicland.Trace);

pointcut dataReturned():
    call(* *.*(..)) && !within(com.fusion.kidsmusicland.Trace);
```

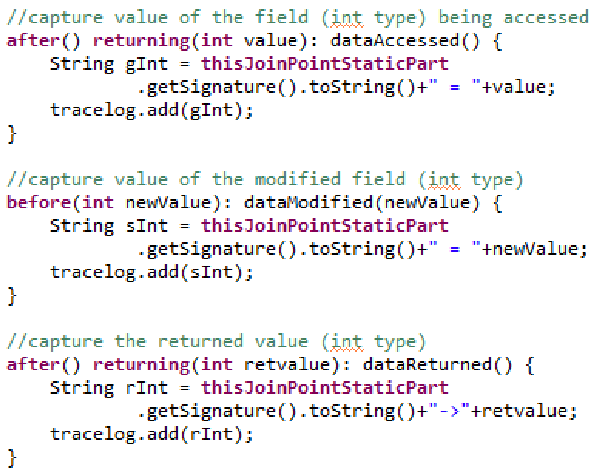
Figure 2. AspectJ pointcuts to capture events



```
before(): methodExec() {
    String mStart = "Method Execution => "
        +thisJoinPointStaticPart.getSignature().toString();
    tracelog.add(mStart);
    Object arguments[] = thisJoinPoint.getArgs();

    for(int i=0; i<arguments.length; i++) {
        Object argument = arguments[i];
        if(argument != null) {
            if(argument instanceof View) {
                String res = "Resource->"
                        +((View) argument).getId();
                tracelog.add(res);
            } else {
                String args = "Args: "+argument;
                tracelog.add(args);
            }
        }
    }
}

before(): methodCall() {
    String imStart = "Method Call => "
        +thisJoinPointStaticPart.getSignature().toString();
    tracelog.add(imStart);
    Object arguments[] = thisJoinPoint.getArgs();

    for(int i=0; i<arguments.length; i++) {
        Object argument = arguments[i];
        if(argument != null) {
            String args = "Args: "+argument;
            tracelog.add(args);
        }
    }
}
```

Figure 3. Advices for methodExec and methodCall pointcuts.

Advices for methodCall and methodExec pointcuts record method's signature and parameters in an array list named tracelog as shown in figure 3. In case of methodExec pointcut an additional event resource ID is captured. Figure 4 shows the advices for the pointcuts dataAccessed, dataModified, and dataReturned.



```
//capture value of the field (int type) being accessed
after() returning(int value): dataAccessed() {
    String gInt = thisJoinPointStaticPart
            .getSignature().toString()+" = "+value;
    tracelog.add(gInt);
}

//capture value of the modified field (int type)
before(int newValue): dataModified(newValue) {
    String sInt = thisJoinPointStaticPart
            .getSignature().toString()+" = "+newValue;
    tracelog.add(sInt);
}

//capture the returned value (int type)
after() returning(int retvalue): dataReturned() {
    String rInt = thisJoinPointStaticPart
            .getSignature().toString()+"->"+retvalue;
    tracelog.add(rInt);
}
```

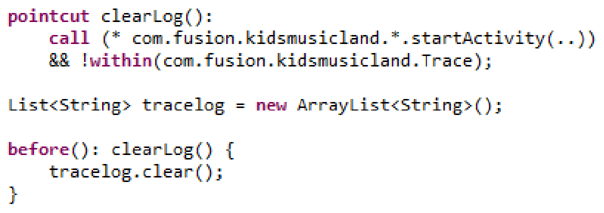
Figure 4. Advices for dataAccessed, dataModified and dataReturned pointcuts.

Advices for pointcuts dataAccessed, dataModified, and dataReturned shown in figure 4 respectively records accessed value, modified value, and returned value. Primitive data types such as boolean, int, float, long, and string can be captured easily with limited performance overhead but capturing objects require serialization. Serializing objects substantially increases the performance overhead. Serialization is a process to convert a data structure or objects into a format which can be stored so finally what we are interested in is primitive data types. As long as we are recording values of primitive data types which affect the objects we don't need to capture them.

Graphical user interface is an integral part of Android application. In Android application an activity represents a single screen or user interface. Elements of user interface are built using View and ViewGroup which are objects that draw something on the screen with which user can interact. There are two ways to intercept events generated from user interaction with the user interface (UI). One way is to capture the events from the View object with which user interacts and another way is to use event listeners. Event listeners are collections of nested interfaces of View class. In both cases callback methods handle the events. These callback methods are called by Android framework when the View to which the listener has been registered is triggered by user interaction with the application. In Android application every View object is identified by a unique ID called resource ID. As shown in figure 3, in our capture phase we record this resource ID so that it can be identified in replay phase.

3.2.2 Checkpoint Detector Component

The target application might run for long period, in such case huge amount of data will be logged in the file and the size of the file will grow substantially. To reduce the size our technique uses activity as a checkpoint because activities are either independent or loosely-coupled with other Android components. When an activity starts we record events in the list and when another activity starts we remove existing data from the list and again we start recording events in the list. This checkpoint technique is implemented by using pointcut and advice shown in figure 5. Appropriate way would have been to record events between start and end of an activity but in Android system may terminate the activity so we don't use this technique since we are not recording system level call in our technique.

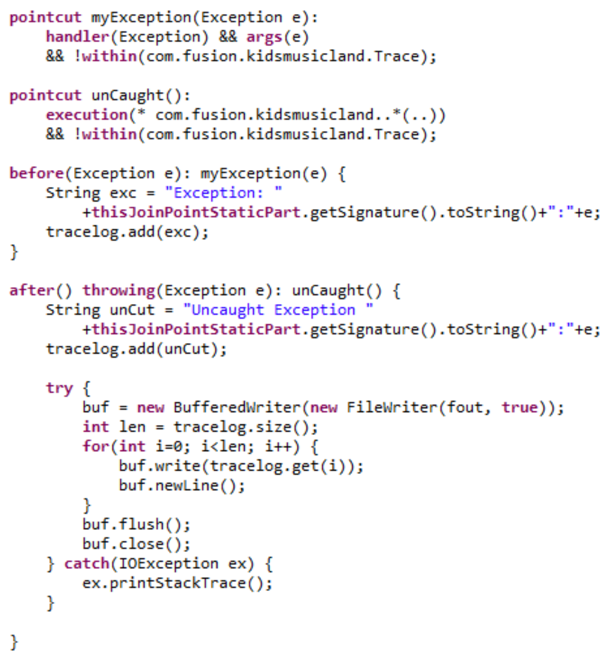


```
pointcut clearLog():
    call (* com.fusion.kidsmusicland.*.startActivity(..))
    && !within(com.fusion.kidsmusicland.Trace);

List<String> tracelog = new ArrayList<String>();

before(): clearLog() {
    tracelog.clear();
}
```

Figure 5. Pointcut and advice for implementing checkpoint technique.

There are two types of activities, one which returns data and another which does not return. The activity which starts returning activity depends on the result of returning activity to perform some task. For example we have one activity named "Play" to play a song and another activity named "Display" to display a list of songs. In this case Play activity starts Display activity which returns a song and then Play activity plays that song. Crash may be caused by dependency relationship between returning and non-returning activities so we only use non-returning activity as our checkpoint.

When one activity starts another activity then the first activity gets paused or stopped but still alive. When the second activity finishes the first activity regains the focus. Suppose application crashes after first activity regains the focus then in the log file we have execution events of second activity and execution events of first activity after it regains focus. Execution events of first activity before it starts second activity get lost due to our checkpoint technique. To reproduce this crash we have to start first activity which in turn starts second activity. Since the execution events of first activity before it starts the second activity have been lost crash may not be reproduced. To overcome this problem we can record the execution events of more than one activity before clearing the event list. Suppose we would like to keep the execution events of two activities then we simply place a counter and increment that counter when AspectJ code intercept *startActivity()* method each time. When the counter becomes three then we clear the list and repeat the process again. This technique increases the accuracy of reproducing crash but also increases the size of log file.



```
pointcut myException(Exception e):
    handler(Exception) && args(e)
    && !within(com.fusion.kidsmusicland.Trace);

pointcut unCaught():
    execution(* com.fusion.kidsmusicland..*(..))
    && !within(com.fusion.kidsmusicland.Trace);

before(Exception e): myException(e) {
    String exc = "Exception: "
        +thisJoinPointStaticPart.getSignature().toString()+":"+e;
    tracelog.add(exc);
}

after() throwing(Exception e): unCaught() {
    String unCut = "Uncaught Exception "
        +thisJoinPointStaticPart.getSignature().toString()+":"+e;
    tracelog.add(unCut);

    try {
        buf = new BufferedWriter(new FileWriter(fout, true));
        int len = tracelog.size();
        for(int i=0; i<len; i++) {
            buf.write(tracelog.get(i));
            buf.newLine();
        }
        buf.flush();
        buf.close();
    } catch(IOException ex) {
        ex.printStackTrace();
    }

}
```

Figure 6. Pointcuts and advices for recording caught and uncaught exceptions.

3.2.3 Crash Detector Component

Our capture technique stores events and data associated with those events in a list during application execution and flush those data into a log file when the application crashes by throwing unhandled exception. Figure 6 shows the pointcuts and advices for recording both caught and uncaught exceptions.

The pointcut myException shown in figure 6 exposes the join points where the exceptions are caught by target applications and its advice records the caught exception whereas the pointcut unCaught exposes method execution join points and its advice records any uncaught exception thrown by the application. After recording the uncaught exception it flushes all the events stored in a list into the log file.The major events of Data Collector, Checkpoint Detector, and Crash Detector components are shown in figure 7.

In Android application there are four components: activities, services, content providers, and broadcast receivers. These components communicate with each other by passing messages which are called intents. Intents are delivered through method calls. Also one part of a component interacts with another part of the component through method calls. In our capture phase we are recording all the events related to method calls so our capture logic can reproduce the crash caused by any components of android application. However in case of service component our checkpoint technique may influence the logic in reproducing crash

Services are generally used to perform long-running operations but our checkpoint technique is designed to record execution events for short period to reduce the size of log file. If crash is caused by long-running interrelated events generated by service then it’s impossible to reproduce the crash by using our capture logic because it’s not feasible to record all those events in our technique.

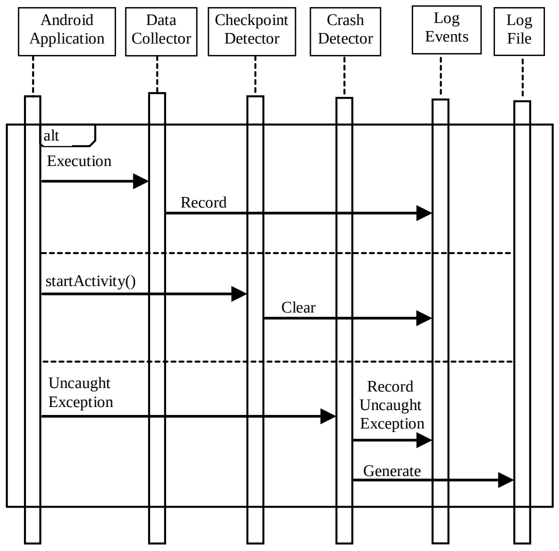


Figure 7. Main scenarios of our capture system

3.3 Replay Phase

In replay phase our technique uses the instrumented version of the application and the log file generated during capture phase for reproducing crash. Replay phase takes place in-house so overhead is not an important factor. Our technique only replays the part of the execution which may have caused the deployed application to crash.

First log entry of the log file is the starting point in replay phase. Due to our checkpoint technique the first log entry is always an event which starts an activity. For example first log entry is like *void SomeActivity.startActivity(Intent)*. It means that *SomeActivity* has started the activity which we want to replay. To know which activity is started by *SomeActivity* we have to check second entry of log file. Second entry is the argument of *startActivity()* method which is like *Intent { flg=0x24000000 cmp=SomeActivity/.AnotherActivity }*. From this entry we know that *SomeActivity* has started *AnotherActivity*. In this way all the events are replayed. In the meantime if any output is generated by application during replay phase then it should be matched with the output generated during capture phase. If both outputs are matching then we are heading in the right direction otherwise there is some problem in reproducing execution. GUI events are also replayed in similar manner except resources in GUI are identified using resource ID recorded during capture phase.

4. Preliminary Experimental Results

To assess the feasibility and effectiveness of our technique, we performed preliminary evaluation in an experimental environment. We used a proprietary android application named *KidsMusicLand* which has 3457 lines of code, 18 activities, and 21 classes as a test subject for our preliminary experiment. Figure 8 shows the layout of main activity of our test subject. The experiment was performed on Intel Core i3 3.10GHz processor, 4 GB RAM, Windows 7, Eclipse Indigo, Android 4.0.3, JDK 1.5, and AspectJ 1.6.12. In the viewpoints of accuracy and efficiency we investigated following two research questions

- **RQ1: Can our technique reproduce crash accurately?**

In our test subject there was not any real crash so we modified the application which caused the application to crash. We modified the application in three different ways which caused the application to crash by throwing *IndexOutOfBoundsException*, *IllegalArgumentException*, and *NullPointerException*. In all the cases we were able to reproduce the crash accurately by using single activity in our checkpoint technique. The sizes of the generated log files in three different cases were 2.45 KB, 6.41 KB, and 2.24 KB.

We were able to reproduce crashes by using single activity in our checkpoint technique. This result may be biased because we introduced the crashes in test subject. In applications with real crash it may require to use more than one activity in checkpoint technique to reproduce crash..

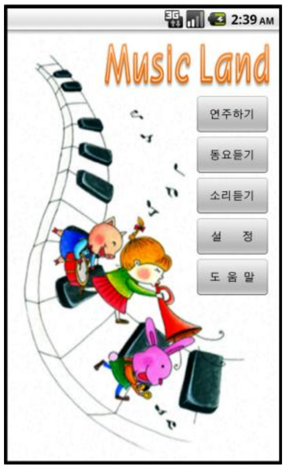


Figure 8. Layout of main activity of our test subject.

▪ RQ2: Can our technique capture executions efficiently?

To measure the efficiency we compared execution time of the original and instrumented version of the application. Measuring execution time of GUI application is difficult because it is affected by user interaction with the application. For this purpose we used debugging tool called Dalvik Debug Monitor Server (DDMS) which comes along with Android.

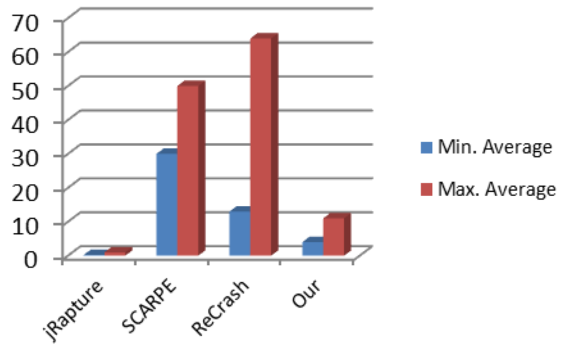


Figure 9. Comparison of average execution time overhead imposed by our and other related techniques.

On average our technique imposed 4% - 11% execution time overhead during capture phase. In few cases it went as high as 55%. It is worth mentioning that the execution overhead increases with increase in data intensive work done by application. During capture phase our technique store events in a list by converting the events into string data type. Of all the execution time overhead around 50% of the overhead is caused by data type conversion. Figure 9 shows the comparison of average execution time overhead imposed by our and other related techniques.

5. Conclusion and Future Work

In this paper we presented a capture and replay technique to reproduce crash in android application. Our technique records the partial execution of deployed application during capture phase and replays the recorded execution in replay phase for debugging. Preliminary experimental results show that the technique can be implemented in deployed applications for reproducing crash. Our approach is simple and easy to implement.

In future we intend to perform experiment on additional applications with real crash. Our technique imposes 4% - 11% execution time overhead which is still high for deployed applications so we intend to improve the performance. At present our technique does not reproduce the failure caused by concurrency related events so we will extend our technique in this direction.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2010-0010606), and the IT R&D program of MKE/KEIT.[10041145, Self-Organized Software-platform (SOS) for welfare devices]

References

[1] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, jRapture: A Capture/Replay Tool for Observation-Based Testing, *Proc. of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000, 158–167.
[2] Shay Artzi, Sunghun Kim, Michael D. Ernst, ReCrash: Making Software Failures Reproducible by Preserving Object States, *Proc. of the 22nd European conference on Object-Oriented Programming*, 2008, 542-565.
[3] S. Joshi and A. Orso, SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions, *ICSM 2007*, 2007, 234-243.
[4] J. Clause and A. Orso, A Technique for Enabling and Supporting Debugging of Field Failures, *Proc. of the 29th International Conference on Software Engineering*, 2007, 261-270.
[5] S. Narayanasamy, G. Pokam, B. Calder, BugNet: Continuously Recording Program Execution for

Deterministic Replay Debugging, *Proc. of the 32nd annual International Symposium on Computer Architecture,* 2005, 284-295.

[6] M. Roneee and K. De Bosschere, RecPlay: A Fully Integrated Practical Record/Replay System, *ACM Transactions on Computer Systems, 17(2),* May 1999, 133-152.

[7] T. J. LeBlanc and J. M. Mellor-Crummey, Debugging parallel programs with Instant Replay, *IEEE Transactions on Computers, C-36(4),* April 1987, 471–482.

[8] R. Konuru, H. Srinivasan, and J.-D. Choi, Deterministic replay of distributed Java applications, *Proc. of the 14th IEEE International Symposium on Parallel & Distributed Processing,* May 2000, 219–228.

[9] J.-D. Choi, B. Alpern, T. Ngo, and M. Sridharan, A perturbation free replay platform for cross-optimized multithreaded applications, *Proc. of the 15th International Symposium on Parallel and Distributed Processing,* April 2001.

[10] J.-D. Choi and H. Srinivasan, Deterministic replay of Java multithreaded applications, *ACM Sigmetrics Symposium on Parallel and Distributed Tools,* August 1998, 48–59.

[11] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: A portable record/replay environment for multi-threaded Java applications, *Software: Practice and Experience, 34(6),* May 2004, 523-547.

[12] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, K. De Bosschere. A Taxonomy of Execution Replay Systems, *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet,* 2003.

[13] Android Developers, http://developer.android.com.

[14] S. M. Srinivasan, S. Kandula, C. R. Andrews and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging, *Proc. of the annual conference on USENIX Annual Technical Conference,* 2004, 3-3.

[15] Marcello Cinque. Enabling On-Line Dependability Assessment of Android Smart Phones. *Proc. of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops,* 2011, 286-291.

[16] Introduction to AspectJ. http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html.

[17] Russell Miles. AspectJ Cookbook. Dec 2004.