VALUE-DETERMINISTIC SEARCH-BASED REPLAY FOR ANDROID MULTITHREADED APPLICATIONS

Ajay Kumar Jha, Sooyong Jeong, and Woo Jin Lee
School of Computer Science and Engineering, Kyungpook National University
1370, Sankyuk-dong, Buk-gu, Daegu, Republic of Korea
ajaykjha123@yahoo.com, kyo1363@naver.com, woojin@knu.ac.kr

ABSTRACT
With the advancement of programming technique like multithreading added with highly efficient memory model design, it is becoming very difficult to understand and analyze the execution behavior of a program. Due to non-determinism in execution behavior introduced by concurrency-related events the program may behave differently than expected which may cause the program to crash. To pinpoint the cause of crash, the execution which caused the crash must be reproduced. Our technique solves this problem by recording the concurrency-related events during program execution and reproducing those events during replay. For this purpose, our technique records thread id and value of the shared variables accessed during program execution while during replay it searches thread space to generate the same value of shared variable which it observed while recording.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids, Monitors, Tracing*.

General Terms
Design, Reliability, Experimentation.

Keywords
Reproducing crash, multithreaded programming, field failure, shared memory, Android.

1. INTRODUCTION
Writing concurrent program is difficult and debugging it is if not highly then equally difficult. Due to sequential thinking of most of the programmers, concurrency-related bugs are common in almost all real-world applications including Android applications. To fix these bugs, reproducing program executions in which those bugs were manifested is highly important. Unfortunately reproducing concurrency-related events in program execution is quite challenging due to non-determinism. Also, Android applications are rich in GUI and programmers are specifically advised not to perform all tasks on the main or UI thread which results into highly multithreaded applications so concurrency-related bugs are common in Android applications.

Debugging is all about reproducing the execution and pin-pointing the bug. Cyclic debugging is still very popular among programmers 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. However in multithreaded applications, different executions with same input may produce different output which is mainly caused by unsynchronized access to shared memory that eventually causes the race condition during program execution. Debugging race condition is highly complex task due to possibility of different threads accessing the shared memory during each execution.

One way to solve this non-deterministic problem in debugging is to make those non-deterministic events deterministic by reproducing the execution and this can be achieved by recording the program execution and with the help of those recorded execution guide a faithful re-execution during replay. 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 purpose then moderate overhead can be acceptable but if this technique is used for field failure 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 applications. These overheads are mainly caused by recording huge volume of data during program execution 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, 18] 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. Another approach for capture and replay technique is order-based [6, 7, 11, 19] in which only order of execution events are recorded and based on those ordered events, execution is reproduced in replay phase. Order-based approach is more efficient but it has also drawback since slight change in

RACS'13, October 1–4, 2013, Montreal, QC, Canada.

input or environment during replay phase can diverge the execution path.

In this paper we present a value-deterministic search-based replay, a content-based technique that reliably reproduces crash in Android applications with acceptable level of execution and memory overhead. Our main goal is to reproduce concurrency-related bugs in Android applications, manifested as crashes, through a record and replay technique for production usage. Our technique records the value of shared variables accessed during program execution while during replay it searches thread space to generate the same value of shared variable which it observed during original program execution. The key observation behind our technique is that the reproduction of same value of shared variable is sufficient to reproduce the crash even if the thread access order differs from the original run.

The rest of the paper is organized as follows. Section 2 introduces the background on android applications and overview on existing techniques. Section 3 presents the detailed procedure of our record 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. 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 in the background. 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).

- **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.

- **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.

2.2 Related Work

Till now large numbers of researches have been performed on reproducing concurrency-related events deterministically. Some of the state-of-art techniques are as follows:

Content-based technique jRapture [1] 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 [12]. ReCrash [2] maintains in memory a shadow of the call stack with copies of the receiver and arguments to each method during program execution. The copies of the receiver objects refer to the original objects on the heap. When the program fails or crashes, ReCrash serializes the shadow stack contents, including all heap objects referred to from the shadow stack. Rather than replaying, ReCrash generates candidate tests. Since ReCrash does not record thread interleaving, it may not reproduce concurrency-related failures. SCARPE [3] 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. ODR [18] is an output deterministic replay system which ensures that the replay run outputs the same values as the original run and it does so by searching the different execution path of the program guided by different sets of events which it records during original run. The technique enables ODR to reproduce data race.

Order-based technique CLAP [19] records the thread access order local to a particular shared program elements instead of global order and by doing so it reduces the execution overhead. The CLAP replayer works by controlling the scheduling of threads to enforce a crash replay under the guidance of thread ids and shared program elements ids which it records during original run. RecPlay [6] records the events only at synchronization level. It is a weak record/replay system because it can only correctly replay programs that are free of data race. However, it automatically detects data race when it occurs and stop the execution. Instant Replay [7] records the version number of each shared object read by a process during original run and during replay it recreates the proper input values for that process. It requires that the operations on each shared object have a valid serialization and for this it uses concurrent-read-exclusive-write (CREW) protocol. DejaVu [10] is a Java based record/replay technique for uniprocessor system. Rather than recording physical thread schedule, it records the logical thread schedule order at each critical event, such as synchronization events and shared variable accesses, in order to reproduce the exact same execution behavior of the program.

3. CRASH REPRODUCTION

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.

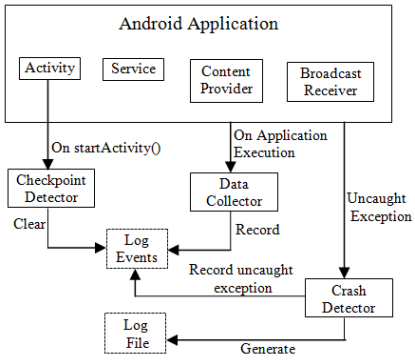

Figure 1 Overall structure of our system

3.1 Instrumentation Phase

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, 19], modifying API [1] or virtual machine [8, 9], and making changes in host operating system [5, 14, 21]. 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 which is an array list implementation in our system. 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 programmer for replaying the execution.

3.2.1 Data Collector

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 triggered 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. In this paper we are not going to discuss further about recording and reproducing these events since our previous work [20] describes this in detail.

For reproducing concurrency-related events our technique records two types of events, thread id and value of shared variables accessed during program execution. Recording the value of shared variable is straight forward that is whenever the value of shared variable is accessed (read/write) during program execution the probe in the code records that value in the list. For recording thread id, the technique first assigns a unique thread id to each thread whenever the probe in the code intercepts thread's start method during program execution. To reduce the execution overhead our technique does not record every instance of thread id when it accesses shared variable during original run instead it records only once instance per thread within a single activity component of android application and rearranges the thread access order with respect to shared variables during replay phase by using search-based technique. Recording only one instance of thread id also reduces thread search space during replay phase.

DataRace.java | Trace.aj

```
1  package com.race.ex;
2
3  public class DataRace extends Thread {
4
5      private static volatile int count = 0; // shared memory
6
7      public void run() {
8          int x = count;
9          count = x + 1;
10     }
11
12     public static void main(String args[]) {
13         Thread t1 = new DataRace();
14         Thread t2 = new DataRace();
15         t1.start();
16         t2.start();
17     }
18 }
19
```

Figure 2 Example code containing data race bug

Figure 2 shows an example in Java programming language which contains data race bug. We chose this example to illustrate our technique because Android applications are also written in Java. The example code has two threads *t1* and *t2* which access unprotected shared variable named *count*. Let's suppose that the example code is executed within a single activity component of Android application.

During capture phase that is during original run, the probe in the example code intercepts thread's start method and it assigns unique thread id 1 and 2 to thread *t1* and *t2* respectively. At this execution point the thread id is not recorded but only assigned a unique id. Thread id is recorded only when it accesses a shared variable. Figure 3 shows the different instances of example code executions with thread id in the first column and the value of shared variable accessed by thread id in the second column in each three different instances.

main :0
1 :0
1 :1
2 :1
2 :2

(a) Normal execution

main :0
1 :0
2 :0
1 :1
2 :1

(b) Race execution

main :0
1 :0
1 :1
2 :0
2 :1

(c) Probe effect

Figure 3 Three different instances of example code execution

Let's go through execution of figure 3 (a). First *main* thread accesses value 0 so our technique records name of the thread that is *main* and value 0 of shared variable. For *main* thread the technique does not assign a unique id so it records name of the thread but for other threads it records thread id. Now thread 1 accesses value 0 so the technique records thread id and value. Next, thread 1 accesses value 1. As this is the second instance for thread 1 accessing shared variable within a single activity so we do not record thread id again. In this case we only record value of shared variable. When the execution terminates, the log file looks like figure 4.

Thread ID: main, 1, 2
Value: 0, 0, 1, 1, 2

Figure 4 Log file for normal execution of example code

3.2.2 Checkpoint Detector

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. For more details on our checkpoint technique interested reader can refer to [20].

For reproducing concurrency-related events, the technique does not remove all the stored events from the list when an activity is started. It keeps two kinds of events that is name of activities and thread ids which started within those activities. Suppose, original run executed three activities A1, A2, and A3 before it crashed and during that execution, suppose four threads T1, T2, T3, and T4 are started and accessed then after implementing checkpoint technique the log file will look like figure 5.

A1, T1, A2, T2, A3, T3, T4
0, 0, 0, 1, 1

Figure 5 Log file after checkpoint technique

First row in figure 5 shows the name of the activities and thread ids accessed during complete execution of original run before crash while the second row shows the value of the shared variables accessed only during the execution of activity A3 before crash. The values of shared variables accessed during execution of activities A1 and A2 are removed by checkpoint technique. Our checkpoint technique can keep the values of shared variables for more than one activity which may increase the accuracy of reproducing crash but it also increases the space overhead.

Keeping the name of the activities and thread ids assures two important things during replay. First, activity name assures that the same execution (activity call sequence) path is followed and second it assures that the same thread is assigned the same unique id as it observed during capture phase.

3.2.3 Crash Detector

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 probe in the code intercepts any unhandled exception thrown by application during original run.

The major events of Data Collector, Checkpoint Detector, and Crash Detector components are shown in figure 6. The Android application 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.

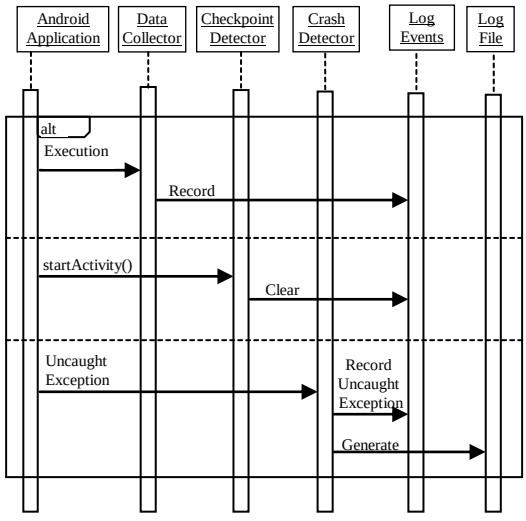

Figure 6 Main scenarios of our capture technique

3.3 Replay Phase

During replay phase, the crash is reproduced by using value-deterministic search-based replay. The technique reproduces the same value of shared variable which it observed during capture phase by searching thread space because it's the value of variables which influence the execution downstream to reproduce the crash.

For this purpose, it uses the original version of the application which was deployed in the field and log file generated during the original run. The technique first introduces probe to assign unique thread id to each thread whenever its start method is called. This is done to identify threads during replay. After instrumentation, the replay technique searches for thread space to generate the same values which are recorded in log file by controlling the scheduling of threads using semaphore.

For race execution shown in figure 3 (b), the capture phase generates log file as shown in figure 7. The replay starts with main thread so when the main thread reaches at line 5 in example code shown in figure 2 it generates value 0 which matches with the first value in log file. Since the value has been matched the execution proceeds further during replay. Now the execution reaches at *run()* method where either threads 1 or 2 can execute the code. We can replay execution either with thread 1 or thread 2. Let's start with thread 1 which generates value 0 at line 8 which is also matching with the second value log file. Next, thread 1 generates value 1 at line 9 which is not matching with the third value stored in log file so the replay technique backtracks the execution to the last matching point which is at line 8 and then it suspends thread 1. After suspending thread 1, replay starts with thread 2 which generates value 0 at line 8 which matches with the third value of log file so the execution proceeds further with thread 2. At line 9, thread 2 generates value 1 which again matches with the fourth value of the log file. Since there is no more execution code remaining for thread 2, it is suspended and thread 1 is resumed which generates value 1 at line 9. As there are no more values remaining in the log file, the replay system terminates.

Thread ID: main, 1, 2
Value: 0, 0, 0, 1, 1

Figure 7 Log file for race execution of example code

As illustrated, either thread 1 or thread 2 can start the replay within the *run()* method so our replay technique provide some relaxation in thread access order. In either case, our replay technique reproduces the execution containing data race which is our prime objective.

4. EXPERIMENTAL RESULTS

To assess the performance of our technique, we conducted preliminary evaluation in an experimental environment. We used an Android application named *KidsMusicLand* which has 3457 lines of code, 18 activities, and 21 classes as a test subject for our preliminary experiment. We chose this test subject because we wanted to know the additional execution overhead caused by concurrency-related events with reference to our previous work [20] which addresses non-determinism caused by input and environment condition. Figure 8 shows the layout of main activity of our test subject.

Figure 8 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 Juno, Android 4.1.2, JDK 1.5, and AspectJ 1.7.2. To measure the efficiency we compared execution time of the original and instrumented version of the application. For this purpose we used debugging tool named Dalvik Debug Monitor Server (DDMS) which comes along with Android. For recording execution time, we executed different activities of the application in sequence then we measured the execution overhead caused by each activity which is shown in table 1. First column represents activity name while second and third column represents execution overhead caused by our existing technique [20] and current technique respectively. The execution overhead of current technique also includes the execution overhead of existing technique.

Table 1 Execution overhead comparison

Activity Name	Overhead 1 (%)	Overhead 2 (%)
KidsMusicLand	55	55.7
PlaySong	5.4	5.4
HelpMakeSong	4.4	4.4
TransportSound	4.4	4.4
AnimalSound	11	11
MakeSong	41	42.1
BirdSound	7	7
ChildMusic	4.6	4.8

The data in table clearly shows no significant difference in execution overhead between our existing and current techniques which is due to the fact that in our current technique the only additional events which are performed during capture phase are assignment and recording of unique thread id. We would like to mention here that our test subject is not highly multithread and rich in shared variables. In our technique, the execution overhead is directly dependent on number of threads accessing shared variables and most importantly on the number of access of shared variables during program execution.

5. CONCLUSION AND FUTURE WORK

In this paper we presented a capture and replay technique to reproduce crash caused by concurrency-related events in android application. Our technique records the partial execution of deployed application during capture phase and re-executes the application during replay with the help of log file for reproducing crash. Preliminary experimental results show that the technique can be implemented in deployed applications for reproducing crash. Our approach is simple and easy to implement.

Android application fails in four different ways [15] namely: freeze, self-reboot, crash, and hang. Currently our technique reproduces the failure caused by crash only. For recording events during original run, the technique introduces probe in the source code which may cause unexpected sequence of events as shown in figure 3 (c). The figure shows that after *thread 1* writes value 1 to shared variable *count*, *thread 2* reads value 0 of the same shared variable *count* which is not possible since the shared variable is declared as *volatile*. This type of probe effect can be exposed by using global counter as shown in figure 9. The first column in the figure represents global counter. The global counter simply increments the count by 1 each time a shared variable gets

accessed during original run. Figure 9 clearly shows that the read event of *thread 2* executed before write event of *thread 1* but it has been recorded in reverse order in log file.

```
1 :main :0
2 :1 :0
4 :1 :1
3 :2 :0
5 :2 :1
```

Figure 9 Execution instance containing probe effect with global counter

In future we intend to perform experiment on additional applications with real crash. Execution overhead caused by our technique is not perfect for production use so will optimize the technique to reduce execution overhead. We will improvise the technique to reproduce other kinds of failures like freeze and hang. Currently our technique does not record global counter to expose the probe effect due to the high overhead caused by recording this event so finally we will come up with technique to minimize the probe effect.

6. ACKNOWLEDGMENTS

This work was supported by the IT R&D program of MISP (Ministry of Science, ICT & Future Planning)/KEIT. [10041145, Self-Organized Software platform (SoSp) for Welfare Devices] and the MSIP, Korea, under the C-ITRC (Convergence Information Technology Research Center) support program (NIPA-2013-H0401-13-1005) supervised by the NIPA (National IT Industry Promotion Agency.)

7. REFERENCES

[1] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. 2000. jRapture: A Capture/Replay Tool for Observation-Based Testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 158-167.

[2] Shay Artzi, Sunghun Kim, Michael D. Ernst. 2008. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, 542-565.

[3] S. Joshi and A. Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. *ICSM 2007*, 234-243.

[4] J. Clause and A. Orso. 2007. A Technique for Enabling and Supporting Debugging of Field Failures. *In Proceedings of the 29th International Conference on Software Engineering*, 261-270.

[5] S. Narayanasamy, G. Pokam, B. Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *In Proceedings of the 32nd annual International Symposium on Computer Architecture*, 284-295.

[6] M. Roneee and K. De Bosschere. 1999. 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. 1987. 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. 2000. Deterministic replay of distributed Java applications. In *Proceedings of the 14th IEEE International Symposium on Parallel & Distributed Processing*, 219–228.

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

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

[11] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. 2004. 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. 2003. A Taxonomy of Execution Replay Systems. *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*.

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

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

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

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

[17] Russell Miles. 2004. AspectJ Cookbook.

[18] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 193-206.

[19] Jeff Huang, Peng Liu, Charles Zhang, Sunghun Kim. *CLAP: Concurrent Lightweight Crash Reproduction*. Technical Report. The HongKong University of Science and Technology.

[20] Ajay K. Jha and Woo J. Lee. 2013. Capture and Replay Technique for Reproducing Crash in Android Applications. In *Proceedings of the 12th IASTED International Conference in Software Engineering*, 783-790.

[21] Min Xu, Rastislav Bodik, and Mark D. Hill. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, 122-135.