Replay Debugger For Multi Threaded Android Applications

By

Rohit Girme

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved February 2011 by the
Graduate Supervisory Committee:

Yann-Hang Lee, Chair
Baoxin Li
Karamvir Chatha

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

Debugging is a hard task. Debugging multi-threaded applications with their inherit non-determinism is all the more difficult. Non-determinism of any kind adds to the difficulty of cyclic debugging. In Android applications which are written in Java, threads and concurrency constructs introduce non-determinism to the program execution. Even with the same input, consecutive runs may not be the same and reproducing the same bug is a challenging task. This makes it difficult to understand and analyze the execution behavior or to understand the source of a failing execution.

This thesis introduces a replay mechanism for Android applications written in Java and is based on the Lamport Clock. This tool provides the user with a controlled debugging environment, where the program execution follows the identical partially ordered happened-before dependency among threads, as during the recorded execution. In this, certain significant events like thread creation, synchronization etc. are recorded during run-time. They can later be replayed off-line, as many times as needed to pinpoint and fix an error in the application. It is software based approach and has been implemented by modifying the Dalvik Virtual Machine in the Android platform. The method of replay described in this thesis is independent of the underlying operating system scheduler.

To My Beloved Family

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor; Dr. Yann-Hang Lee, without whose guidance, encouragement and support, this thesis would not have been possible. I have gained a lot of knowledge about my field, improved my research skills and otherwise, working under him. In all it has been a very satisfying and a fulfilling experience.

I would also like to thank Dr. Karamvir Chatha and Dr. Baoxin Li for their time and effort to help me fulfill my degree requirements; as part of my thesis committee.

I am grateful to all my friends at Arizona State University who made me feel at home and making this stay an enjoyable experience. I am thankful to all my lab mates, especially Young Song and Preetham who helped me with my queries.

Finally, I am indebted to my parents and my family. They all have constantly encouraged me and been understanding through all the tough times. I wish to express my utmost and deepest gratitude to them for being patient, warm and supportive.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 Motivation 2

1.2 Document Outline 5

2 BACKGROUND 7

2.1 Multi-Threaded Applications 7

2.2 Non Deterministic Order of Execution 7

2.3 Synchronization Race 9

2.4 Deterministic Replay 10

2.5 Lamport Clock 11

2.6 Android Architecture 12

2.7 Android Application Architecture 18

2.7.1 Processes and Threads 19

2.7.2 Processes and Lifecycles 19

2.7.3 Threading Model 20

2.7.4 Inter Process Communication 21

2.7.5 Lifecycle of Application 23

3 RELATED WORK 25

3.1 General Replay Techniques 25

CHAPTER Page

3.2 DejaVu 27

3.3 JaRec 28

3.4 JReplay 28

4 DESIGN OF REPLAY DEBUGGER 30

4.1 Record Replay in Java 31

4.2 Definitions 33

4.3 Record 34

4.4 Replay 36

4.5 Recording Events 38

4.5.1 Synchronization 38

4.5.2 Wait/Notify 40

4.5.3 Thread Creation 42

4.5.4 Message Passing between Threads 43

4.5.5 External Events 44

5 IMPLEMENTATION 47

5.1 Initialization 48

5.2 Record 50

5.2.1 Synchronization in Java 50

5.2.2 Message Passing 52

5.2.3 Thread Creation 53

5.2.4 Wait/Notify 55

5.3 Replay 57

CHAPTER Page

5.4 Lifecycle of Application 59

5.5 Log Structure 59

6 COMPARISON AND CONCLUSION 61

7 FUTURE WORK 64

REFERENCES 65

LIST OF TABLES

Table Page

1. Various States of Android Application 23

LIST OF FIGURES

Figure Page

1. Data Race in a Multi Threaded Program 8
2. Synchronization Race in a Multi Threaded Program 10
3. A Happened Before Relation between Events 12
4. Major Components of Android 13
5. Application Layer of Android 13
6. Application Framework of Android 14
7. Libraries Layer of Android 15
8. Layers of Android Application 16
9. Android Runtime Layer 17
10. Linux Kernel in Android 17
11. Binder in Action 22
12. Record Process Overview 30
13. Replay Process Overview 31
14. Identify Ordering of Events 35
15. Using Ordering of Events for Replay 37
16. Instrumentation of MONITOR ENTER 39
17. Instrumentation of MONITOR EXIT 40
18. Instrumentation of Wait/Notify 41
19. Incorrect Handling of Wait/Notify 42
20. Instrumentation of Thread Start 43
21. Instrumentation of Message Passing 44

Figure Page

22. Overview of Working of Android 46

23. Double Buffering while Recording Events 50

24. Synchronization in Android 52

25. Message Passing in Android 53

26. Thread Creation in Android 54

27. Wait/Notify in Android 56

CHAPTER 1

INTRODUCTION

Debugging has been an imperative step to ensure the correctness of software programs. Based on the data from a survey, in a 2002 NIST report [1], on an average a bug found in coding/unit testing takes 4.9 hours to fix, whereas an average bug found in post-product release takes 15.3 hours to fix. In addition, using the distribution of where bugs are found, we calculate that the overall average time to investigate and fix a bug is 17.4 hours. This is quite costly in terms of the engineer's time as well as the impact on the software products and users.

A very common debugging technique is cyclic debugging. Here the programmer re-executes a program several times to pinpoint an error that occurred. Paramount to this scheme is that the erroneous execution can be replayed at will. Unfortunately, this is not always the case. In single threaded applications, the flow of execution is sequential. When dealing with multi-threaded programs this property of repeatability is usually lost. In multi-threaded Android applications several threads run concurrently and may compete for work or to enter critical sections. The order of execution of these threads also depends on the underlying scheduling events, which may change from execution to execution of the same application. Also it may be hard (or impossible) to regenerate the input (e.g. a key press from the hard keyboard or a touch event from the screen of the Android device). Non-determinism may also be due to race conditions where multiple threads modify shared data outside a critical section.

This form of non-determinism is generally considered an error, or at least a bad programming practice. In this thesis, it is assumed that an execution is free of data races. This can be verified by one of the existing automatic data race detection systems such as [2].

This inherent non-determinism makes debugging of multi-threaded Android applications very hard. Even with same input to the application, the different thread execution order may disallow a bug that appeared in one execution instance of the program from appearing in another execution instance of the same program. There may also be deadlocks, starvation which are conventional issues with multithreaded programs. These should be taken care of during the development stage but are often forgotten about. Due to the hurdle of using cyclic debugging for such programs, engineers are forced to look into the trace data from a failed run and to understand thread dependency. This tactic is certainly problematic and time consuming.

1.1 Motivation

Recent years have seen rapid evolution of mobile phones as computation platforms (beyond the basic voice telephony), which could soon rival traditional desktop computing in many respects. This has shifted focus of major players in the area of general purpose desktop computing viz. Google, Microsoft, Apple, and so on towards mobile computing.

Such a product from Google is the Android mobile platform which is increasingly being seen on mobile devices. The importance and popularity of Android in current software systems is increasing day by day. This has

necessitated the development of advanced programming tools for writing efficient and correct Android applications. Many Android applications are multi-threaded Java programs which also receive external inputs like network events/messages, windowing events, keyboard events etc. Building tools for such applications is non-trivial because of non-determinism in Java and also the aforementioned external inputs.

"It's been proven that when you can simulate the system, you can reproduce the same behavior if you apply the same input," Genard noted (Based on a survey conducted by Virtutech Distributed at Embedded Systems Conference [3]). There are many basic tools available for debugging and profiling Android applications. Android Debug Bridge (ADB), Dalvik Debug Monitor Server (DDMS), Traceview and logcat [4] among others are some of the tools available. They have various characteristic uses like graphically setting up breakpoints in your code in your IDE, screen captures on the emulator, thread and stack information, and many other useful features. They have been integrated with the Eclipse IDE [5] or can be used stand alone**.** An important element missing is the ability to provide a deterministic replay of a non-deterministic execution instance.

Classical approach of single stepping cannot be used with Android applications as it would affect the temporal component of the system while interacting with the outside world. This is because; the system interacts with, and is sometimes dependent on external real time context. [6] gives a nice example of difficulty in keeping temporal component intact.

“Breaking the execution will only break the internal execution while the external process will continue”.

Thus even if we stop execution of our application, the outside world with all its evens is still running. With more & more pressure for time-to-market & with shorter product cycle, engineers are in fray with inherent non-deterministic nature of multithreaded applications & lack of control to replay the program.

The inability of classical approach to deal with the debugging of multi threaded real time software has encouraged several researchers to look into this problem. Numerous approaches have been developed and published to deal with inherent complexity of debugging such Java applications. Out of those mechanisms, one of the most well known approaches is deterministic replay or trace based replay. In this sufficient information is collected during one execution of a multi-threaded application to ensure that the concurrent behavior can be deterministically reproduced. Cyclic debugging can then be used in the same manner as for sequential programs. Several solutions have been proposed, that capture and deterministically replay the execution of a Java multi-threaded application [7-9]. However none of them have talked about applying the deterministic replay methods to a multi threaded application in the context of the Android platform.

The proposed replay mechanism is obtained by instrumenting the Dalvik Virtual Machine [10]. It runs in two modes: (1) The record mode, wherein, the tool records the non-deterministic events made during the first program execution and then to impose this order during subsequent executions i.e. (2) replay mode.

During the record phase, the main goal is to record the behavior of the program with as little intrusion as possible. This ensures that a representative execution is recorded. During replay, the main goal is to do a faithful replay of the recorded execution. To provide an ordering to events that happened during record phase, Lamport Clock [11] is used. It describes a happened before relation that gives the partial ordering between different events of the same thread as well as with events in other threads. By recording synchronization events and applying this relation ordering to them the execution of the program can be reproduced offline. This can then be used to monitor the program for finding bugs. This approach is independent of the underlying Linux scheduler. Although described in the context of Java, this technique can be applied to multi threaded applications with similar synchronization primitives. Note that events in a single thread are implicitly ordered according to their temporal order during execution of the thread. The manner in which non-critical events are scheduled during a replay does not affect the execution behavior of the program.

1.2 Document Outline

The rest of the document is organized as follows.

Chapter 2 provides Background information about Android, its architecture, the Android application. It also explains some terms to better understand the document.

Chapter 3 talks about the Related Work done in the area of record replay in general. It also mentions certain works which describe the specific mechanism of record/replay in terms of Java programs.

Chapter 4 talks about the design and techniques of the replay debugger as a whole. It mentions the different events that we need to record and replay in the context of an Android application.

Chapter 5 describes the implementation details of the debugger. It gives insights on the working of critical events and how they are recorded.

Chapter 6 gives a brief comparison about this tool with other similar tools. It concludes the thesis as a whole.

Chapter 7 presents some features and enhancements that can be done to the existing implementation. This will further enhance the tool.

CHAPTER 2

BACKGROUND

2.1 Multi-Threaded Applications

These are applications which have more than one thread running concurrently. In a single threaded application the flow of execution is in order. On a single CPU system, threads are run by interleaving while on multiple CPU system all the threads run concurrently, depending on the number of CPU's. This concurrency is the subtle source of bugs and can lead to a several issues like race condition, deadlocks, starvation. The thread execution order is decided by the scheduler and some other factors. This introduces inherent non-determinism in multi-threaded applications. Multithreaded applications are hard to design and harder to debug due to the above reasons. Android applications can be multi-threaded using Java threads.

2.2 Non deterministic order of execution

In a multi-threaded application, the order of execution of threads depends on the scheduler and some other factors. There may also be a lot of inter thread communication. This order may differ each time the application is run. In a multi threaded application it is common that threads share data. If two threads access a shared data structure in a non-synchronized manner, where at least one of the threads changes the data structure, we say that a race condition occurs [12]. The outcome of the program is timing dependent and hence a source of non determinism. This is called a data race. Refer Figure 1 for an example of such a data race. In this figure, two threads, T1 and T2, are manipulating a common

object s, of class S, but there is no synchronization whatsoever between the threads. The execution speed of the two threads can vary greatly: the scheduler might decide to let one thread run and block the other. One thread may be running exclusively on a processor while the other thread needs to share its processor with threads from other processes. This timing difference can result in a different order of operations of the two threads. In Figure 1 this is illustrated. Here the threads perform the same operation on the fields *i* and *j* of object s. However after the two executions, the final result is different. The threads are said to be 'racing' to control the object. In this thesis it is assumed that the application is free of data races.

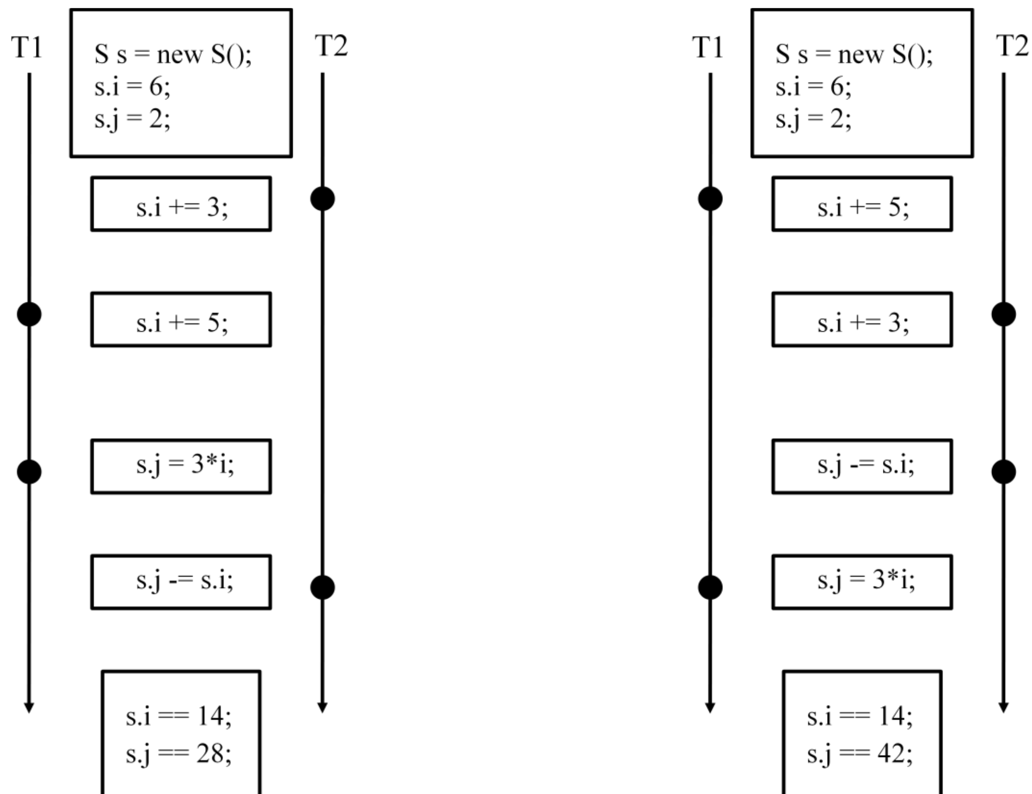

Figure 1: Data Race in a Multi Threaded Program.

2.3 Synchronization Race

In order to avoid such data races, sufficient synchronization must be added to control the accesses made by threads to shared data structures. Many synchronization constructs have been designed for this purpose such as monitors, semaphores, mutexes, condition variables, etc. However, the use of synchronization operations, introduces another type of race called a ‘synchronization race’. To illustrate this, Figure 2 shows an example of a synchronization race. Here, two threads, T1 and T2, use a lock to synchronize their actions. Only one thread can acquire the lock (indicated by an open bracket) at one point in time. Other threads have to wait until the lock is released (indicated by a closing bracket) by the thread holding it. Using the lock, the accesses to the shared data structures are now mutually exclusive or atomic. Still the execution is not deterministic. The threads T1 and T2 are now racing to obtain the lock. In the left execution T1 wins, while in the right execution T2 arrives and locks first. The result of these two executions is different and as a result the execution of the program depending on the value of *s* could be entirely different as well.

The non-determinism introduced by synchronization is mostly seen as a useful feature of a parallel program. Thus it is perfectly normal for these race conditions to occur. However they create a big problem while debugging such a program. The use of a debugger must not interfere with the program’s execution behavior. Replaying the synchronization operations is an effective solution to this problem.

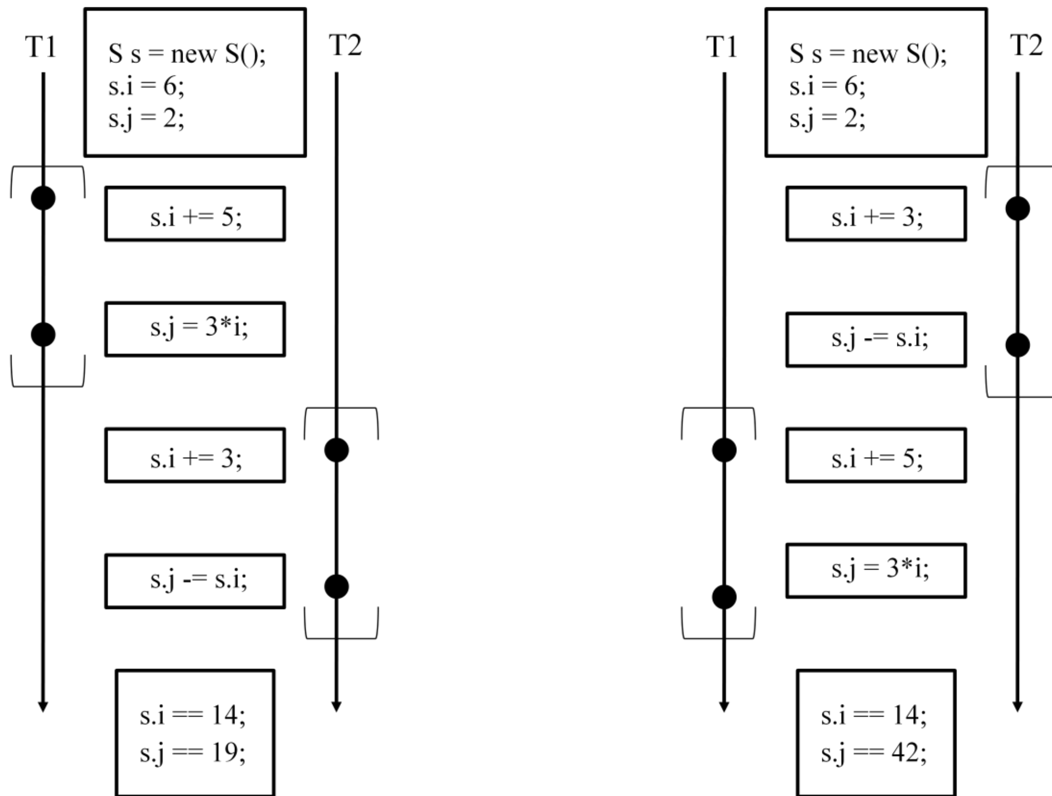

Figure 2: Synchronization Race in a Multi Threaded Program.

2.4 Deterministic Replay

Deterministic replay is a technique widely used for debugging non-deterministic applications. Such a technique operates in two phases: record and replay. The point of the initial record phase is to capture information about events in the monitored application and to record all the data required to reproduce it at some later stage. The application is then forced to execute in a similar way by using the captured information, during the subsequent replay phases. The main challenge (which is an active research topic) with such an approach is in determining what all events to capture and the quantity of information that should be recorded to ensure deterministic replay.

2.5 Lamport Clock

It is a simple method to define a partial order between events in a distributed system. A Lamport Clock is used for the same. It is a logical clock, which is actually a monotonically incrementing software counter maintained in each process with the following rules:

1. A process increments its counter (clock) before each event in that process.
2. When a process sends a message, it includes its counter value with the message.
3. At the receiving end, the process sets its counter to be one more than, the maxima of its own value and the received value.

By following the aforementioned rules, the "happened-before" relation → can be observed in the below situations:

1. If *a* and *b* are events in the same process, and *a* occurs before *b*, then $a \rightarrow b$ is true.
2. If *a* is the event of a message being sent by one process, and *b* is the event of the message being received by another process, then $a \rightarrow b$ is true.
3. If $a \rightarrow b$, $b \rightarrow c$ then $a \rightarrow c$

In the Figure 3, T1 and T2 are two threads in an application program. In thread T1, there is some critical section in the form of a synchronized block. This is synchronized on the object – objA. After some processing, the synchronized block ends. The start and end of the block is similar to acquiring a lock and then releasing it, in Java. Thread T2 has a block synchronized on the same object – objA. Assume that during a recording run, the synchronized block in thread T1

starts and ends before that in thread T2 starts. Then a happened before relationship is formed between end of synchronized block of thread T1 and start of the block in thread T2.

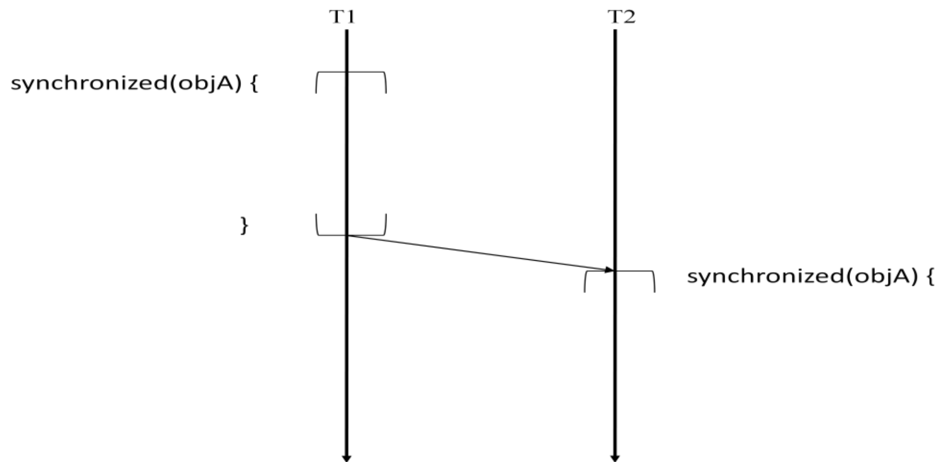

Figure 3: A Happened Before Relation between Events

For more information on the workings of the Lamport Clock, please refer [11].

2.6 Android Architecture

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. Figure 4 shows the major components of the Android operating system [13]. Each section is described in more detail below.

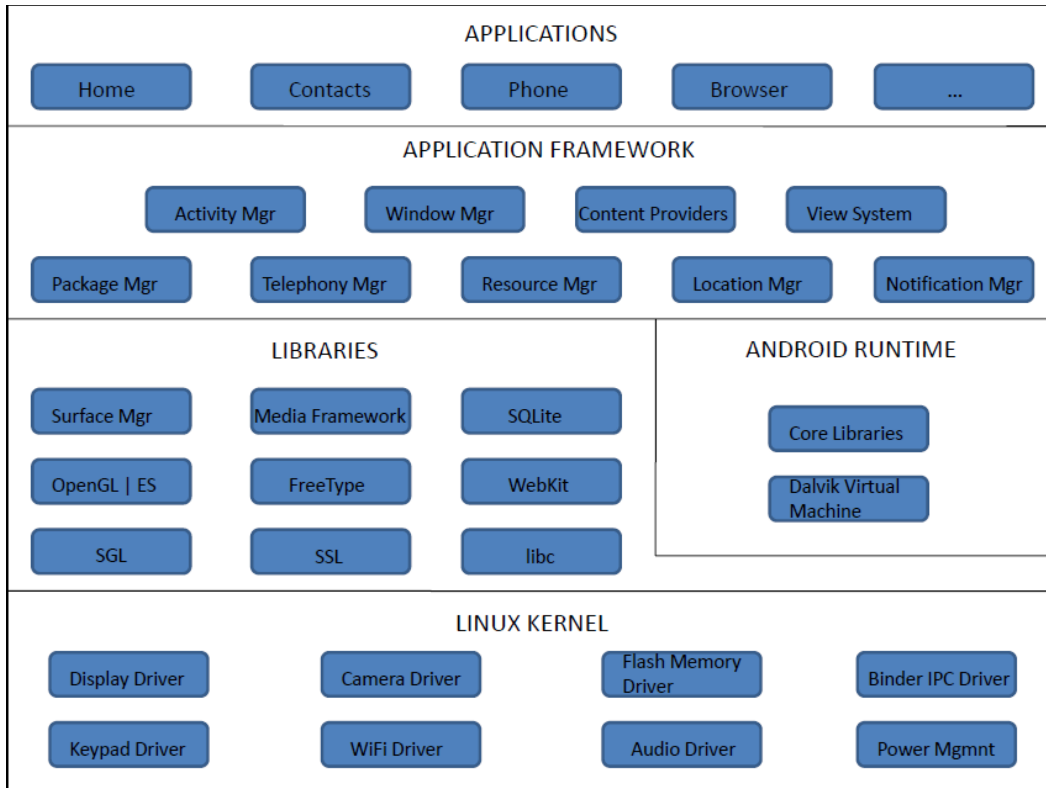

Figure 4: Major Components of Android

Applications

Android has included a basic set of applications (Figure 5) like the email client, calendar, maps, browser etc. All these applications are written in the Java programming language. These applications can be multi-threaded depending on their purpose. Other users/developers may add such applications according to their need. These are the applications that this tool will be used for.

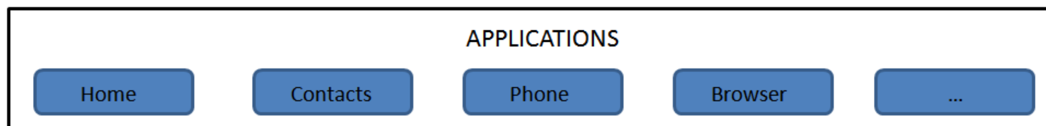

Figure 5: Application Layer of Android

Application Framework

This includes the programs that manage the phone's basic functions like resource allocation, voice applications, window allocation for applications, managing lifecycle of applications and keeping track of the phone's physical location. This layer is majorly written in the Java programming language. This layer is the main Java API used by developers in their applications. Application developers are allowed full access to Android's application framework (Figure 6). This allows them to take advantage of Android's processing capabilities and support features.

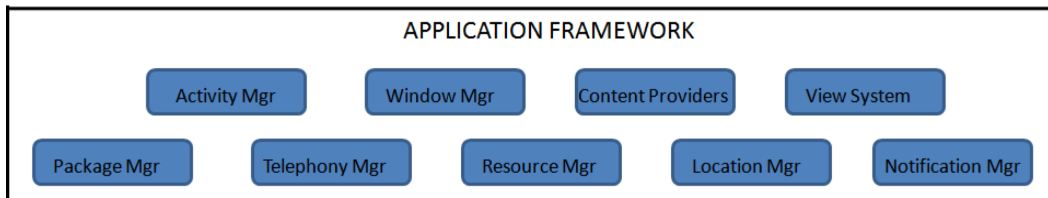

Figure 6: Application Framework of Android

Libraries

Android has a set of C/C++ libraries used within the Android system by many components. These are exposed to developers through the Android application framework. Most of the CPU intensive work of the framework is accomplished using native C/C++ libraries (Figure 7). Some of the core libraries are:

- System C library - a tuned implementation of the standard C system library (libc), for embedded Linux-based devices
- Media Libraries - these libraries support playback and recording of many popular audio and video formats and image files.

- 3D libraries - the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer
- SQLite - a powerful and lightweight relational database engine.

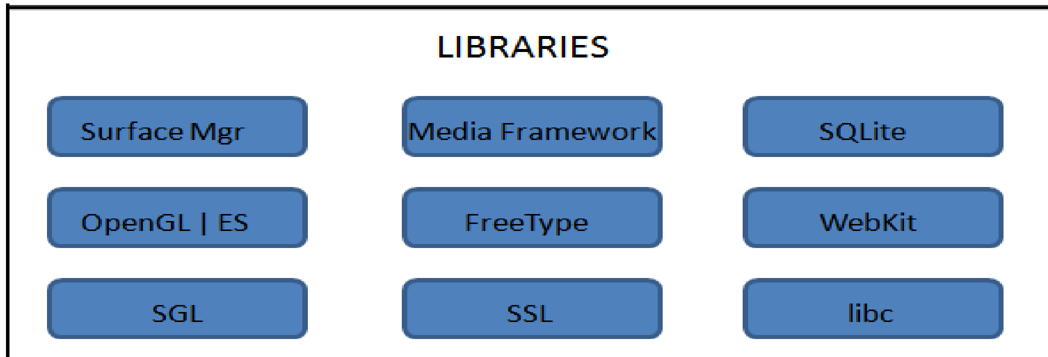

Figure 7: Libraries Layer of Android

These libraries have been optimized for embedded use: E.g., fast pthread implementation using 4-byte mutex rather than the 12-byte mutex (since there may not be as many total threads compared to a desktop environment).

Android Runtime

Android has a set of core libraries that provide most of the functionality available in the core libraries of the Java programming language. Each Java application needs an interpreter for the compiled Java byte code. Every Android application runs in its own process, along with its own instance of the Virtual Machine [14]. Android has its own Virtual Machine called Dalvik Virtual Machine. Refer Figure 8 below.

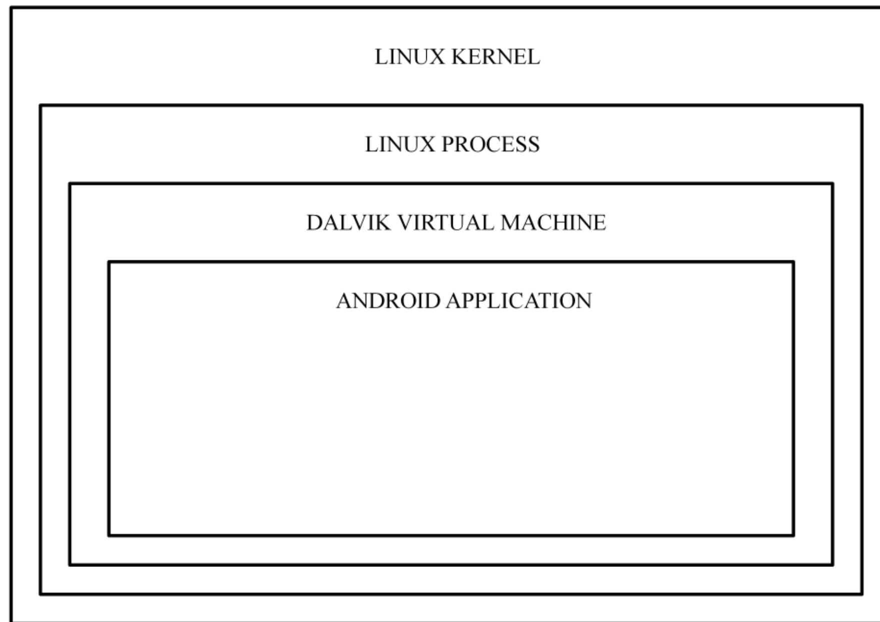

Figure 8: Layers of Android Application

Dalvik [15] primarily is a process virtual machine. Refer Figure 9 below. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format [16] which is optimized for minimal memory footprint. The .dex files are obtained from the .class files by the “dx” tool. The VM is register-based, and runs classes compiled by a Java language compiler. The Dalvik VM is written in C. The Dalvik VM relies on the Linux kernel for functionality such as threading and low-level memory management.

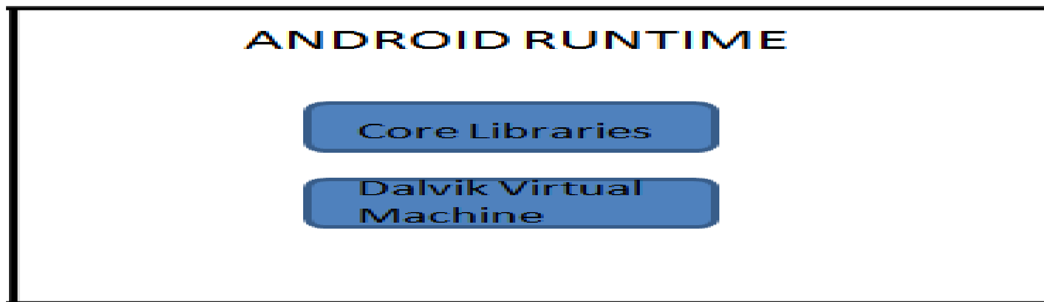

Figure 9: Android Runtime Layer

Linux Kernel

Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel acts as an abstraction layer between the hardware and the rest of the software stack. A set of kernel drivers is added as patch to Linux kernel to be able to meet some special requirements of Android as a whole. Refer Figure 10 below.

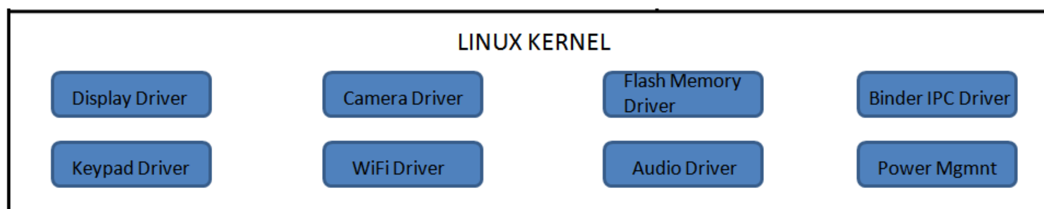

Figure 10: Linux Kernel in Android

Hardware Abstraction Layer

There is an abstraction layer present in between the Linux kernel and above layers. This enables certain core default system applications and services to be replaced by third party/ custom implementations. Most mobile OEMs have the basic drivers to control their audio, video etc. Android defines this hardware abstraction layer on top of kernel and standardizes Android's interface. This

hardware abstraction layer exists as a user-space C/C++ library. This probably means that Android implements a standard interface for Audio, irrespective of the technology supported by the underlying hardware, just asking implementations of features that it needs from the particular hardware.

2.7 Android Application Architecture

Android applications are written in the Java programming language. The compiled Java code, all the necessary data and resource files of the application are bundled by the "aapt" tool into an *Android package.* This file has a .apk suffix. This file can then be used for installing the application on devices. All the code in a single .apk file is considered to be one *application*.

Android application sand-box model

Android uses the process separation provided by Linux kernel as the primary means of achieving isolation against other suspicious applications. Each application runs in its own Linux process. Furthermore, each managed piece of code executes in a virtual machine (DVM). As a result each application is sand-boxed from the other applications running at any given time. All IPC is achieved via the mechanisms provided by Binder.

A second level of isolation builds upon the capability of underlying Linux to strongly isolate data/files of one user from the other. This is achieved by allocating a unique user-id to each installed application on a particular system. Android starts the process when any of the application's code needs to be executed, and shuts down the process when it's no longer needed and other applicaitions are in need of resources. Unlike applications on most other systems,

Android applications don't have a single entry point for everything in the application (no main() function, for example).

2.7.1 Processes and Threads

When the application needs to run, Android starts a Linux process for it with a single thread of execution. By default, all components of the application run in that process and thread. However, additional threads for any process can be spawned as and when required. All components are instantiated in the main thread of the specified process. Consequently, methods like View.onKeyDown() that handle external events like key press are always run by the main thread of the process. This means that no component should perform long or blocking operations when called by the system, since this will block other components in the process. Separate threads can be spawned for long operations. Android may decide to shut down a process at some point, when memory is low and required by other processes that are more immediately serving the user. A process is restarted when there's work for them to do. Android provides the developer the functionality to save the state of the application. If the functionality is implemented by user the state is recreated, otherwise a new instance is started.

Android decides which process to terminate based on a number of factors like the relative importance to the user. For example, it more readily shuts down a process that is not visible than a process which is visible.

2.7.2 Processes and Lifecycles

A **foreground process** is one that is required for what the user is currently doing. They are killed only when memory is so low that no one can continue to

run. At this point killing some foreground processes is necessary to keep the user interface responsive.

A **background process** is one holding an activity that is currently not visible to the user. These processes do not have any direct impact on the user interface. Hence they can be killed more readily to reclaim resources for other processes. There may be many background processes running, so they are kept in an LRU (least recently used) list. The process that was last seen by the user is more likely to be killed. If an activity implements the lifecycle methods correctly, then the current state will be saved before killing it.

Also Android has a concept of "Application Not Responding" (ANR). If an application does not respond within a specified time period to the system, Android may show an ANR dialog box, asking the user to Force Close the application.

2.7.3 Threading model

There may be times when an additional thread needs to be spawned to do some background work. Since the user interface must always be quick to respond to user actions, the main thread should not do time-consuming operations. If something cannot be completed in a short period of time, a different thread should be spawned for that purpose.

Threads are created in code using standard Java Thread objects. Android provides a number of convenience classes for managing threads. Looper for running a message loop within a thread. Handler for processing messages and HandlerThread for setting up a thread with a message loop. The Looper and the

Handler together are responsible for handling all the messages and data received by a thread. This code is run by the main thread itself. There is generally one message queue per main thread and hence per application.

2.7.4 Inter Process Communication

Android has a lightweight mechanism for Inter Process Communication(IPCs) — where a method is called locally, but executed remotely (in another process). This involves decomposing the method call and all its corresponding data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, and reassembling and reenacting the call there. This is done by the Binder [17] driver running in the kernel. Android provides the necessary code to do that work, so that user/developer can concentrate on defining and implementing the IPC itself.

If the call originates in one process and goes back to the same process it is executed in the caller thread. However, when the call originates in another process, an IPC in the real sense, the method is executed in a thread selected from a pool of threads maintained by Android in the same process. It's not executed in the main thread of the process. The message is then handed over to the main thread. Refer Figure 11 for an illustration of the above explanation.

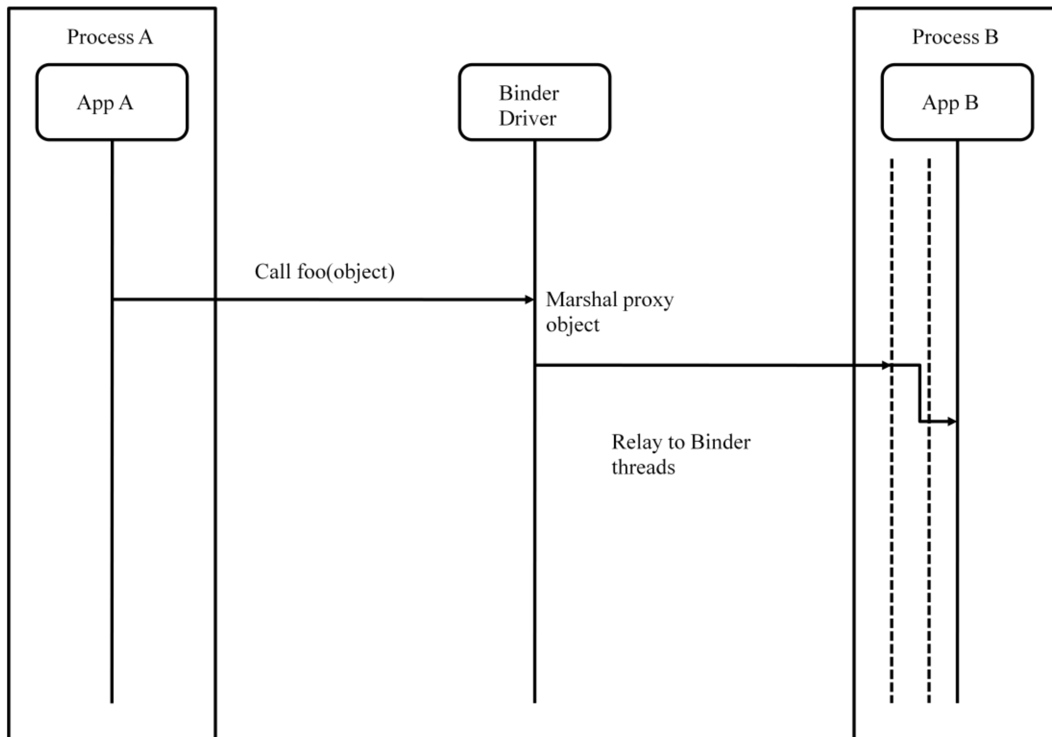

Figure 11: Binder in Action

In the Java API for Android, for using IPC in Android an interface called Parcelable has to be implemented. The IPC is then translated and its state is saved into something called a Parcel. A Parcel is essentially a C-Structure representation of the protocol understood by the Binder driver. Binder driver then forwards this parcel to the destination process. This driver is a character driver, and all forms of IPC used in Android user applications are routed through this driver module. It has been optimized for embedded use in that it does not use the general purpose Java serialization for marshalling objects. Instead a more simplified protocol that is sufficient for system level IPC is used.

2.7.5 Lifecycle of Application

Applications have a lifecycle. Lifecycle starts with the beginning when Android instantiates them through to an end when the instances are destroyed. The following Table 1 lists the lifecycle methods an application can implement. They symbolize the various states an application may go through.

Table 1

Various states of Android application

Method	**Description**
onCreate()	Called at start of application. Must be implemented by user. The application is in foreground.
onStart()	Called when activity becomes visible.
onResume()	Called before user interaction with the application. This is the current application with user input going to it.
onPause()	Intermediate state. Called when another application is going to be started.
onStop()	Called when application is not visible to user. Application goes to background.
onDestroy()	Called when application is going to be destroyed.

onRestart()	Called when application comes from background to foreground.

The **entire lifetime** happens between the first call to onCreate() through to a single final call to onDestroy(). Activity Manager Service [18] is a Java service that manages all aspects of the application lifecycle, and sits on top of the window manager to tell it what to do with the windows coming from various applications.

CHAPTER 3

RELATED WORK

This chapter investigates the research that has been done on execution record/replay frameworks. It also reviews existing projects relevant to this thesis. Based on the discussion before, it is clear that cyclic debugging in the case of multithreaded application is difficult. This is because the order of execution may be different for subsequent executions. For the record/replay approach to solve such an issue, the important thing is how much information should be traced in the record phase. On the one hand, enough information should be generated about the execution so that a faithful re-execution of the program is possible. On the other hand, the amount of information traced and the computation required should be minimized as much as possible in time and in space in order to avoid any probe effect [19].

3.1 General Replay techniques

On the general topic of replay, some researchers have been relying on special hardware to reproduce the program behavior. A noninterference monitoring architecture has been developed which collects data about the execution of a real-time program without affecting its execution. A replay mechanism has been designed to control the reproduction of the program behavior. In addition it examines the states of the target system and its behavior [20]. The software instruction counter approach [21] records the backward branches and is used to identify the exact location of the required event occurs (e.g. interrupt). Another technique relies on using a software counter which is

used to compute the number of instructions executed between nondeterministic events during a normal execution [22]. If a failure or bug occurs, the computed instruction counts are used to force the replay of these events at the same execution points. The execution of the application can thus be replayed to recreate the pre-failure state. Pan and Linton [23] record and replay the values of shared objects whenever anyone accesses them. LeBlanc and Mellor-Crummey described a system that records the order of accesses based on version counters in Instant Replay [24]. In this, the relative order of significant events is saved as they occur, not the data associated with such events. Tai et. al. [25] use a source-to-source transformation of an Ada program to replay a sequence of synchronization events. It is a language-based approach to deterministic replay of concurrent Ada programs. The approach is to define synchronization sequences of a concurrent Ada program in terms of Ada language constructs and to replay such sequences without the need for system-dependent debugging tools. Russinovich and Cogswell [26] block all threads except one to replay the program. In this way they reconstruct the scheduling of an execution on a uniprocessor on the Mach operating system. They use a software instruction counter [21], i.e., the number of backward control transfers and the current instruction pointer to decide when a thread switch should happen.

The most similar work to this thesis is the Replay Debugger [27]. The debugger considers a multithreaded execution as a partially ordered sequence of interactions and reproduces the sequence in cyclic debugging process. The record framework is a wrapper for IO and IPC (inter-process communication) library

calls. This technique is more based on immediate happened before relations, than actual Lamport Clock values. Moreover, to make the Replay Debugger more practical, the proposed replay mechanism has been incorporated into GDB with an extended debugging interface for thread control and observation.

The following sections describe techniques specifically for the Java programming language.

3.2 DejaVu

In the specific area of record/replay of Java programs, the first solution to achieving deterministic replay of executions of multithreaded Java programs was proposed by Choi and Srinivasan [28]. This paper introduces a notion of a logical thread schedule, which is based on counting the number of critical events occurring between thread switch. Critical events in this context are all synchronization events performed by threads, for example monitorenter and monitorexit and shared variable accesses. The approach to capturing the logical thread schedule is based on using global clocks. This clock ticks at every execution of a critical event to uniquely identify each critical event. A local clock per thread is used to allow each thread to identify schedule intervals that belong to that thread. Their implementation is based on a modified Java virtual machine. The approach has been extended to include record/replay of networking events in distributed applications [29]. The common thing between this paper and this thesis is the modification of the Virtual Machine for record and replay.

3.3 JaRec

JaRec [8] aims at portable replay for multi-threaded Java applications. JaRec uses bytecode instrumentation to capture and replay schedules based on Lamport clocks for objects. It assumes a data-race free program and, therefore, only needs to instrument the synchronization operations. This technique has originally been used by the same group to implement RecPlay [30], a tool that combines record/replay and data race detection for Solaris. Instrumentation of the bytecode is performed on the fly by the virtual machine. This enables dynamic loading and replaying of classes over a network and prevents having several versions of a class. If present, JaRec uses the JVMPI [31] for instrumentation.

3.4 JReplay

JReplay [9] tries to achieve similar functionality to that of DejaVu. It is a deterministic replay of multi-threaded applications which forces the application to execute according to a specified thread schedule. Solution proposed by JReplay is independent of the underlying operating system and of the JVM implementation the program is running on. Replay of a schedule is achieved by instrumenting the original application according to a given schedule and supplying a library containing a replay engine in addition to the instrumented class files to the virtual machine at start-up. No separate thread is created to control replay. Rather, each application thread performs method calls to the replay engine at appropriate times. To achieve deterministic replay, JReplay only allows one thread to run at a time with all other threads blocked. The running thread is specified by the given schedule. To control which thread is currently running, JReplay assigns a lock

object to each thread during the replay of the instrumented program. Threads are blocked and unblocked using these locks and Java synchronization mechanism. In order to transfer control from one thread to another, JReplay unblocks the next thread scheduled to run and then blocks the current thread. No work for the record/replay of multi-threaded Android applications has been published to date.

CHAPTER 4

DESIGN OF REPLAY DEBUGGER

This chapter talks about the overview as well as some details about the design of the debugger tool. The technique of the debugger consists of two main phases: record and replay. Figures 12 & 13 informally depict the two stages, record and replay respectively. The record stage takes place when the application is running. The application runs in an environment around it. The environment consists of the Dalvik Virtual Machine, other applications running in different processes and many other events taking place around the application. The application and the external events/inputs that affect its execution behavior are part of the captured subsystem. The recorded information is then stored in a file to be used during replay.

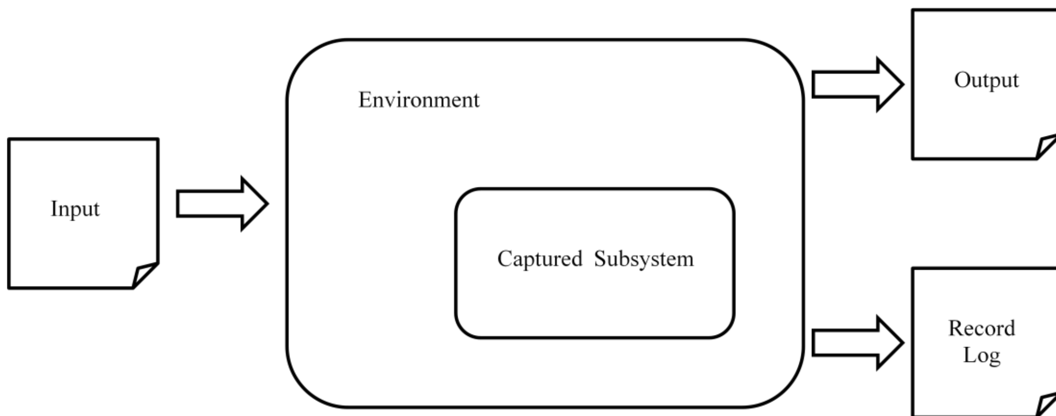

Figure 12: Record Process Overview

In the replay stage, the captured subsystem remains the same. The recorded information is then used as input to the application. The data of the recorded external events/inputs is fed back to the application. The order of events recorded in the Log, is useful when replaying the application. The same order is enforced on the application events for faithful replay.

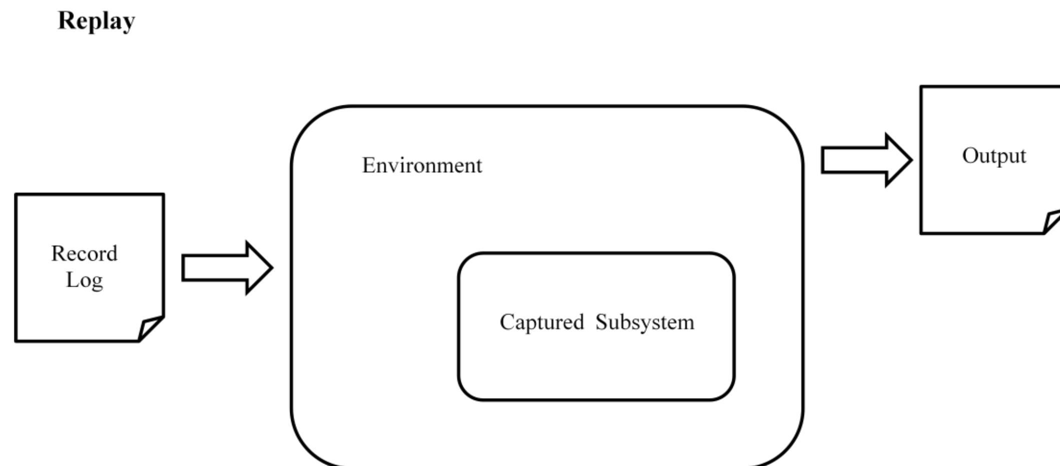

Figure 13: Replay Process Overview

4.1 Record Replay in Java

There are essentially two main approaches to record/replay.

The first approach is a 'content based' record/replay system. During the record phase in such a system, all the data that is read by a thread from main memory is stored in a trace file. While during replay, the same stored data is fed back to the thread. This method is very intrusive since each and every read operation performed by the thread is traced. So essentially it generates huge trace files. However, the advantage of a content based record/replay system is that there is no dependency while replaying. A thread can be replayed without re-executing any

other thread. It also automatically traces synchronization races and the input, provided that the input is also stored in memory.

The second approach is 'ordering based' record/replay. Here, only the order in which data is exchanged or accessed with the main memory is traced. If during replay the same order can be imposed along with the same input, the same program execution can be recreated. The amount of data to be traced can be varied. The exact order in which all instructions are executed can be logged. This becomes a problem on multi-processor systems, where there are many threads executing simultaneously.

The JVM guarantees that the instructions executed by a single thread are seen by that thread to happen in program order. However other threads may or may not see the same order. The only guarantee is that all operations performed by one thread before the release of a lock will be visible to another thread when it acquires the same lock [32]. Due to this it is not possible to obtain the total ordering of the executed instructions. The other possibility is to record the order in which the synchronization operations are executed [24]. If all data accesses are properly synchronized (no data race conditions), this approach allows for a faithful replay, with much smaller trace files. It is precisely this approach that is used in this thesis. The following section will touch upon certain definitions which are required to understand the proposed record/replay technique.

4.2 Definitions

Any event that is recorded in the trace file is called a *thread interaction event*, E. It is characterized by type of event (e.g. Thread Creation event, Synchronization event and so on), thread ID, Object ID and the Lamport clock. Object ID, O_{id}, identifies the object on which the event takes place or the object used for communication between threads. Thread ID, T_{id}, identifies the thread performing or involved in the event. The set of events performed by the same thread is called thread event set, TES_{id} for thread T_{id}, while the set of events performed on the same object, O_{id}, is called object event set, OES_{id}.

When doing ordering based record/replay, the order in which threads perform synchronization operations must be recorded. To accomplish this, A Lamport clock is maintained for each O_{id} or T_{id} and is equal to the Lamport clock of the most recent event happened on O_{id} or invoked by T_{id}. $LC(a)$ is a function that returns a Lamport clock timestamp taking a parameter as E_i, O_{id} or T_{id}. The Lamport clock has been explained in previous sections. Every time a thread performs a synchronization operation, the Lamport clock value is updated and saved in a trace file.

In addition to the happened-before relation we define an *immediate happened-before* relation, denoted by " |→ ", for two successive thread interaction events as follows. For two thread interaction events E_b and E_a, there is an immediate happened-before relation $E_b \mid\rightarrow E_a$, if

1) E_b, $E_a \in Set_{id}$ where Set_{id} is OES_{id} or TES_{id}, and

2) For all $E_i \in Set_{id}$ $(i \neq b)$ that satisfies

$LC(E_a) - LC(E_i) > LC(E_a) - LC(E_b) > 0$

According to the above expression, E_b is the last event before E_a in the Set_{id}.

4.3 Record

In the record phase, the happened-before relations between events are captured by chains of immediate happened-before relations. The Lamport clock is updated for each event. The computation of the Lamport clock is as follows

$LC(E_i) = max(LC(T_{id}), LC(O_{id})) + 1$

$LC(T_{id}) = LC(E_i)$

$LC(O_{id}) = LC(E_i)$

Hence, both the thread and the synchronization object get a new clock value that is 1 higher than the maximum of their previous clock values. Hence, the logical time of this event is bigger than the logical time of any object that was involved in this event. In practice, each thread keeps track of its own logical time (increasing for every new event), and the logical time of synchronization objects *O* is used to communicate the logical time from one thread to another. A link to the event that happened before this event is also stored. The happened before relation is established using the *immediate happened before* relation explained in the Definitions section. The unique combination of *<Type of Event, Thread ID, Object ID, Event Happened Before, Lamport Clock>* is saved in the trace file. If it is an external event, then the associated data can be saved too.

An example of a record phase using Lamport clocks is given in Figure 14, which shows an application that is using three threads. These threads have already been started by the main thread. Initially the threads have clock values as

indicated at the top of the figure. The monitor operations of entering and exiting a monitor are given by the brackets opening to the bottom and the top respectively. Arrows between monitor operations indicate a dependency, i.e. the source of the arrow must precede the target. This dependency in the form of immediate happened before is logged for each event. If there is not dependency this is marked as -1. Notice that at each synchronization operation two clock values are updated: (i) the $LC(T_{id})$ field which represents the Lamport clock time stamp of the thread performing the synchronization operation, and (ii) the $LC(O_{id})$ value, representing the Lamport clock time stamp passed from one thread to another using the object associated with the synchronization operation.

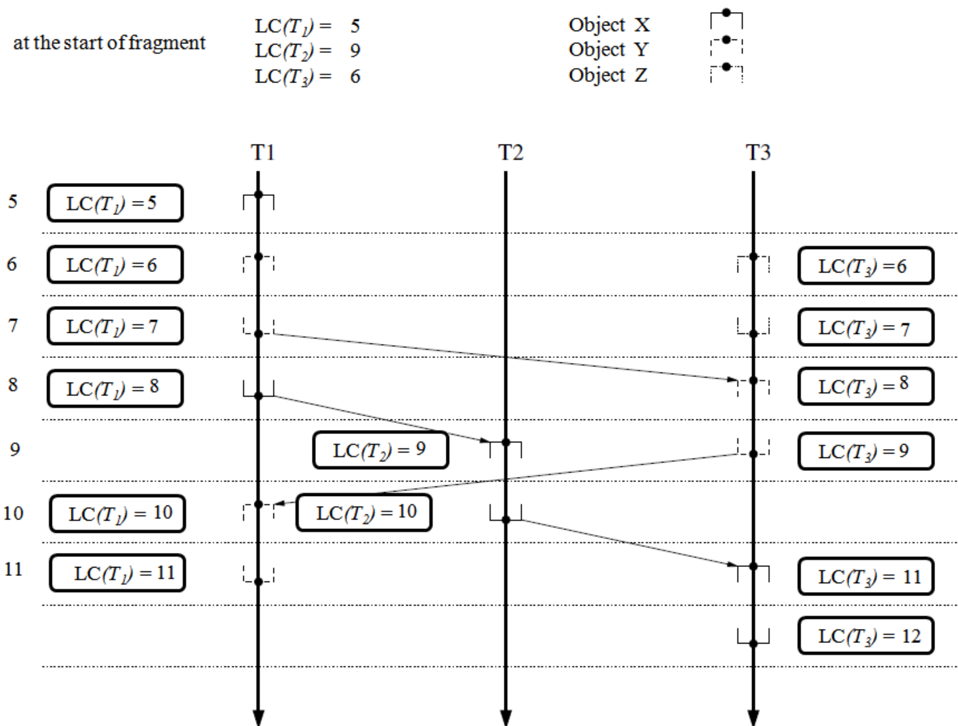

Figure 14: Identify Ordering of Events

After the recording stage, the information collected in the trace file represents the partial ordering of events. The partial ordering can be represented as a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{L})$, where $\boldsymbol{V}$ is a set of events recorded, $\boldsymbol{L}$ is a set of edges, and each $l = (E_1, E_2) \in L$ denotes the immediate happened-before relation between E_1 and E_2. So now we have a list of how many events were executed by each thread and the partial order/dependency between these events.

4.4 Replay

During this phase the operations executed by all threads must be executed in the same causal order as during the record phase: this can be guaranteed by executing the events in the order they were recorded in the trace file. Each time a thread *T* executes a synchronization operation; its LC value is updated in a way similar to that during record. After the event executes, it is marked as "executed", while others are "not-executed". A thread can only resume its execution if all its immediate happened before relations are executed. Threads that are not yet allowed to execute the next events are suspended by the replay engine. Every time a thread arrives at a critical event, it calls the replay engine before executing the event and waits [27]. The replay engine then checks the immediate happened before relations for that event and thread, using the trace file information. The replay engine goes through all the suspended threads one by one. If a thread is running, it is not touched. It looks for the first non executed event for these threads. These events correspond to the events that the threads are waiting to execute.

If the relation has been satisfied i.e. the happened before event has executed then the replay engine signals the waiting thread to move ahead. If not, the thread is kept waiting till the relation has been satisfied.

The replay phase for the example from Figure 14 is given in Figure 15.

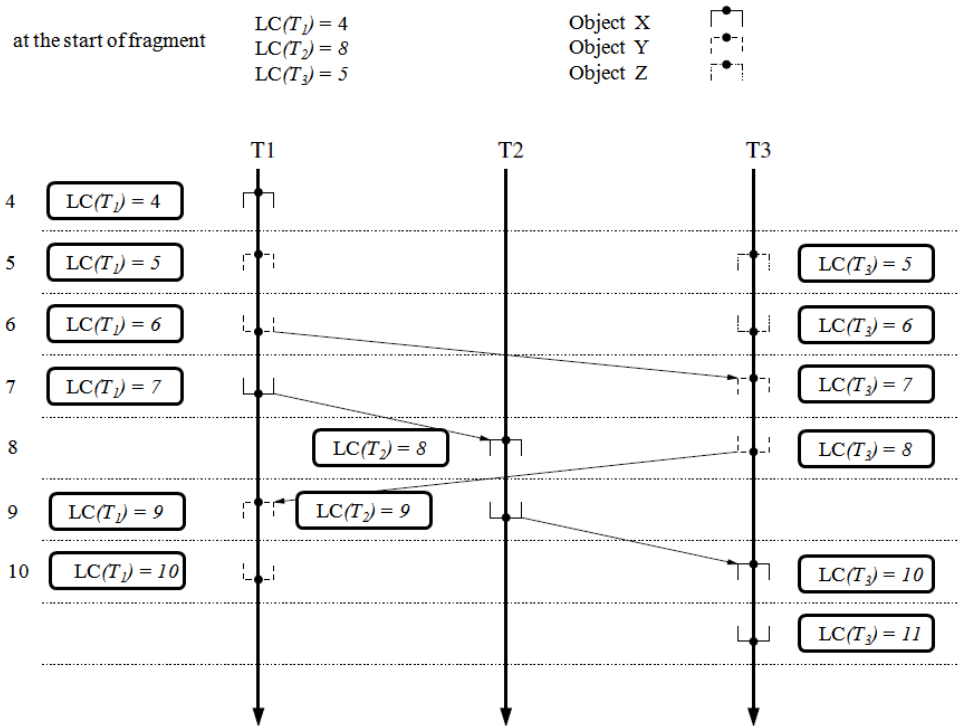

Figure 15: Using Ordering of Events for Replay

Assume that replay is at a state with clock values (4,8,5) for the three threads respectively. All threads may execute up to the point where they would perform their first recorded synchronization operation. Assume that T1 is the main thread. All threads are allowed to execute. Each time a thread executes an event, its Lamport Clock is updated. When Thread T3 arrives at the event at Lamport Clock 7, the immediate happened before relation is checked using the trace file. If Thread T1 has executed event at Lamport Clock 6, then T3 is allowed

to move ahead. Similarly for T2, replay system checks if T1 has executed event at Lamport Clock 7. If yes, T2 is allowed to move ahead. This continues until all threads have exhausted the values available from the trace and have replayed the original synchronization order faithfully.

4.5 Recording Events

As explained previously, the ordering based approach of record/replay is used. The events recorded to generate the trace information which are later used for deterministic and faithful replay are mentioned below.

4.5.1 Synchronization

Synchronization events can potentially affect the thread execution order and thus are a source of non-determinism. Android applications are written in the Java language. Java provides several flavors of synchronization.

Synchronization construct

The main synchronization construct in Java is a 'monitor'. It suffices to record the use of the monitors to be able to perform a faithful replay of an application. A monitor in Java is basically a block of statements guarded by a lock on an object. Entering the monitor means that the lock on the guarding object is taken. Leaving or exiting the monitor means the lock is released. The same object may be used to guard multiple monitors. Each time a monitor is entered by a thread T_i, the lock on the object O guarding the monitor is acquired by T_i. Another thread T_j trying to enter a monitor with the same guard, object O, has to wait until the lock on O is released by T_i.

In Java, a monitor can be expressed in two ways.

1. The synchronized(O){block} statement.

2. A synchronized method.

The design for recording and replaying the synchronized construct is shown in Figure 16 & 17. The logging of data and the actual event should be atomic. The reason is explained in the oncoming sections. In Java, entering a synchronized section is indicated by the MONITOR ENTER bytecode, while exiting it is done by MONITOR EXIT. As discussed before, we have to log the immediate happened before event, update the Lamport Clock

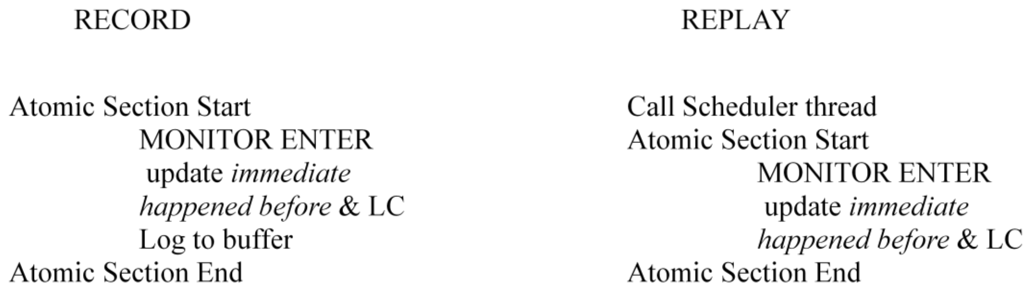

Figure 16: Instrumentation of MONITOR ENTER

and log to the buffer. However, all of this has to be done atomically since there are many threads running concurrently. Hence the updating has to be unique and atomic. During Replay, we enforce the ordering of events to achieve faithful replay. Hence there should be a call to the replay engine before the event actually executes. The replay engine will then decide based on the trace information, whether to allow the thread to move ahead or block it to ensure correct ordering.

RECORD	REPLAY
Atomic Section Start MONITOR EXIT update *immediate* *happened before* & LC Log to buffer Atomic Section End	Call Scheduler thread Atomic Section Start MONITOR EXIT update *immediate* *happened before* & LC Atomic Section End

Figure 17: Instrumentation of MONITOR EXIT

4.5.2 Wait/Notify

Wait/notify can be used to co-ordinate the execution order of multiple threads. A thread *T* can execute a wait() method on an object *O*. This puts the thread in wait state and releases the lock on the object *O*. Thread T can execute wait on an object *O*, only when it has acquired the lock on object *O*. Hence it is necessary that *T* has a lock on object *O* before it can invoke the wait() method on that object. It is also necessary that a thread has a lock on the object on which it invokes the notify(). Invoking notify() on an object *O* sends a signal to a thread waiting in the queue of object *O*. Upon receiving such a notification, *T* once more competes to reacquire the lock on *O*. As soon as the lock is acquired, *T* will continue its execution in the monitor. Hence, wait and notify have an effect on the order of thread execution. Also, notify wakes up a thread waiting on an object *O*. This does not guarantee, that the signaled thread will start running immediately. In the case of more than one threads waiting, the notify on an object wakes up an arbitrary thread waiting on the same object. The thread woken up is based on scheduling policy. Thus for wait(), there have to be two recordings, one before

wait and one when the thread starts running after wait. On the other hand one recording event suffices for notify().

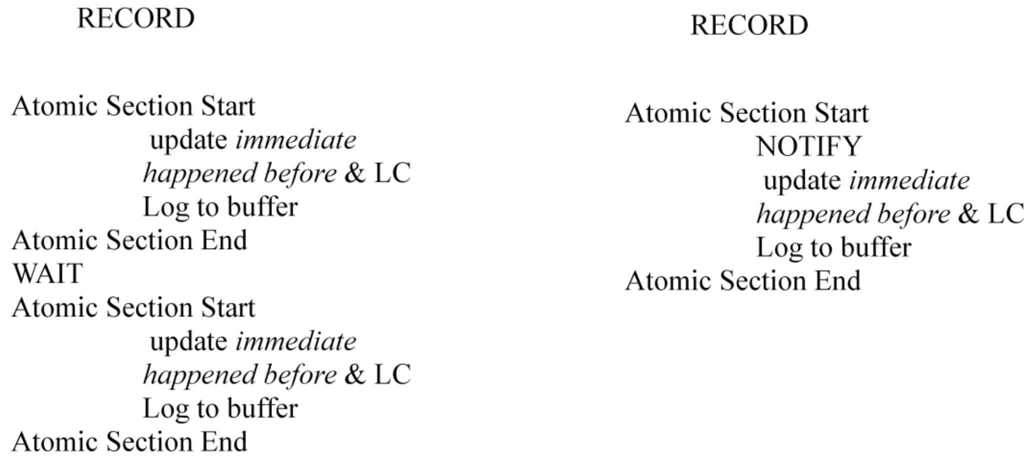

Figure 18: Instrumentation of Wait/Notify

Figure 18 illustrates the recording policy to be used in case of wait/notify. As previously mentioned, recording for wait has to be done twice while once suffices for notify. Since wait is a blocking event (i.e. the thread that executes wait() will suspend and hence cannot proceed until some another thread does a notify). If we make it atomic, this will result in a deadlock. Refer Figure 19 below.

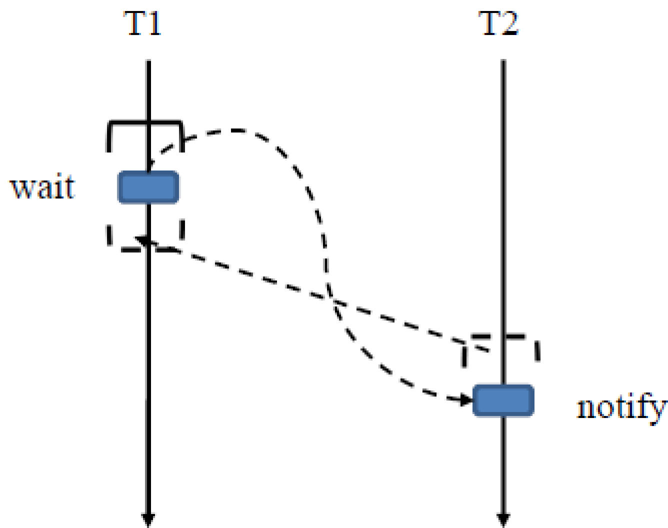

Figure 19: Incorrect Handling of Wait/Notify

T1 does not leave the atomic section since it executes a wait. T2 which is supposed to execute a notify, cannot do so because T1 is present in the atomic section. This causes a deadlock. A thread can successfully invoke wait on an object only if it is inside a synchronized block. Similarly, a thread can invoke notify inside on an object when it already owns the object in a synchronized block. Because of this, wait or the code for record need not be inside an explicit atomic section. A global log lock is only required while writing to the log, since some other thread can log an unrelated event to the log.

Currently, only the recording of wait/notify has been derived. The algorithm for replay of wait/notify events has yet to derived.

4.5.3 Thread Creation

In Java and thus in Android, threads are supported by the language itself – the java.lang.Thread class. A new thread is created by using the “new” keyword.

Thread thr = new Thread();

The above line of code will only create a Thread object. This does not correspond to the actual start of the thread.

thr.start();

The above line will issue the actual starting or running of the created thread. This event should be recorded and stored in the trace to be used later on during the replay.

RECORD	REPLAY
Atomic Section Start	Call Scheduler thread
THREAD START	Atomic Section Start
update *immediate happened before* & LC	THREAD START
Log to buffer	update *immediate happened before* & LC
Atomic Section End	Atomic Section End

Figure 20: Instrumentation of Thread Start

Figure 20 illustrates the design for record and replay of thread start. The explanation is same as mentioned in previous sections.

4.5.4 Message Passing between threads

In Android, there is one main/UI thread which handles all the UI components of the application. No other thread is allowed to access the UI part of the application. This is to avoid inconsistent display of UI components, e.g. two threads updating the contents of a Textbox simultaneously will result in incorrect data being displayed. If any other thread wants to update the UI of the application, it has to communicate with the main thread. Every main thread in an application has a Queue associated with it. The main thread keeps checking this Queue for

messages. A Looper and Handler class; together are responsible for handling of messages. They are run by the main thread.

Rather than storing the content of the messages, the order of each access of the Queue is stored. This is because the data/message sent will be the same during replay. This reduces the size of the trace file generated during recording of the application. Any other message passing between the application threads also takes place using this Message Queue. Figure 21 illustrates the record and replay strategy for message passing.

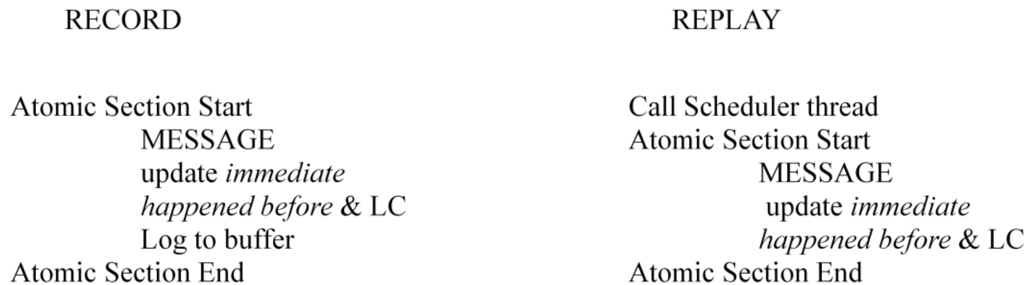

Figure 21: Instrumentation of Message Passing

4.5.5 External Events

Recording of external events is essential to replaying of an Android application. For a faithful replay, the time of the external event and the data associated with it should be the same during replay, as it was for record. Also, a bug may be caused by a particular execution order of threads interrupted by an external event. Hence it is necessary to store the order of these events relative to others along with the data associated with it. During replay the I/O event can be played back using this information during the re-execution of a program.

In Android, external events are delivered to the application by the Android system. The part of the Android system that takes care of reading and dispatching events runs in a process called System Server. This runs in a separate process and VM instance. The System Server is a process that is important to the core of Android system. It provides important services like Activity Manager, Power Manager, Alarm Manager etc to all the Android applications. Each service runs in a separate thread. There is a service called Window Manager Service which opens input devices and dispatches the raw event data into events delivered to the application windows. The Window Manager Service starts two threads, viz Input Device Reader and Input Dispatcher Thread. Input Device Reader actually reads from a Linux device "/dev/input" and enqueues the data. The Input Dispatcher Thread reads from this queue and dispatches event to currently focused application window. Since the Android application runs in another process, conveying of the event involves inter process communication. The way of inter process communication in Android is Remote Procedure Calls. This involves copying data from the System Server address space to the address space of the Android application. All this functionality is provided by Android using the Binder utility. To handle this RPC in the Android application, a group of threads called Binder threads are provided. These are similar to daemon threads, in that they provide some service to the main application. These Binder threads then communicate to the main thread of the application regarding the event.

The main thread runs special code called event handlers to handle the communication between the Binder threads and the application. Currently, the

external events are key press events generated by the keyboard of the device.

Only the application threads are recorded for a faithful replay of the application.

To put all of the above discussion in perspective, consider the figure 22 below.

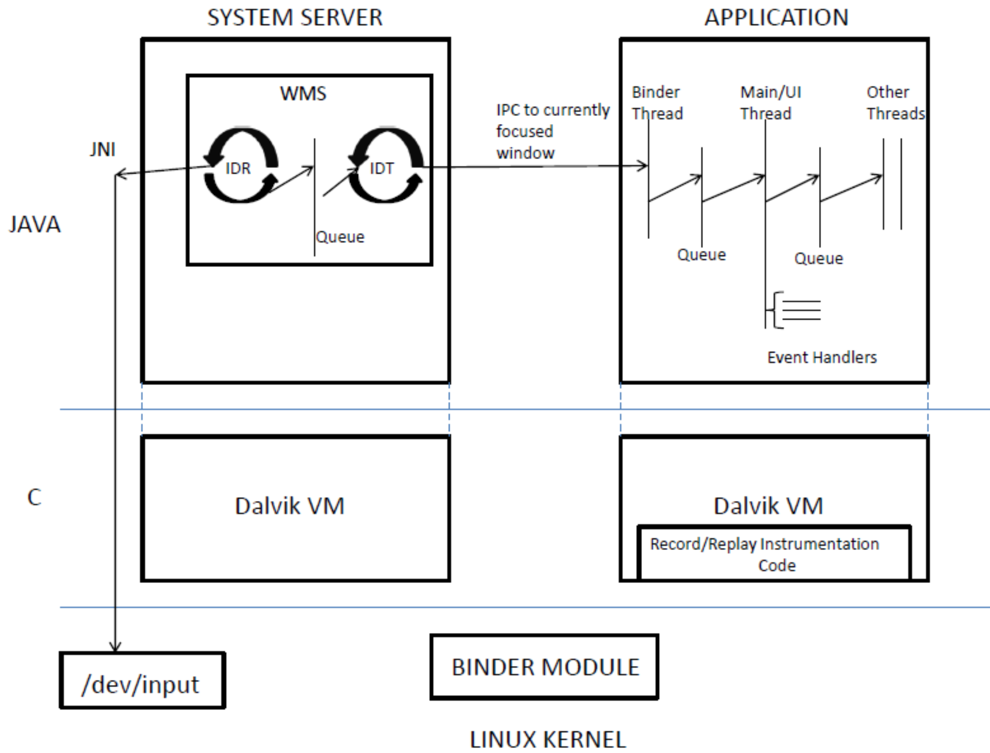

Figure 22: Overview of Working of Android

CHAPTER 5

IMPLEMENTATION

On a uniprocessor system, execution behavior of an application can be uniquely identified by the sequence of execution events. Events could include thread creation events, synchronization events, message passing, external events and so on. Therefore, two execution behaviors of an application are identical if their execution sequences are identical. For a multi-threaded application, events can be executed by different threads. Here, the order of execution of events matters along with the thread executing them.

In Java, an (execution) event can be defined as an execution of a Java bytecode by an interpreter. Android applications are written in the Java language. The Java language is an interpreted language. The Java files are compiled into bytecode, which is then interpreted in the virtual machine. Android has its own version of this virtual machine called Dalvik Virtual Machine. All Java code goes through this Dalvik VM. Record and replay is achieved by adding code to the Dalvik Virtual Machine. This is where the instrumentation for record and replay mechanisms is present.

As explained in previous sections, each Android application has its own instance of the Dalvik Virtual Machine. Before the actual application starts, the Virtual Machine should be up and running. It is here that record/replay mechanism is initialized and started. The initialization or startup of the Virtual Machine is required before the application.

5.1 Initialization

Initialization is the list of things to do before the actual execution of a program starts. There are many things that need to be set up before an instance of the Dalvik VM starts. For instance, the garbage collector thread needs to be set up and started, certain classes need to loaded etc. There are many methods called in sequence for initialization of the VM. It is in one of these methods that the specific method for record/replay is called. For record, initialization includes allocating two identical sized buffers for double buffering scheme used for logging the events. Initializing certain data structures and fields to their start values, to be used in recording is also done. It also creates the Logger thread, which would be responsible to dump the data from buffer to a dump file when buffer is full. During replay, the scheduler thread is created, which ensures the partial order of events is maintained as it was during recording. The memory dump file created during record is read. Certain data structures which are used to determine the happened before relations of an event are initialized with the correct values from the file.

All these essential things should be completed. They are done in a function at the start of the VM. It is in this function after all the necessary VM initialization; the record/replay initialization code is added.

A double buffering mechanism is implemented for the logging process. The idea behind double buffering is to divide the buffer into two segments and to present these two segments as two separate buffers. When the first buffer fills up, data is written into the second segment while the Logger thread is writing out the

contents of the segment that is full to a dump file. When the second segment becomes full, the Logger should have finished writing out the contents of the first segment to disk and therefore the first segment is now available for writing data into it. However, it should be noted that the double buffering has its limitations and there are problems, which cannot be solved with double buffering. These problems arise from situations where the traced application generates execution data to be recorded at a much higher rate than the rate with which these data can be written to disk. In this case, both buffer segments will become full before the Logger thread will be able to write one of them to disk. The way round these situations is to block the recorded application and therefore prevent it from generating any more data before the Logger has an opportunity to write out the buffer segment to disk. Whilst this solution has an adverse effect on performance, the situations causing this problem are unlikely to occur very often. The implementation of double buffering uses a well known producer-consumer model, which employs condition variables (pthread library). Once the application to be recorded starts running, the Logger thread takes on the role of the consumer whereas the application threads are the producers. The Logger thread sits in a while loop, waiting for a signal if none of the buffers are full. Once any of the buffers' is full, the Logger thread is signaled. The Logger thread breaks from the while loop when the application goes to the background. It also logs the half filled buffers to the file for the last time. Refer Figure 23 for above explanation.

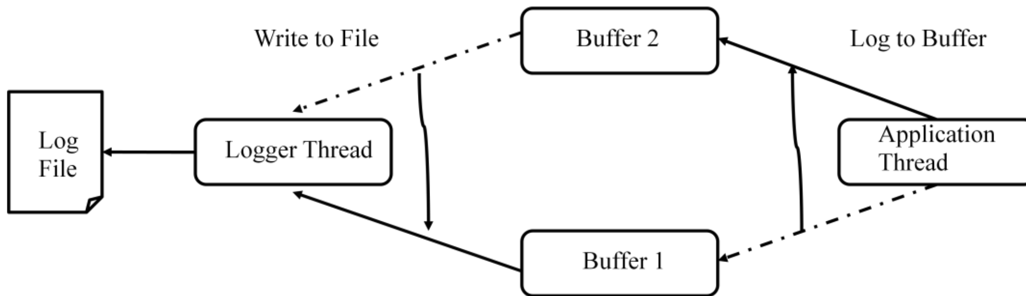

Figure 23: Double Buffering while Recording Events

During replay, the initialization code will create the scheduler thread.

5.2 Record

Since multiple threads execute these events and update the same global clock, the following three actions must be executed as a single atomic action during the record phase:

Logging of Event and associated data to trace file.

Update immediate happened before relation

Updating the Lamport clock

The actual event.

The thread executing the above actions may be interrupted for a couple of reasons, and there may be a thread switch. Some other thread may update the Lamport Clock and log another event. This will cause an incorrect order of events to be recorded. Since this log is used while replaying, the events and threads may be executed in an incorrect order. Hence there is no deterministic replay. To avoid such things, the above must be atomic while recording.

5.2.1 Synchronization in Java

As discussed in previous section, the synchronized construct is used to implement critical sections in Java. This can be a synchronized block or a method. Java's

byte code has two instructions to help implement critical sections: monitor enter and monitor exit. In the Dalvik Virtual Machine, the critical sections for synchronized methods and blocks are implemented using monitor enter and monitor exit. Each MONITOR ENTER/EXIT results in a call to the methods *dvmLockObject* and *dvmUnlockObject* respectively. The methods in turn correspond to a *pthread_mutex_lock/unlock* respectively, in the underlying code of the Dalvik Virtual machine. This is shown in Figure 24. Each of these are events have to be recorded. Reproducing the same order in which threads enter a critical section is necessary to reproduce the same order in which threads access shared variables within the critical section. Since we are interested in the partial order of events, the instrumented code will be as follows for entering of synchronized section:

Monitor enter

Log to buffer

Update immediate happened before relation

Update Lamport Clock.

And for leaving the synchronized section

Log to buffer

Update immediate happened before relation

Update Lamport Clock

Monitor Exit

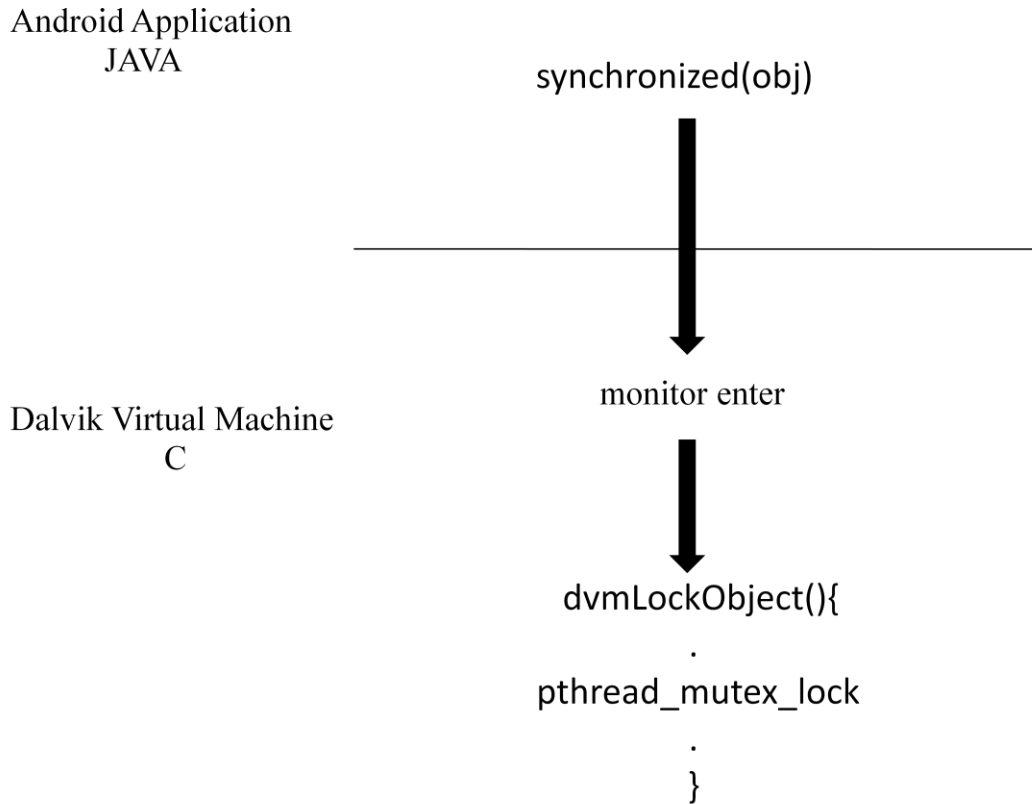

Figure 24: Synchronization in Android

We are interested in the partial order of events and not the exact order. Hence we need to maintain the correct happened before relation pertaining to locking/unlocking using an object. The order of locking/unlocking of one object with respect to that of another object is of no consequence. Only one thread is allowed to enter the critical section pertaining to a particular object, at the same time. We achieve implicit atomicity while locking/unlocking of the same object. Hence no explicit lock/mutex is required for ensuring atomicity.

5.2.2 Message Passing

Message passing between Android application threads takes place through a Message Queue. This is associated with the main thread. All accesses to this

queue are synchronized. Hence, the recording of events takes place through recording of synchronized access of the queue, as explained in the above section.

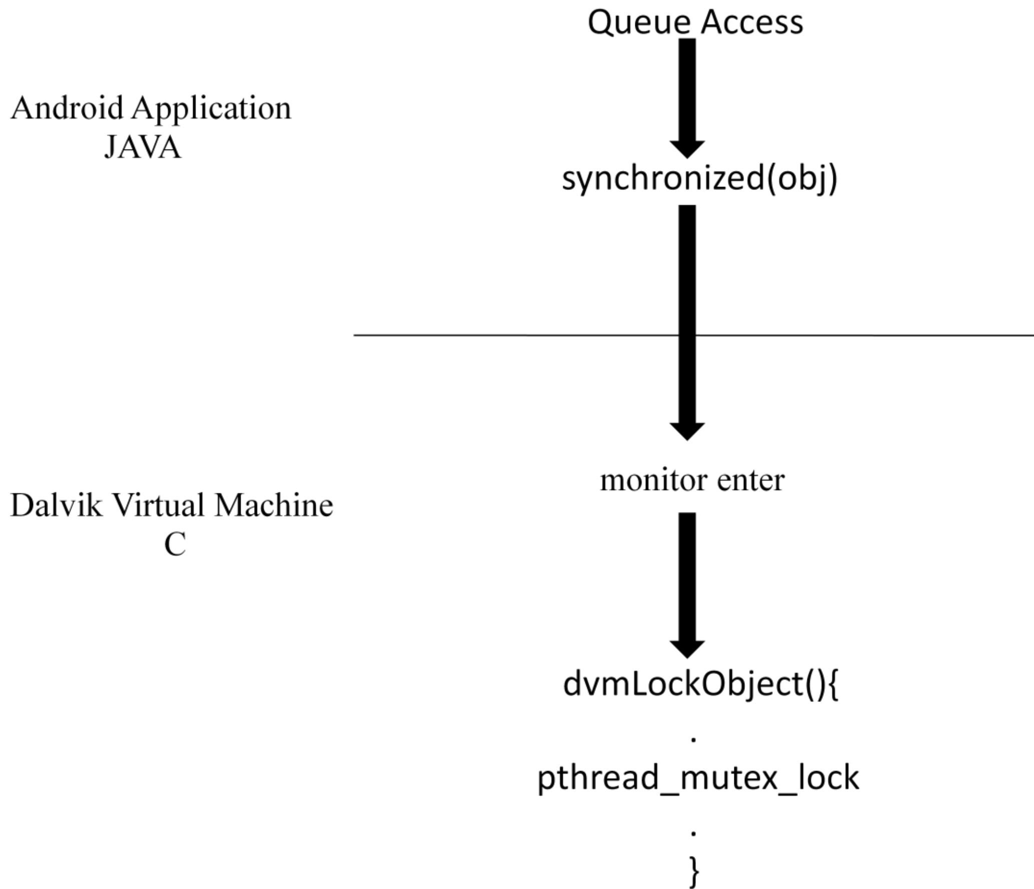

Figure 25: Message Passing in Android

Also the order of each queue access is recorded instead of the actual message content. This reduces the size of the trace file. Since each queue access is synchronized the recording takes place in a similar way as actual synchronization. Figure 25 illustrates this mechanism.

5.2.3 Thread Creation

In Android, every Java thread is mapped to a native thread. There is a one to one mapping between Java threads and native threads. Since the Java bytecode

is interpreted by the DVM, every thread creation call gets interpreted in the DVM as well. Each *Thread.start()* method ultimately becomes a *pthread_create()* method call as shown in Figure 26. This *pthread_create* is responsible for creating and starting a native thread corresponding to a Java thread. Inside the Dalvik VM, a Thread structure is maintained for each thread. This structure maintains all the information about the thread, like the actual Thread object corresponding to the thread, thread id, a pthread_t handle to the thread, current status of the thread and so on. This is where the instrumentation code for logging to buffer and updating Lamport Clock is placed.

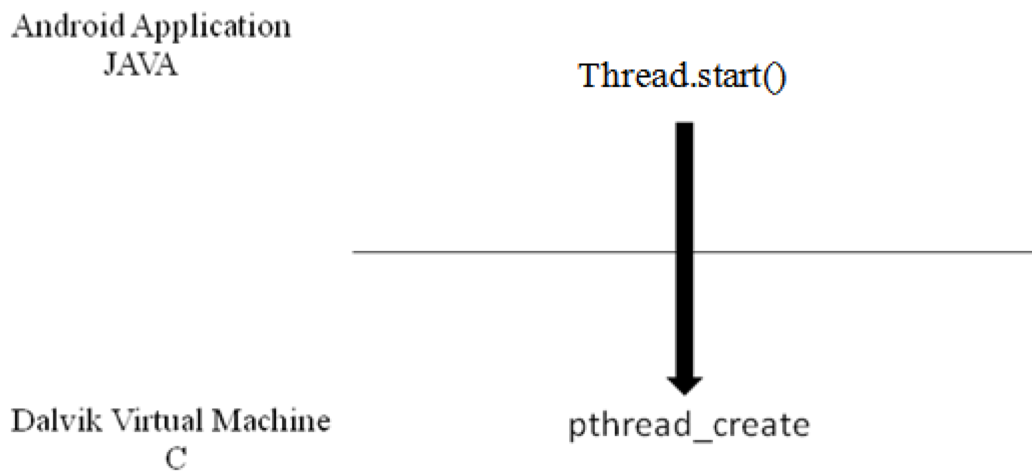

Figure 26: Thread Creation in Android

For all the above, it is necessary to ensure that the atomicity conditions mentioned at the start of this section are met. This is done using a global mutex lock from the pthread library around the necessary lines of code.

All this instrumentation for record/replay is inside the DVM. To make sure that only a particular application is recorded, the following is done.

Instrumentation of the *java.lang.Thread* class.

This helps to determine which threads and hence which application should be recorded. The instrumentation includes addition of a field and a setter method to assign a value to this field. The application should include a single line of code for each thread, setting the value of this field appropriately. Underneath the VM, the value of this field is the deciding factor on whether to record a particular event. If the event is invoked by a thread whose field value is set, that event is recorded. The other methods possible are addition of an extra Thread constructor, with the value of the field as the parameter. For this to work every application will have to change the way threads are created. In contrast to this just a single line of code with a call to the said method would suffice.

In the Dalvik Virtual Machine, a table is maintained for each application thread and each synchronization object, storing essential data like the current Lamport Clock and previous event executed or immediate happened before event. This is logged and updated after each event execution. Even though the thread ID of the application threads remains the same in each run of the application, there is no consecutive ordering of threads IDs. Hence a mapping of thread IDs is required to identify application threads in subsequent runs of the application.

5.2.4 Wait/Notify

Wait/notify are Java language constructs that can be used to coordinate the execution order of multiple threads. A thread that has executed a wait on an object must wait till a different thread executes notify on the same object. These constructs can change the order of thread execution and introduce non-

determinism in an application. Hence these must be recorded and used to replay the same partial order of threads.

In Android, each wait/notify is handled by the Dalvik Virtual Machine running underneath each application. In the Dalvik VM, each wait/notify is handled by the methods *dvmObjectWait/Notify*. To summarize, each wait/notify boils down to a *pthread_cond_wait/signal* call. Refer Figure 27. These methods are instrumented accordingly for record and replay. The instrumentation code and the actual event should be atomic to maintain the partial order.

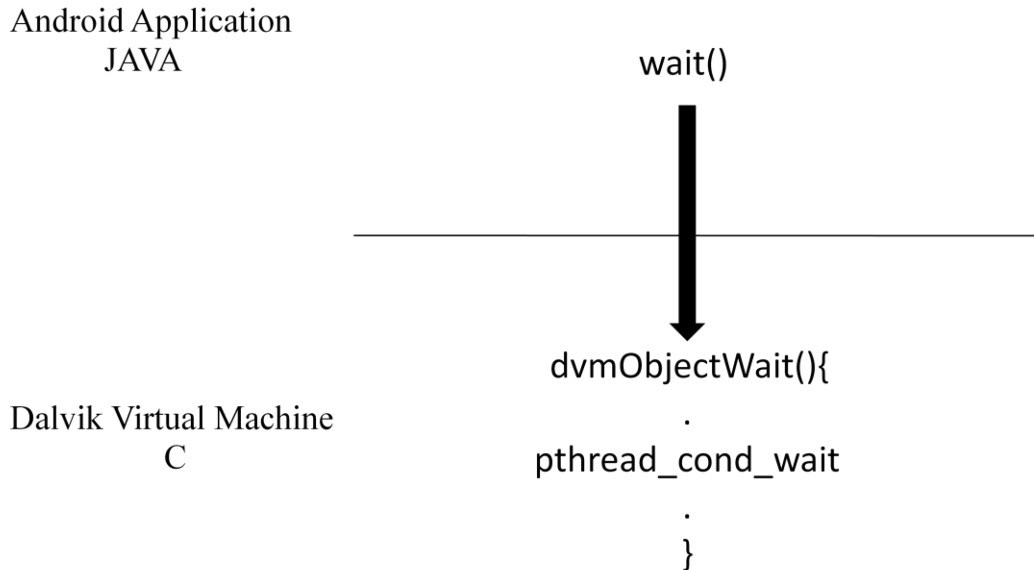

Figure 27: Wait/Notify in Android

The recording of wait/notify events is done inside the DVM as specified in the wait/notify section of the Design chapter. The wait event has to be recorded twice, once before the actual wait and once after the thread wakes up and reacquires the lock. As mentioned the actual mechanism is using *pthread_mutex_lock* and *pthread_cond_wait*. The implementation of the methods *dvmObjectWait/Notify* is

used to our advantage for atomic logging. Currently, only recording of wait/notify can be achieved. However, replay of wait/notify is yet to be achieved.

The flow of the record phase is as shown below:

Initialize the buffers, Logger thread and certain other data structures essential for record when the VM starts up.

Start the application execution.

Whenever application thread arrives at significant events, check whether they should be recorded.

If Yes, record, update immediate happened before relation and update the Lamport clock

Log to buffer.

If current buffer is full

Then switch buffer and signal the Logger thread

Continue this till end of application.

5.3 Replay

Replay is achieved by enforcing the ordering of events using the information in the trace file. During replay, all the non-deterministic choices made during recording are applied while executing the application. The replay is carried out in a way similar to that described in the Design chapter. The replay mechanism is the same for all the events recorded.

Similar to the record phase, the decision of whether to replay an application is taken by the value of a particular field in the Thread class. If this field is set to a particular value, all the critical events executed by that particular

thread are replayed. Whenever such a thread arrives at a critical event, it calls the replay engine and waits for the replay engine to get back. The replay engine is implemented in a separate thread running in the DVM instance. This thread is called the scheduler thread. This thread goes over the trace information and checks the immediate happened before relations for the waiting threads. If the relations are satisfied, that particular thread is signaled to move ahead with its execution. Else if a particular thread is running, the scheduler thread does not interfere with it. Thus the scheduler thread decides which thread should proceed and which should not based on the trace information. This thread does not interfere with the underlying Linux scheduler. All threads including the scheduler thread are handled by the Linux scheduler. The scheduler thread can be thought of a VM level scheduler.

The flow of replay phase is as below:

Arrive at critical event

Call replay engine

Wait for signal from Replay Scheduler thread

Replay Scheduler thread consults recorded log

Checks for immediate happened before relations

If all immediate happened before events are executed OR If none found

Signal application thread to proceed.

Repeat till end of program.

5.4 Lifecycle of Application

The Lifecycle of an Android application is very different from a normal Java application. A normal Java application has a fixed entry point in the form of a main method. When the main method ends, it signifies the end of the application. However for an Android application, there are more than one entry and exit points. An important and unusual feature of Android is that an application process's lifetime is not directly controlled by the application itself. Instead, it is determined by the system through a combination of the parts of the application that the system knows are running, how important these things are to the user, and how much overall memory is available in the system. Currently, recording of the application takes place from the Start of a new instance of the application till it goes to the background. The Start of a new instance is marked by execution of the *onCreate()* method of the application, that has been explained before. There is no exit button for an Android application. The user does not have any control on exiting the application. The application becomes a background process when triggered by the back key press. This back key press is recorded to mark the end of application recording. The replay is done till all recorded events are executed.

5.5 Log Structure

The recorded information for critical events is stored in a file called memory dump. The recorded information is stored in a particular format so that later on it can be used in the replay phase. Each entry in the log is of a fixed size and format. The log structure has the following fields

Serial Number – Stores the serial number of the recorded event. Serial numbers start from 0 and are incremented for each event. These are later on used during replay to identify happened before events.

Event Type – Helps identify the type of event. The recorded event may be a Thread creation event, a synchronization event or an external event and so on.

Thread ID – Signifies the unique ID of the thread that executed the event. This helps to decide which thread should be scheduled to execute the event.

Lamport Clock – The Lamport Clock value associated with that particular event. It may not be the same as the Serial Number field.

Event Index – This helps identify the type of a particular event. The Event Type field signifies the type of event e.g. Synchronization event. This field will symbolize whether it is Synchronization lock or Synchronization unlock. The sign of the field is used for the same.

Different Thread Dependency –This defines the immediate happened before relation. It points to the event this current event is directly dependent on.

Status – This particular field is more helpful during replay. Before Replay, this field of all events is initialized to NOT_EXECUTED. As replay progresses, all events executed are marked as such.

CHAPTER 6

COMPARISON AND CONCLUSION

Multi-threaded programs are in general non-repeatable due to races. This makes debugging multi-threaded programs very hard. Android applications are such multi-threaded Java programs.

In this thesis, the mechanisms that allow deterministic execution of multi threaded Android applications for debugging purposes have been presented. The contributions by this tool are enhanced debugging capabilities by tackling non-deterministic events like synchronization constructs, wait & notify, message passing between threads & external events like key press. Currently, record/replay of synchronization construct, message passing between application threads and thread creation have been considered. All these contribute to non-determinism like synchronization races. The deterministic replay is achieved by using a modified Dalvik Virtual Machine. The tool uses Lamport's happened before relation among events to establish a partial order. This partial order is then enforced on the thread execution during replay. Tools have been presented for multi threaded Java applications but none specifically for Android applications. Notable among these are JReplay, JaRec and DejaVu.

JReplay is similar to DejaVu as it tries to achieve deterministic replay of multi threaded applications by forcing the applications to execute according to a particular thread schedule. But instead of modifying the VM similar to this research, it instruments the bytecode of the original program. Solution of the JReplay is similar to this thesis, in that it is independent of the underlying

operating system. But it is also independent of the JVM. JReplay takes care of thread switches in the bytecode invocations. However, JReplay allows only one thread to run at a time according to the schedule compared to this research which only controls execution order of threads. Hence even if two threads normally run concurrently, they run one at a time during replay. This may be a block for performance reasons. Also, the information to record location of thread switches is assumed to be provided. It does not specify how to achieve this information.

JaRec achieves replay by instrumenting Java class files at load time. This causes a lot of record/replay overhead in the form of bytecode size increase. That increases the file size by 20% in record phase and 15% in replay phase, while the worst case increase is 120%. Increase in the number of synchronization operations, will increase the bytecode size by a considerable amount. All the instrumentation in this research is inside the Virtual Machine. Hence, there is no concern of increase in bytecode size. Also, the instrumentation takes place using JVMPI. All VM's do not support this feature, especially the Dalvik Virtual Machine. This could be a deterrent in using this system.

DejaVu is another notable tool designed for deterministic replay of Java applications. DejaVu was the first such tool to replay Java applications. It has also been extended for distributed applications. The approach presented in this thesis is similar to DejaVu, in that the replay is achieved by instrumenting the VM. However it does not use Lamport's happened before relation for keeping track of events, instead a combination of local clock and global clock is used. Also,

none of above mechanisms takes into account I/O operations, which are likely to impact the execution of the program.

This is the first such tool, capable of value additions in the form of deterministic replay of Android applications.

CHAPTER 7

FUTURE WORK

In this research a tool is presented to replay multi threaded Android applications which have been written in Java. The debugger uses Lamport's principle of happened before relation, to form a partial order of events. This partial ordering leads to smaller size of record logs along with correct order of events.

Currently, the recording takes place from start of application till it goes to the background. An enhancement worthy of consideration is to continue recording even after the application goes to the background. In this way, if that particular application is resumed by Android, a complete recorded picture can be obtained. Recording should stop only when the application process is killed by Android. Future implementation will undertake recording of the application for the entire lifecycle. It will involve recording of the application when it moves from background to foreground in addition to the current implementation.

External events considerably add to the non-determinism as has been explained. Logging of external events is orthogonal to recording of other events. Also, as an enhancement for further accurate recording of an application, it is essential to implement the record/replay of external events along with replay of wait/notify. This will widen the range of applications that can use this debugger tool for development.

REFERENCES

[1] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," U. S. D. o. Commerce, Ed.: National Institute of Standards and Technology, 2002.

[2] C. Mark and B. Koen De, "TRaDe, a topological approach to on-the-fly race detection in java programs," in *Proceedings of the 2001 Symposium on Java^{TM} Virtual Machine Research and Technology Symposium - Volume 1*. Monterey, California: USENIX Association, 2001.

[3] R. Goering, "Embedded developer survey reveals debugging challenges," 2007.

[4] Google, "Tools Overview," 2011.

[5] "Eclipse<http://www.eclipse.org/>."

[6] T. Henrik, S. Daniel, H. Joel, and P. Anders, "Replay Debugging of Real-Time Systems Using Time Machines," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*: IEEE Computer Society, 2003.

[7] A. Bowen, N. Ton, C. Jong-Deok, and S. Manu, "DejaVu: deterministic Java replay debugger for Jalapeno Java virtual machine," in *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*. Minneapolis, Minnesota, United States: ACM, 2000.

[8] A. Georges, M. Christiaens, M. Ronsse, and K. D. Bosschere, "JaRec: a portable record/replay environment for multi-threaded Java applications," *Softw. Pract. Exper.*, vol. 34, pp. 523-547, 2004.

[9] S. Viktor, B. Marcel, and B. Armin, "JVM Independent Replay in Java," *Electron. Notes Theor. Comput. Sci.*, vol. 113, pp. 85-104, 2005.

[10] Wikipedia, "Dalvik (software)."

[11] L. Leslie, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558-565, 1978.

[12] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *Electronic Computers, IEEE Transactions on*, vol. EC-15, pp. 757-763, 1966.

[13] P. Brady, "Anatomy & Physiology of an Android," 2008.

[14] Wikipedia, "Virtual Machine."

[15] D. Bornstein, "Dalvik VM Internals," 2008.

[16] Google, "Controlling the Embedded VM."

[17] D. Hackborn, "Binder IPC Mechanism," 2005.

[18] D. Morrill, "Inside the Android Application Framework," 2008.

[19] G. Jason, "A probe effect in concurrent programs," *Softw. Pract. Exper.*, vol. 16, pp. 225-233, 1986.

[20] J. J. P. Tsai, K. Y. Fang, H. Y. Chen, and Y. D. Bi, "A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 897-916, 1990.

[21] J. M. Mellor-Crummey and T. J. LeBlanc, "A software instruction counter," in *Proceedings of the third international conference on Architectural support for programming languages and operating systems*. Boston, Massachusetts, United States: ACM, 1989.

[22] J. H. Slye and E. N. Elnozahy, "Supporting nondeterministic execution in fault-tolerant systems," in *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*: IEEE Computer Society, 1996.

[23] Z. P. Douglas and A. L. Mark, "Supporting reverse execution for parallel programs," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*. Madison, Wisconsin, United States: ACM, 1988.

[24] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Comput.*, vol. 36, pp. 471-482, 1987.

[25] T. Kuo-Chung, H. C. Richard, and E. O. Evelyn, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 45-63, 1991.

[26] R. Mark and C. Bryce, "Replay for concurrent non-deterministic shared-memory applications," in *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*. Philadelphia, Pennsylvania, United States: ACM, 1996.

[27] Y. W. S. Yann-Hang Lee, Rohit Girme, Sagar Zaveri, Yan Chen, "Replay Debugging for Multi-threaded Embedded Software," in *The IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*: IEEE Computer Society/International Federation for Information Processing, 2010.

[28] C. Jong-Deok and S. Harini, "Deterministic replay of Java multithreaded applications," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*. Welches, Oregon, United States: ACM, 1998.

[29] "Deterministic Replay of Distributed Java Applications," in *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*: IEEE Computer Society, 2000.

[30] R. Michiel and B. Koen De, "RecPlay: a fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, pp. 133-152, 1999.

[31] Oracle, "Java Virtual Machine Profiler Interface (JVMPI)."

[32] F. Y. Tim Lindholm, "The JavaTM Virtual Machine Specification," 1999.

This document was generated using the Graduate College Format Advising tool. Please turn a copy of this page in when you submit your document to Graduate College format advising. You may discard this page once you have printed your final document. DO NOT TURN THIS PAGE IN WITH YOUR FINAL DOCUMENT!