Applying Desktop Race Detector Techniques to Mobile Systems

Dan Jonik
University of Michigan, Ann Arbor
djonik@umich.edu

Jason Varbedian
University of Michigan, Ann Arbor
jpvarbed@umich.edu

ABSTRACT

Mobile systems have recently become prevalent with more smart phones being sold than any other computing device. Development for these systems has increased as fast as the sales with applications and systems requiring more powerful hardware. Smart phones are being released with multicore architectures and come with the same multithreaded programming difficulty as desktop systems face. Debugging and developing applications for these multicore mobile platforms is a difficult task and few if any tools exist specifically for concurrency detection.

This paper presents an implementation and port of the FastTrack race detector for the Android mobile system. Android lacks any true instrumentation library and most Java instrumentation is done at runtime instead of compile which don't work on Android. The tool is able to detect errors in simple applications we wrote ourselves while running on an Android emulator.

1. INTRODUCTION

Multiple cores and multithreaded programming has become the standard on desktop systems. Chips have had to increase the number of cores to increase computing power. To take advantage of these cores multiprogramming is often used. Programming for multiple cores is often non-trivial even for so-called 'embarrassingly parallel' calculations. The challenge is greater for large applications and services like browsers and servers. Several tools have been developed for desktop systems to help with debugging these programs.

Smart phones are becoming multicore and apps are running with multiple threads. Many data race detectors and multithreaded programming tools have been developed for desktop systems, but few exist for mobile systems. The introduction of multicore processors to mobile space introduces non-deterministic bugs. These bugs inhibit development as the current technique for solving non-deterministic concurrency errors is the use of exhaustive, expensive, and time-consuming tests. These tools will run an app for a significant period of time with many different environment variables and output diagnostic logs. There are exponentially-many interleavings to explore and it can be difficult to pinpoint when and where an error occurred; additionally, it is just as difficult to reproduce a bug once it has been found. There is an obvious need for tools to air developers in finding data races before they manifest into multithreaded bugs.

There have been many race detectors, record and replay systems, and other tools to monitor and debug concurrent programs for desktop systems. In our research and our time working in industry with commercial Android products, we have found no such tools for mobile systems. It is possible there are proprietary tools that Apple and Microsoft use for their mobile systems, bu there isn't yet a publicly-available one for Android.

Our hypothesis is that data race techniques implemented on desktop systems can be applied to mobile systems. Our project ports a low-overhead, fast, and thorough dynamic race detector called FastTrack that runs on the Java Virtual Machine to Android's Dalvik Virtual Machine. The detector FastTrack [2] implements an algorithm that, used with the dynamic analysis framework RoadRunner, can monitor memory accesses and synchronization operations in concurrent Java programs. FastTrack checks that programs follow a *happens-before* relationship[3] with lightweight vector clocks that, in most cases, use constant space and time operations. RoadRunner provides an API for communicating an event stream to back-end analysis tools; FastTrack monitors this event stream to monitor an application's behavior. The tool makes debugging multithreaded applications much faster and easier to fix by providing detailed information about each detected data race.

In this paper we make two contributions:

1. Describe issues with porting data race detection systems to Android
2. A partial port of FastTrack and RoadRunner to Android

The rest of the paper is organized as follows. The following section reviews tools used for multithreaded program debugging and development and explains our choice of which tool is the most appropriate to port to Android. Section 3 describes the RoadRunner instrumentation framework. Section 4 explains the algorithm used by FastTrack to build and monitor a *happens-before* graph. Section 5 presents the modifications we made to the FastTrack tool to run it on Android, followed by an evaluation of our work in Section 6.

2. SURVEY OF EXISTING TOOLS

There are many techniques for detecting data races and providing record and replay capabilities for desktop multithreaded applications. We surveyed different tools and systems to determine which would be the most appropriate for use in a mobile computing environment. Mobile systems, compared to desktops and laptops, have less memory and fewer cores since energy is a limiting resource for system performance; therefore, we were most interested in debugging tools that had relatively small memory footprints and performance overheads. False positives, when a benign data race is erroneously reported as an error, greatly limit a tool's usefulness and usability, so we favored tools that guaranteed to report only legitimate errors. Several debugging tools require developers to make modifications to their application's source code before the tool can be used. This is not only a burden to the programmer, but also could possibly introduce additional bugs that were not previously present. Our ideal tool requires no source code modifications to preserve both correctness and usability. Our analysis of five different desktop multithreaded debugging tools is presented below.

DoublePlay [9] is a deterministic system that provides guaranteed record and replay with minimal overhead at the cost of using more cores. It does so by running the original multithreaded program on half the of cores and taking "checkpoints", while at the same time running multiple time-slices of a single-processor execution on the other half of the cores. The key insight is that by running different time-slices of a program each on their own individual core, multithreaded code can be executed and then replayed deterministically without a severe performance hit. In case of a mismatch between the single-processor execution and the multiprocessor execution, both executions are rolled back to the last checkpoint and restarted. DoublePlay is very effective because it offers full replay capabilities to multithreaded software, does not report false positives, and introduces little slow down. However, since DoublePlay is based on the assumption that the system on which it is running has unused cores (an assumption that most definitely does not apply to mobile systems), we decided that it was not the best option for porting to Android.

Eraser [8] is a system used to detect data races in programs that use locks for synchronization by using the *lock-set* algorithm. Eraser keeps track of which locks are held at each access to a shared object; if there is ever an access in which the set of locks held is empty, a data race is reported. Unfortunately, there are several circumstance in which false positives may be reported, such as user-implemented synchronization techniques. When these situations arise, developers are forced to add statements to their code telling Eraser to ignore certain objects and accesses. Eraser implements a straightforward data race detection algorithm, but ultimately fails on two of our predefined criteria, requiring source code modification and reporting of false positives.

Kendo [7] is a tool that guarantees weak determinism for multithreaded applications. It modifies the POSIX thread library to ensure a deterministic order of all lock acquisitions for a given set of inputs based on the current progress of each thread. Unfortunately, Kendo requires programmers to insert library calls around intentionally racy reads to flag them as benign so that Kendo ignores them. Perhaps an even bigger limitation is that Android applications run on the native Java Thread library; therefore, porting Kendo to Android would require a complete reimplementation as well as redefining a native Java library, which is discourage and limits the system's portability.

CHESS [6] is a system for systematically testing concurrent systems for bugs that appear only in certain thread interleavings. It does so by taking control of the thread scheduler and exhaustively enumerating all possible thread interleavings. To reduce the number of program execution paths that must be explored, threads are preempted only at certain key instructions (instead of at every instruction) that are most likely to be central to any bugs that appear. CHESS offers no false positives and near-complete code coverage, but requires the creation of wrappers to allow the tool to run on top of different development platforms Additionally, it is a resource-hungry application which would experience performance problems on mobile devices.

FastTrack [2] is able to monitor any concurrent Java program and report only true data races with full *happens-before* coverage. By instrumenting compiled Java class files at load time, it uses a shadow memory system to watch all synchronization and memory operations. It is able to identify the type of data race that has occured, which threads were involved, and which variable was the cause. FastTrack uses a shortened version of vector clocks, known as *epochs*, to monitor the *happens-before* relationship. *Epochs* reduce the memory overhead of the system, as well as speed up the comparison operations used to check for data races. FastTrack is the ideal algorithm choice for an Android race detector since it requires no source modifications, it has a small memory and performance overhead, and it reports only true data races.

3. ROADRUNNER

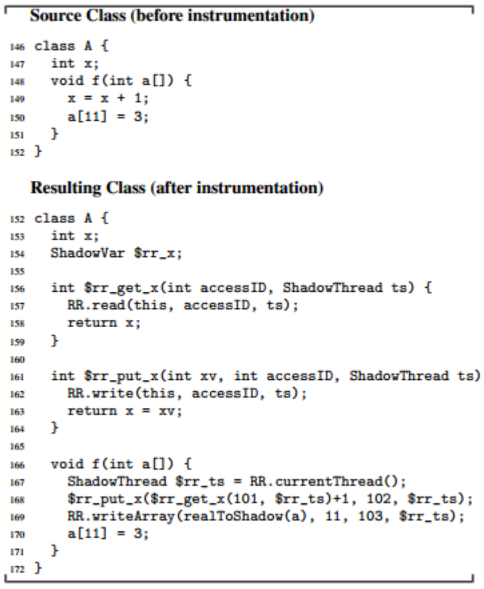
RoadRunner is a dynamic analysis framework for monitoring concurrent Java programs. It provides an interface for compiled Java bytecode instrumentation without requiring any changes to the underlying virtual machine. RoadRunner outputs updated class files that have additional function calls to the RoadRunner framework wrapped around each synchronization operation and memory access. When an instrumented class file is executed, it will be making calls to RoadRunner, which then broadcasts the events to any programs, such as FastTrack, that have subscribed to the event stream. These programs can then monitor and handle the events in whatever way they desire.

3.1 RoadRunner API

RoadRunner provides an abstract class, Tool, that allows different types of programs to be built on top of RoadRunner. Inside of this class, developers must specify how to handle different types of events that come from RoadRunner by overwriting virtual event handler functions; although a wide variety of events are broadcast, tools only need to write handlers for the events that they care about. FastTrack is an extension of the Tool class that monitors all memory operations and synchronization operations. The fastTrack event handlers use the events to construct a *happens-before* graph which is then used to monitor for data races.

3.2 Instrumentation

RoadRunner uses the ASM [1] library to instrument Java classes at load time from a compiled Java .class file. A shadow variable is created for each static member of the class, as well as getter and putter functions to access the variable. *ShadowThread* objects and function calls are inserted to the class to keep track of variable accesses between threads. An example instrumented class is below in Figure 1.



Source Class (before instrumentation)

```
class A {
  int x;
  void f(int a[]) {
    x = x + 1;
    a[11] = 3;
  }
}
```

Resulting Class (after instrumentation)

```
class A {
  int x;
  ShadowVar $rr_x;

  int $rr_get_x(int accessID, ShadowThread ts) {
    RR.read(this, accessID, ts);
    return x;
  }

  int $rr_put_x(int xv, int accessID, ShadowThread ts) {
    RR.write(this, accessID, ts);
    return x = xv;
  }

  void f(int a[]) {
    ShadowThread $rr_ts = RR.currentThread();
    $rr_put_x($rr_get_x(101, $rr_ts)+1, 102, $rr_ts);
    RR.writeArray(realToShadow(a), 11, 103, $rr_ts);
    a[11] = 3;
  }
}
```

Figure 1: An example of an instrumented class. A ShadowVar x is created along with its get and put functions. The current thread is stored in the ShadowThread and used in the access operations.

4. FASTTRACK

FastTrack is an extension of RoadRunner that uses a modified *happens-before* relationship to monitor concurrent programs. It monitors an event stream and reports if there are any race conditions. A *happens-before* relation is represented by vector clocks [5] which use considerable memory and require costly comparison operations. Given an application with n threads, each vector clock requires $O(n)$ storage space and each vector clock comparison operation requires $O(n)$ time. Previous tools have chosen between the precision (no false positives) and large overhead of full vector clocks and the imprecision of lightweight *happens-before* techniques which produces many false alarms. FastTrack leverages a new lightweight vector clock design, *epochs*, that, in most cases, uses constant time and space to provide accurate race detection and low overhead suitable for mobile environments.

4.1 Happens-Before Relationship

The *happens-before* relationship is an important relation in distributed and parallel systems. As defined by Leslie Lamport [4], it is a 'relation between the result of two events, such that if one event should happen before another event, the result must reflect that even if those events are in reality executed out of order'. If the happens-before relation is respected during execution, then a program has no data races.

4.2 FastTrack Algorithm

The key to the FastTrack algorithm's low overhead is the observation that most operations don't require a full vector clock to check for *happens-before* violations on an object, just the information about the last time the object was accessed. The algorithm stores an *epoch* for the last access in most cases, which is a tuple of the vector clock time (c) and thread (t) denoted *c@t.* The insight is that writes are ordered until there is a date race, so the algorithm just needs to check if a new read or write proceeds the last recorded *epoch.* Let *V1* and *V2* be the vector clocks of threads 1 and 2, respectively. Figure 2 shows a sample trace and how the epochs *Wx* and *Rx* are updated between two threads with vector clocks *C0* and *C1*.

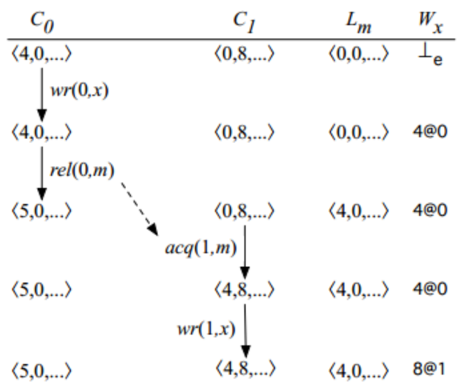


Figure 2: Sample Trace

Figures 3 and 4 shows how FastTrack handles reads and writes for different types of accesses, and the observed percentage of each operation type from several benchmarks. Since read operations aren't always ordered (one could finish after with a earlier clock time) and a program would still be correct, it is sometimes necessary to keep a full vector clock to make sure the very last read isn't changed by a write. The takeaway here is that less than 1% of reads and writes require operations on full vector clocks; the rest require only a comparison involving an epoch which runs in $O(1)$ time.

5. IMPLEMENTATION

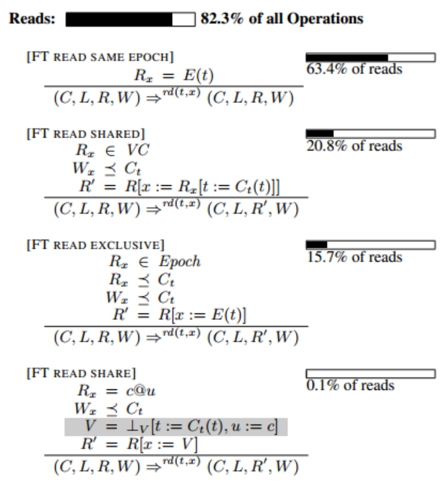

Reads: 82.3% of all Operations
[FT READ SAME EPOCH]
$\dfrac{R_x = E(t)}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R,W)}$
63.4% of reads
[FT READ SHARED]
$\dfrac{R_x \in VC \quad W_x \preceq C_t \quad R' = R[x := R_x[t := C_t(t)]]}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R',W)}$
20.8% of reads
[FT READ EXCLUSIVE]
$\dfrac{R_x \in Epoch \quad R_x \preceq C_t \quad W_x \preceq C_t \quad R' = R[x := E(t)]}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R',W)}$
15.7% of reads
[FT READ SHARE]
$\dfrac{R_x = c@u \quad W_x \preceq C_t \quad V = \bot_V[t := C_t(t), u := c] \quad R' = R[x := V]}{(C,L,R,W) \Rightarrow^{rd(t,x)} (C,L,R',W)}$
0.1% of reads

Figure 3: Read Accesses

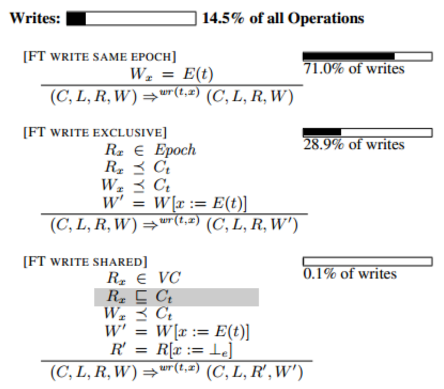

Writes: 14.5% of all Operations
[FT WRITE SAME EPOCH]
$\dfrac{W_x = E(t)}{(C,L,R,W) \Rightarrow^{wr(t,x)} (C,L,R,W)}$
71.0% of writes
[FT WRITE EXCLUSIVE]
$\dfrac{R_x \in Epoch \quad R_x \preceq C_t \quad W_x \preceq C_t \quad W' = W[x := E(t)]}{(C,L,R,W) \Rightarrow^{wr(t,x)} (C,L,R,W')}$
28.9% of writes
[FT WRITE SHARED]
$\dfrac{R_x \in VC \quad R_x \sqsubseteq C_t \quad W_x \preceq C_t \quad W' = W[x := E(t)] \quad R' = R[x := \bot_e]}{(C,L,R,W) \Rightarrow^{wr(t,x)} (C,L,R',W')}$
0.1% of writes

Figure 4: Write Accesses

This section describes the changes we made to FastTrack and RoadRunner to work with Android applications on Android 2.1. Unfortunately, Android does not provide any libraries for bytecode instrumentation, which is absolutely necessary to run FastTrack. Because of this, we essentially split the original desktop implementation of FastTrack so that it runs in two separate phases:

1. Instrument .class files in Java Virtual Machine and store updated bytecode files
2. Run Android application on Dalvik Virtual Machine with updated bytecode files with FastTrack included in application as external library

The details are provided in the following sections.

5.1 Instrumentation

The java.lang.Instrument library is used by RoadRunner to instrument the target application's .class files [2]. When running RoadRunner on a desktop machine, a user specifies which tools they want to use, as well as the target application. For example, the line

```
rrrun -tool=FT test.Test
```

tells FastTrack (FT) to monitor the execution of class *Test* in the package *test*. As soon as RoadRunner and FastTrack have both been initialized, the .class file for the class *Test* is instrumented using the native Java Instrument library. During the instrumentation, a *metadata* file is also generated that records the static members fields and member functions for all of the target application's classes. This file is used later to look up class members as they are accessed while the application executes. As soon as the instrumentation has completed and the *metadata* file has been created, *Test* is executed.

As mentioned before, Android provides no libraries that support binary instrumentation. Because of this, we were forced to instrument the target Android application code in the Java Virtual Machine. Since the instrumented files would eventually be running inside an Android application, RoadRunner was modified to store the instrumented files into a temporary directory where they could be loaded back into the Android application that they belong to. This is phase 1 of our FastTrack implementation as described earlier.

5.2 Tool Startup Modifications

A minor problem arises from the way the original FastTrack tool was invoked; the sample *rrrun* command shows that the target class to run is passed in at runtime as a parameter. This model of execution does not translate well to Android applications. When users want to run an application (whether on a physical Android device or in an emulator), they simply click on the icon for that application and it gets loaded into the Dalvik Virtual Machine.

To stay consistent with our goal of creating a tool with seamless usability, we needed to restructure the boot sequence of FastTrack. Our new implementation of FastTrack automatically initializes itself the first time that it receives an event from the underlying RoadRunner framework. Since the application code has been instrumented, it contains calls to the FastTrack tool. If the tool receives a call before it has loaded its tool properties file and the application's *metadata* file, it will do so, and upon finishing, continue to handle the function call from the application.

5.3 Exposed Interface

Although our implementation of FastTrack is able to initialize itself as described in the previous section, it does require a small modification to the application's source code. We expose our entire FastTrack interface as a single function call that must be made by the Android application's top level class. It is defined as follows:

```
void initFTSession(InputStream propertyFileHandle,
                   InputStream metadataFileHandle);
```

Since the FastTrack library is included in the application as a .jar file that is compiled in the Java Virtual Machine, it does not have the ability to open resources from the *assets* folder of the target Android application, which is where the tool property and *metadata* files are stored. The user is required to simply created *InputStream* objects to each of these two files and pass them to FastTrack. Although a completely transparent implementation would have been preferred, this single function call is much simpler than other race detector tools that require extensive updates to an application's source code.

5.4 Integration with Android Application

In order to run an Android application with our implementation of FastTrack, there are two phases as mentioned before. The first phase handles the bytecode instrumentation. Since this is done in the Java Virtual Machine (due to the dependency on java.lang.Instrument), the application's source code must be compiled with the Java compiler *javac*; once this has been done, FastTrack can be invoked with the same command that the original implementation used. However, our FastTrack tool will write out the newly instrumented .class files to a temporary directory that also contains the compiled FastTrack source.

Once the instrumentation step is done, this directory is sanitized and compressed to a .jar file. Sanitation refers to the removal of classes that are not needed in our FastTrack implementation in order to minimize its size in memory. The .jar file can then be added to the Android application with a simple *Import* operation in the Eclipse IDE. When the application is run, the compiled source code contained in the .jar file will be executed, which includes calls to the FastTrack library. This is considered phase two of our FastTrack implementation; at this point, if any data races are detected, error messages are printed to Eclipse's Logcat console, which can be dumped to a text file for preservation.

6. EVALUATION

This project started out as a proof-of-concept that FastTrack could be ported to run on Android's Dalvik Virtual Machine. After we had completed our implementation, we were able to run a set of multithreaded test cases on an Android emulator. Since we had written the test cases manually, it was known which tests contained legitimate data races. Our FastTrack implementation was able to correctly identify such data races and printed the appropriate error messages.

6.1 Resource Usage

To supplement our evaluation of FastTrack's correctness, we also observed the amount of computing resources that were required by the race detector. There was a risk of too much memory being consumed by the application's source code since instrumentation adds a considerable amount of shadow variables and member functions to each file. However, we noticed that instrumented files were only, on average, 30% larger than their original version. Additionally, we package up the instrumented source as a .jar file, which involves a compression algorithm. We noticed that file sizes decreased to up to 40% of their original size when being added to the .jar file. This effectively cancels out any memory overhead due to instrumentation.

A major concern with many race detectors is the slowdown incurred by the race detection algorithm running in the background. FastTrack employs *epochs* to help reduce this overhead, which makes a significant improvement compared to tools such as Eraser than can slowdown up to 100% [8]. We noticed slowdowns around 10%, which is consistent to the performance overhead of the original FastTrack system running on a desktop system [3].

6.2 Usability

A significant limit of our system is the inability to instrument code that uses Android libraries. Since the instrumentation happens in the Java Virtual Machine, any code that cannot be compiled with *javac* cannot be instrumented. Therefore, our implemented of FastTrack will only be able to detect data races in the files of an application that contain exclusively native Java code.

Although we have proved the concept that FastTrack can be applied to the Android mobile operating system, it is not yet a complete data race detector. Before a fully robust race detector can be written for Android, a instrumentation class must be added to Android that allows the instrumentation of .dex files (compiled Java code format used on the Dalvik VM).

7. CONCLUSION

We believe that the FastTrack algorithm is a good fit for mobile systems, but the instrumentation system of RoadRunner and the lack of instrumentation in Android limits its effectiveness. If the RoadRunner system was rewritten to always statically load classes or an instrumentation system was made for Android, then more apps could be monitored.

8. REFERENCES

[1] Bruneton, E., Lenglet, R., and Coupaye, T. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems* (2002).

[2] Flanagan, C., and Freund, S. N. Fasttrack: efficient and precise dynamic race detection. *SIGPLAN Not. 44*, 6 (June 2009), 121–133.

[3] Flanagan, C., and Freund, S. N. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2010), PASTE '10, ACM, pp. 1–8.

[4] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[5] Mattern, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms* (1988).

[6] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 267–280.

[7] Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not. 44*, 3 (Mar. 2009), 97–108.

[8] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (Nov. 1997), 391–411.

[9] Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P. M., Flinn, J., and Narayanasamy, S. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 3:1–3:24.