Studying Behaviour of MultiThreaded Programs Using Active Testing

amit nagarkoti
amitn@iitk.ac.in
gaurav mamgain
mamgain@iitk.ac.in

Abstract

In this project we studied the behaviour of multi threaded programs under a randomized scheduler with a bound on the number context switches. Here instead of using a model checking framework which tries to explore all possible program states iteratively we will be using a technique called Active Testing. Active Testing combined with iterative context bounding is highly efficient as it imposes a polynomial bound on the number of executions w.r.t. context bound c. We consider only preemptive context switches and will use a simple dynamic race detection algorithm to find scheduling points.

This project uses a tool called CalFuzzer, developed by University of California, Berkeley and is based on the principle of active testing. Main problem with techniques used for concurrency testing is resource limitations and the inability to reproduce the erroneous executions with certain guarantee. This project tries to target both of them as number of executions are polynomial in context bound c. Next we give some examples which will show certain errors can only be discovered when the bound is above a certain value. Finally we will give some experimental results which shows behaviour of example programs using this methodology.

I. Definitions

First we give some definitions used to define the states of a concurrent program and state evolution during an execution.

State (S) :- The state of a multithreaded program is defined by the current Register values, Heap Memory, Global Variables, Stack memory along with the value of program counter. A program goes from one state to another during an execution.

I. Events and Execution

An execution of a program can be seen as a sequence of events and an event is defined by three actions

• MEM(σ, m, a, t, L) :- denotes that thread t performed an access a ∈ {WRITE, READ} to memory location m while holding the set of locks L and executing the statement σ.
• SND(g, t) denotes the sending of a message with unique id g by thread t.
• RCV(g, t) denotes the reception of a message with unique id g by thread t.

II. Happens-Before Relation

The happens-before relation is defined\(^1\) by the nature of events and their order during an execution of a concurrent program and is the smallest relation satisfying following conditions :-

• If \(e_i\) and \(e_j\) are events from the same thread and \(e_i\) comes before \(e_j\) in the sequence \(⟨e_1⟩\), then \(e_i \prec e_j\).
• If \(e_i\) is the sending of the message g and \(e_j\) is the reception of the message g, then \(e_i \prec e_j\).
• \(\prec\) is transitively closed.

II. Common methods for concurrency testing

Brief overview of methods used for concurrency testing

\(^1\)These are definitions as given in reference 1 by K.Sen
• **Stress Testing** :- Run a program for a sufficiently long time in hope of actually finding a real bug, this method may consume a lot of time and resources and still fail to actually detect a buggy interleaving.

• **Systematic Scheduling** :- Controlling scheduler to generate a particular interleaving.

• **Depth bounding** :- Limit the number of steps in program execution, this generally doesn’t work well for multithreaded programs as execution itself is non deterministic.

• **Randomized scheduling** :- It involves introducing random delays, or context switches or priority inversions so that certain executions are more favourable.

### III. Active Testing

CalFuzzer the framework used in this project is based on the principle of active testing. For our analysis we will be using a simple race detection algorithm. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. CalFuzzer can be extended to implement other analysis be it for atomicity or deadlocks but here we will only be focussing on simple race detection.

In the first phase an imprecise analysis is done which identifies possible statements with bugs in the program. This analysis is based on certain heuristics which involves for example monitoring how a memory location is being accessed. In the final step this information is written to a file which is used in the next phase of active testing.

According to the above algorithm a race condition occurs when two threads tries to access the same memory location and using separate locks and such that they do not follow the happens-before relation. Two accesses to same memory location are concurrent if they do not follow the happens before relation, this tracking can be done using Vector Clocks as is used in distributed systems when a lack of causality implies concurrency.

In the second phase CalFuzzer reads the data from file and uses that information to direct scheduling in such a way that the possible racing statements pairs can be brought next to each other. Its a dynamic analysis technique depending on the actual execution so the results are precise. Main advantage of this technique is it gives an idea to control the scheduler behaviour which results in kind of partial order reduction.

Consider the following example if we run this simple race detection algorithm on it we will get a set of pairs of statements which are in race with each other.

```java
int x, y, z;

Thread threadB = new Thread () {
    public void run() {
        int a;
        x = 2;
        a = 3;
        y = 2;
        a = 6;
    }
};

Thread threadA = new Thread () {
    public void run() {
        x = 3;
        y = 3;
    }
};
```

<table>
<thead>
<tr>
<th>Table 1: A Simple Race Detection Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_i = \text{MEM}(\sigma_i, m_i, a_i, t_i, L_i)$ $\land$</td>
</tr>
<tr>
<td>$e_j = \text{MEM}(\sigma_j, m_j, a_j, t_j, L_j)$ $\land$</td>
</tr>
<tr>
<td>$t_i \neq t_j \land m_i = m_j$ $\land$</td>
</tr>
<tr>
<td>$(a_i = \text{WRITE} \lor a_j = \text{WRITE})$ $\land$</td>
</tr>
<tr>
<td>$L_i \cap L_j = \emptyset$ $\land$</td>
</tr>
<tr>
<td>$\neg(e_i &lt; e_j) \land \neg(e_j &lt; e_i)$</td>
</tr>
</tbody>
</table>
Table 2: Racing statements in above program

<table>
<thead>
<tr>
<th>Racing pairs</th>
<th>statement 1</th>
<th>statement 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestRace.java#6</td>
<td>TestRace.java#14</td>
<td></td>
</tr>
<tr>
<td>TestRace.java#8</td>
<td>TestRace.java#15</td>
<td></td>
</tr>
</tbody>
</table>

The way to actually implement this in cal-fuzzer is to define some static methods which are then added just before the required program points. A simple such instrumented function may look like:

```java
public void readBefore (Integer iid, Integer thread, Long memory, boolean isVolatile) {
    synchronized (ActiveChecker.lock) {
        eb.addEvent (iid, thread, memory, true, vcTracker.getVectorClock (thread));
        eb.checkRace (iid, thread, memory, true, vcTracker.getVectorClock (thread));
    }
}
```

This says just before reading a variable identified by Long memory in the thread identified by its integer ID thread, save this event and at the same time check for a possible race using the algorithm defined earlier.

The phase two of active testing can identify between a real race and a false warning a simple example illustrates the fact. This is so because in a real execution of this program the reported conflicting statements can never be brought together.

```java
int x, y, z;
Thread threadB = new Thread () {
    public void run () {
        int a;
        x = 2;
        a = 3;
        y = 2;
        a = 6;
    }
};
Thread threadA = new Thread () {
    public void run () {
        x = 3;
        if (x < 2)
            y = 3;
    }
};
```

In this example phase one will report line number #8 and #16 as a racing pair but a real execution will never be able to bring these statements together as if condition in #15 is always false.

IV. Iterative Context Bounding

This paper introduced the idea of iterative context bounding to explore the possible program states in a multithreaded program. By iterative context bounding we mean start execution with bound 0 and after running your program for a certain number of iterations increase the bound to 1 and so on. With each execution capture the statements execution order observed. We bound only preemptive switches that can take place in an execution, context switches due to a process blocking on a resource are not considered. The main idea here is to study the program behaviour as the context bound is increased by exploring new execution orders produced by different inter-leavings of threads in the process.

Unlike depth based search strategies where there is a limit on how far the execution can go, in context bounding the program is free to go deeper into program states. Even bound of zero may lead to a program termination.

2References to reference 2 by Madan Musuvathi, Shaz Qadeer
Another advantage of limiting the number of context switches is that the possible thread inter-leavings become polynomial in number of steps taken by a thread w.r.t. the current bound value say $c$. A simple proof follows as:-

Consider a program which has $n$ threads and suppose each thread executes $k$ steps each. Provided a random scheduler with no bounds on the number of switches there will be an exponential number of possible program executions given by

$$\frac{(nk)!}{k!^n} \geq (n!)^k \tag{1}$$

This bound is certainly exponential in both $n$ and $k$. So the possibility of exploring orderings in multithreaded program is a very hard task. Now lets see how applying a context bound $c$ simplifies the order of above equation. With a bound $c$ the possible inter-leavings will be given by :-

$$\binom{n}{k}(n+c)! \tag{2}$$

the term $n+c$ is due to the fact that termination of a thread is equivalent to a blocking call. Now incase $c \ll n$ and $c \ll k$ the above equation further reduces to

$$(n^2k)^c \tag{3}$$

a much weaker bound when $c$ is small. So this makes exploring executions of a multithreaded program much easier. As scheduling at every step can still lead to large number of executions a simple heuristic is to add a scheduling decision only at a program points where a synchronization or shared variables are being accessed and these points were provided by the phase one of active testing. The reasoning is that any interleaving of program statements which say are not accessing a shared variable for ex in our simple race detection are not important and such do not provide any insight to the overall program behaviour. A blocking decision can be added in calfuzzer much in the same way as was done in phase one say for example :-

```java
public void readBefore(Integer iid, Integer thread, Long memory, boolean isVolatile)
{
    synchronized (ActiveChecker.lock) {
        if (rand.nextBoolean())
            block(0);
        else
            ActiveChecker.executionOrder.add(idX);
    }
}
```

Here the scheduler on random decides to block a thread or let it continue. This random decision will introduce different program executions as presented in table below.

<table>
<thead>
<tr>
<th>Table 3: execution orders for a sample example with 25 iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence order</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>29 30 31 32</td>
</tr>
<tr>
<td>29 31 32 30</td>
</tr>
<tr>
<td>29 30 31 29</td>
</tr>
</tbody>
</table>

As we observe increasing context bounds makes it possible to discover new inter-leavings in during an execution. The numbers in above table are just IDs given to program statements during instrumentation phase. A mapping may look like
Table 4: IDs to statement mapping

<table>
<thead>
<tr>
<th>ID</th>
<th>statement #</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>TestRace.java#14</td>
</tr>
<tr>
<td>29</td>
<td>TestRace.java#20</td>
</tr>
<tr>
<td>31</td>
<td>TestRace.java#12</td>
</tr>
<tr>
<td>30</td>
<td>TestRace.java#21</td>
</tr>
</tbody>
</table>

V. IMPLEMENTATION

I. Framework and tools used

- **Apache Ant** :- A tool for automating software build processes and configuration tasks easier in Java.
- **CalFuzzer** :- The active testing framework developed at Berkeley. Provides facility to easily instrument java bytecode and APIs to implement an active testing algorithm for example active race testing as we used in our analysis.

II. Work Done

We implemented a SimpleRaceAnalysis as part of the imprecise phase one of active testing. To keep things simple we only kept track of memory access during any event and avoided checking for locks, the rest is same as algorithm given in table1. This provided us with the list of racing pairs which we then used with iterative context bounding to introduce custom breakpoints and study the program execution orderings possible. For a sample example we obtained the following results where the number of inter-leavings discovered increased with an increasing bound. We then used an already implemented algorithm in calfuzzer to get more precise racing pairs and for any sequence of feasible executions of these statements implemented a deterministic scheduler class which takes a feasible execution and simulates it. The execution sequences are generated using classes implemented for context bounding phase.

REFERENCES

