Race Directed Random Testing of Concurrent Programs

a paper by Koushik Sen

presented by Michail Famelis

December 4, 2008

RaceFuzzer is an algorithm for determining real races in concurrent programs that uses a randomized scheduler to create real race conditions.

Petter Solberg in the 2008 WRC Rally Acropolis *(photo by greekadman, some rights reserved)*

The problem

Finding bugs caused by data races.

- ... but existing techniques are either imprecise (static, dynamic and hybrid race detectors)
- ... and require manual inspection
- ... or are precise but cannot predict potential races (*happens-before* based detectors)

Proposed solution

Combine race detection with a randomized thread scheduler:

RaceFuzzer

RaceFuzzer controls a random scheduler based on the results of a race detector, to create real race conditions, and then resolve them randomly at runtime.

Some definitions

- **Enabled** thread: a thread that is not waiting to get a lock held by another thread
- **Alive** thread: a thread that has not yet terminated its execution

For a state s, if:

- Enabled(s) is the set of all enabled threads in s
- Alive(s) is the set of all alive threads in s

then:

- $(Enabled(s) = \emptyset \wedge Alive(s) \neq \emptyset) \rightarrow Deadlock$

...and a few more

$\texttt{MEM}(\sigma, m, \alpha, t, L)$ predicate

- thread t executes statement σ
- ...performing a memory access $\alpha \in \{\text{WRITE, READ}\}$
- ...to a memory location m
- ...while holding the set of locks L

happens-before relation

- if e_i, e_j in same thread: $e_i \prec e_j$
- if e_i sends message g and e_j receives it: $e_i \prec e_j$
- $\prec$ is transitively closed

Phase I: Hybrid Race Detection

For each pair of events (e_i, e_j), check the conjuction of:

- $e_i = \text{MEM}(\sigma_i, m_i, \alpha_i, t_i, L_i)$
- $e_j = \text{MEM}(\sigma_j, m_j, \alpha_j, t_j, L_j)$
- $t_i \neq t_j \wedge m_i = m_j$
- $\alpha_i = \text{WRITE} \vee \alpha_j = \text{WRITE}$
- $L_i \cap L_j = \emptyset$
- $\neg(e_i \prec e_j) \wedge \neg(e_j \prec e_i)$

If the condition holds, (σ_i, σ_j) is a racing pair.

Phase II: the 'Fuzzer

Algorithm 1 Algorithm RACEFUZZER

```
1:  Inputs: the initial state s_0, a set of two racing statements RaceSet
2:  s := s_0
3:  postponed := ∅
4:  while Enabled(s) ≠ ∅ do
5:      t := a random thread in Enabled(s) \ postponed
6:      if NextStmt(s, t) ∈ RaceSet then
7:          R := Racing(s, t, postponed)
8:          if R ≠ ∅ then /* Actual race detected */
9:              print "ERROR: actual race found"
10:             /* Randomly resolve race */
11:             if random boolean then
12:                 s := Execute(s, t)
13:             else
14:                 postponed := postponed ∪ {t}
15:                 for all t' ∈ R do
16:                     s := Execute(s, t')
17:                     postponed := postponed \ {t'}
18:                 end for
19:             end if
20:         else /* Wait for a race to happen */
21:             postponed := postponed ∪ {t}
22:         end if
23:     else
24:         s := Execute(s, t)
25:     end if
26:     if postponed = Enabled(s) then
27:         remove a random element from postponed
28:     end if
29: end while
30: if Active(s) ≠ ∅ then
31:     print "ERROR: actual deadlock found"
32: end if
```

Algorithm 2 Function $\mathbf{Racing}(s, t, postponed)$

```
1: Inputs: program state s, thread t, and set postponed
2: return {t' | t' ∈ postponed s.t. NextStmt(s, t) and
   NextStmt(s, t') access the same memory location and
   at least one of the accesses is a write}
```

...in other words

For a given racing pair, execute the threads in a random schedule

- Whenever a thread is about to execute one of the racing statements, the scheduler **postpones** it
- The execution stays postponed, until some other thread also tries to execute one of racing statements and creates a race (ie accesses the same memory location and is a write)
- **Randomly resolve** the race: execute one and keep postponing the other

So we reported a real race and by random resolution we checked if something bad can happen

...in the meantime

- While postponing threads, we may happen to get several ones trying to execute a racing statement but are not racing as they access different dynamic shared memory locations.
- Those threads are postponed.
- If all threads are postponed, the scheduler randomly picks one to break the deadlock.

Other points of interest

Deadlocks

- RaceFuzzer keeps looping until there is no *enabled* thread
- If at termination there exist *active* threads: **deadlock**.

Replay of specific executions

- Using the same seed for random number generation
- Works because the RF scheduler allows only one thread to execute at a time
- ...and resolves non-determinism by picking randomly generated numbers

Example 1

```
Initially: x = y = z = 0;

thread1 {                    thread2 {
1:   x = 1;                  7:    z = 1;
2:   lock(L);                8:    lock(L);
3:   y = 1;                  9:    if (y==1) {
4:   unlock(L);              10:      if (x != 1){
                             11:        ERROR2;
5:   if (z==1)               12:      }
6:     ERROR1;               13:   }
}                            14:   unlock(L);
                             }
```

Racing pair (1, 10)

```
Initially: x = y = z = 0;

thread1 {
1:   x = 1;
2:   lock(L);
3:   y = 1;
4:   unlock(L);

5:   if (z==1)
6:     ERROR1;
}
```

```
thread2 {
7:    z = 1;
8:    lock(L);
9:    if (y==1) {
10:       if (x != 1){
11:           ERROR2;
12:       }
13:   }
14:   unlock(L);
}
```

- Hybrid detection would report this as a race
- ...but x is implicitly synchronized by y
- RaceFuzzer will not report this as a race

Racing pair (1, 10)

```
Initially: x = y = z = 0;

thread1 {                    thread2 {
1:   x = 1;                  7:    z = 1;
2:   lock(L);                8:    lock(L);
3:   y = 1;                  9:    if (y==1) {
4:   unlock(L);              10:       if (x != 1){
                             11:           ERROR2;
5:   if (z==1)               12:       }
6:     ERROR1;               13:   }
}                            14:   unlock(L);
                             }
```

- t2 cannot reach (10) first
- If t1 reaches (1) first it will delay until t2 reaches (10)
- ...t2 will terminate, so t1 will be resumed

Racing pair (5, 7)

```
Initially: x = y = z = 0;

thread1 {                    thread2 {
1:   x = 1;                  7:    z = 1;
2:   lock(L);                8:    lock(L);
3:   y = 1;                  9:    if (y==1) {
4:   unlock(L);              10:       if (x != 1){
                             11:          ERROR2;
5:   if (z==1)               12:       }
6:     ERROR1;               13:   }
}                            14:   unlock(L);
                             }
```

- If `t1` reaches (5) first it is postponed
- ...and `t2` reaches (7).
- ...and RaceFuzzer reports a real race

Racing pair (5, 7)

```
Initially: x = y = z = 0;

thread1 {                    thread2 {
1:   x = 1;                  7:    z = 1;
2:   lock(L);                8:    lock(L);
3:   y = 1;                  9:    if (y==1) {
4:   unlock(L);              10:       if (x != 1){
                             11:          ERROR2;
5:   if (z==1)               12:       }
6:     ERROR1;               13:   }
}                            14:   unlock(L);
                             }
```

- (5) or (7) is randomly chosen for execution
- ...so `ERROR1` is executed with probability 0.5
- The same pattern holds if `t2` reaches (7) first

Example 1

- RaceFuzzer does not create false warnings
- It creates multiple scenarios that illustrate the race
- One such scenario shows the reachability of `ERROR1`
- Scenarios are replayable, by the appropriate random seed.

Example 2

```
Initially: x = 0;

thread1 {                    thread2 {
1.   lock(L);                10.    x = 1;
2.   f1();                   11.    lock(L);
3.   f2();                   12.    f6();
4.   f3();                   13.    unlock(L);
5.   f4();                   }
6.   f5();
7.   unlock(L);
8.   if (x==0)
9.     ERROR;
}
```

Example 2

```
Initially: x = 0;

thread1 {                    thread2 {
1.   lock(L);                10.    x = 1;
2.   f1();                   11.    lock(L);
3.   f2();                   12.    f6();
4.   f3();                   13.    unlock(L);
5.   f4();                   }
6.   f5();
7.   unlock(L);
8.   if (x==0)
9.     ERROR;
}
```

Example 2

- With the default scheduler the probability of (8) and (10) happening temporally near is very small
- With high probability (8) and (10) would be separated by the acquisition and release of lock L
- ...so a *happens-before* detector would not detect this with high probability
- Also, `ERROR` has low probability of ever being reached
- These probabilities depend heavily on the number of statements before (8)

Example 2

```
Initially: x = 0;

thread1 {                    thread2 {
1.   lock(L);                10.    x = 1;
2.   f1();                   11.    lock(L);
3.   f2();                   12.    f6();
4.   f3();                   13.    unlock(L);
5.   f4();                   }
6.   f5();
7.   unlock(L);
8.   if (x==0)
9.     ERROR;
}
```

- RaceFuzzer will create the race condition with probability 1
- ...and will resolve the case and reach `ERROR` with probability 0.5

Some implementation details

- Implemented for Java
- Hybrid algorithm not optimized (not an objective)
- Works with Java bytecode
  - Cannot track acquire/release of locks in native code
  - Can go to deadlock inside native code
  - Use of monitor thread
- Can go into livelocks (ie all threads disabled except one that does not care to synchronize with any other thread)
  - Use of monitor thread

Experiments

RaceFuzzer was evaluated on many Java benchmark programs. The inescapable table of results:

Program Name	SLOC	Average Runtime in sec.			# of Races			# of Exceptions		Probability of hitting a race
		Normal	Hybrid	RF	Hybrid	RF (real)	known	RF	Simple	
moldyn	1,352	2.07	> 3600	42.37	59	2	0	0	0	1.00
raytracer	1,924	3.25	> 3600	3.81	2	2	2	0	0	1.00
montecarlo	3,619	3.48	> 3600	6.44	5	1	1	0	0	1.00
cache4j	3,897	2.19	4.26	2.61	18	2	-	1	0	1.00
sor	17,689	0.16	0.35	0.23	8	0	0	0	0	-
hedc	29,948	1.10	1.35	1.11	9	1	1	1	0	0.86
weblech	35,175	0.91	1.92	1.36	27	2	1	1	1	0.83
jspider	64,933	4.79	4.88	4.81	29	0	-	0	0	-
jigsaw	381,348	-	-	0.81	547	36	-	0	0	0.90
vector 1.1	709	0.11	0.25	0.2	9	9	9	0	0	0.94
LinkedList	5979	0.16	0.26	0.22	12	12	-	5	0	0.85
ArrayList	5866	0.16	0.26	0.24	14	7	-	7	0	0.55
HashSet	7086	0.16	0.26	0.25	11	11	-	8	1	0.54
TreeSet	7532	0.17	0.26	0.24	13	8	-	8	1	0.41

Notes on experimental results

- RaceFuzzer is a tool for testing and debugging, so runtimes are not *too* important
- It is demonstrated that RF only reports real races
  - For `moldyn` 2 new -but benign- races were discovered
- RaceFuzzer is far more effective in discovering insidious errors than when only JVM's default scheduler is used
- A number of previously unknown uncaught exceptions were discovered, in JDK1.4.2 classes `LinkedList`, `ArrayList`, `HashSet`, `TreeSet`

Why RaceFuzzer rocks

Classifies real races from false alarms

- Actively controls a randomized scheduler and creates real races with high probability
- Automatically separates real races from false alarms
- ...which would otherwise be done manually

Provides for easy replication of races

- The execution can be replayed using the same random seed
- ...it therefore requires no recording and is lightweight
- Replay useful for debugging real races

Why RaceFuzzer rocks

Separates (some) harmful races from benign ones

- Randomly resolves a race
- Races that lead to errors get detected

Gives no false warnings

- Creates actual racing conditions by bringing racing events temporally near

"Embarrassingly" (actual quote) parallel

- Different invocations for different racing pairs are independent
- Performance can be increased linearly with more processors

My take

- Simple enough idea, great benefits
- It does rock
- A point: as it resolves randomly a race, many runs of the same test should be done so as to maximize the chance that a potential error state is reachable

Questions?

(photo by thanos tsimekas, some rights reserved)