Predicting Data Races from Program Traces

Luis M. Carril
Concurrency & debugging

- Concurrency programming is hard
  - Non-determinism
  - Multiple control flows

- New types of errors: **data races**, deadlocks, atomicity violations…

- Non-determinism makes debugging a difficult task
  - Probe effect [Gait86]
  - Developer cannot reproduce result of analysis tool
Approach

- Predict errors from a single execution:
  - Infer alternative interleavings from an observed execution
  - Find errors in this set of interleavings
  - Produce a history of the race to enable deterministic replay
Example – Captured trace

Thread 1
1: write (y)
2: lock (m)
3: write (x)
4: unlock (m)
5: 
6: 
7: 
8: 

Thread 2

lock (m)
write (x)
unlock (m)
read (y)
Predict

- Encode trace as a process in a process algebra (CSP)
  - Process represent alternative reorderings of the trace

- Define data race patterns in CSP terms
  - Patterns: read-write / write-write

- Is any of the data race patterns possible in the process?
Example model

THREAD1 = write.t1.y → lock.t1.m → write.t1.x → unlock.t1.m → SKIP

THREAD2 = lock.t2.m → write.t2.x → unlock.t2.m → read.t2.y → SKIP

THREAD_INTERLEAVING = THREAD1 ||| THREAD2

MUTEX(i) = lock.t1.i → unlock.t1.i → MUTEX(i)
   □ lock.t2.i → unlock.t2.i → MUTEX(i)

PROGRAM = THREAD_INTERLEAVING ||\{lock,unlock\} MUTEX(m)
Alternative traces

Program Process

write (t1,y)
lock (t1,m)
write (t1,x)
unlock (t1,m)
read (t2,y)

write (t2,m)
write (t2,x)
unlock (t2,m)
read (t2,y)

write (t1,y)
lock (t2,m)
write (t2,x)
unlock (t2,m)
read (t2,y)

write (t1,y)
lock (t1,m)
write (t1,x)
unlock (t1,m)
read (t2,y)

write (t1,y)
lock (t2,m)
write (t2,x)
unlock (t2,m)
read (t2,y)

write (t1,y)
lock (t1,m)
write (t1,x)
unlock (t1,m)
read (t2,y)
Other synchronization constructs

FORK2 = fork.t1.t2 → start.t2 → SKIP

JOIN2 = end.t1.t2 → join.t1.t2 → SKIP

SIGNAL_C = signal.t2.c → wait.t1.c → SKIP

BARRIER_B = barrier_enter.t1.b → barrier enter.t2.b
           → barrier_exit.t1.b → barrier_exit.t2.b → SKIP
Race detection

- Refinement relationship

\[ \text{SPEC} \sqsubseteq \text{IMPL} \iff \text{behavior(IMPL)} \sqsubseteq \text{behavior(SPEC)} \]

\[ \text{STOP} \sqsubseteq_T (\text{PROGRAM} \parallel \text{RACE(y)}) \setminus (\text{AllEvents} - \{\text{race}\}) \]

- If the event **race** is reachable, then we have a data race

- One refinement check per shared variable (not per racy-pair) \( \Rightarrow \) FDR3 refinement checker
Race detection II

- Represents all read and write combinations between the two threads on shared element v

RACE_ERR(v) = read.t1.v → write.t2.v → race → STOP
  □ write.t1.v → read.t2.v → race → STOP
  □ write.t1.v → write.t2.v → race → STOP
  □ read.t2.v → write.t1.v → race → STOP
  □ write.t2.v → read.t1.v → race → STOP
  □ write.t2.v → write.t1.v → race → STOP

RACE(v) = RACE_ERR(v) ∆ (□ x:sync_ops@x → RACE(v))
Counterexample

Race found on $y$, with counterexample:
$$\text{lock(t2,m) } \rightarrow \text{unlock(t2,m)}$$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>lock (m)</td>
</tr>
<tr>
<td>3:</td>
<td>write (x)</td>
</tr>
<tr>
<td>4:</td>
<td>unlock (m)</td>
</tr>
<tr>
<td>5: write (y)</td>
<td></td>
</tr>
<tr>
<td>6: lock (m)</td>
<td></td>
</tr>
<tr>
<td>7: write (x)</td>
<td></td>
</tr>
<tr>
<td>8: unlock (m)</td>
<td></td>
</tr>
</tbody>
</table>
Replay & confirmation

- Enables coarse replay of the program
  - only enforcement of synchronization operations order
  - other operations still happen in parallel

- Deterministic execution until error point, non-deterministic afterwards

- Simultaneous execution of a happens-before detector
  - confirms the data race
  - provides more detailed information: source lines, stack...

- Debugging does not alter the replay
**Target & implementation**

- **Target:** C programs with pthreads
- **Tracing and replay in LLVM**
  - Instrumentation of pthread calls and memory accesses
  - Instrumentation of pthread_wait loops
- **Trace reduction:**
  - Variable grouping as single shared variables (online and offline)
  - Filtering using relaxed happens-before & lockset
- **Scalability:**
  - Trace windowing -> inter-window false negatives
# Application benchmark

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Counterexamples</th>
<th>Confirmed errors</th>
<th>TSan x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>aget</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>blackscholes</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>boundedBuffer</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ctrace</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>fft</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fmm</td>
<td>190</td>
<td>50</td>
<td>36</td>
</tr>
<tr>
<td>lu</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lu-non</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>qsort</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>streamcluster</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>water-nsquared</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Conclusion

- Data race prediction
  - modelled in CSP to observe alternative interleavings
  - reduced timing effects on detection

- Error witness generation
  - enables re-execution of data race prefix
  - reduction on debugging effort

- Finds more races than multiple re-execution of classical approaches