Karlsruhe Institute of Technology

Predicting Data Races from Program Traces

Luis M. Carril

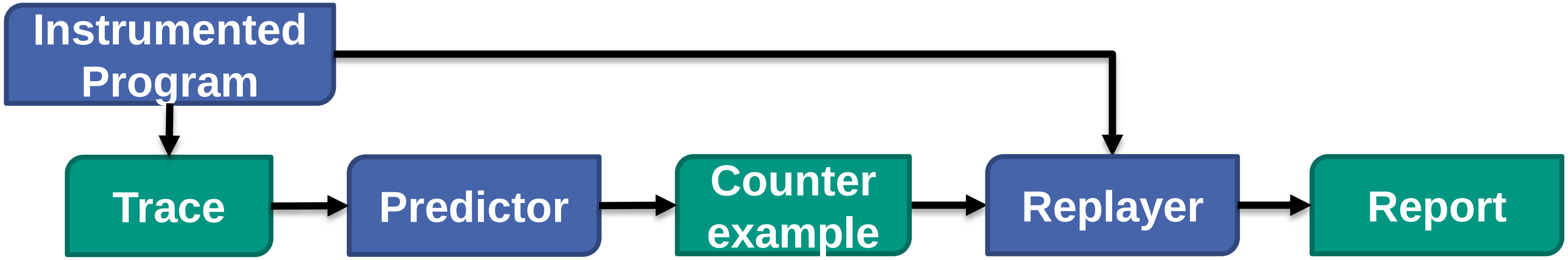
IPD Tichy – Lehrstuhl für Programmiersysteme

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

$$THREAD1 = write.t1.y \rightarrow lock.t1.m \rightarrow write.t1.x \rightarrow unlock.t1.m \rightarrow SKIP$$

$$THREAD2 = lock.t2.m \rightarrow write.t2.x \rightarrow unlock.t2.m \rightarrow read.t2.y \rightarrow SKIP$$

$$THREAD\_INTERLEAVING = THREAD1 \mid\mid\mid THREAD2$$

$$\begin{aligned} MUTEX(i) = {} & lock.t1.i \rightarrow unlock.t1.i \rightarrow MUTEX(i) \\ & \square\ lock.t2.i \rightarrow unlock.t2.i \rightarrow MUTEX(i) \end{aligned}$$

$$PROGRAM = THREAD\_INTERLEAVING \mid\mid_{\{lock,unlock\}} MUTEX(m)$$

Alternative traces

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

PROGRAM Process

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

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

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

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

lock (t2,m)
write (t2,x)
unlock (t2,m)
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

$$SPEC \sqsubseteq IMPL \leftrightarrow behavior(IMPL) \subseteq behavior(SPEC)$$

$$STOP \sqsubseteq_T (PROGRAM \mid\mid RACE(y)) \ \backslash \ (AllEvents - \{race\})$$

- If the event **race** is reachable, then we have a data race
- One refinement check per shared variable (not per racy-pair) => 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) Δ (□ x:sync_ops@x → RACE(v))

Counterexample

- Race found on **y**, with counterexample:

  lock(t2,m) → unlock(t2,m)

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

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

Scenarios	Counterexamples	Confirmed errors	TSan x100
aget	2	2	3
blackscholes	0	0	0
boundedBuffer	0	0	0
ctrace	4	3	2
fft	1	0	0
fmm	190	50	36
lu	0	0	0
lu-non	0	0	0
qsort	2	6	1
streamcluster	1	1	1
water-nsquared	0	0	0

Conclusion

Karlsruhe Institute of Technology

- 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