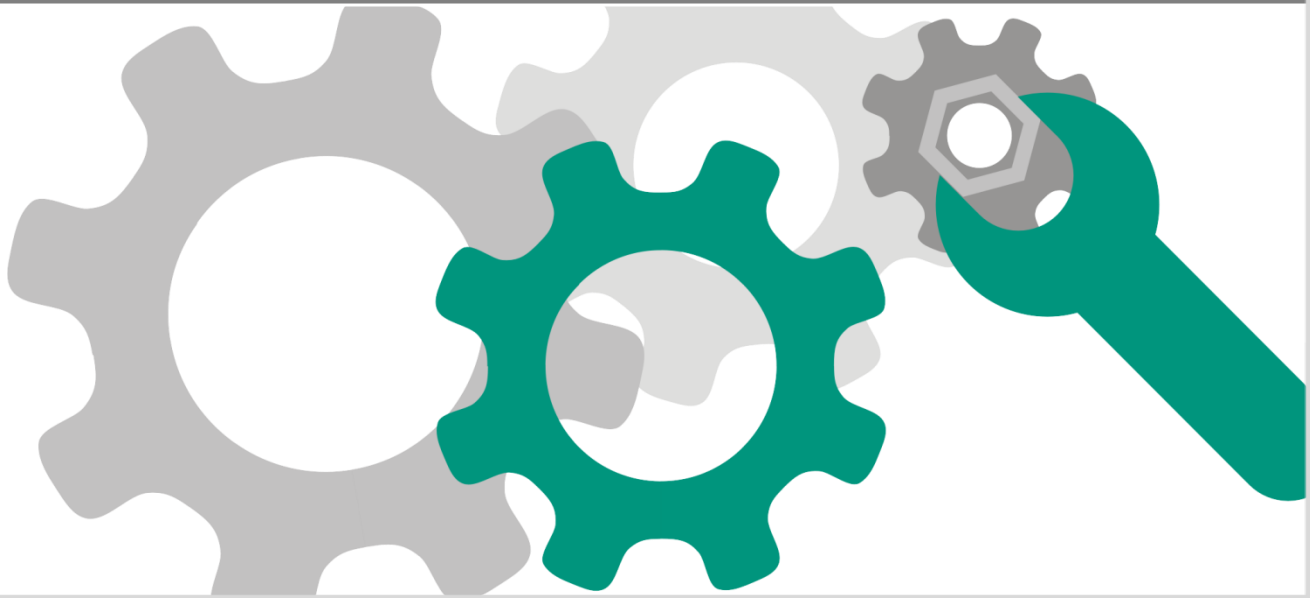
Karlsruhe Institute of Technology

Predicting and witnessing data races using CSP

Luis M. Carril
Walter F. Tichy

IPD Tichy – Lehrstuhl für Programmiersysteme



Concurrency & debugging

- Concurrency programming is hard
  - Non-determinism
  - Multiple control flows
- New types of errors: **data races**, deadlocks, atomicity violation…
- Non-determinism makes debugging a difficult task
  - Probe effect [Gait86]
  - Developer cannot reproduce result of analysis tool

Goals

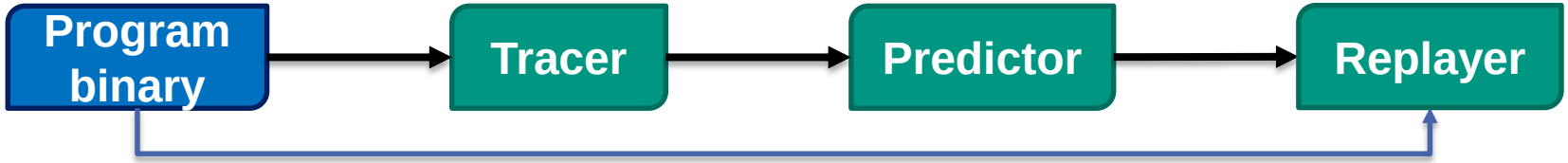
- Predict races from a single execution:
  - infer alternative interleavings from an observed execution
  - find races in this set of interleavings
- Reproduce data race:
  - produce a history of the race to enable deterministic replay

Data race

- A data race occurs when:
  - two threads access the same memory location concurrently, and
  - at least one is a write access, and
  - there is no explicit mechanism ordering the accesses.

- Typical approaches:
  - Lockset [Savage97]
  - Happens-before [Lamport78]

Approach



- Capture:
  - Execution trace from a single program execution
  - Trace model:
    - Memory operations: read and write
    - Synchronization operations:

      fork – start

      join – end

      lock – unlock

      signal – wait

      …

Example – Observed execution

	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
- Data race patterns defined as another CSP process
- Race detection: is any data race pattern possible in the trace process?

Communicating Sequential Processes

- Description and analysis of concurrent systems [Hoare78]
- Processes describe behavior of systems (upper case)
- The behavior is observed by the emission of atomic events (lower case)
- Operators for process composition: ||, |||, □ …

  e.g.: $P = a \rightarrow b \rightarrow STOP$ $\quad$ $Q = P \,\square\, c \rightarrow d \rightarrow STOP$

- Processes communicate trough synchronous event sharing

Generalizing trace - threads

- Map trace events to CSP events
- Each thread is encoded as an independent CSP process

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

- Combined using the interleaving operator

INTERLEAVINGS= THREAD1 ||| THREAD2

- INTERLAVINGS process contains all reorderings, including infeasible, e.g.: … lock.t1.m, lock.t2.m…

Generalizing trace - synchronization

- Synchronization constructs defined as additional processes

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

- Synchronization processes impose happens-before orderings between THREAD processes

  PROGRAM = INTERLEAVINGS $||_{\{sync\_ops\}}$ MUTEX(m)

- PROGRAM process has a more restricted set of behaviors

- Additional processes for: fork-start, end-join, signal-wait…

Example model

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

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

$$MUTEX(i) = lock.t1.i \rightarrow unlock.t1.i \rightarrow MUTEX(i)$$
$$\square\ lock.t2.i \rightarrow unlock.t2.i \rightarrow MUTEX(i)$$

$$PROGRAM = (THREAD1 \,|||\, THREAD2\ ) \,||_{\{sync\_ops\}}\, MUTEX(m)$$

Alternative traces

Karlsruhe Institute of Technology

```
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)
```

Data race patterns

- Represents all read-write combinations

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:reset_set@x → RACE(v))

Race detection

- Enforce process PROGRAM with RACE process, with interfaced parallel operator ||

  PROGRAM ||{} RACE(y)

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

- Refinement relationship

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

  $STOP \sqsubseteq_T (PROGRAM \parallel RACE(y)) \setminus \Sigma - \{race\}$

- One refinement check per shared variable (not per racy-pair) => FDR3 refinement checker

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

- Enables coarse replay of the program
  - only enforcement of synchronization operations order
- Race confirmation: simultaneous happens-before detector
- Debugging does not alter the replay

Preliminary Evaluation

Scenarios	Real races	Predicted Races x1	Helgrind x10
48	31	31	21

- Target: C programs with pthreads
- Tracing and replay implemented as Valgrind plugins

Conclusion

- Data race prediction
  - modelled in CSP to observe alternative interleavings
  - reduced timing effects on detection
- Data race witness generation
  - enables re-execution of data race prefix
  - reduction on debugging effort