

Testing Concurrent Programs on Relaxed Memory Models

Jacob Burnim, Koushik Sen, Christos Stergiou

Introduction

- Programmers assume sequential consistency
- High performance concurrent programs:
 - synchronization libraries
 - lock-free data structures
- Memory model related bugs

Relaxed Memory Models

$$x = y = 0$$

T1:	T2:
$x = 1$	$y = 1$
$a = y$	$b = x$

$$a = b = 0 ?$$

Goals

- Testing tool that finds memory model bugs
- Provide a trace of the buggy execution
- Distinguish harmful from benign sequential consistency violations
- Find bugs exhibited under rare conditions
- Work for different memory models

Our approach

- C/C++ programs using pthreads
- Operational semantics for memory models
- Simulate program under relaxed memory model
 - Random testing, no guarantees
 - Exhaustive search, a lot of non-determinism

Our approach

- Active Testing
 - Two phases:
 - Find potential sequential consistency violations
 - Direct testing using potential violations
 - Not random and scalable
- How to find potential violations?
- How to create the violations?

Sequential Consistency

- Trace is a sequence of loads and stores
- Program order $e_1 \rightarrow_p e_2$
 - same thread, e_1 issued before e_2
- Conflict order $e_1 \rightarrow_c e_2$:
 - same memory location, e_1 or e_2 is write
 - e_1 “happens before” e_2 from main memory perspective
- happens-before relation $\rightarrow_{hb} \stackrel{\text{def}}{=} \rightarrow_p \cup \rightarrow_c$
- A trace is sequentially consistent
iff \rightarrow_{hb} is acyclic

Technique Overview

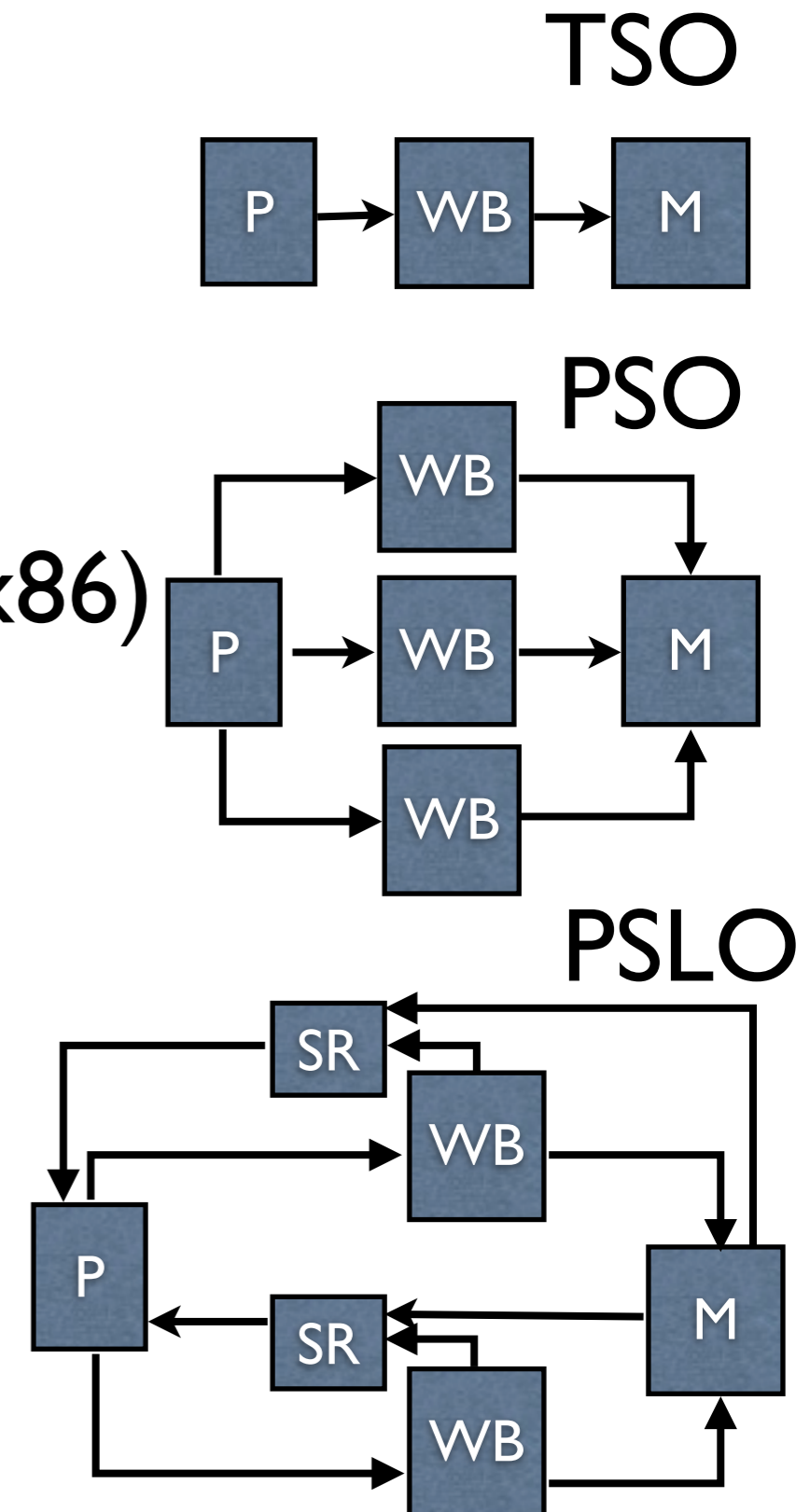
- Phase I
 - Execute program under sequential consistency
 - Find potential hb cycle: e_1, \dots, e_n, e_1 from trace
 - Real cycle : $e_1 \rightarrow_p e_2 \rightarrow_c e_3 \dots e_n \rightarrow_c e_1$
 - race edge: $e_1 \leftrightarrow_r e_2$: can be $e_1 \rightarrow_c e_2$ or $e_2 \rightarrow_c e_1$
 - Potential : $e_1 \rightarrow_p e_2 \leftrightarrow_r e_3 \rightarrow_p e_4 \leftrightarrow_r e_5 \dots e_n \leftrightarrow_r e_1$
 - Successive pairs (e_i, e_{i+1}) in cycle alternate between: program order & potential data races

Technique Overview

- Phase II
 - Execute program on relaxed memory model using biased random scheduler
 - $e_1 \rightarrow_p e_2 \leftrightarrow_r e_3 \rightarrow_p e_4 \leftrightarrow_r e_5 \dots e_n \leftrightarrow_r e_1$
 - Resolve (e_2, e_3) race as $e_2 \rightarrow_c e_3$
 - e_3 : delay execution
 - load \rightarrow pause thread
 - store \rightarrow buffer value
 - e_2 : execute quickly, commit immediately

Memory Models

- Our tool intercepts loads & stores
 - Can simulate any memory model with operational semantics
- TSO: total store ordering (SPARC, ~x86)
 - store-load reordering
- PSO: partial store ordering (SPARC)
 - TSO + store-store reordering
- PSLO: partial store load ordering
 - PSO + loads reordered before previous loads and stores



Example

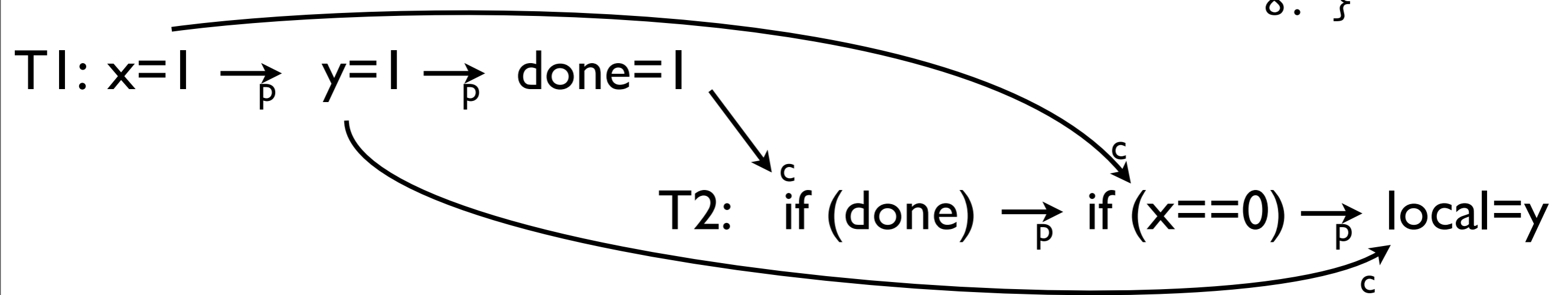
$x = y = \text{done} = 0$

```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```

```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

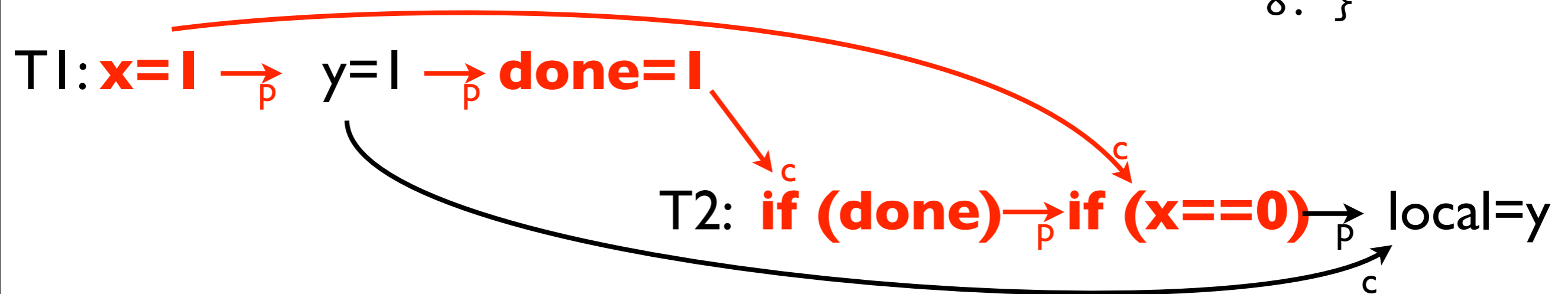
Example

```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;  
thread2 {  
4: if (done) {  
5:   if (x==0)  
6:     ERROR;  
7:   local = y;  
8: }
```



Example

```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;  
thread2 {  
4: if (done) {  
5:   if (x==0)  
6:     ERROR;  
7:   local = y;  
8: }
```

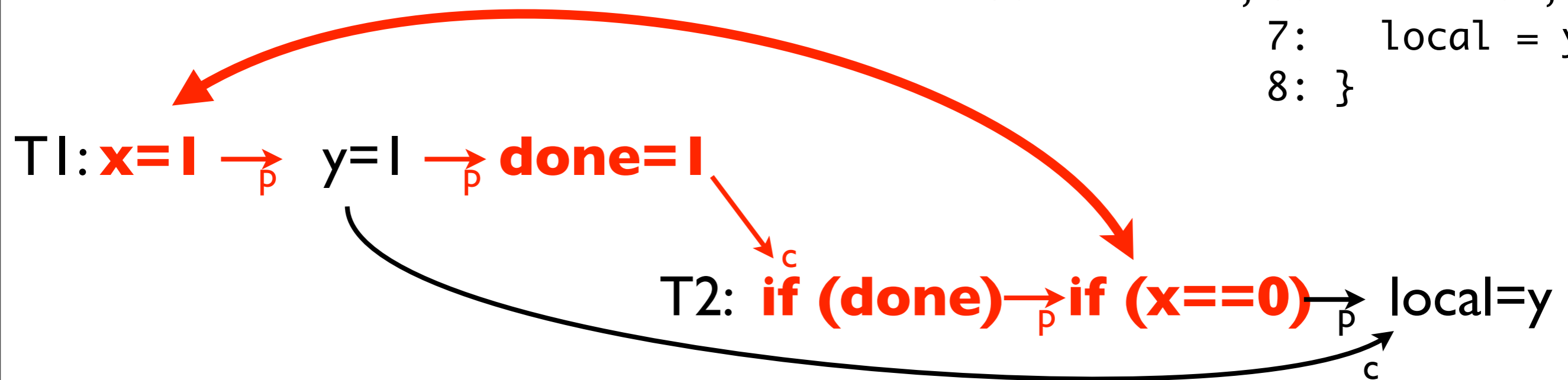


potential happens-before cycle
not real cycle in sequential consistent execution

Example

```
thread1 {  
  1: x = 1;  
  2: y = 1;  
  3: done = 1;  
thread2 {  
  4: if (done) {  
  5:   if (x==0)  
  6:     ERROR;  
  7:   local = y;  
  8: }
```

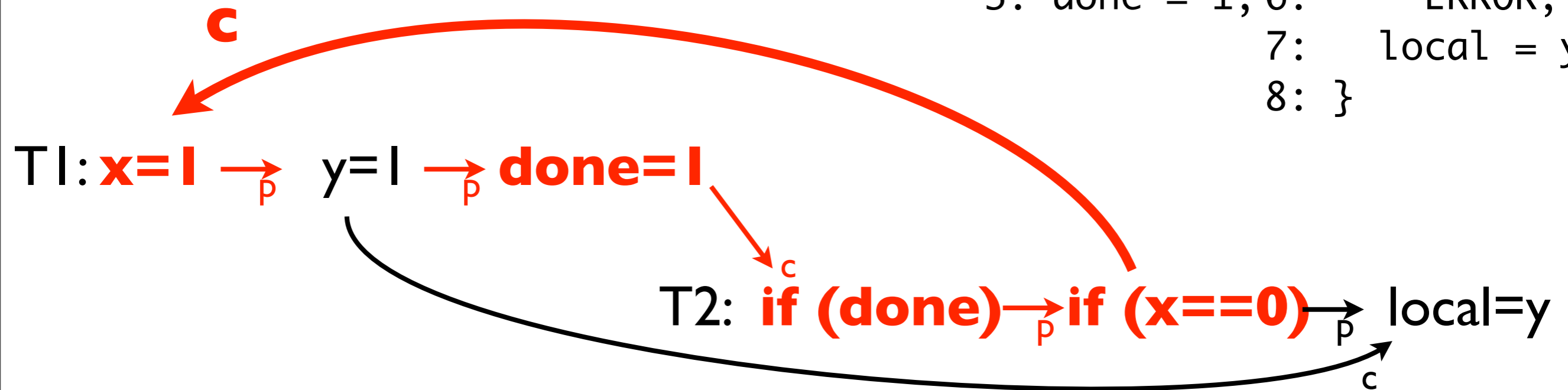
race edge



race edge can be ordered either way
depending on when x is committed

Example

```
thread1 {  
  1: x = 1;  
  2: y = 1;  
  3: done = 1;  
thread2 {  
  4: if (done) {  
  5:   if (x==0)  
  6:     ERROR;  
  7:   local = y;  
  8: }
```



race edge can be ordered either way
depending on when x is committed

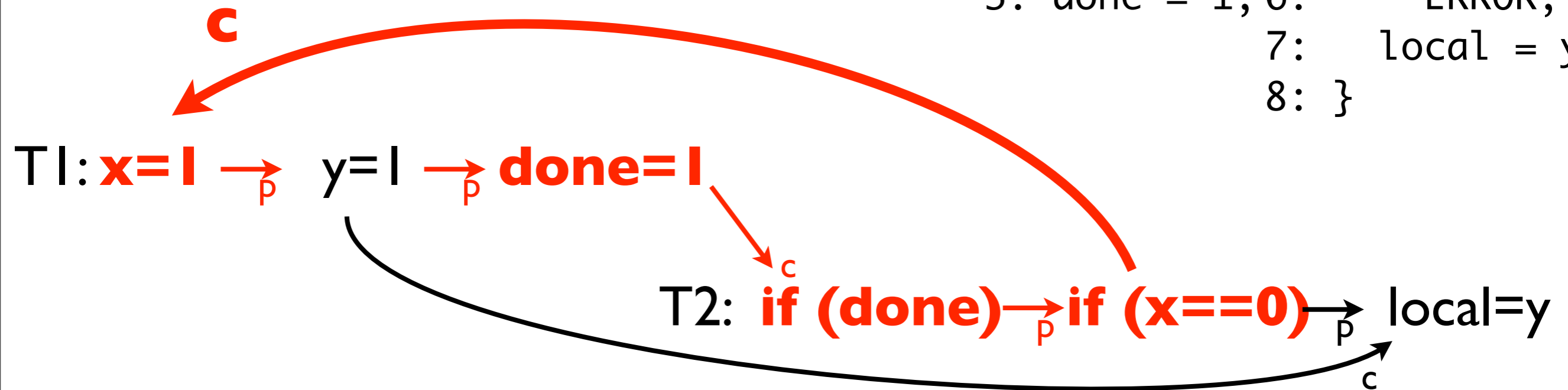
Example

```

thread1 {
1: x = 1;
2: y = 1;
3: done = 1;
}

thread2 {
4: if (done) {
5:   if (x==0)
6:     ERROR;
7:   local = y;
8: }

```



```

1: x=1;
3: done=1;
4: if (done) {
5: if (x==0) {

```

Example

thread1 {		thread2 {
1: x = 1;	➡	4: if (done) {
2: y = 1;		5: if (x==0)
3: done = 1;		6: ERROR;
		7: local = y;
		8: }

Potential:

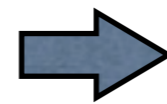
$I \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r I$ - **PSO**

Goal:

$I \rightarrow_p 3 \rightarrow_c 4 \rightarrow_p 5 \rightarrow_c I$

Example

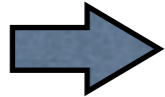
```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```



```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

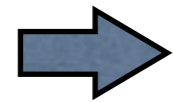
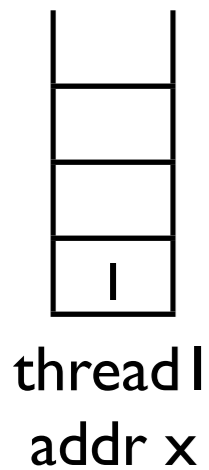
$I \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r I$ - PSO

Example

 thread1 { 1: x = 1; 2: y = 1; 3: done = 1;	thread2 { 4: if (done) { 5: if (x==0) 6: ERROR; 7: local = y; 8: }
---	---

| \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r | - PSO

Example

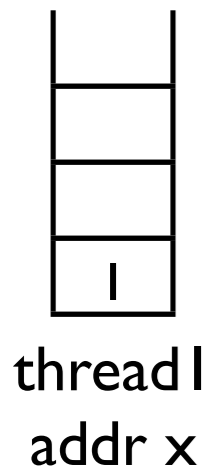


```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```

```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

| \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r | - PSO

Example

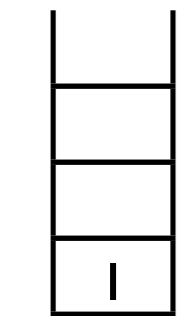


```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```

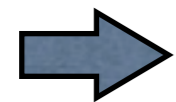
```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

$I \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r I$ - PSO

Example



thread I
addr x

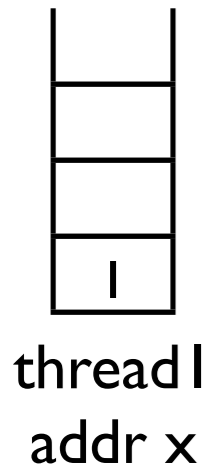


```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```

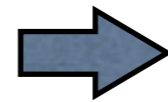
```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

$I \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r I$ - PSO

Example



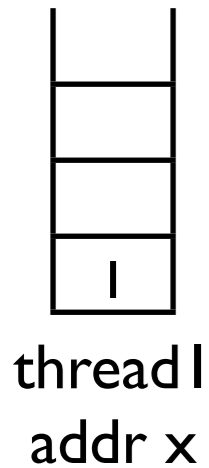
```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```



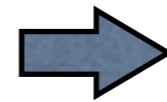
```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

$I \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r I$ - PSO

Example

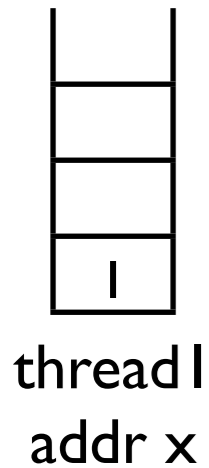


```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```

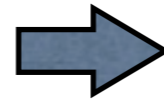


```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

Example



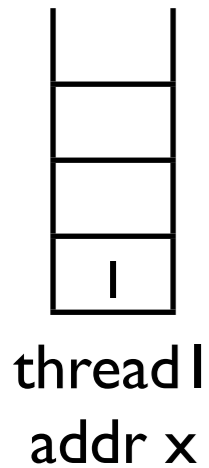
```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```



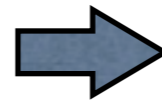
```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

$1 \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r 1$ - PSO

Example



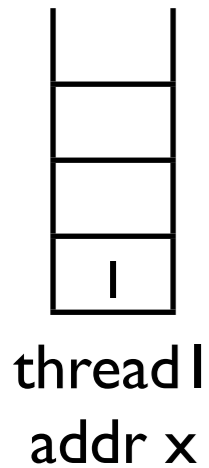
```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```



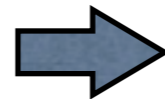
```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

$I \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r I$ - PSO

Example



```
thread1 {  
1: x = 1;  
2: y = 1;  
3: done = 1;
```



```
thread2 {  
4: if (done) {  
5:     if (x==0)  
6:         ERROR;  
7:     local = y;  
8: }
```

$I \rightarrow_p 3 \leftrightarrow_r 4 \rightarrow_p 5 \leftrightarrow_r I$ - PSO

$I \rightarrow_p 3 \rightarrow_c 4 \rightarrow_p 5 \rightarrow_c I$

Potential cycle is realizable.

SC violation is not benign

Summary

- Testing tool that simulates program under different memory models
- Active Testing
 - Phase 1: Examine sequential consistent executions and find potential violations
 - Phase 2: Execute program under relaxed memory models, try to create violations using biased scheduler

Benchmarks

- dekker, bakery: mutual exclusion algorithms
- msn: non-blocking queue
- ms2: two-lock queue
- lazylist: list-based concurrent set
- harris: non-blocking set
- snark: non-block double-ended queue

Benchmarks

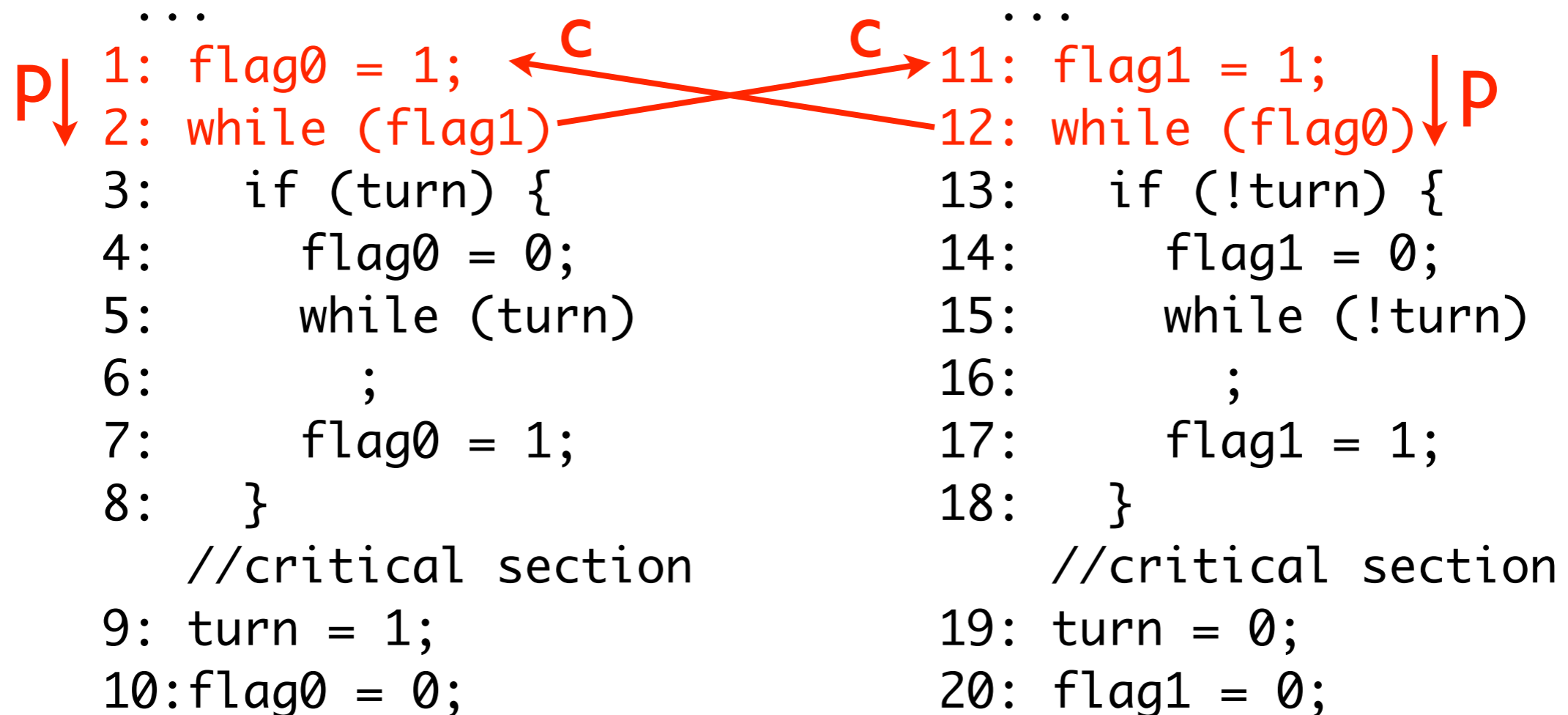
- Manually constructed test harnesses
- dekker and bakery
 - two threads access critical section 3 times
 - assert never concurrently in critical section
- concurrent data structures
 - multiple operations in parallel
 - assert results are consistent with atomic run of operations

Read-After-Delayed-Write Bug

Initially flag0 = flag1 = 0

thread0:

thread1:



cycle under TSO: $1 \rightarrow_p 2 \rightarrow_c 11 \rightarrow_p 12 \rightarrow_c 1$

Results

Benchmark	Cycles predicted	Cycles Confirmed			# of Bugs TSO PSO PSLO			Estimated probability of confirming a cycle		
		TSO	PSO	PSLO	TSO	PSO	PSLO	TSO	PSO	PSLO
dekker	112	23	32	52	17	16	46	0.26	0.17	0.27
bakery	208	24	56	75	20	40	43	0.43	0.19	0.46
msn	350	0	79	93	0	77	89	-	0.13	0.15
ms2	74	0	2	1	0	2	1	-	0.56	0.24
lazylist	157	0	7	6	0	4	4	-	0.07	0.21
harris	93	0	7	23	0	3	10	-	0.09	0.22
snark	1677	0	268	201	0	142	75	-	0.13	0.14

Discussion

- No false warnings, but false negatives possible
 - Fail to predict feasible cycle
 - Fail to confirm feasible cycle
 - Feasible cycles not classified as buggy

Related Work

- Random testing for concurrent bugs
 - ConTest, CTrigger, Active Testing
- Program verification under relaxed models
 - explicit state model checking (D.L.Dill)
 - bounded model checking (Checkfence)
- Runtime monitoring algorithms (Sober)

Conclusions

- Our tool uses operational vs. axiomatic semantics, easier to understand and debug
- Works with any memory model if operational semantics are provided
- Quickly triggers real bugs even under rare schedules or operation reorderings

Thank you

Questions?