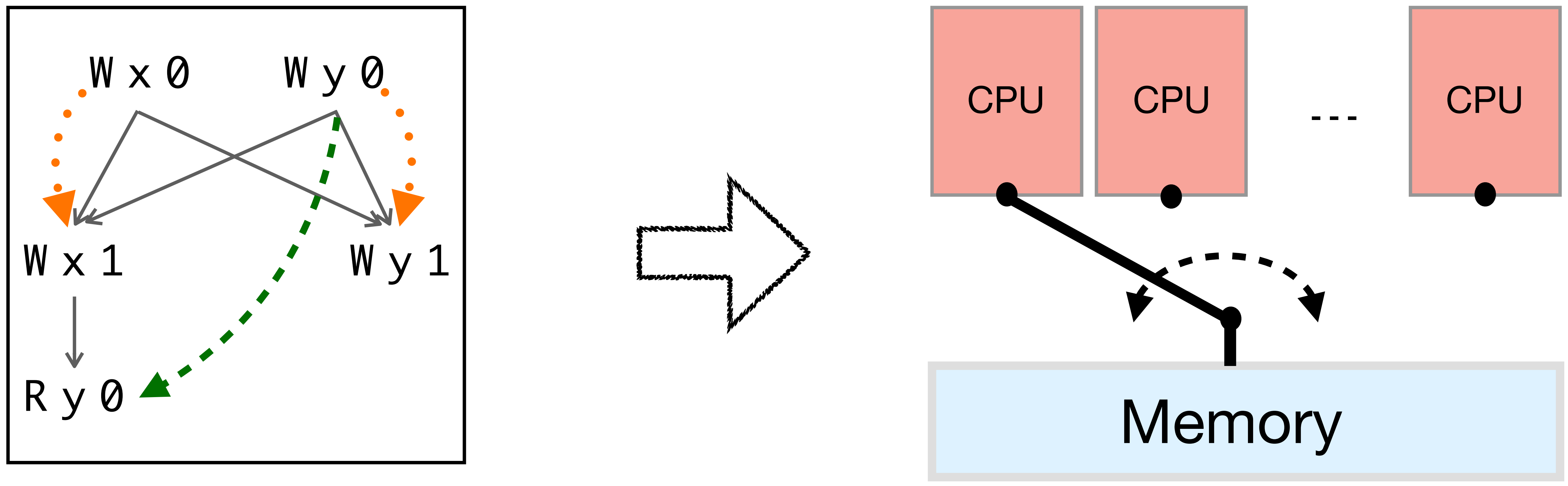
Robustness Against Release/Acquire Semantics

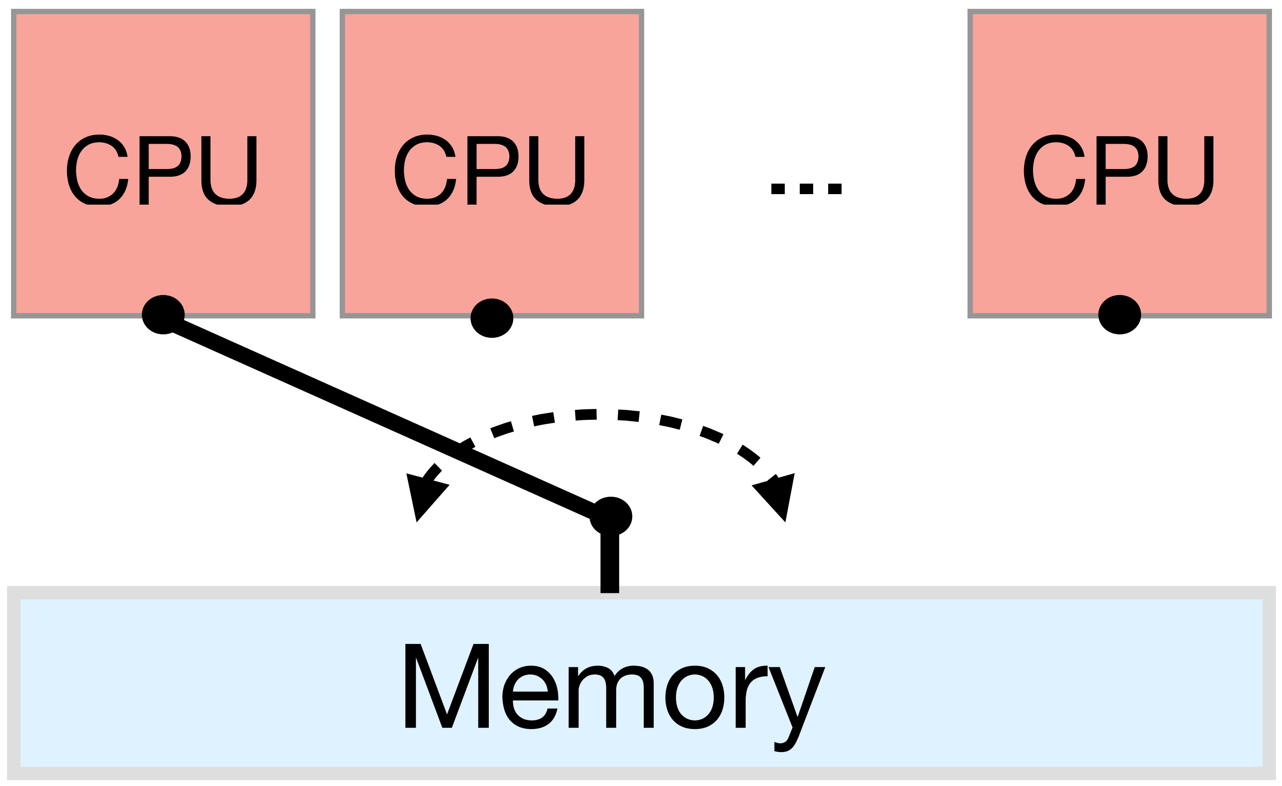
Ori Lahav   Roy Margalit

TEL AVIV UNIVERSITY



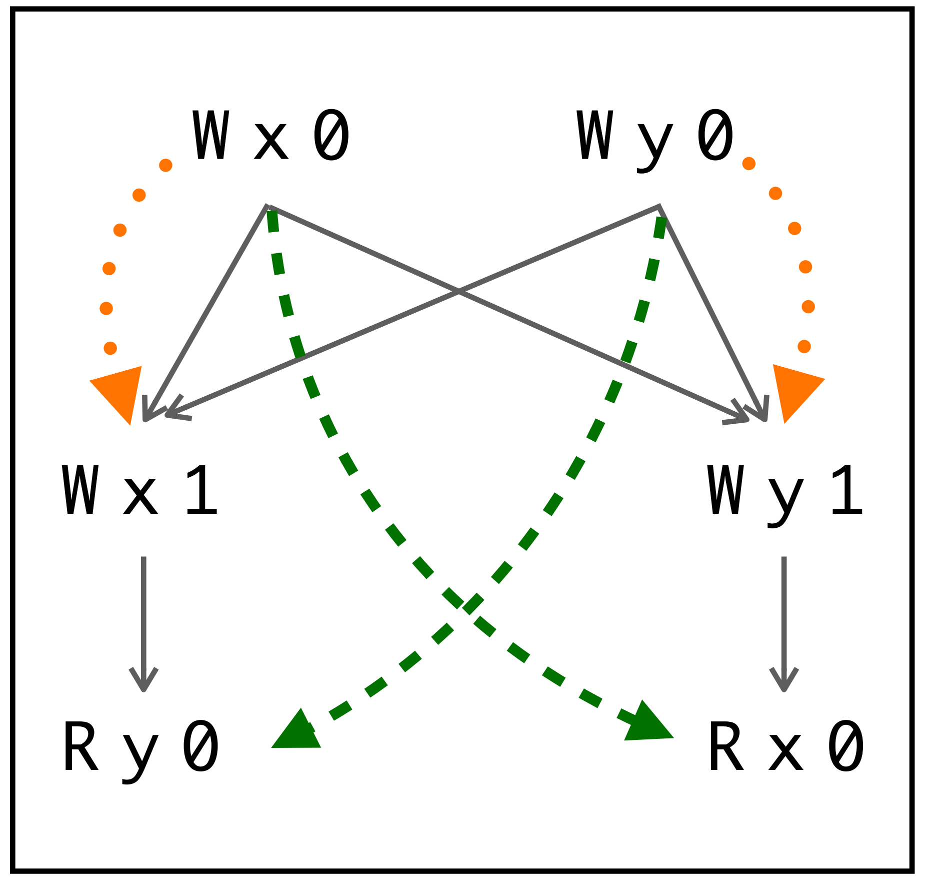
A short story: *Peterson's algorithm in C++*

- In 1981, Peterson proposed a simple algorithm for critical section in shared memory.



- It assumes sequential consistent shared memory (**SC**).

- Q: How to implement Peterson's algorithm in *C/C++11*?



A short story: *Peterson's algorithm in C++*

```
Peterson::Peterson() {
   _victim.store(0, memory_order_release);
   _interested[0].store(false, memory_order_release);
   _interested[1].store(false, memory_order_release);
}
void Peterson::lock() {
   int me = threadID; // either 0 or 1
   int he = 1 – me; // the other thread
   _interested[me].exchange(true, memory_order_acq_rel);
   _victim.store(me, memory_order_release);
   while (_interested[he].load(memory_order_acquire)
       && _victim.load(memory_order_acquire) == me)
      continue; // spin
}
```

7. Dmitriy V'jukov Says:

December 3, 2008 at 4:55 am

Memory ordering in your implementation of Peterson's algo is both insufficient and excessive at the same time. In both permits races and contains unnecessary fences.

11. Bartosz Milewski Says:

So even though I don't have a formal proof, I believe my implementation of Peterson lock is correct. For all I know, Dmitriy's implementation might also be correct, but it's much harder to prove.

C++ atomics and memory ordering, blog post by **Bartosz Milewski**
https://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/

A short story: *Peterson's algorithm in C++*

A subsequent post by Anthony Williams analyzed both algorithms:

- Bartosz's implementation is indeed *wrong*.
- Dmitriy's implementation is *correct*.

https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html

"Any time you deviate from SC, you increase the complexity of the problem by orders of magnitude."

Goal

Automatically establish **robustness** of programs against a weak memory model

verification under weak memory	=	verification under sequential consistency	+	robustness

- Key ingredient in **automatic fence insertion**
- Our focus: C/C++11's **Release/Acquire** fragment
- Previous work: hardware models (especially **x86-TSO**)

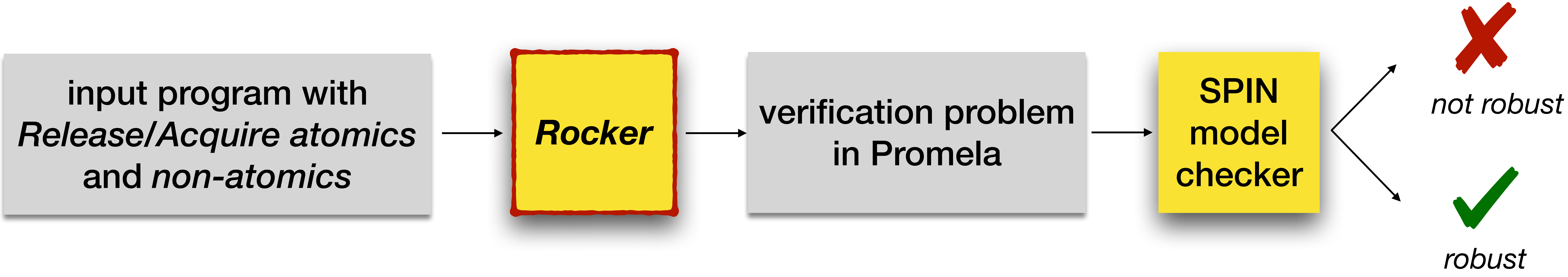
Our Contribution

Theorem

Execution-graph robustness against Release/Acquire is **decidable** and **PSPACE-complete**.

- as verification under SC
- as robustness against x86-TSO

- A **tool** for verifying execution-graph robustness



- **Evaluation** on several challenging synchronization algorithms

Release/Acquire in C/C++11

<u>Implementability</u> ⟷ <u>Programmability</u>

- allows ***cheaper implementation*** (w.r.t. SC):
  - **x86-TSO**: use primitive accesses
  - **IBM Power**: use “lightweight” fences

- ensures the **DRF property**
- often ***sufficiently strong***:
  - *but not always…* (e.g., Perterson’s algorithm)
  - supports “**message passing**” idiom

Syntax

```
atomic_store_explicit(&x, r, memory_order_release)

r = atomic_load_explicit(&x, memory_order_acquire)

atomic_fetch_add_explicit(&x, r, memory_order_acq_rel)

b = atomic_compare_exchange_strong_explicit(&x, &r1, r2,
      memory_order_acq_rel, memory_order_acquire)

atomic_thread_fence(memory_order_seq_cst)
```

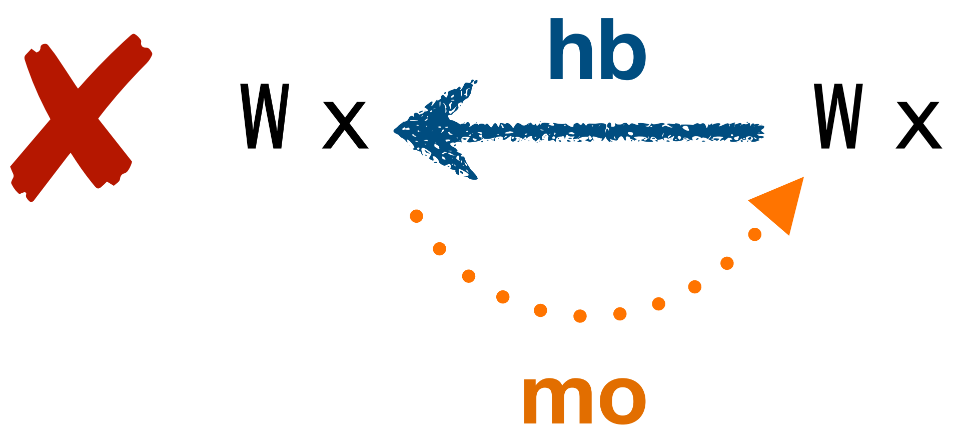
Semantics (one-slide course)

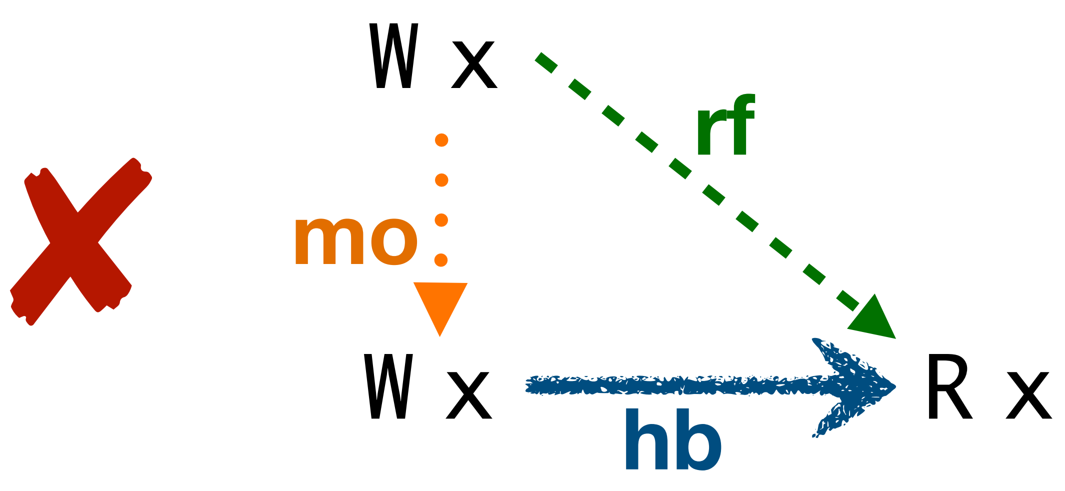
- A form of ***causal consistency***
- Defined **declaratively** using ***execution graphs***

happens-before =

(**program-order** ∪ **reads-from**)$^+$

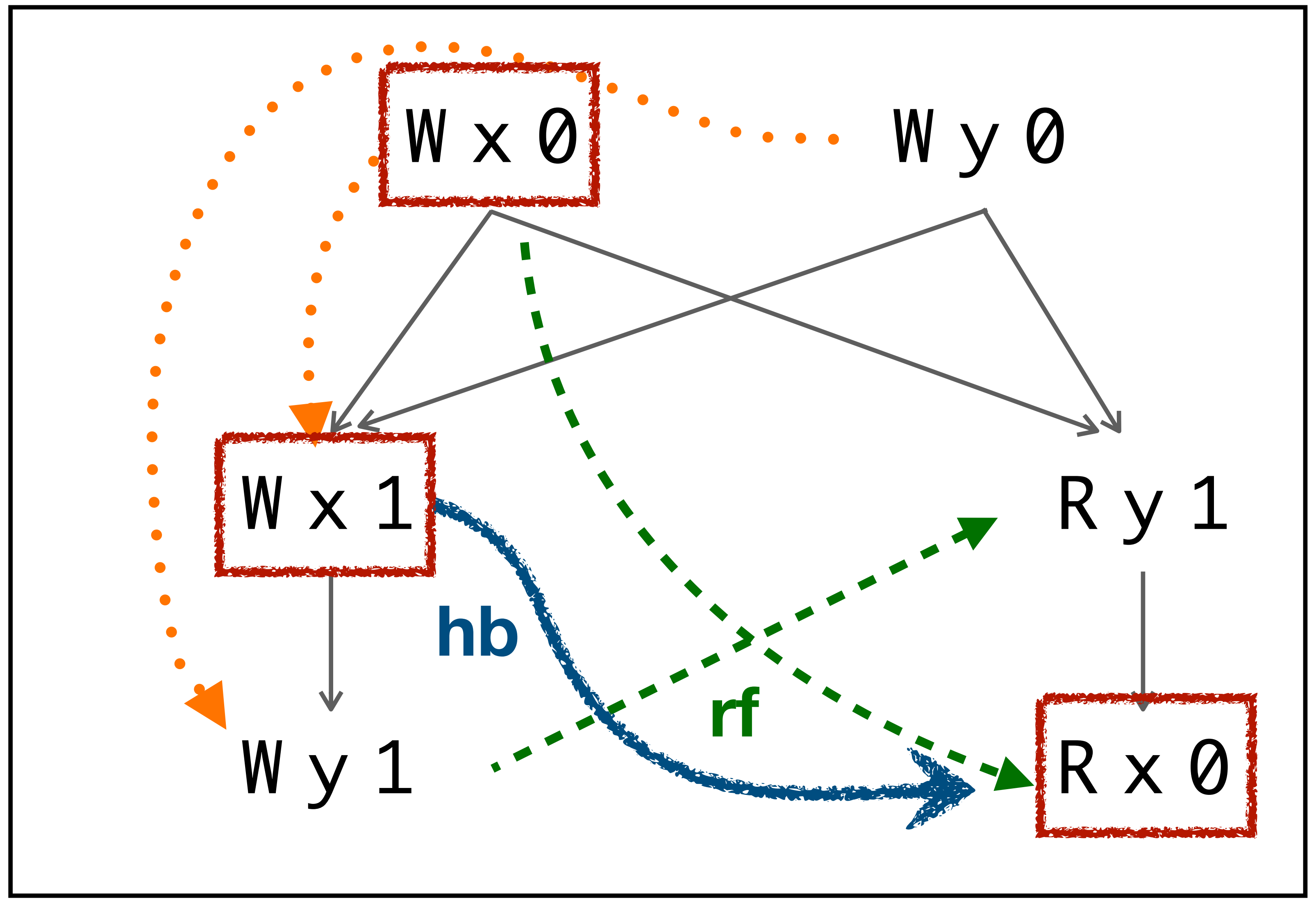
modification-order - total order on writes to the same location





Message passing litmus test

```
          x = y = 0
x = 1   ||   a = y // 1
y = 1   ||   b = x // 0
```



inconsistent execution graph

disallowed program outcome

Operational version

machine state (q, G)

machine step $(q, G) \to_{\mathsf{RA}} (q', G')$

program left to run

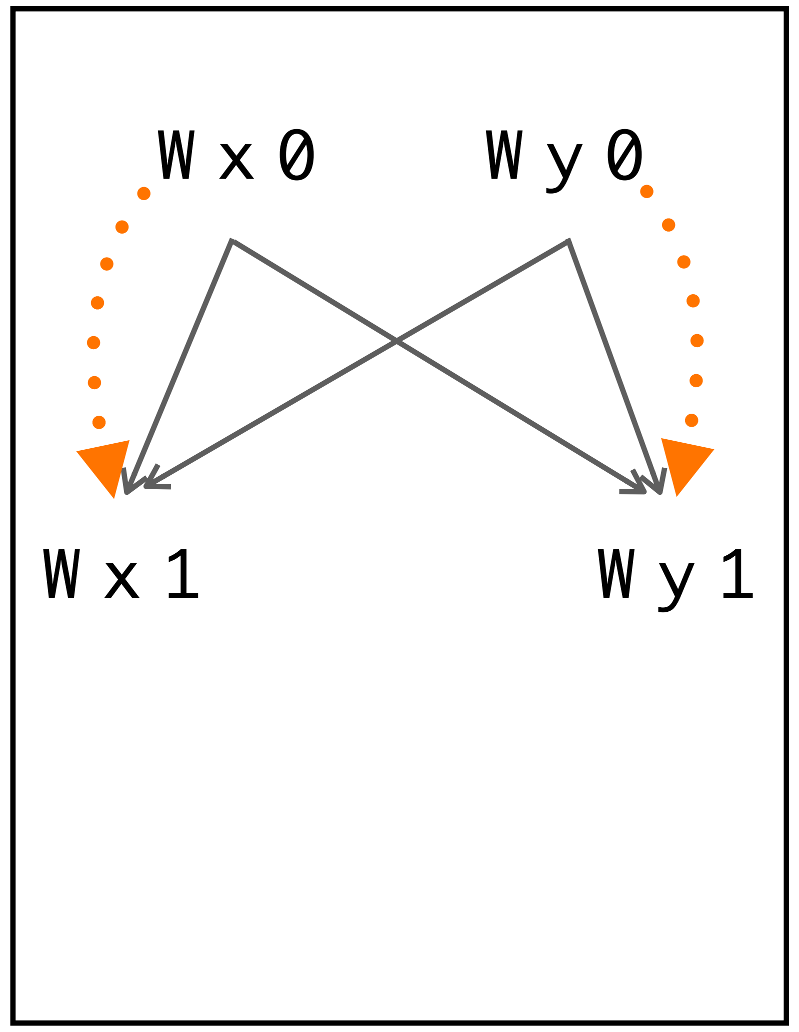
```
a = y || b = x
```

finite state space

local store

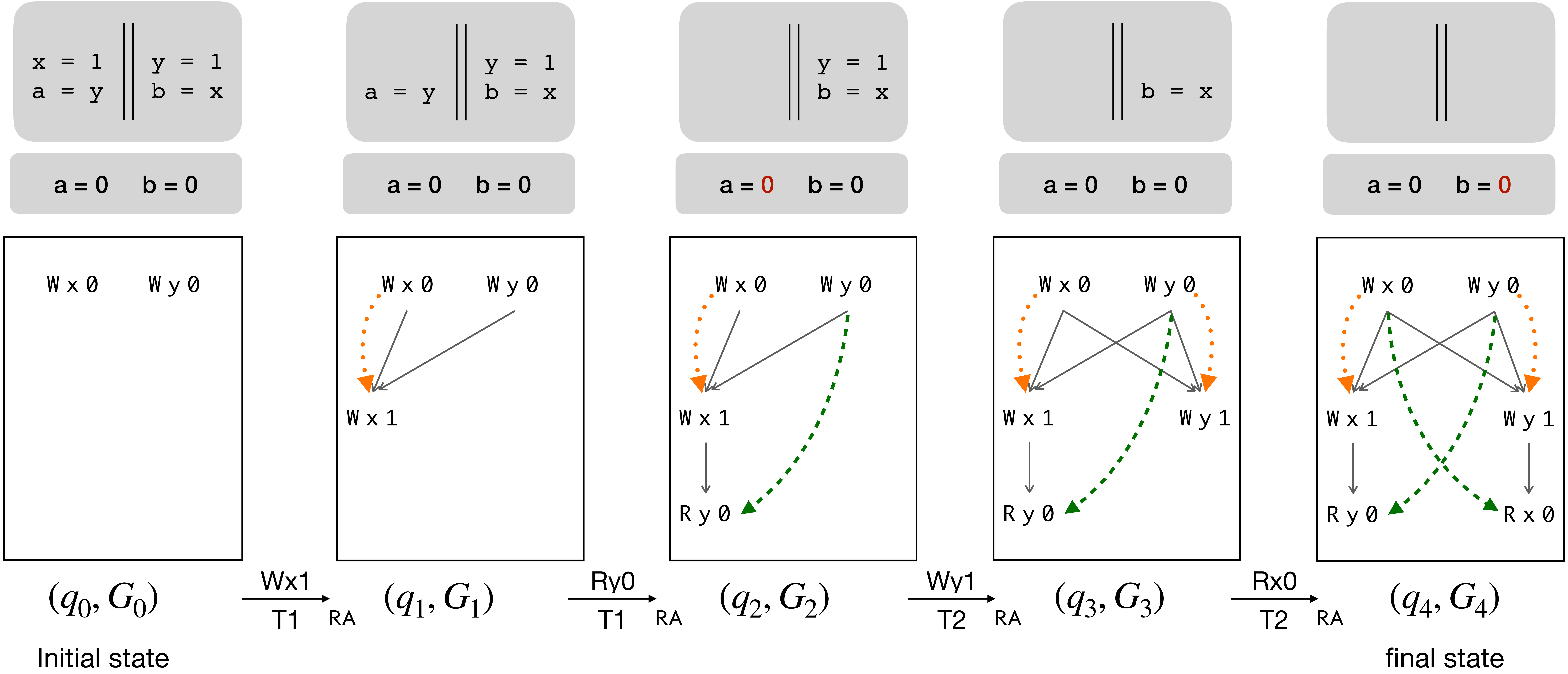
```
a = 0   b = 0
```

current **consistent** execution graph

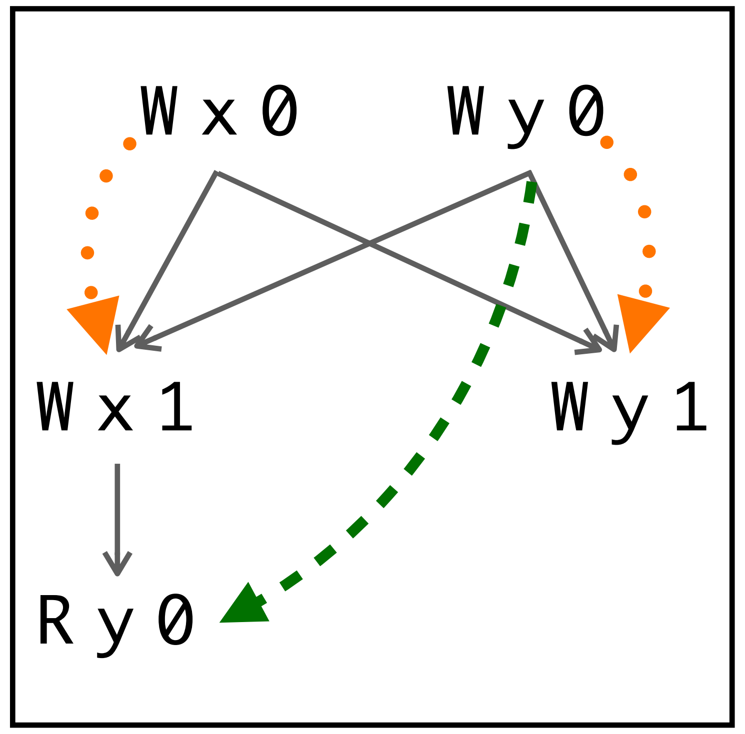


infinite state space

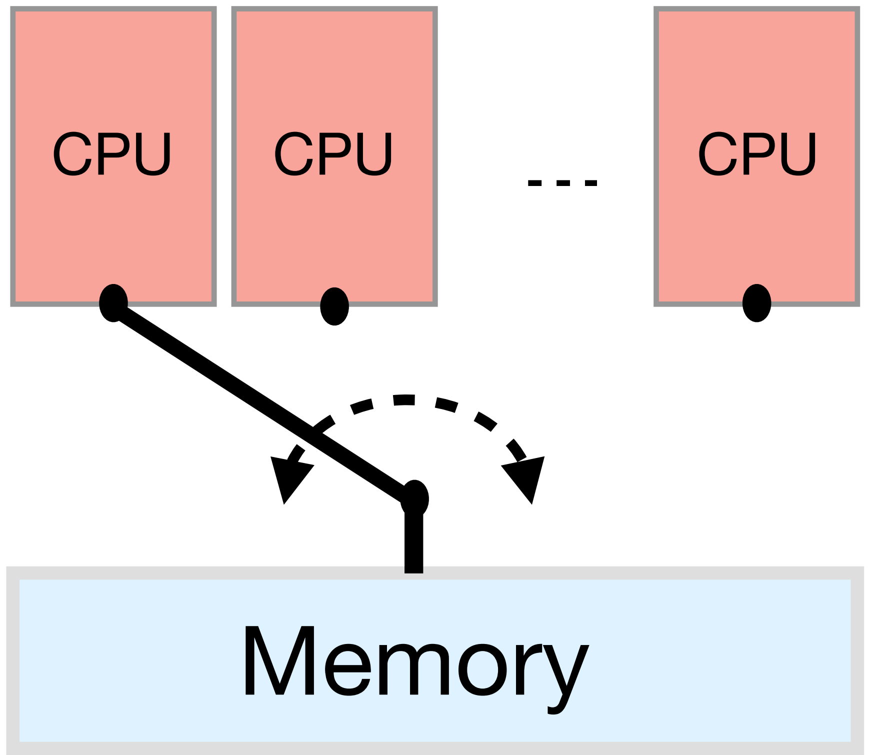
Example: “store-buffer” litmus test



Robustness



Release/Acquire



Sequential consistency

$$\forall q\,.\ \left(\exists G\,.\,(q_0, G_0) \to^*_{\mathsf{RA}} (q, G)\right) \implies \left(\exists M\,.\,(q_0, M_0) \to^*_{\mathsf{SC}} (q, M)\right)$$

Bad news…

- Reduction from **state reachability** [Bouajjani, Derevenetc, Meyer ESOP'13]
- State-reachability for Release/Acquire is **undecidable**! [Abdulla, Arora, Atig, Krishna ***PLDI'19***]

Execution-graph robustness

hb ∪ **mo** can be linearized to an execution order of an SC-run

$\forall q, G \,.\; (q_0, G_0) \rightarrow^*_{\mathsf{RA}} (q, G) \implies G$ **describes an SC-history**

$$\Downarrow$$

State robustness

$$\forall q \,.\; \left(\exists G \,.\, (q_0, G_0) \rightarrow^*_{\mathsf{RA}} (q, G) \right) \implies \left(\exists M \,.\, (q_0, M_0) \rightarrow^*_{\mathsf{SC}} (q, M) \right)$$

Theorem

Execution-graph robustness against Release/Acquire is **decidable** and **PSPACE-complete**.

Reduction to ***reachability*** under ***an instrumented SC semantics***

allowed by SC

a "minimal" robustness violation:

$$(q_0, G_0) \rightarrow_{\mathsf{RA}} (q_1, G_1) \rightarrow_{\mathsf{RA}} (q_2, G_2) \rightarrow_{\mathsf{RA}} \cdots \rightarrow_{\mathsf{RA}} (q_n, G_n) \rightarrow_{\mathsf{RA}} (q_{n+1}, G_{n+1})$$

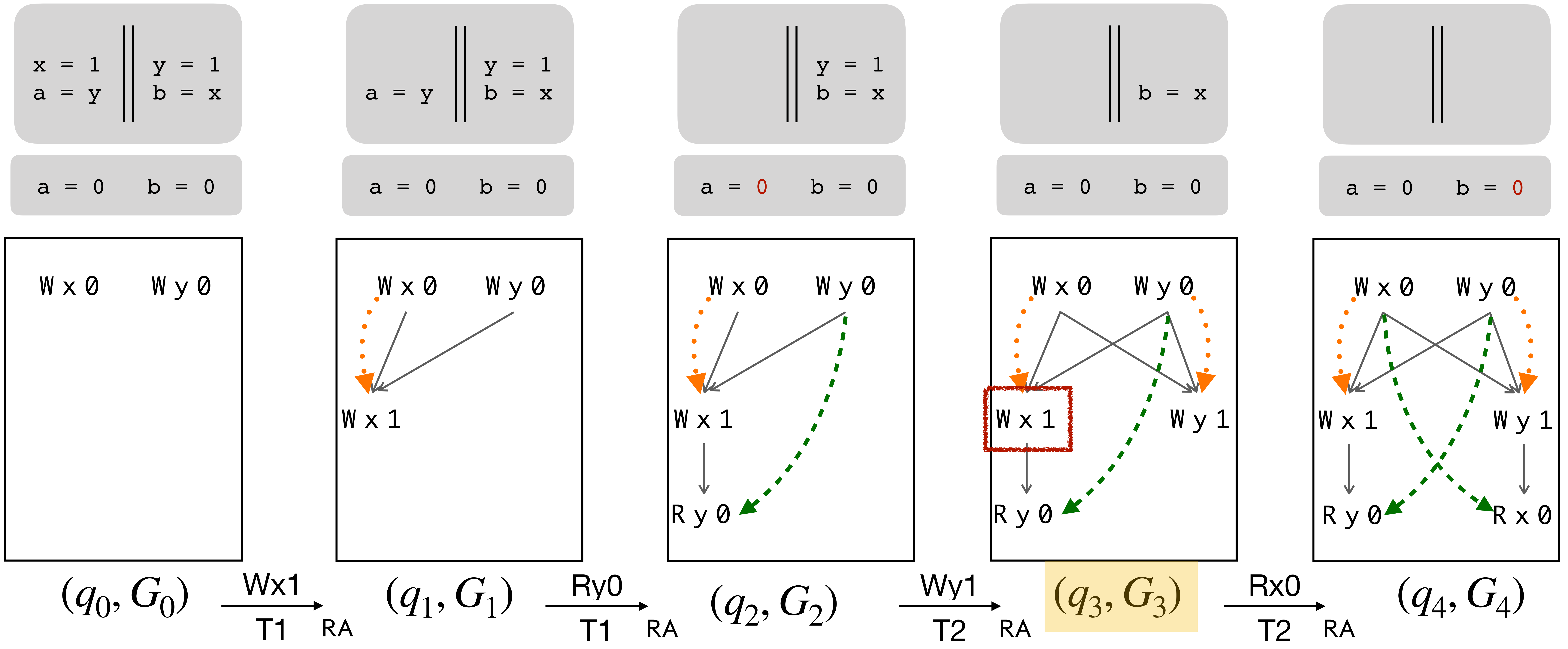
disallowed by SC

can take an RA-step to a non-SC execution graph

$$(q_0, M_0) \rightarrow_{\mathsf{SC}} (q_1', M_1) \rightarrow_{\mathsf{SC}} (q_2', M_2) \rightarrow_{\mathsf{SC}} \cdots \rightarrow_{\mathsf{SC}} (q_n, M_n)$$

I_0 I_1 I_2 I_n

robustness instrumentation



$(q_0, M_0) \rightarrow_{\mathsf{SC}} (q_1', M_1) \rightarrow_{\mathsf{SC}} (q_2', M_2) \rightarrow_{\mathsf{SC}} (q_3, M_3)$

I_0 $\quad I_1$ $\quad I_2$ $\quad I_3$

robustness viloation!

For w = the **mo**-maximal write to `x` (`W x 1`):

- w has no **hb** T2
- Every SC-run producing G_3 executes w before the current last event of T2

Instrumented SC Semantics

	$\langle \tau, \mathsf{W}(x, v)\rangle$ or $\langle \tau, \mathsf{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}})\rangle$	$\langle \tau, \mathsf{R}(x, v)\rangle$
$\mathsf{V}'_{\mathsf{SC}} = \lambda \pi.$	$\begin{cases} \mathsf{V}_{\mathsf{SC}}(\tau) \cup \mathsf{M}_{\mathsf{SC}}(x) & \pi = \tau \\ \mathsf{V}_{\mathsf{SC}}(\pi) \setminus \{x\} & \pi \neq \tau \end{cases}$	$\begin{cases} \mathsf{V}_{\mathsf{SC}}(\tau) \cup \mathsf{W}_{\mathsf{SC}}(x) & \pi = \tau \\ \mathsf{V}_{\mathsf{SC}}(\pi) & \pi \neq \tau \end{cases}$
$\mathsf{M}'_{\mathsf{SC}} = \lambda y.$	$\begin{cases} \mathsf{M}_{\mathsf{SC}}(x) \cup \mathsf{V}_{\mathsf{SC}}(\tau) & y = x \\ \mathsf{M}_{\mathsf{SC}}(y) \setminus \{x\} & y \neq x \end{cases}$	$\begin{cases} \mathsf{M}_{\mathsf{SC}}(x) \cup \mathsf{V}_{\mathsf{SC}}(\tau) & y = x \\ \mathsf{M}_{\mathsf{SC}}(y) & y \neq x \end{cases}$
$\mathsf{W}'_{\mathsf{SC}} = \lambda y.$	$\begin{cases} \mathsf{M}_{\mathsf{SC}}(x) \cup \mathsf{V}_{\mathsf{SC}}(\tau) & y = x \\ \mathsf{W}_{\mathsf{SC}}(y) \setminus \{x\} & y \neq x \end{cases}$	$\mathsf{W}_{\mathsf{SC}}(y)$

Figure 5. Maintaining V_{SC}, M_{SC} and W_{SC} in SCM transitions.

	$\langle \tau, \mathsf{W}(x, v)\rangle$ where $v_{\mathsf{R}} = \mathsf{M}(x)$	$\langle \tau, \mathsf{R}(x, v)\rangle$	$\langle \tau, \mathsf{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}})\rangle$
$\mathsf{V}' = \lambda \pi, y.$	$\begin{cases} \emptyset & \pi = \tau,\ y = x \\ \mathsf{V}(\pi)(x) \cup \{v_{\mathsf{R}}\} & \pi \neq \tau,\ y = x \\ \mathsf{V}(\pi)(y) & y \neq x \end{cases}$	$\begin{cases} \mathsf{V}(\tau)(y) \cap \mathsf{W}(x)(y) & \pi = \tau \\ \mathsf{V}(\pi)(y) & \pi \neq \tau \end{cases}$	$\begin{cases} \mathsf{V}(\tau)(y) \cap \mathsf{W}(x)(y) & \pi = \tau \\ \mathsf{V}(\pi)(x) \cup \{v_{\mathsf{R}}\} & \pi \neq \tau,\ y = x \\ \mathsf{V}(\pi)(y) & \pi \neq \tau,\ y \neq x \end{cases}$
$\mathsf{W}' = \lambda z, y.$	$\begin{cases} \mathsf{V}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}(z)(x) \cup \{v_{\mathsf{R}}\} & z \neq x,\ y = x \\ \mathsf{W}(z)(y) & \text{otherwise} \end{cases}$	$\mathsf{W}(z)(y)$	$\begin{cases} \mathsf{W}(x)(y) \cap \mathsf{V}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}(z)(x) \cup \{v_{\mathsf{R}}\} & z \neq x,\ y = x \\ \mathsf{W}(z)(y) & \text{otherwise} \end{cases}$
$\mathsf{V}'_{\mathsf{RMW}} = \lambda \pi, y.$	$\begin{cases} \emptyset & \pi = \tau,\ y = x \\ \mathsf{V}_{\mathsf{RMW}}(\pi)(x) \cup \{v_{\mathsf{R}}\} & \pi \neq \tau,\ y = x \\ \mathsf{V}_{\mathsf{RMW}}(\pi)(y) & y \neq x \end{cases}$	$\begin{cases} \mathsf{V}_{\mathsf{RMW}}(\tau)(y) \cap \mathsf{W}_{\mathsf{RMW}}(x)(y) & \pi = \tau \\ \mathsf{V}_{\mathsf{RMW}}(\pi)(y) & \pi \neq \tau \end{cases}$	
$\mathsf{W}'_{\mathsf{RMW}} = \lambda z, y.$	$\begin{cases} \mathsf{V}_{\mathsf{RMW}}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}_{\mathsf{RMW}}(z)(x) \cup \{v_{\mathsf{R}}\} & z \neq x,\ y = x \\ \mathsf{W}_{\mathsf{RMW}}(z)(y) & \text{otherwise} \end{cases}$	$\mathsf{W}_{\mathsf{RMW}}(z)(y)$	$\begin{cases} \mathsf{W}_{\mathsf{RMW}}(x)(y) \cap \mathsf{V}_{\mathsf{RMW}}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}_{\mathsf{RMW}}(z)(y) & \text{otherwise} \end{cases}$

Figure 6. Maintaining V, W, $\mathsf{V}_{\mathsf{RMW}}$, and $\mathsf{W}_{\mathsf{RMW}}$ in SCM transitions.

Complications

- **Read-modify-write (RMW) instructions**

  require much more ***refined instrumentation***
  (depends on values being read)

```
a = CAS(x,0,1)
```

R x v
(v ≠ 0)

RMW x 0 1

- **Masking benign violations**

```
X = Y = 0
X = 1                      || Y = 1
do a = Y while (a ≠ 1)     || do b = X while (b ≠ 1)
```

✗ *not robust*

  masked using blocking instructions:

```
X = Y = 0
X = 1            || Y = 1
wait (Y == 1)    || wait (X == 1)
```

✓ *robust*

- **Sequentially consistent fences** modelled as RMWs

Evaluation

number of threads

number of lines

robust?

w/o robustness instrumentation

robustness against x86-TSO

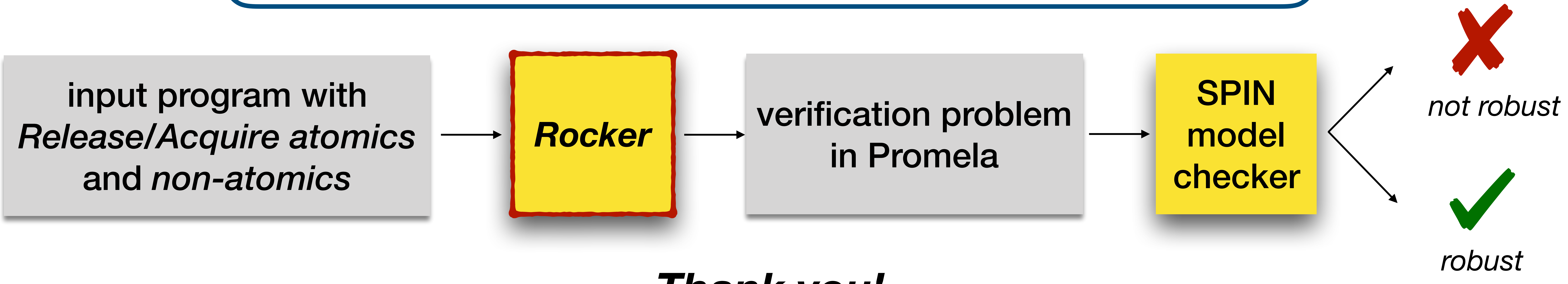
	#T	LoC	Result	Time (sec)	SC (sec)	Trencher (TSO)	
						Result	Time (sec)
spin-lock	2	34	✓	1.6	1.2	✓	5.4
seq-lock	4	49	✓	20.7	3.4	✓	8.9
Peterson	2	28	✗	2.5	1.2	✗	5.6
Peterson for x86-TSO	2	30	✗	3.3	1.3	✓	5.6
Peterson - Dmitriy	2	36	✓	4.3	1.2	✓	5.5
Peterson - Bartosz	2	28	✗	3.4	1.1	✗	5.6
RCU	4	74	✓	67.6	2.2	✗ *	-
RCU (offline)	3	215	✓	137.9	18.3	✗ *	-

requires blocking instructions

Summary

- We developed a **sound and precise reduction** from **execution-graph robustness against Release/Acquire** semantics to a **reachability problem under SC**.
- Execution-graph robustness against Release/Acquire is **PSPACE-complete**.
- We **implemented** the reduction and verified several challenging algorithms, demonstrating in particular that **execution-graph robustness is not overly strong**.

verification under weak memory = verification under sequential consistency + robustness



Thank you!

```
        X = Y = 0
X = 1        ||   Y = 1
a = Y // 0   ||   b = X // 0
```

```
        X = Y = 0
X = 0        ||   Y = 0
a = Y // 0   ||   b = X // 0
```

state robustness	***execution graph robustness***
✗	✗
✓	✗