What's Decidable about Causally Consistent Shared Memory?

Ori Lahav

TEL AVIV UNIVERSITY

Udi Boker

IDC HERZLIYA

OWLS
January 20, 2021

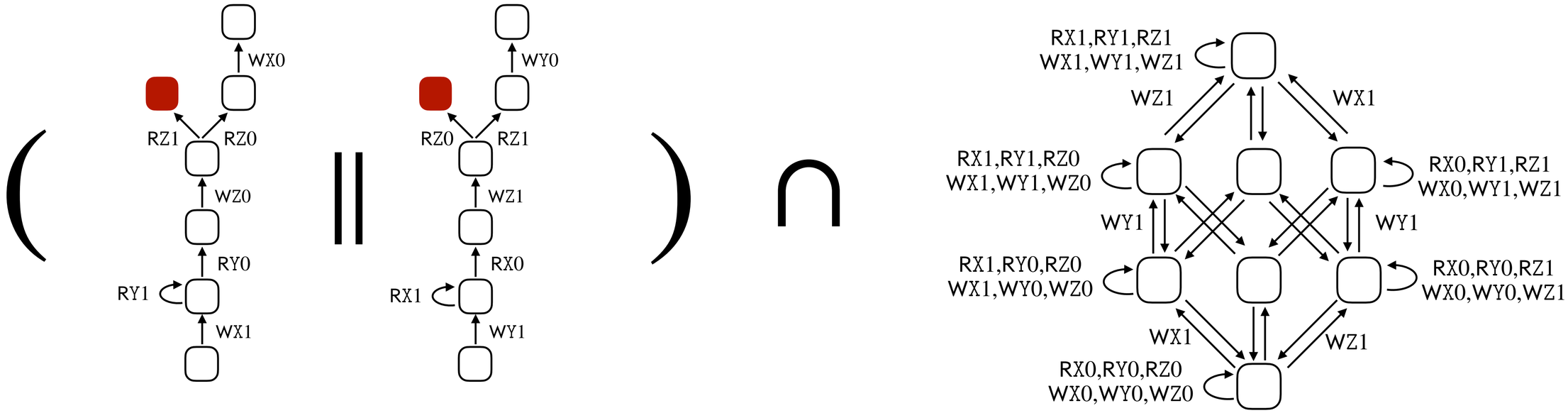
Safety Verification

```
X := 1;
repeat
   a := Y;
until (a = 0);
Z := 0;
assert (Z = 0);
X := 0;
```

||

```
Y := 1;
repeat
   b := X;
until (b = 0);
Z := 1;
assert (Z = 1);
Y := 0;
```

Simple Solution

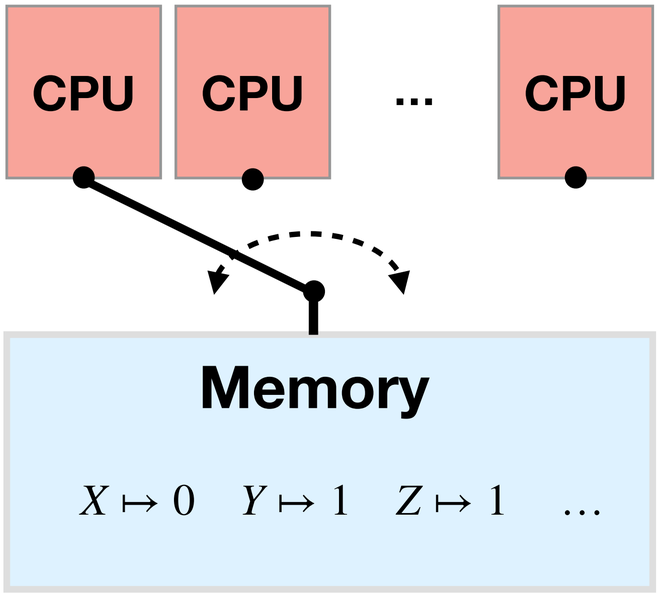


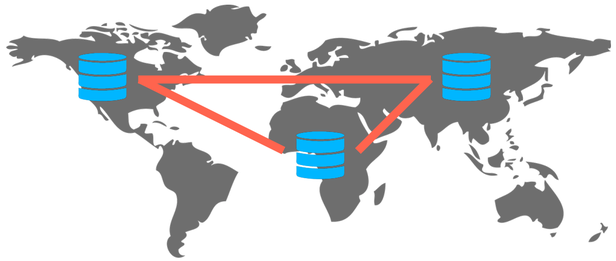
For programs with a **bounded data domain**, this problem is clearly **decidable**:

- Reduction to reachability in finite-state systems
- PSPACE-complete

Sequential Consistency (SC)

- We assumed the classical shared-memory model.
- But it is **unrealistic**:
  for better **performance/scalability/availability/ fault-tolerance** shared-memory implementations provide weaker semantics.
- Similar situation in *distributed data-stores*





Weaker Memory Models

- x86-TSO
- POWER
- ARM
- RISC-V
- C/C++11
- many many more…

Decidable

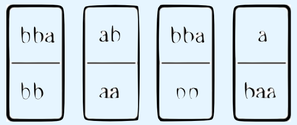
[Atig, Bouajjani, Burckhardt, Musuvathi. POPL'2010]

[Abdulla, Atig, Bouajjani, Ngo. CUNCUR'2016]

Undecidable

[Abdulla, Arora, Atig, Krishna. PLDI'2019]

- Even for the **Release/Acquire** fragment:
  `memory_order_release` & `memory_order_acquire`
- Reduction from *Post correspondence problem*



Causal Consistency

- A classical model originated from ***replicated data stores***:
  - nodes may disagree on the order of some operations
  - consensus on the order of *"causally related"* operations
- Relatively simple and intuitive but more scalable than SC

Sequential Consistency

|

x86-TSO

|

Causal Consistency

Safety Verification

```
X := 1;
repeat
  a := Y; // 0
until (a = 0);
Z := 0;
assert (Z = 0);
X := 0;
```

||

```
Y := 1;
repeat
  b := X; // 0
until (b = 0);
Z := 1;
assert (Z = 1);
Y := 0;
```

Both can read 0 in the same execution!

Is it a Problem?

https://en.wikipedia.org/wiki/Dekker's_algorithm

- How come ***airplanes don't crash***?
  - There are ways to **demand sequential consistency** when we need it.
  - We often **don't need** sequential consistency in its full power.

We have to define and understand the semantics of shared-memory concurrency.

Flag-Based Synchronization

```
X = Y = 0

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

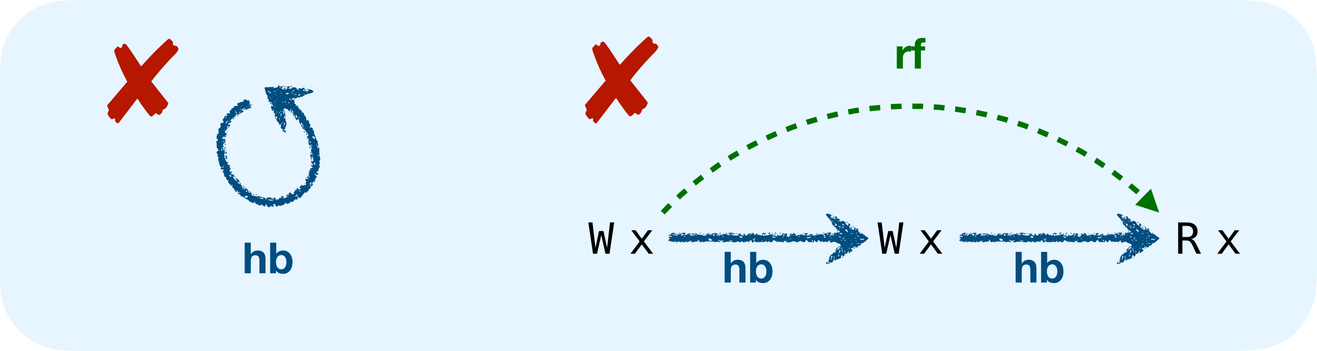
This behavior is forbidden under causal consistency

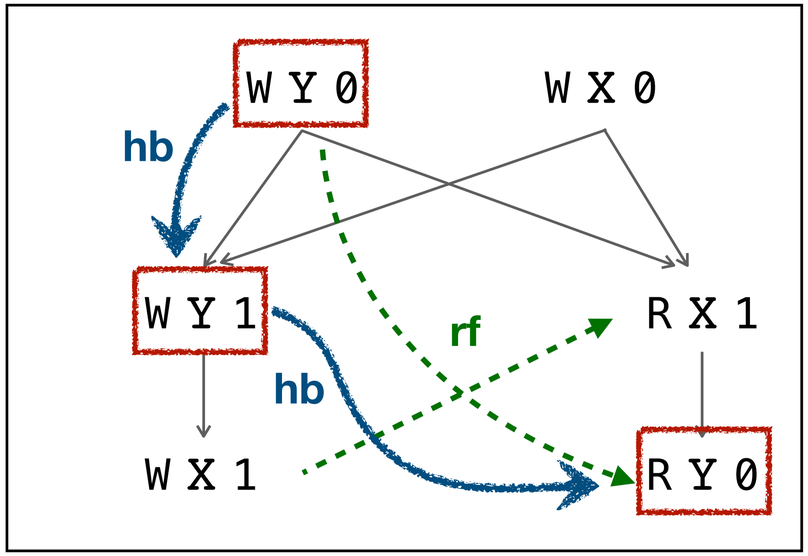
Formal Semantics

Defined **declaratively** using ***execution graphs***

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

happens-before = (**program-order** ∪ **reads-from**)$^{+}$

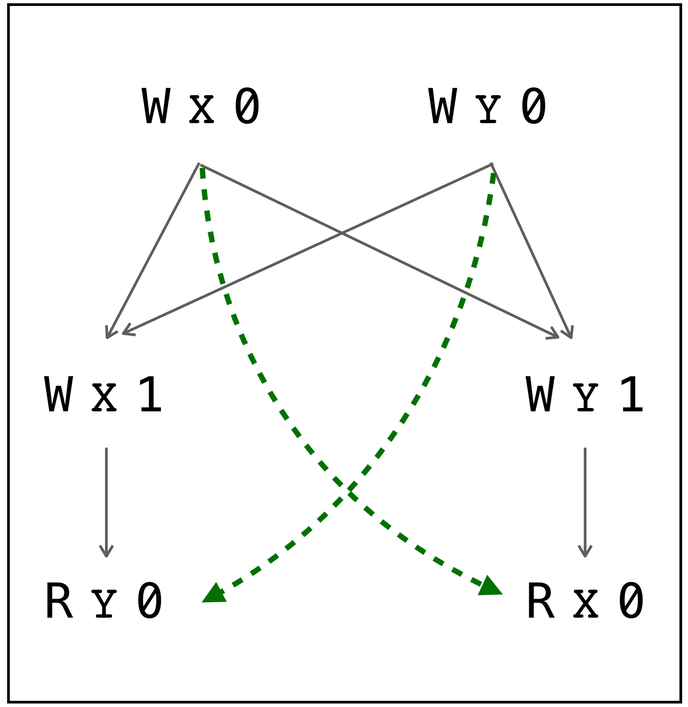




inconsistent execution graph

disallowed program outcome

```
X := 1;                  ||   Y := 1;
repeat                   ||   repeat
   a := Y;               ||      b := X;
until (a = 0);           ||   until (b = 0);
Z := 0;                  ||   Z := 1;
assert (Z = 0);          ||   assert (Z = 1);
```



program-order

reads-from

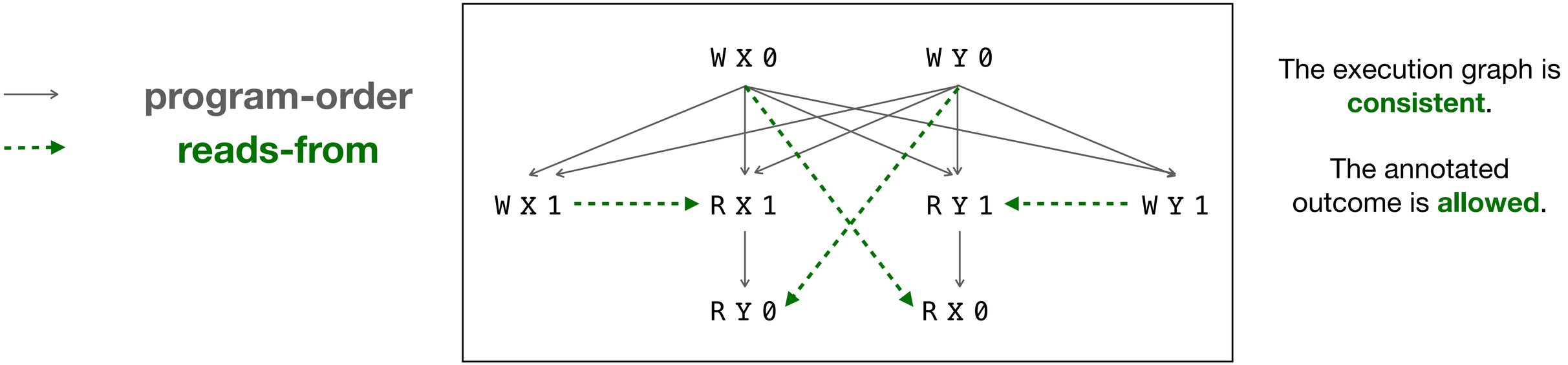
The execution graph is **consistent**.

The annotated outcome is **allowed**.

Non-Multi-Copy-Atomicity

Weaker than x86-TSO: different threads can observe writes in different orders

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



The execution graph is **consistent**.

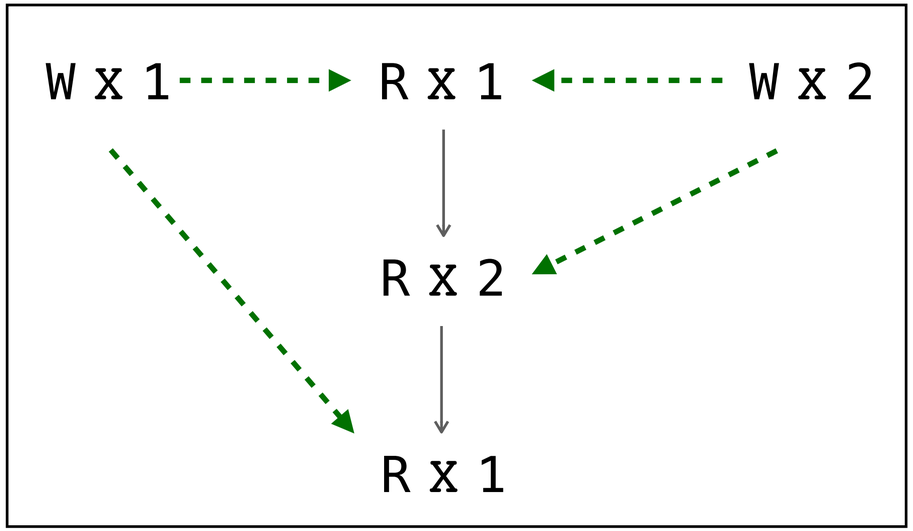
The annotated outcome is **allowed**.

What about *concurrent writes*?

```
x = 1  ||  a = x // 1  ||  x = 2
       ||  b = x // 2  ||
       ||  c = x // 1  ||
```

program-order

reads-from



The execution graph is **consistent**.

The annotated outcome is **allowed**.

What about *concurrent writes*?

nothing

make sure that they are totally ordered

Weak Release/Acquire (**WRA**)

Release/Acquire (**RA**)

Strong Release/Acquire (**SRA**)

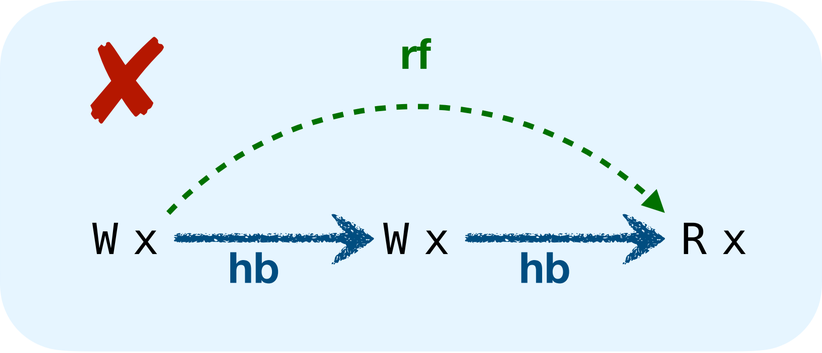
Three Variants

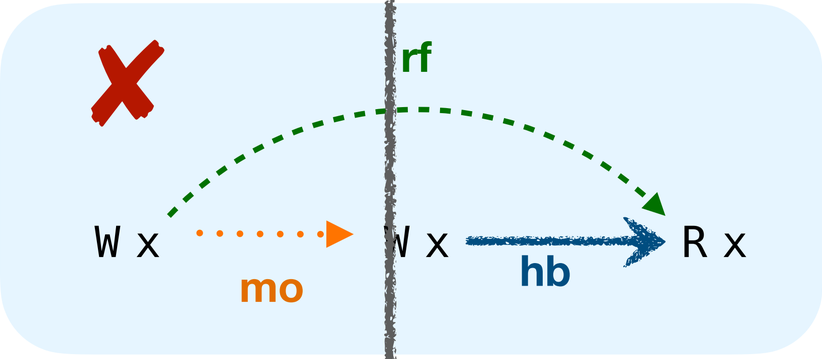
Weak Release/Acquire
(**WRA**)

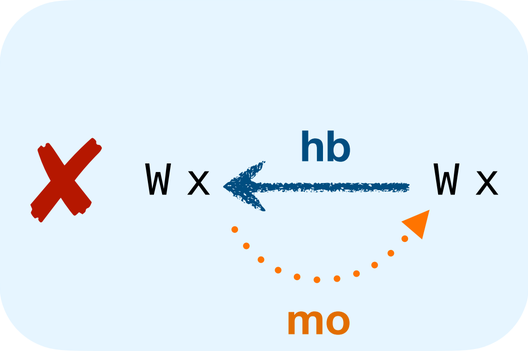
Release/Acquire
(**RA**)

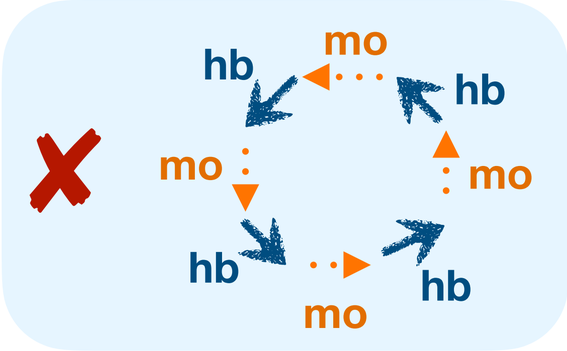
Strong Release/Acquire
(**SRA**)

∃ total order on writes to the same location (**modification-order**) s.t.:

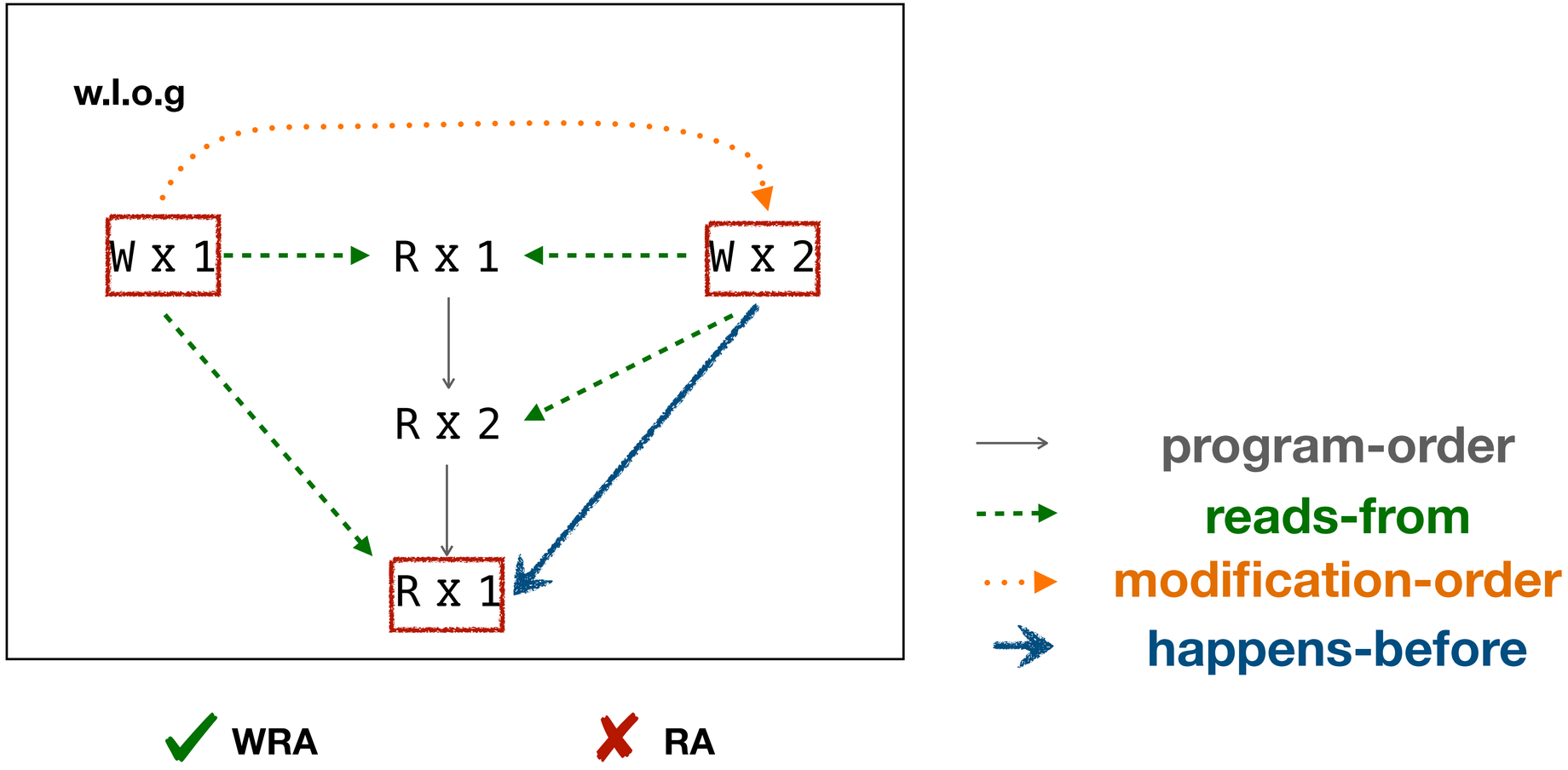






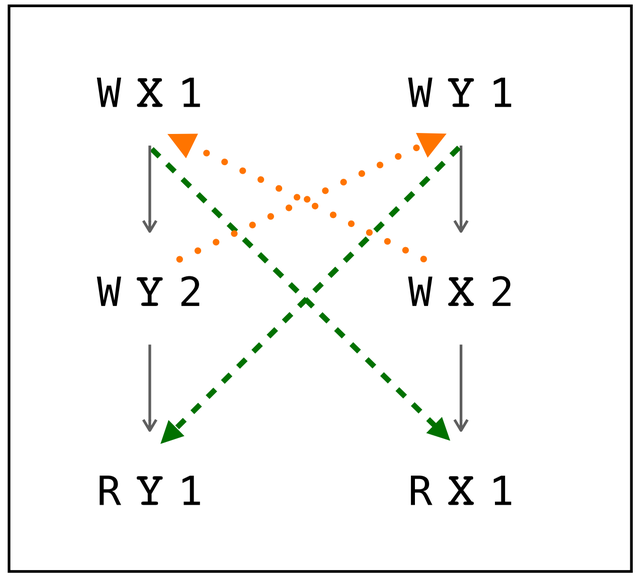


WRA is strictly weaker than RA



RA is strictly weaker than SRA

```
X = 1          ||  Y = 1
Y = 2          ||  X = 2
a = Y // 1     ||  b = X // 1
```



✓ RA ✗ SRA

program-order

reads-from

modification-order

Causal Consistency

Sequential Consistency

x86-TSO

Causal Consistency

SRA

RA

WRA

Distributed data-stores
POWER architecture [L, Giannarakis, Vafeiadis. POPL'16]

C/C++11
Java 9

CC in [Bouajjani, Enea, Guerraoui, Hamza. POPL'17]
[Kokologiannakis, L, Sagonas, Vafeiadis. POPL'18]

Write-Write-Race Freedom

Theorem

$Prog$ has no WW-races under **SRA** $\implies [\![Prog]\!]_{\text{WRA}} = [\![Prog]\!]_{\text{RA}} = [\![Prog]\!]_{\text{SRA}}$

Results

Theorem

The verification problems under **SRA** and **WRA** are **decidable**.

- In contrast with **RA** [Abdulla, Arora, Atig, Krishna. PLDI'2019]
- To obtain this result we develop a **new semantics** for SRA and WRA.

Corollary

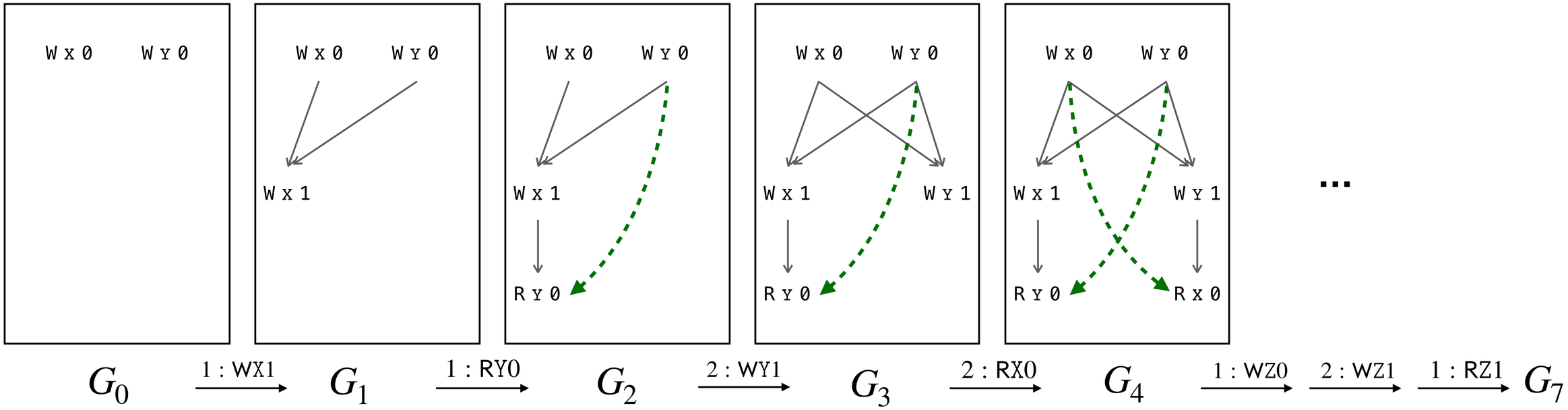
The verification problem under **RA** is **decidable** for write-write-race-free programs.

Lower Complexity Bound

- For causal consistency, the problem is **non-primitive recursive**

- We can simulate a lossy FIFO channel machine
  - as for x86-TSO [Atig, Bouajjani, Burckhardt, Musuvathi. POPL'2010]

Even when the program is finite state,
its synchronization with a causally consistent memory is **infinite** state.



Initial state

Error state

- To establish **decidablity**:
  - We use the framework of ***well-structured transition systems*** (WSTS)

    [Abdullah] [Finkel, Schnoebelen] …
  - **Challenge**: find a WSTS equivalent to a casually consistent memory

Backward Reachability

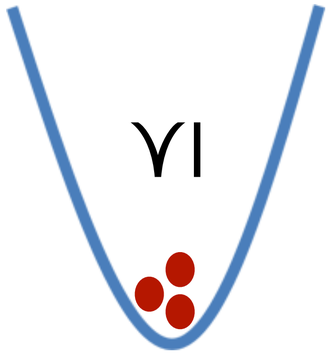
Input: LTS $(Q, Q_0, \rightarrow)$, state q_{bad}
Output: is q_{bad} reachable?

$S := \{q_{\text{bad}}\}$
repeat
    $S_{\text{prev}} := S$
    $S := S \cup \text{pre}(S)$
until $(S = S_{\text{prev}})$
return $(Q_0 \cap S \neq \emptyset)$

To make this an algorithm we need to work with *finitely representable sets* & *guarantee termination*

Well-Structured Transition Systems

- Equip the transition system with a **well quasi-order** $\preceq$
- Work with **upward closed** sets of states represented by their finite basis



$\preceq$ should be compatible with $\rightarrow$

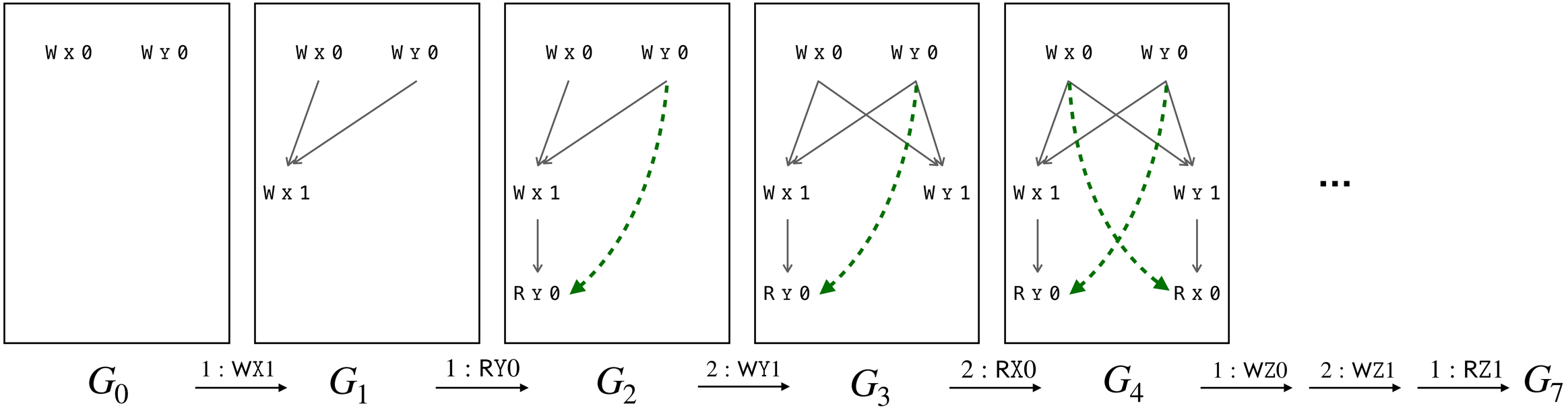
$$\begin{array}{ccc} q_1' & \xrightarrow{*} & q_2' \\ \rotatebox{90}{$\preceq$} & & \rotatebox{90}{$\preceq$} \\ q_1 & \longrightarrow & q_2 \end{array}$$

Compatibility is guaranteed in "lossy" systems

$$\textbf{(lose)}\ \frac{q' \succeq q}{q' \rightarrow q}$$

- **Challenge:** characterize casually consistent shared memory as a ***lossy*** system

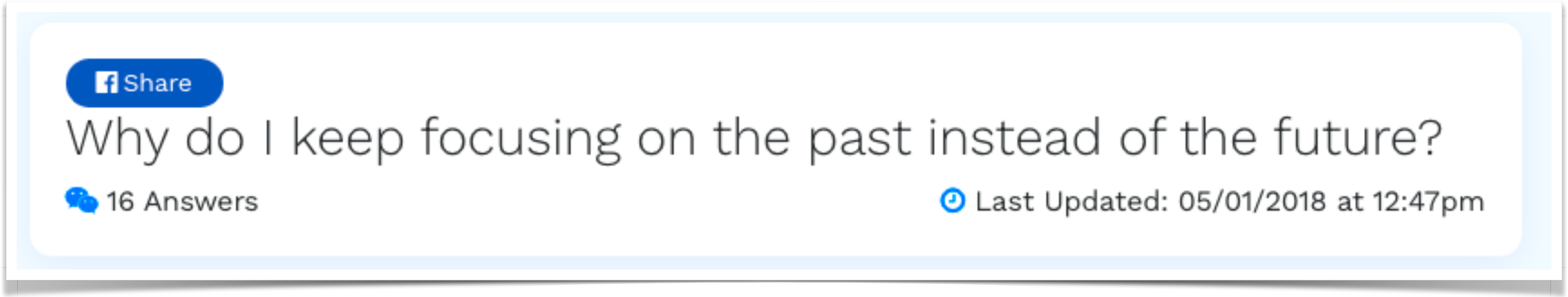
Causal Consistency as a WSTS



Two critical obstacles:

- Partial order embedding is ***not*** a well-quasi-order
- Execution histories are ***not*** lossy

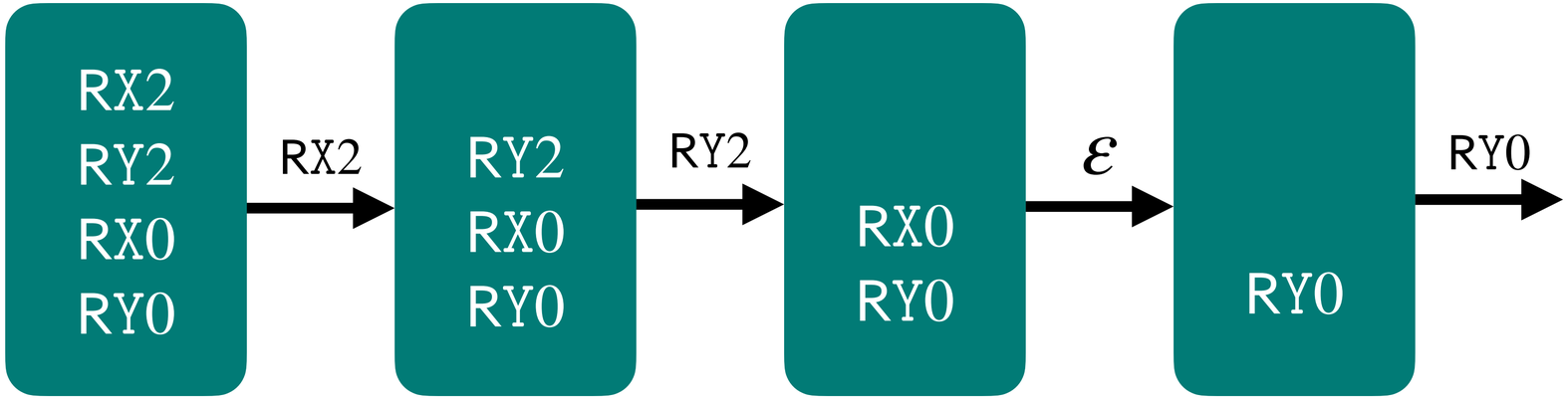
Key Idea



Record the **threads' potentials** in memory states:

what possible sequences of reads each thread can execute?

Lossy SRA



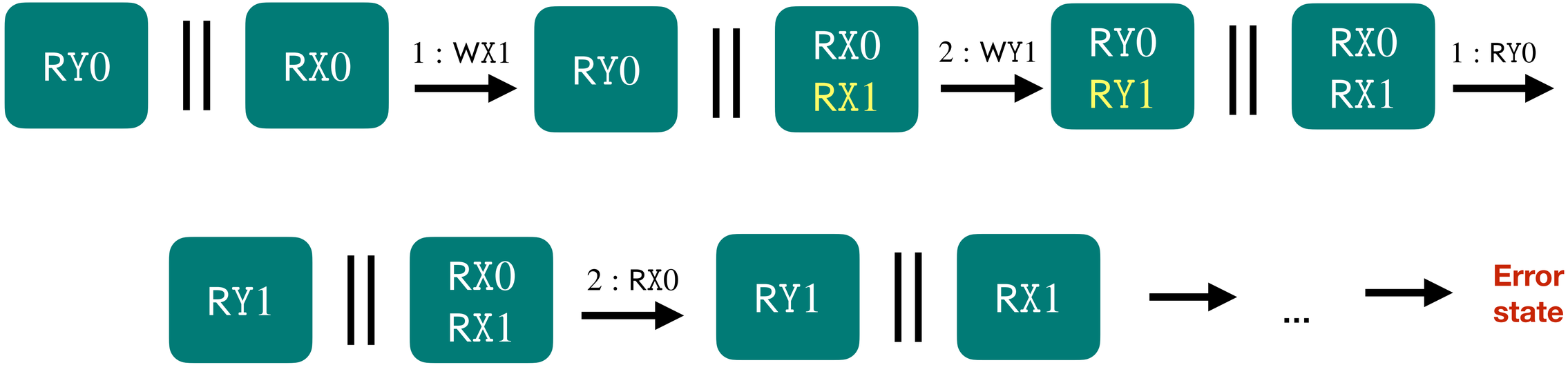
$$Q = \text{Threads} \to \{\text{R}xv \mid x \in \text{Var}, v \in \text{Val}\}^*$$

$$Q_0 = \text{Threads} \to \{\text{R}x0 \mid x \in \text{Var}\}^*$$

$$q \preceq q' \iff \forall \tau \in \text{Threads}\,.\; q(\tau) \sqsubseteq q'(\tau)$$

subsequence

```
X := 1;                 ||   Y := 1;
repeat                  ||   repeat
    a := Y;             ||       b := X;
until (a = 0);          ||   until (b = 0);
Z := 0;                 ||   Z := 1;
assert (Z = 0);         ||   assert (Z = 1);
```



Potential Maintenance for SRA

$\xrightarrow{\varepsilon}$

Lose step

Remove some elements from the potentials

$\tau : \mathsf{R}xv \longrightarrow$

Read steps **Deterministic**

Precondition: first element in τ's potential is $\mathsf{R}xv$

$\tau : \mathsf{W}xv \longrightarrow$

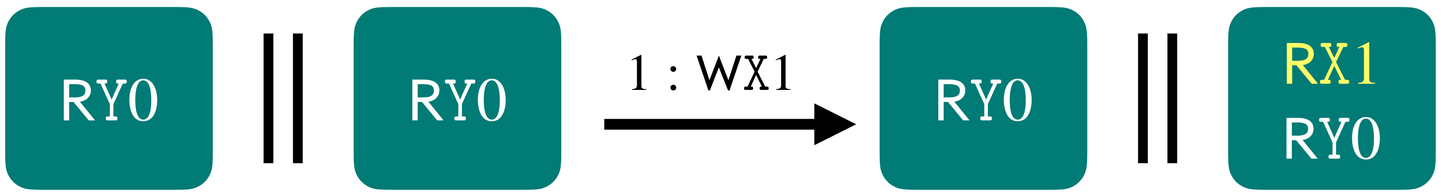
Write steps **Non-Deterministic**

Precondition: no $\mathsf{R}x\_$ in τ's potential

All threads may get new options $\mathsf{R}xv$ → **where?**

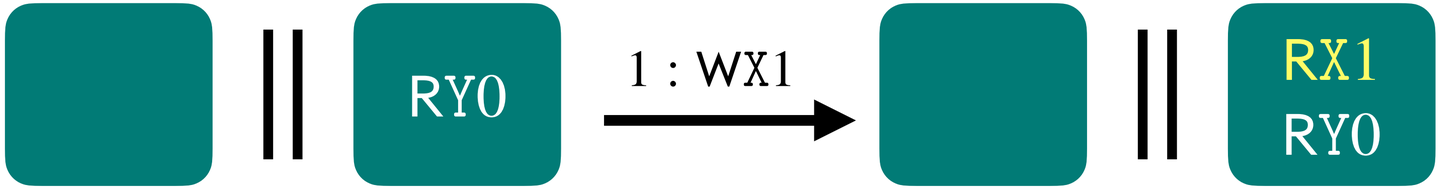
Where?

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



This transition **should be allowed**

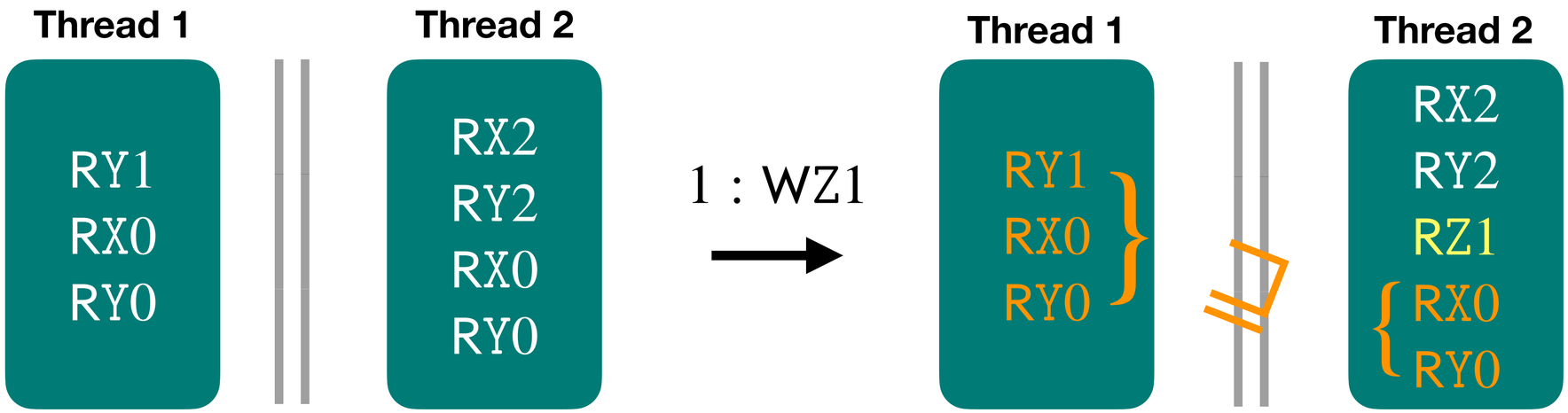
```
X = Y = 0
Y = 1   ||  a = X // 1
X = 1   ||  b = Y // 0
```



This transition **should not be allowed**

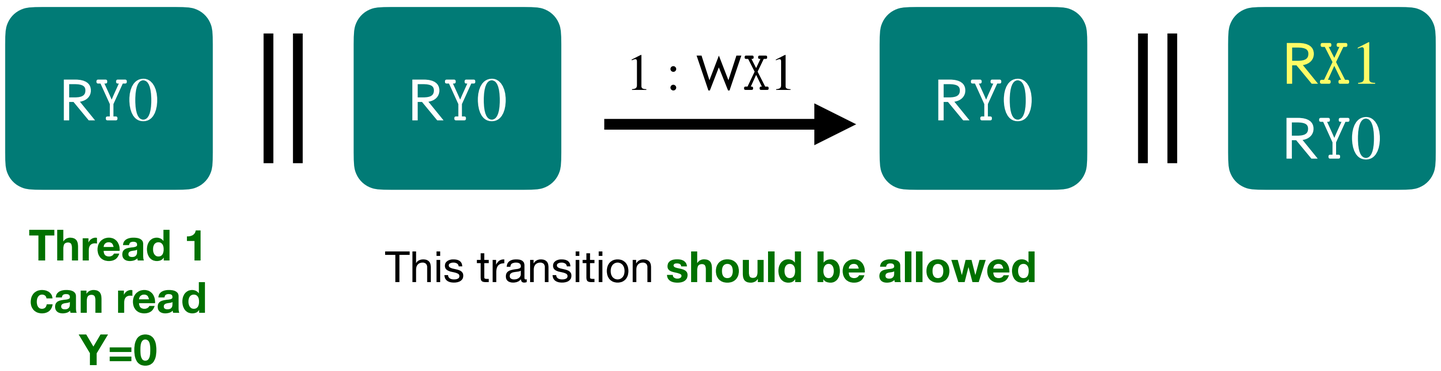
Shared-Memory Causality Principle

Every sequence of reads that thread π can perform after reading from a certain write executed by thread τ could be performed by thread τ immediately after it executed the write.



Where?

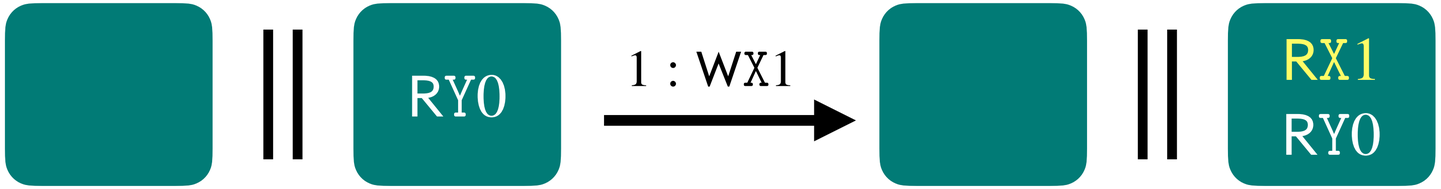
```
X = Y = 0
X = 1   ||  a = X // 1
        ||  b = Y // 0
```



Thread 1 can read Y=0

This transition **should be allowed**

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



Thread 1 cannot read Y=0

This transition **should not be allowed**

More Details

- ***Multiple lists*** per thread
- **Writer thread's id** in "read options"
- Additional flags to handle **RMWs** (atomic Read-Modify-Writes)

WRITE

$$\frac{\begin{array}{l}\forall \pi \in \mathsf{Tid}, L' \in \mathcal{B}'(\pi).\ \exists n \geq 0, u_1, \ldots, u_n, L_0, \ldots, L_n. \\ \quad L' = L_0 \cdot \langle \tau, x, v_{\mathtt{W}}, u_1 \rangle \cdot L_1 \cdot \ldots \cdot \langle \tau, x, v_{\mathtt{W}}, u_n \rangle \cdot L_n \\ \wedge\ L_0 \cdot \ldots \cdot L_n \in \mathcal{B}(\pi)\ \wedge\ L_1 \cdot \ldots \cdot L_n \in \mathcal{B}(\tau) \\ \wedge\ \forall o \in L_1 \cdot \ldots \cdot L_n.\ \mathtt{loc}(o) \neq x \\ \wedge\ \forall o \in L_0.\ \mathtt{loc}(o) = x \implies \pi \neq \tau \wedge \mathtt{rmw}(o) = \mathtt{R}\end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{loSRA}} \mathcal{B}'}$$

RMW

$$\frac{\begin{array}{c}\mathtt{loc}(o) = x \qquad \mathtt{val}(o) = v_{\mathtt{R}} \\ \mathtt{rmw}(o) = \mathtt{RMW} \\ \mathcal{B} = \mathcal{B}_{\mathrm{mid}}[\tau \mapsto o \cdot \mathcal{B}_{\mathrm{mid}}(\tau)] \\ \mathcal{B}_{\mathrm{mid}} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{loSRA}} \mathcal{B}'\end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})}_{\mathsf{loSRA}} \mathcal{B}'}$$

READ

$$\frac{\mathtt{loc}(o) = x \qquad \mathtt{val}(o) = v_{\mathtt{R}} \qquad \mathcal{B} = \mathcal{B}'[\tau \mapsto o \cdot \mathcal{B}'(\tau)]}{\mathcal{B} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\mathsf{loSRA}} \mathcal{B}'}$$

LOWER

$$\frac{\mathcal{B}' \sqsubseteq \mathcal{B}}{\mathcal{B} \xrightarrow{\varepsilon}_{\mathsf{loSRA}} \mathcal{B}'}$$

Multiple Lists per Thread

$x := 0$	$y := 0$	$z := 0$	$d_1 := x$ // 1	$e_1 := y$ // 1	$f_1 := z$ // 1
$x := 1$	$y := 1$	$z := 1$	$d_2 := y$ // 1	$e_2 := z$ // 1	$f_2 := x$ // 1
$a_1 := z$ // 1	$b_1 := x$ // 1	$c_1 := y$ // 1	$d_3 := z$ // 0	$e_3 := x$ // 0	$f_3 := y$ // 0
$a_2 := y$ // 0	$b_2 := z$ // 0	$c_2 := x$ // 0			

Ry1
Rx0

Rx1
Ry0

Rx1
Ry0

Writer Thread in “Read Options”

x := 0	y := 0	a := z // 1	c := w // 1
x := 1	y := 1	w := 1	d := y // 0
z := 1	z := 1	b := x // 0	

Rz1 Ry0	Rz1 Rx0

1 : Rz1 Ry0	2 : Rz1 Rx0

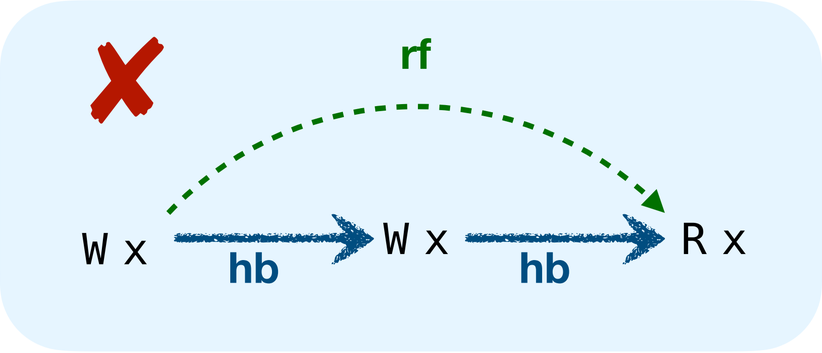
Three Variants

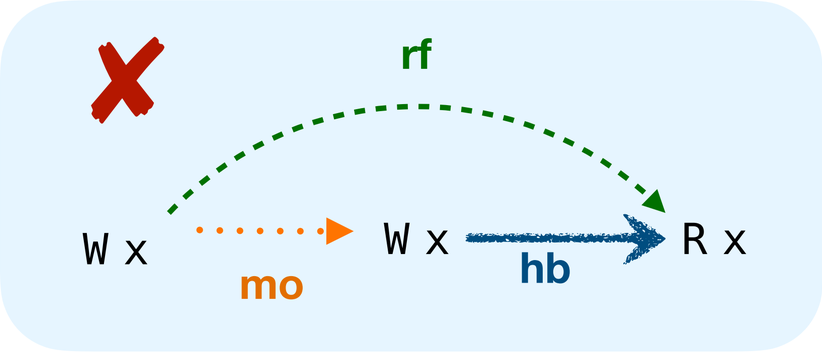
Weak Release/Acquire
(**WRA**)

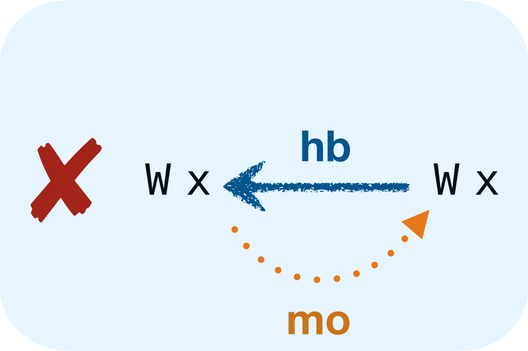
Release/Acquire
(**RA**)

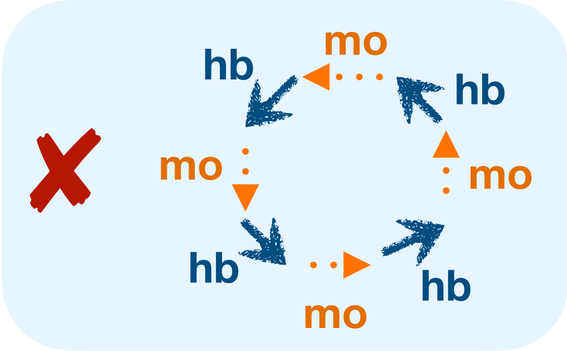
Strong Release/Acquire
(**SRA**)

∃ total order on writes to the same location (**modification-order**) s.t.:









Potential-Based System for WRA

τ : Wxv →

Write steps

Precondition: no Rx_ in τ's potential

All threads may get new options Rxv

```
x = 1   ||   x = 2   ||   x = 3
```

RX1
RX2
RX1

```
x = 1   ||   x = 2   ||   a = x // 1
                          x = 3
                          a = x // 1
```

~~RX1~~
RX2
RX1

Potential-Based System for WRA

Use "**write options**" to mark when writes are allowed

+ Simple constraints on where read options are added wrt write options

```
x = 1 || x = 2 || x = 3
```

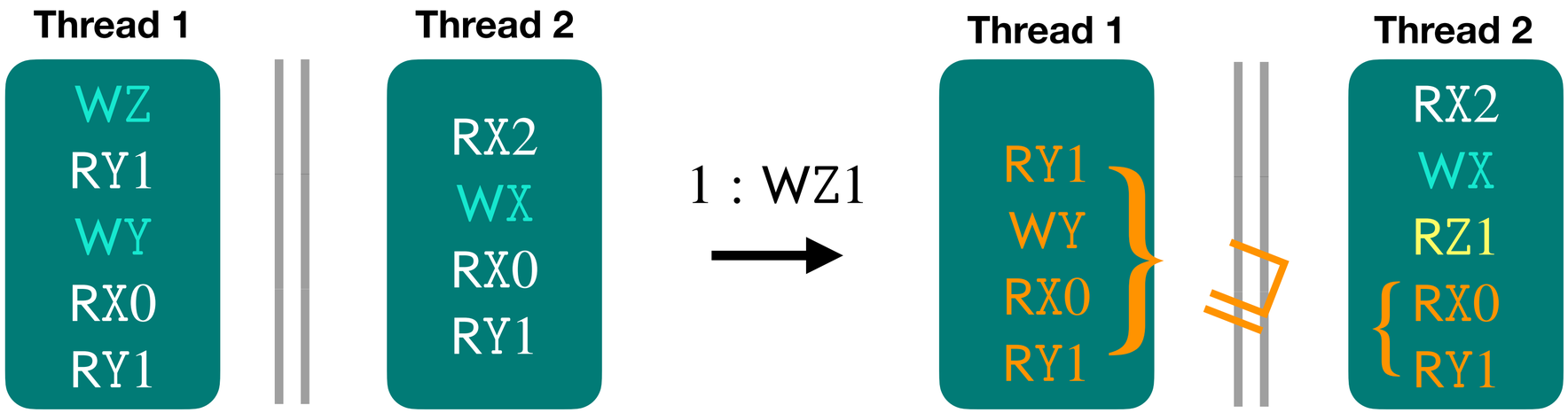
WX
RX1
RX2
RX1

```
x = 1 || x = 2 || a = x // 1
                   x = 3
```

~~RX1~~
RX2
RX1

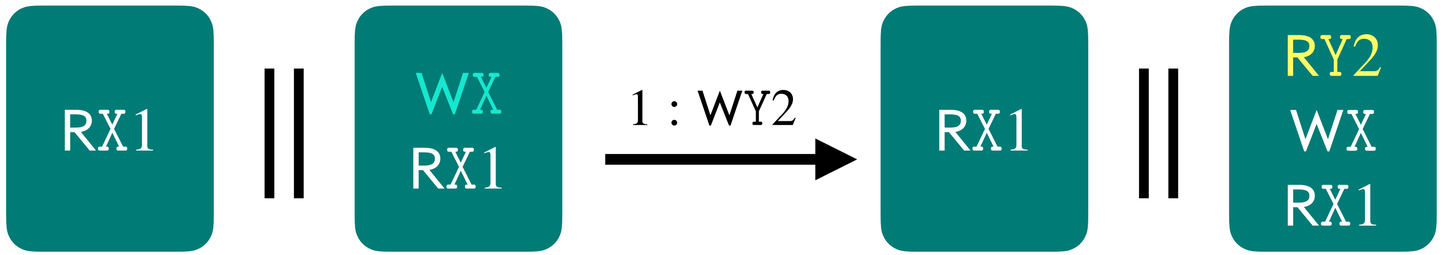
Shared-Memory Causality Principle

Every sequence of reads ***and writes*** *that thread π can perform after reading from a certain write executed by thread τ could be performed by thread τ immediately after it executed the write.*



```
X = Y = 0

X = 1          a = Y // 2
Y = 2          X = 3
               b = X // 1
```



Thread 1 cannot write to X and then read X=1

This transition **is disallowed**

Results

Theorem

The potential-based memory systems are **equivalent** to the SRA/WRA systems.

Theorem

When synchronized with a (finite-state) concurrent program, the potential-based memory systems form **WSTS**.

Results

Theorem

The verification problems under **SRA** and **WRA** are **decidable**.

Corollary

The verification problem under **RA** is **decidable** for write-write-race-free programs.

Research Questions

- Useful **implementation**
- **RA** without RMWs?
- **Other models** and extensions of causal consistency (get closer to RA?)
- **Parametrized** programs
- Use the potential-based semantics for **other verification approaches**


erc
Interested in concurrency & verification?
I'm looking for students / postdocs!