June 21 2024
PODC, Nantes, France

Tutorial

Weak memory models in programming language semantics

Ori Lahav

About me

https://www.cs.tau.ac.il/~orilahav/ orilahav@tau.ac.il

- PhD from TAU in **logic in CS**

- Postdocs: **formal verification** (TAU) and **weak memory concurrency** (MPI-SWS)

- Now professor at TAU,
  main areas of research: programming languages theory, concurrency, verification

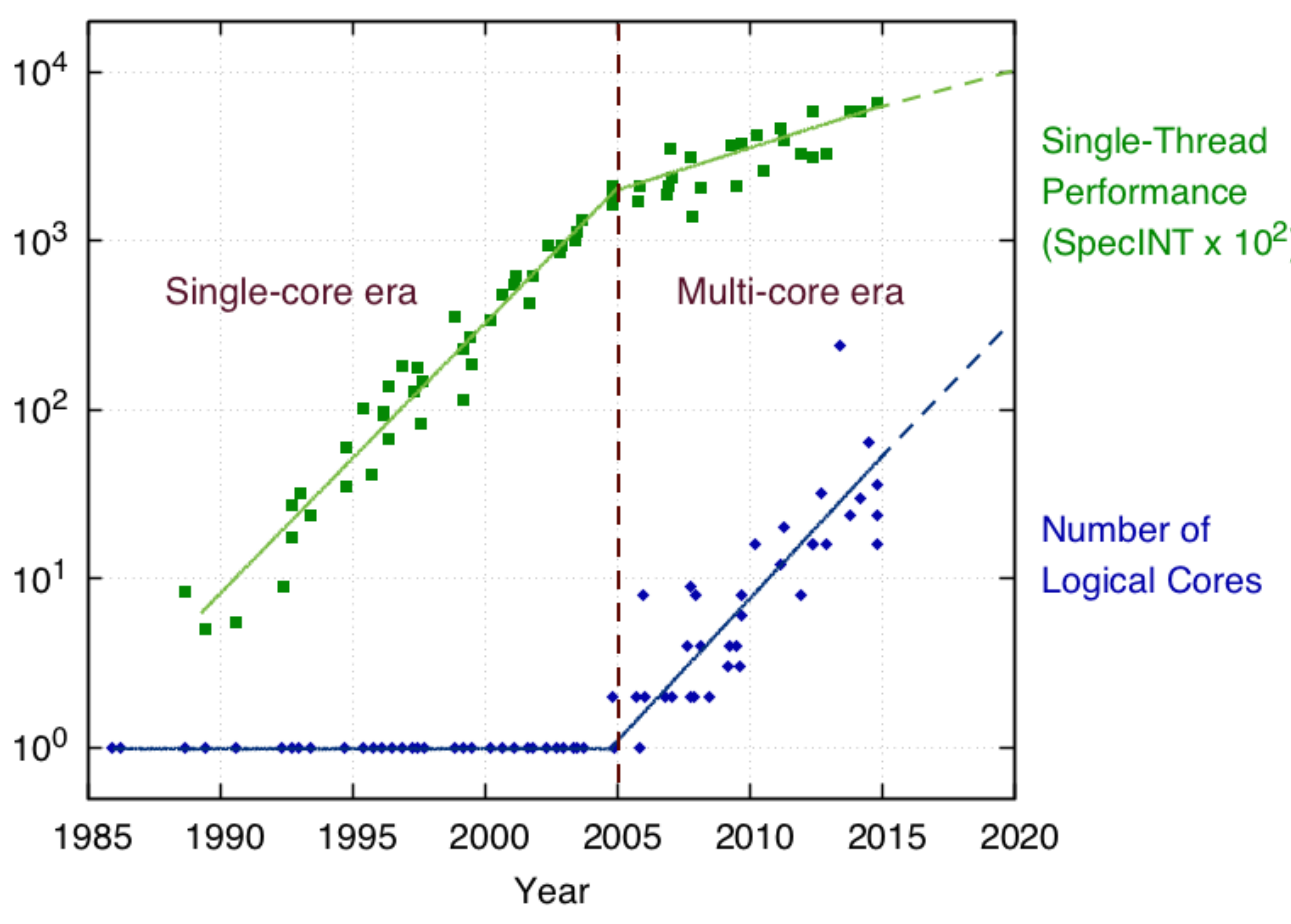
- ERC Starting Grant (hiring students/postdocs)

Agenda

1. Introduction
2. The C/C++11 memory model
3. The out-of-thin-air problem & RC11
4. Implementability of (R)C11: compiler optimizations and mapping to hardware
5. Programmability guarantees: DRF theorems, library abstraction
6. Verification (short survey of problems and results)

Parallelism is here

"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software"/ Herb Sutter (2005)



Concurrent programming is hard!

"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."

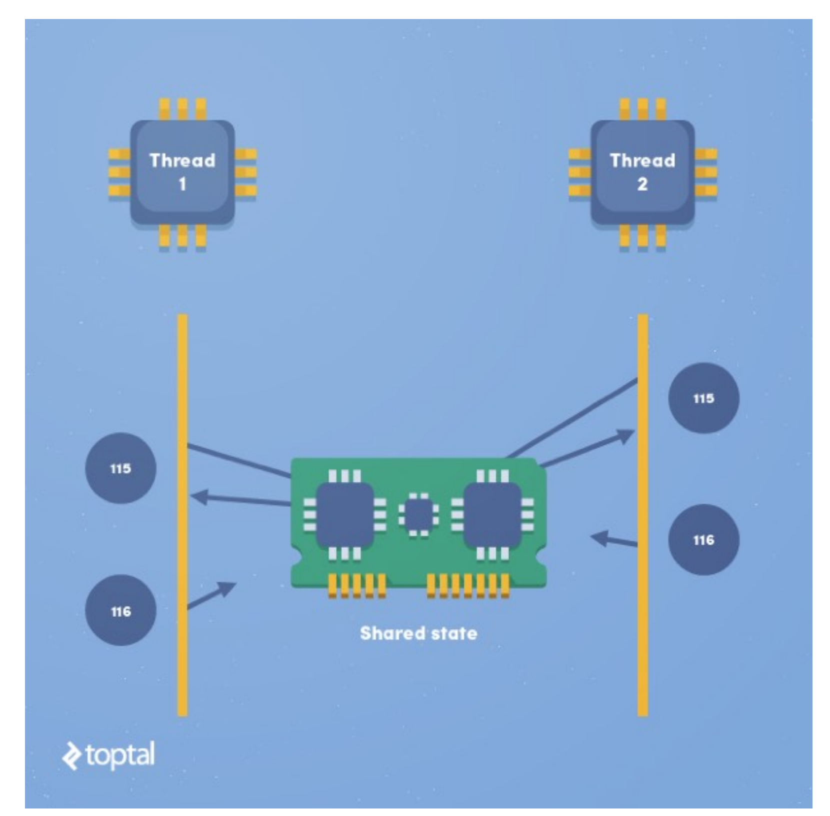
(Java documentation, ~25 years ago)

Why?

- Requires a fundamentally different way of thinking
- Interference among threads
- Inevitable non-determinism
- Testing is ineffective
- Reproducing bugs and debugging is hard

Concurrent programming

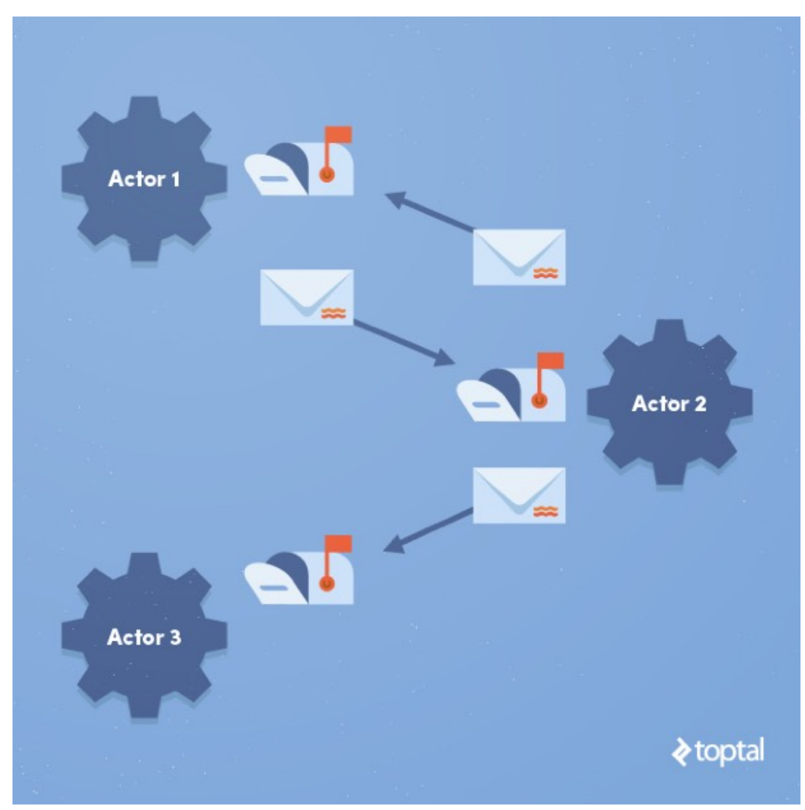
shared memory



interaction by reading and
writing shared objects in memory

```
store/write
load/read
read-modify-write (e.g. CAS, FADD)
lock & unlock
```

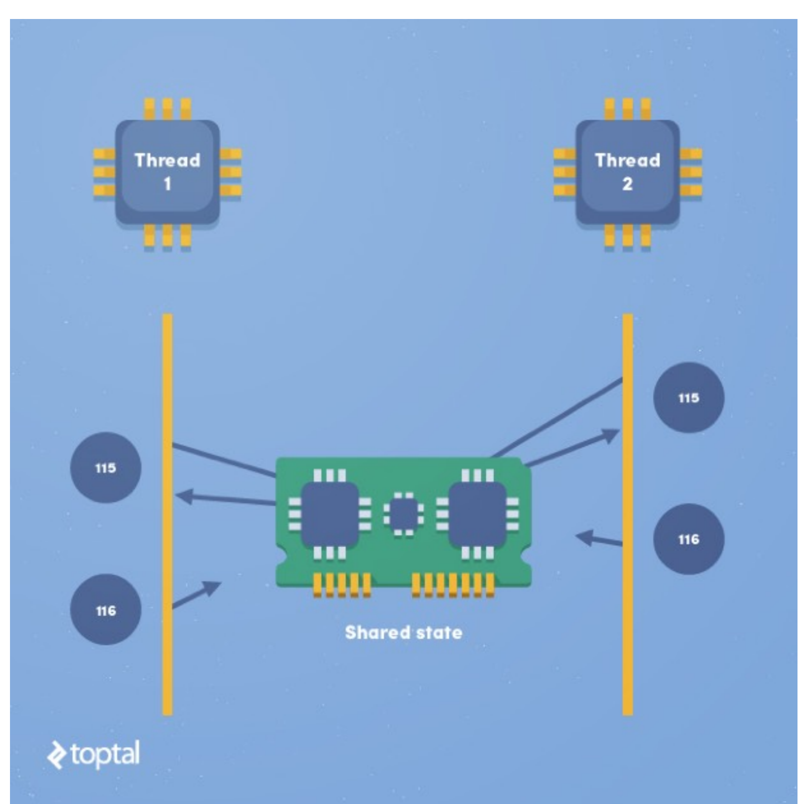
message passing



interaction by sending messages to each
other through a communication channel

Concurrent programming

shared memory



interaction by reading and
writing shared objects in memory

```
store/write
load/read
read-modify-write (e.g. CAS, FADD)
lock & unlock
```

Dekker's mutual exclusion

wants_to_enter[0] ← false, wants_to_enter[1] ← false, turn ← 0   // or 1

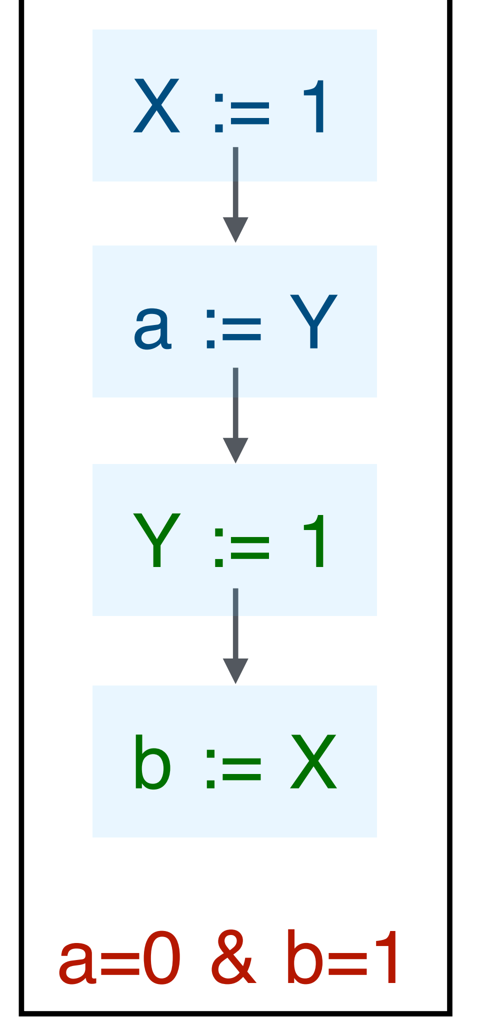
```
wants_to_enter[0] ← true
 while (wants_to_enter[1]) {
   if (turn ≠ 0) {
     wants_to_enter[0] ← false
     while (turn ≠ 0) { // busy wait }
     wants_to_enter[0] ← true
   }}
 // critical section
 ...
 turn ← 1
 wants_to_enter[0] ← false
 // remainder section
```

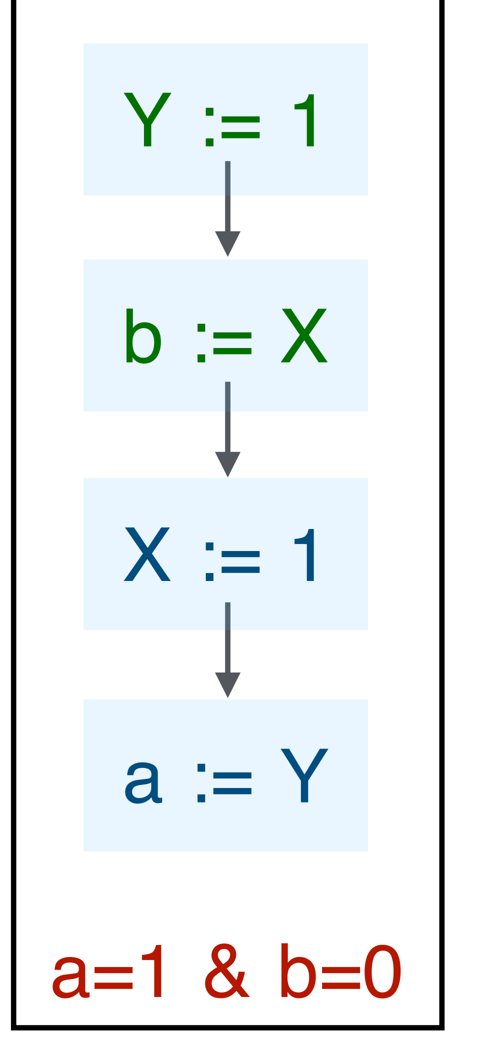
```
wants_to_enter[1] ← true
 while (wants_to_enter[0]) {
   if (turn ≠ 1) {
     wants_to_enter[1] ← false
     while (turn ≠ 1) { // busy wait }
     wants_to_enter[1] ← true
   }}
 // critical section
 ...
 turn ← 0
 wants_to_enter[1] ← false
 // remainder section
```

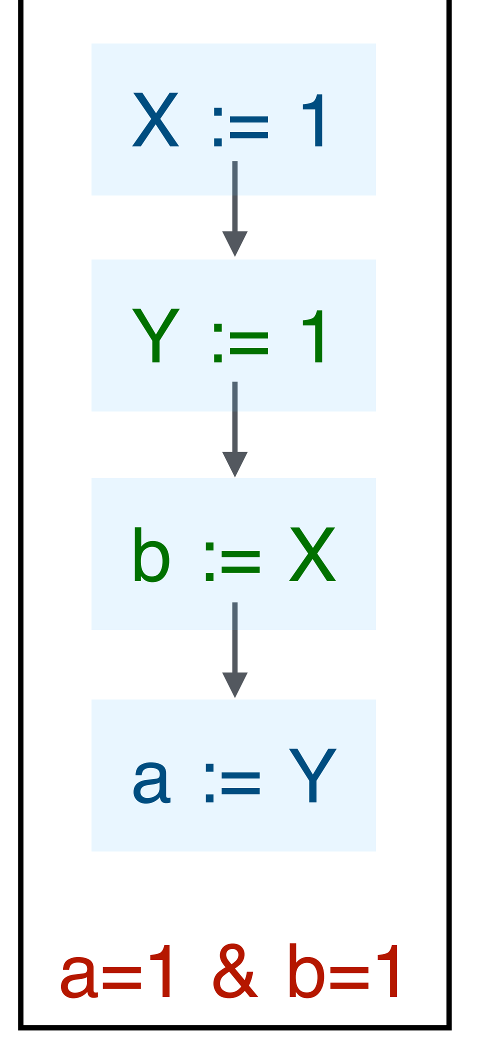
Example

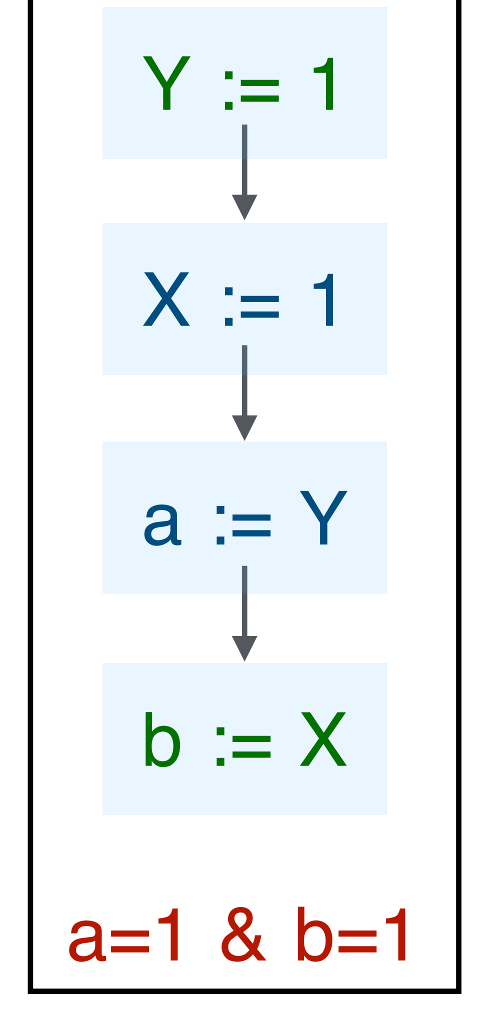
initially, `X=Y=0`

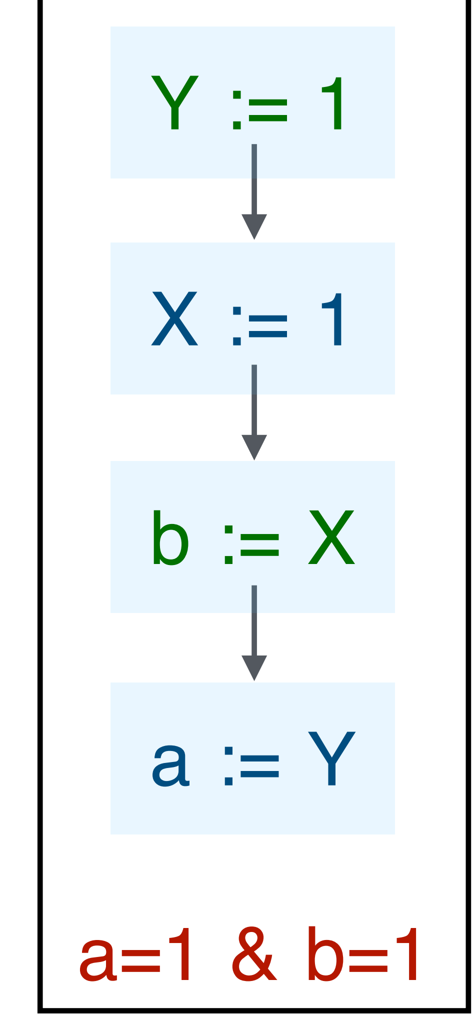
`X := 1` `a := Y // 0`	`Y := 1` `b := X // 0`

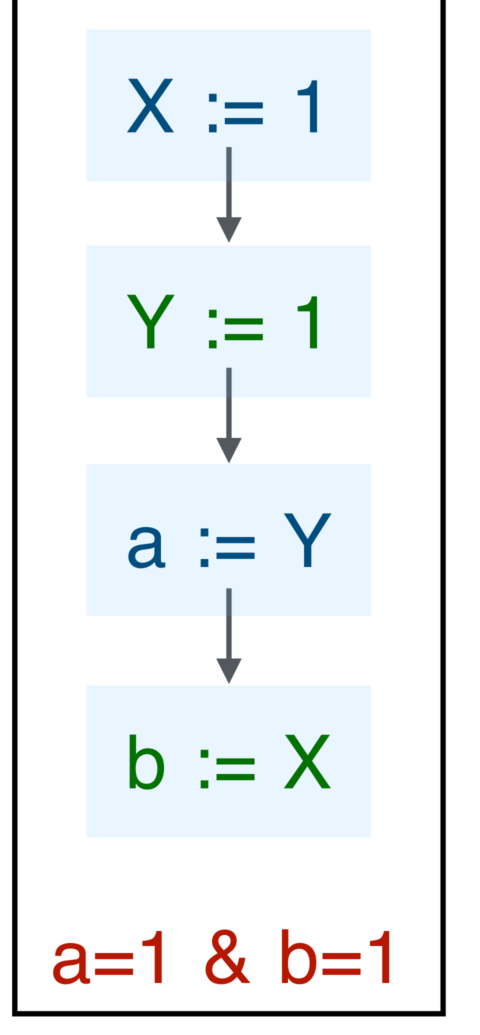












Demo

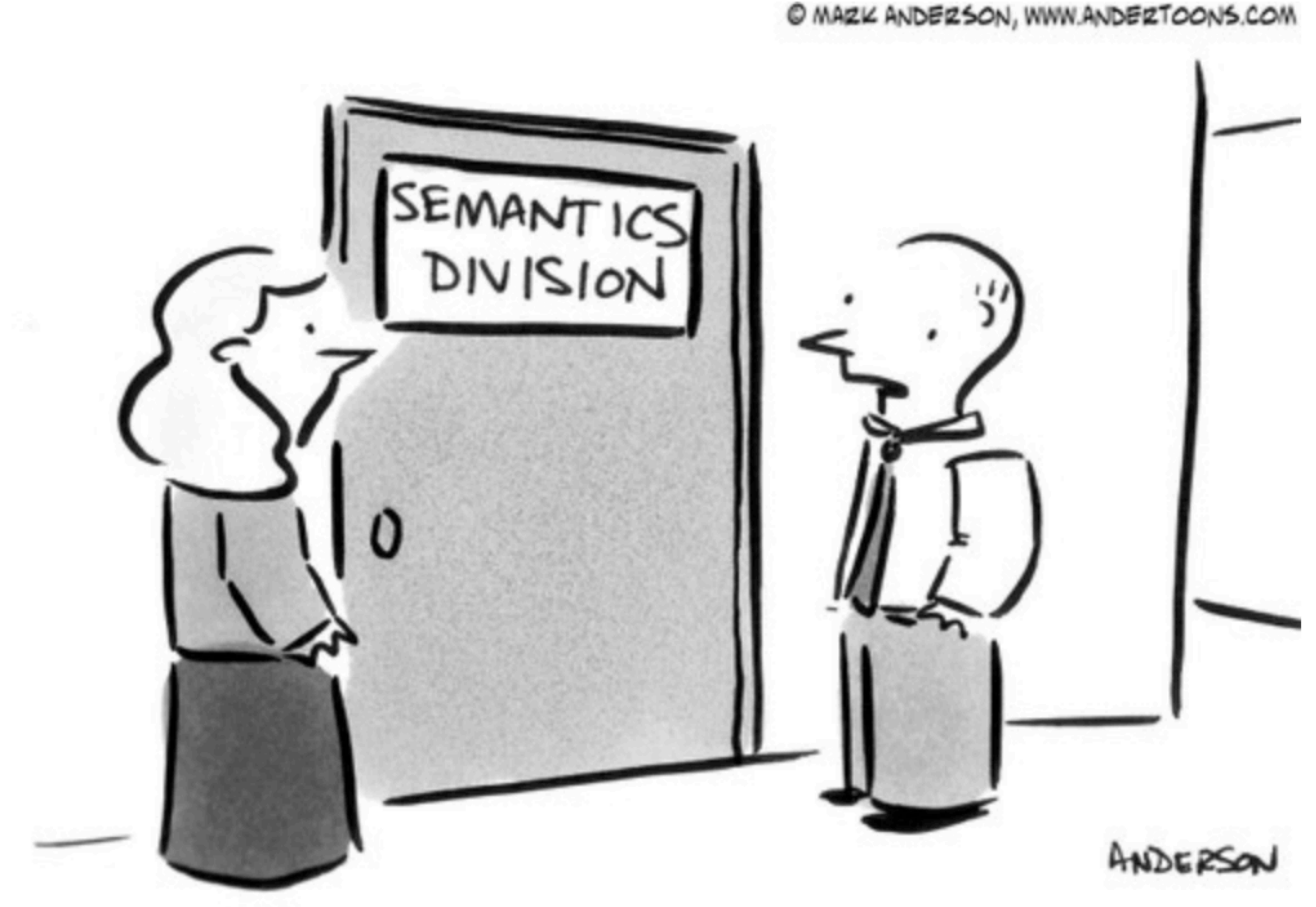
```
int X, Y, a, b;

void thread1() {
    X = 1;
    a = Y;
}

void thread2() {
    Y = 1;
    b = X;
}
```

```
int main () {
    int cnt = 0;

    do {
        X = 0; Y = 0;

        thread first(thread1);
        thread second(thread2);

        first.join();
        second.join();
        cnt++;

    } while (a != 0 || b != 0);

    printf("%d\n",cnt);
    return 0;
}
```

How come airplanes don't crash?

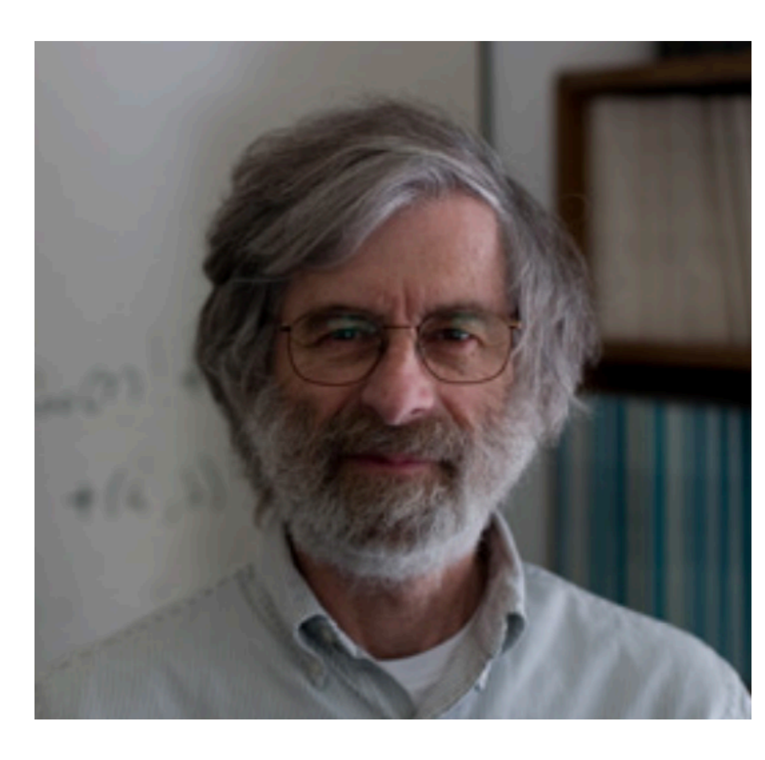
- There are ways to demand strong semantics when we need it
- We often don't need strong semantics in its full power

Before programming/verification, we need semantics

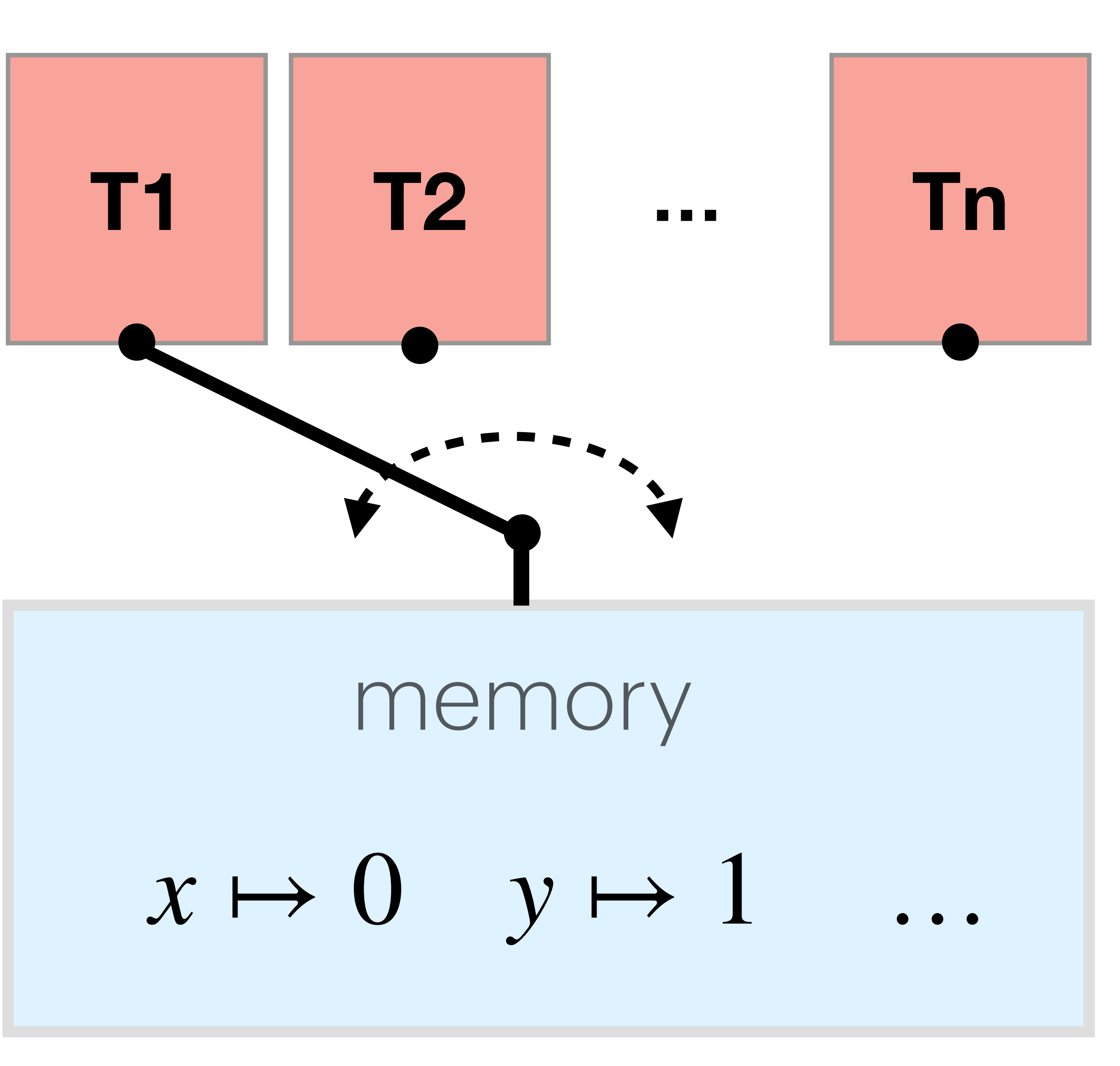


"We're really more of a department."

Sequential consistency (SC)



...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...



Lamport. 1979. **How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.** IEEE Trans. Comput. https://doi.org/10.1109/TC.1979.1675439

The requirements needed to guarantee sequential consistency rule out some techniques which can be used to speed up individual sequential processors. For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs. Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task.

Lamport. 1979. **How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.** IEEE Trans. Comput. https://doi.org/10.1109/TC.1979.1675439

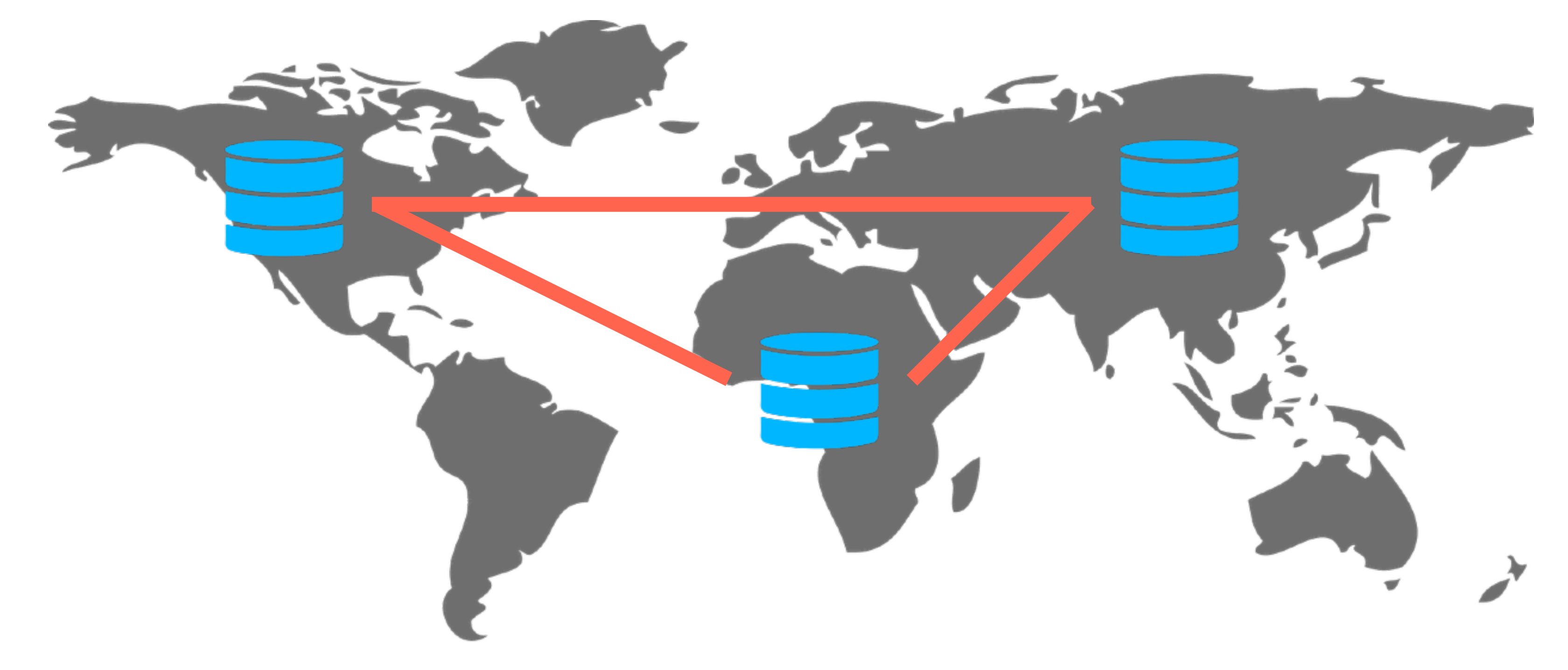
SC is unrealistic

- for better performance/scalability shared-memory implementations perform various optimizations:
  - local store buffers
  - out-of-order execution
  - hierarchies of caches
  - ...
- Compilers further stir the pot by performing thread-local program optimizations
- These optimizations are:
  - unobservable in sequential programs
  - but can be observed by concurrent code!

Weak consistency in distributed systems

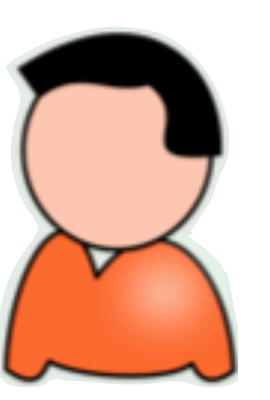
```
send(X = 1)
get(Y) // 0
```

```
send(Y = 1)
get(X) // 0
```



```
Email := “dear bob, ...”
Sms := “check your email”
```

```
a := Sms // “check your email”
b := Email // “no new email”
```



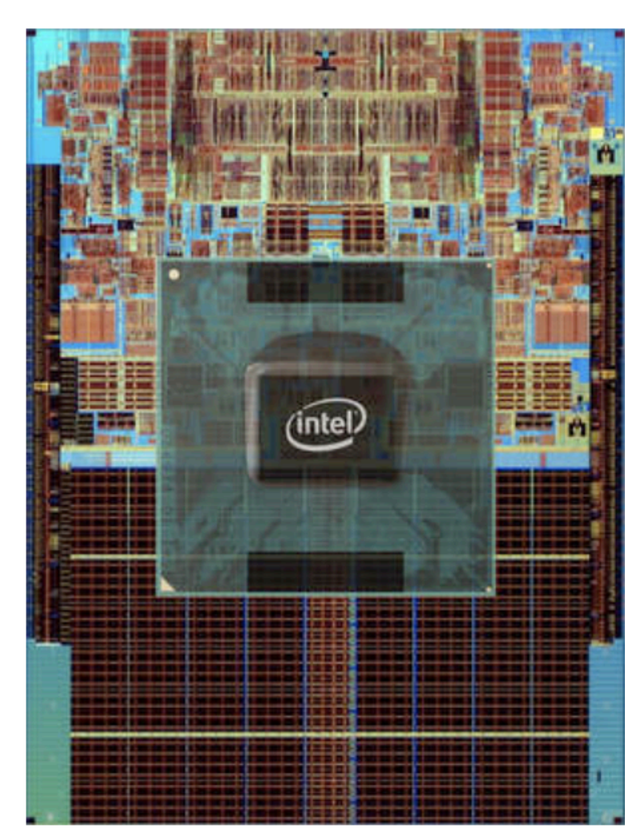
Weak memory models

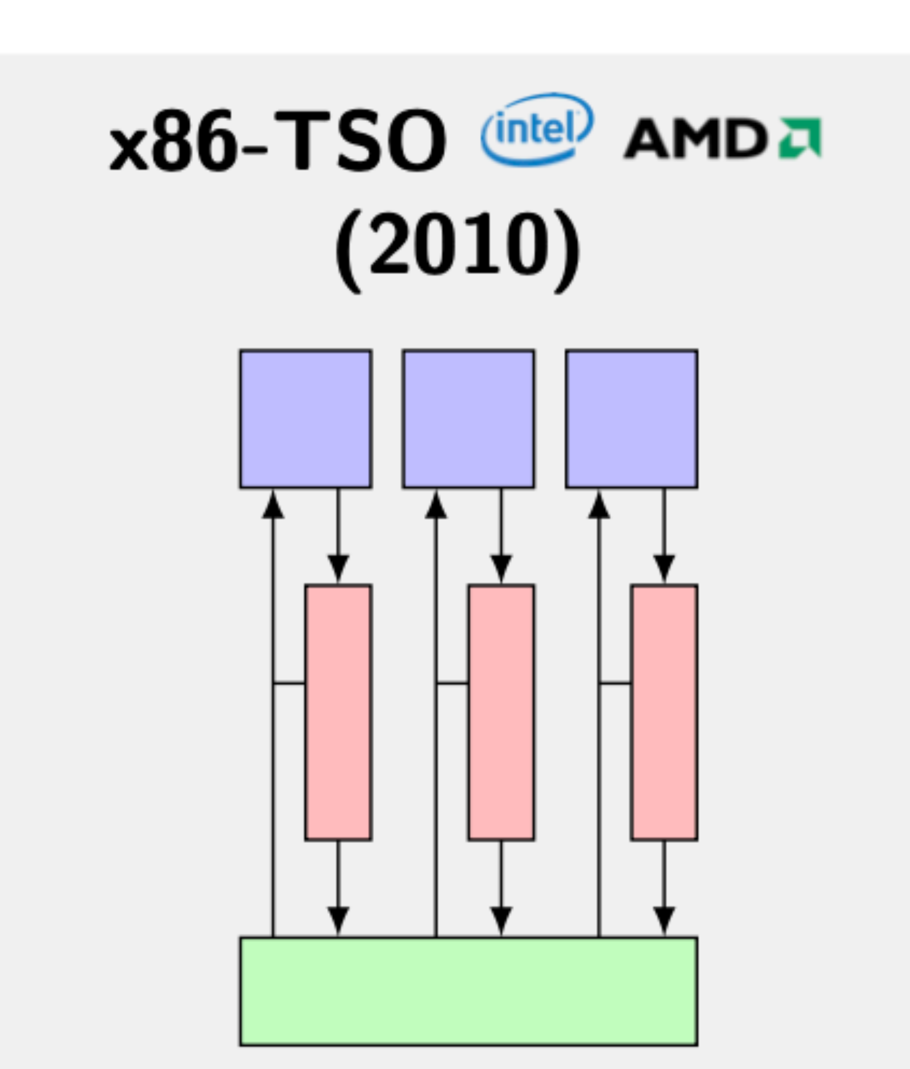
- A formal interface between the user and the implementation:
  - What are the possible behaviors of a concurrent program?
  - More concretely, what values each read may return?

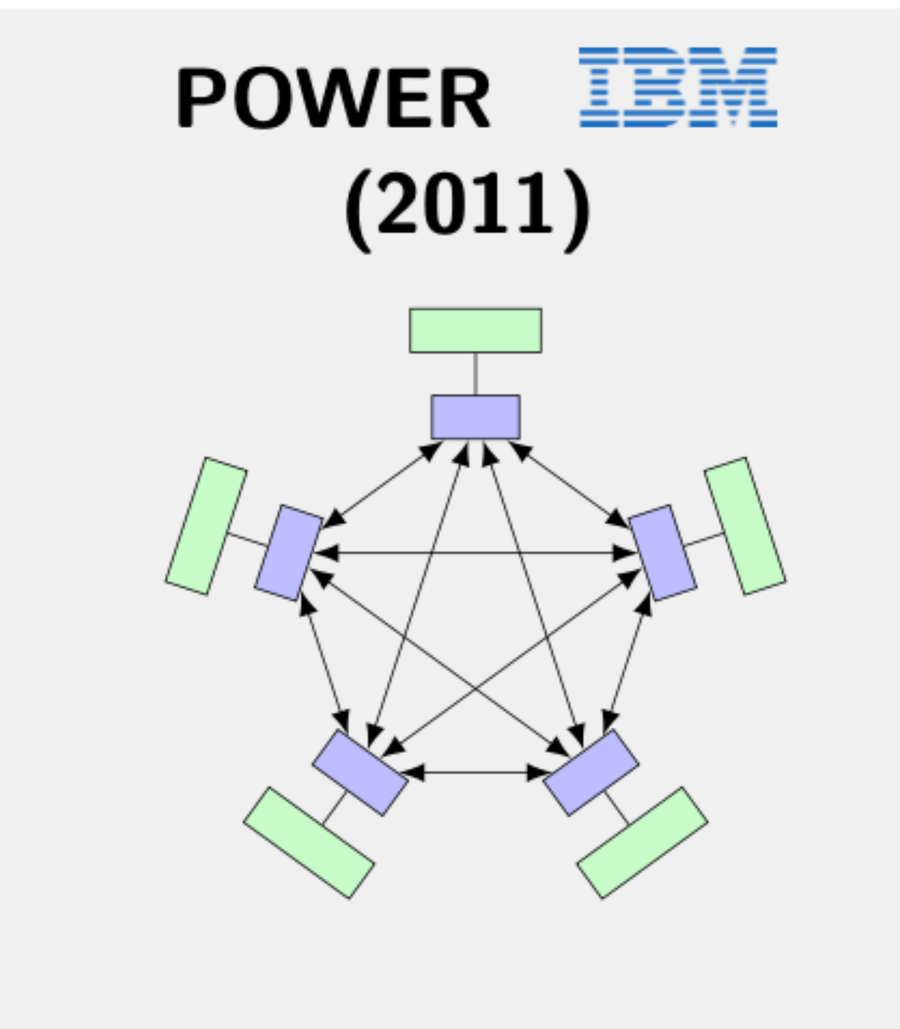
- A weak memory model (WMM) allows all outcomes allowed by SC *and more*

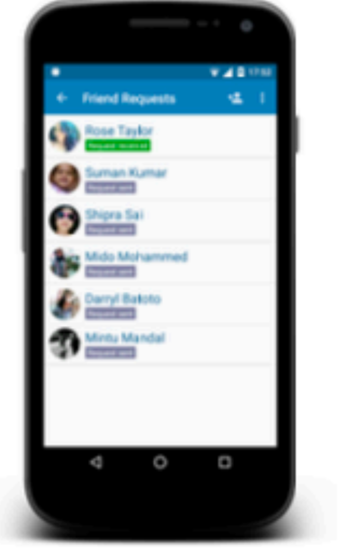
Hardware memory models

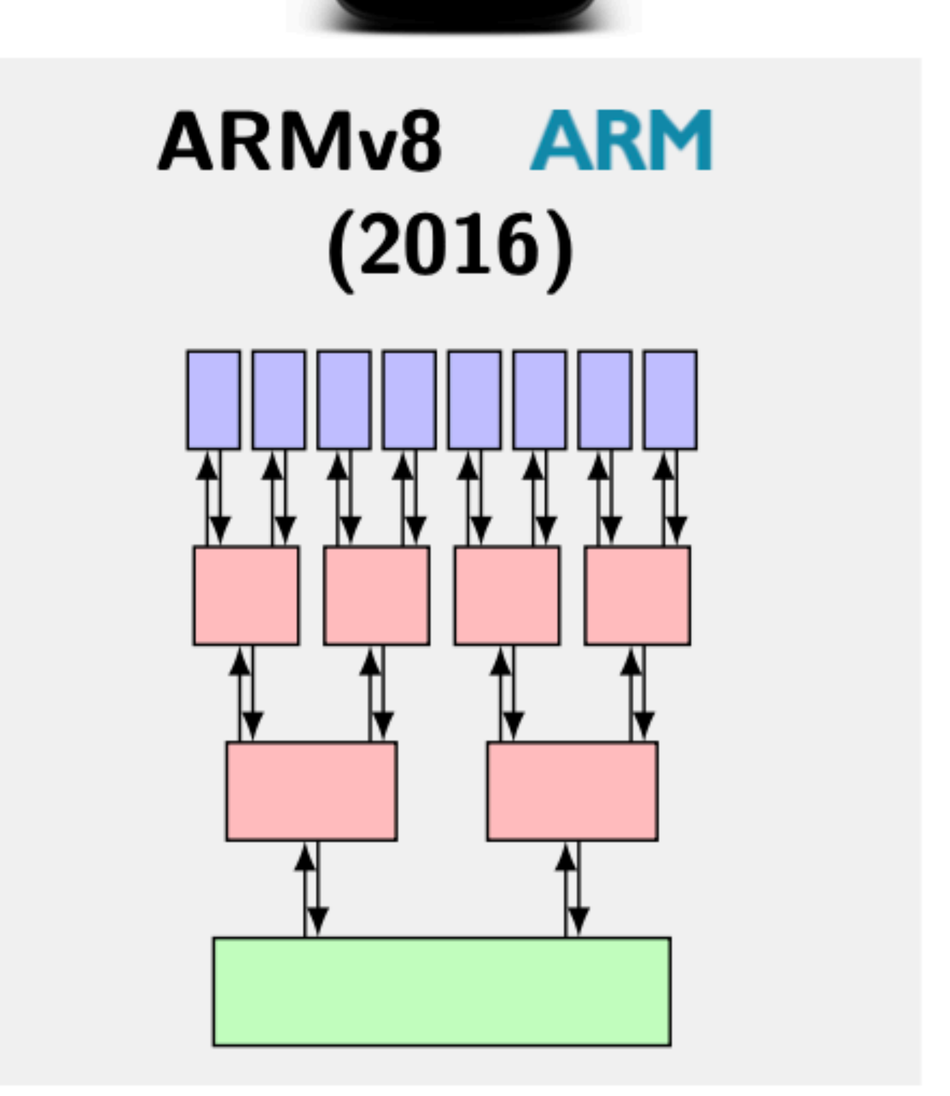
- Each architecture has its own WMM: x86-TSO, ARM, Power, RISC-V...
- Often: subtle differences
- None of them is SC







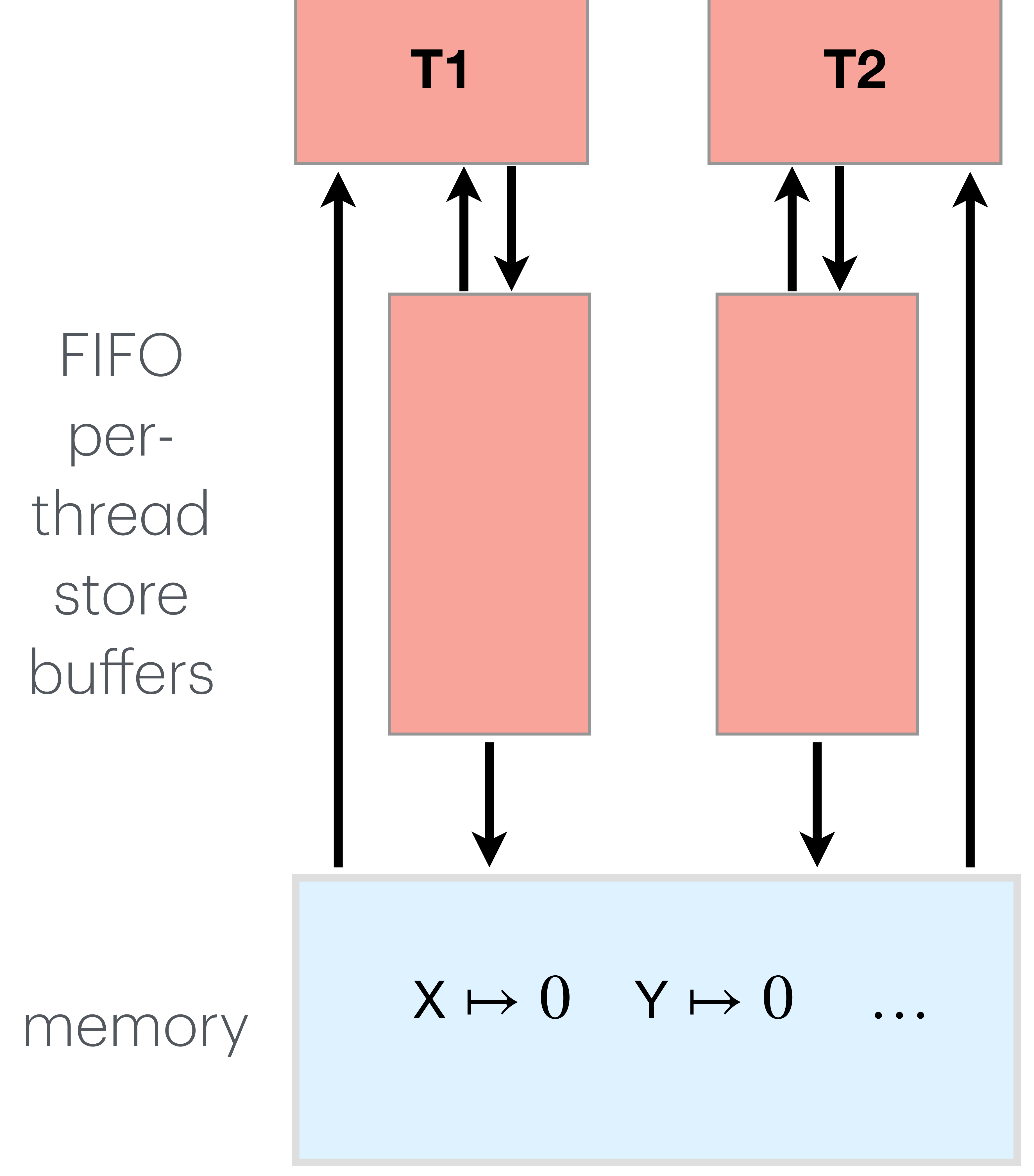




x86-TSO

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

`X := 1`
`fence()`
`a := Y // 0`
	`Y := 1`
	`fence()`
	`b := X // 0`

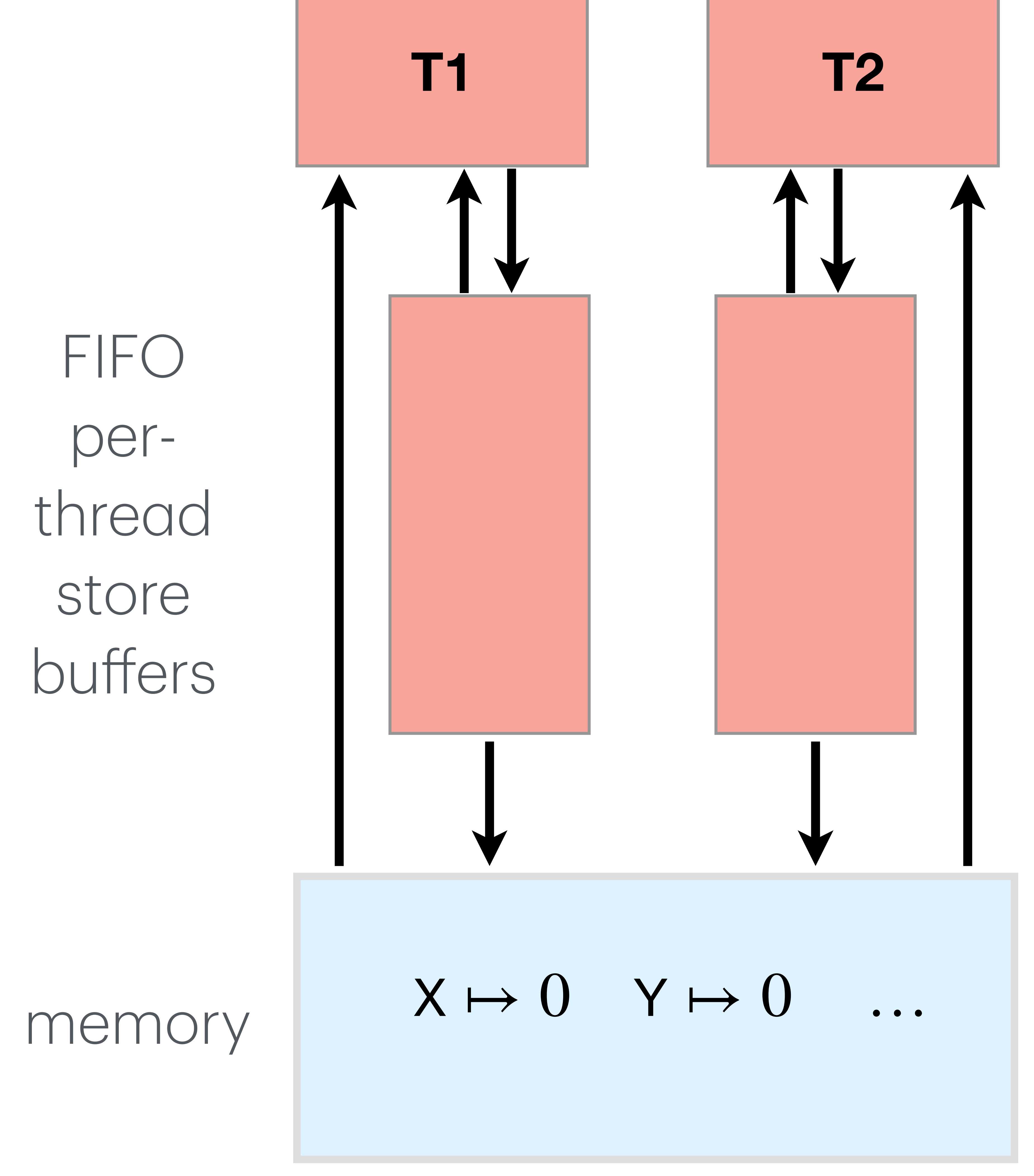


Sewell, Sarkar, Owens, Zappa Nardelli, Myreen: **x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors.** Commun. ACM 53(7) 2010. https://doi.org/10.1145/1785414.1785443

x86-TSO

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

```
X := 1          Y := 1
fence()         fence()
a := Y // 0     b := X // 0
```



Sewell, Sarkar, Owens, Zappa Nardelli, Myreen: **x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors.** Commun. ACM 53(7) 2010. https://doi.org/10.1145/1785414.1785443

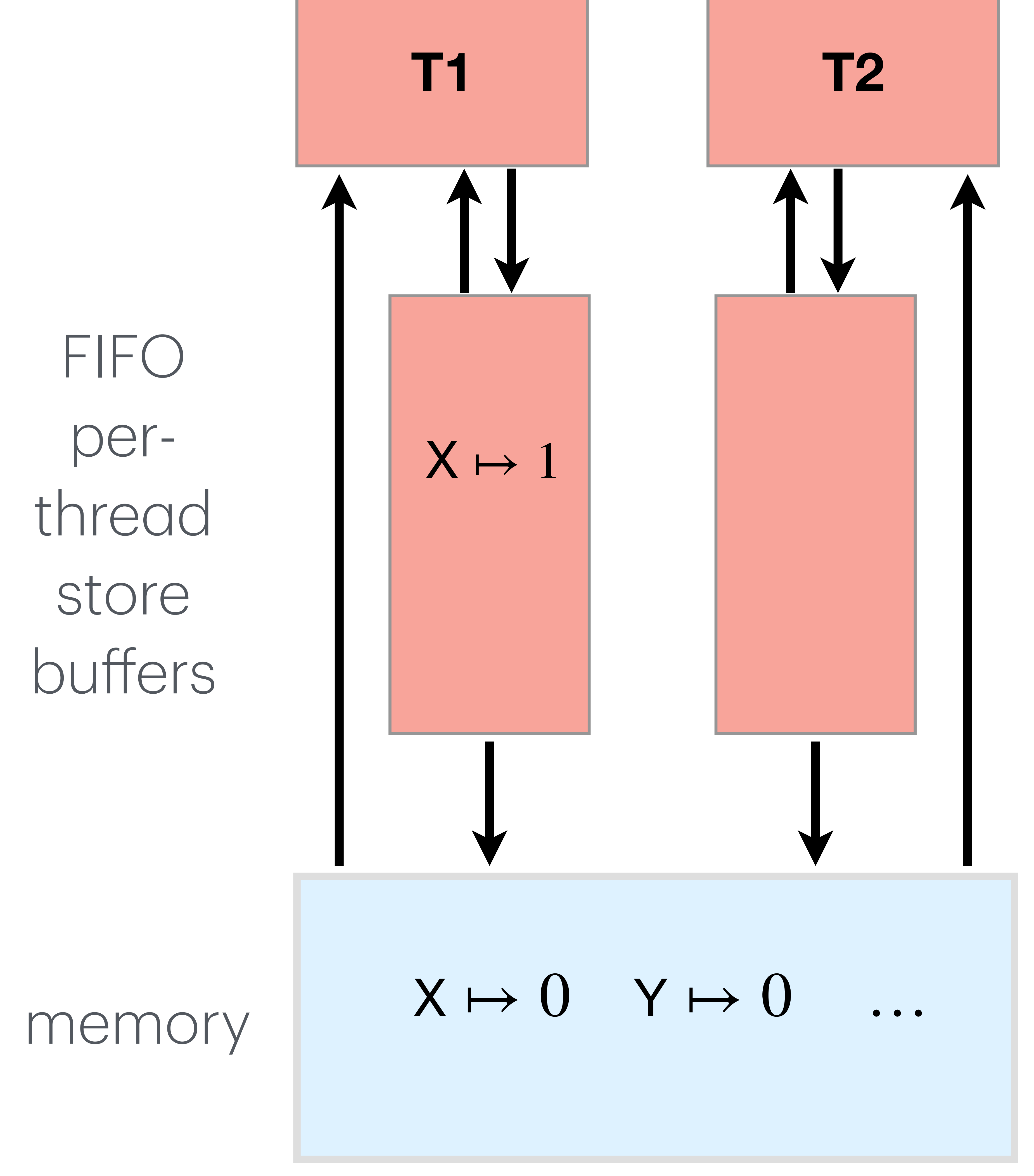
x86-TSO

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

```
X := 1          Y := 1
fence()         fence()
a := Y // 0     b := X // 0
```

FIFO per-thread store buffers

memory



Sewell, Sarkar, Owens, Zappa Nardelli, Myreen: **x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors.** Commun. ACM 53(7) 2010. https://doi.org/10.1145/1785414.1785443

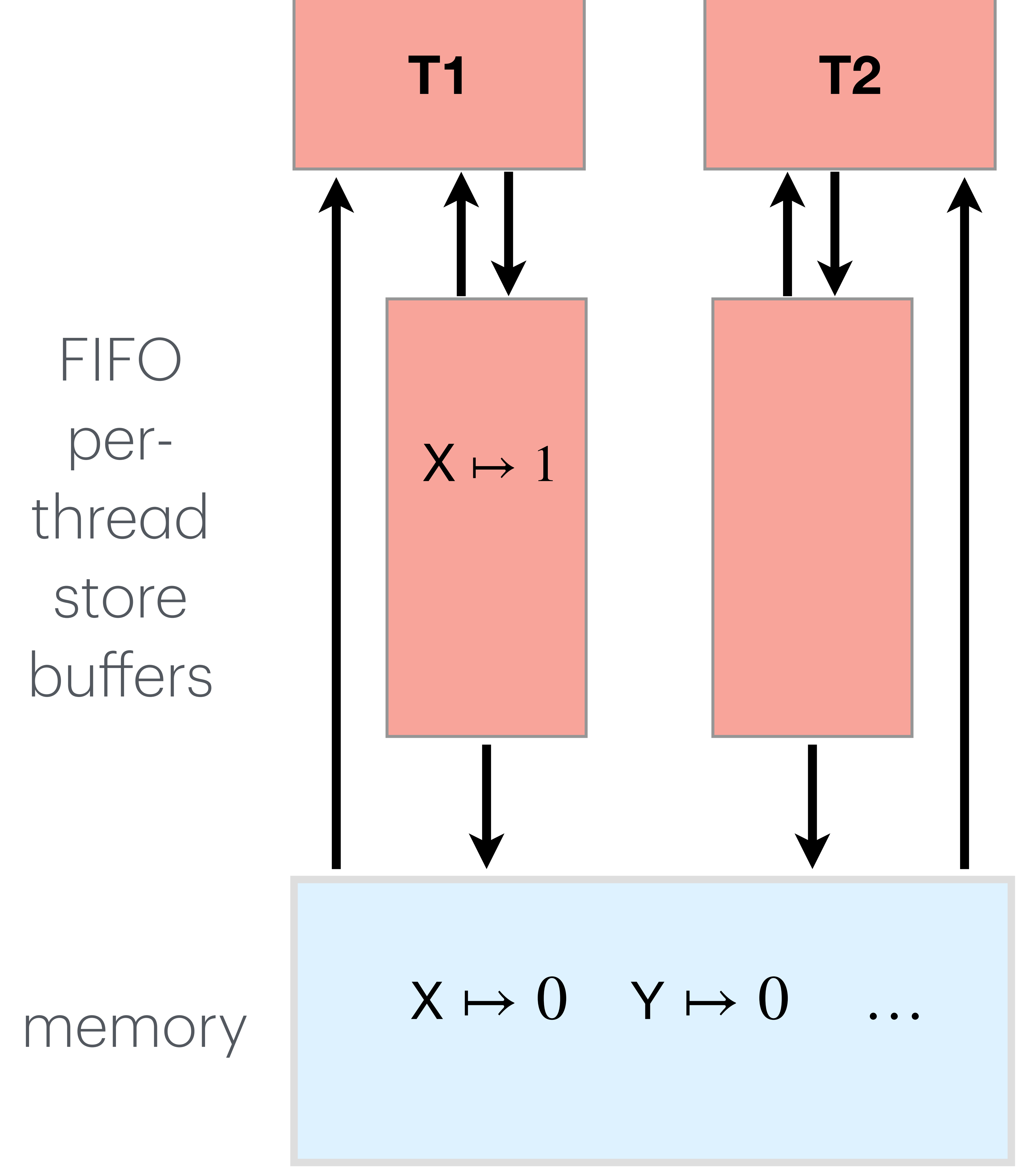
x86-TSO

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

```
X := 1            Y := 1
fence()           fence()
a := Y // 0       b := X // 0
```

FIFO per-thread store buffers

memory



Sewell, Sarkar, Owens, Zappa Nardelli, Myreen: **x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors.** Commun. ACM 53(7) 2010. https://doi.org/10.1145/1785414.1785443

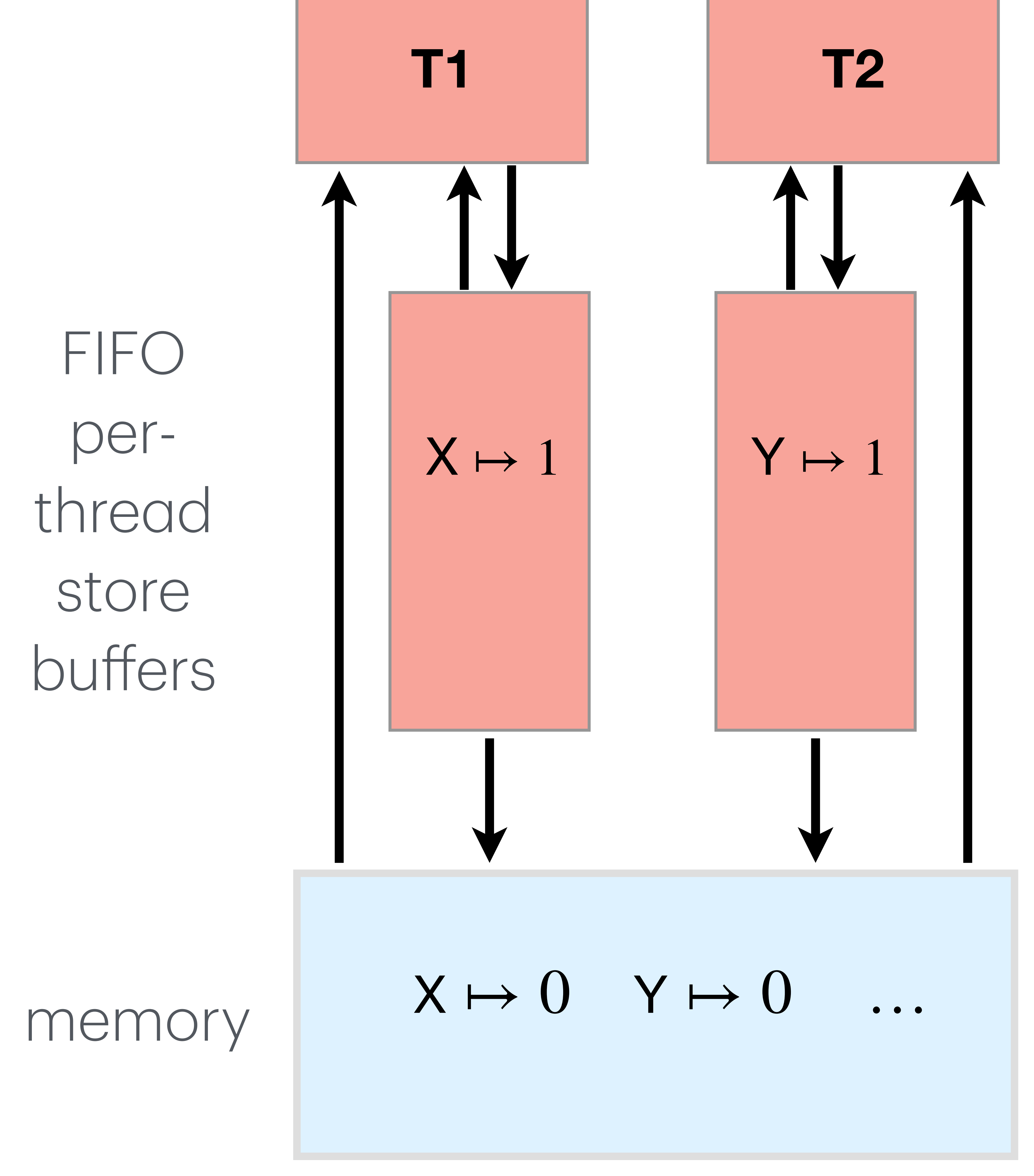
x86-TSO

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

```
X := 1              || Y := 1
fence()             || fence()
a := Y // 0         || b := X // 0
```

FIFO per-thread store buffers

memory



Sewell, Sarkar, Owens, Zappa Nardelli, Myreen: **x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors.** Commun. ACM 53(7) 2010. https://doi.org/10.1145/1785414.1785443

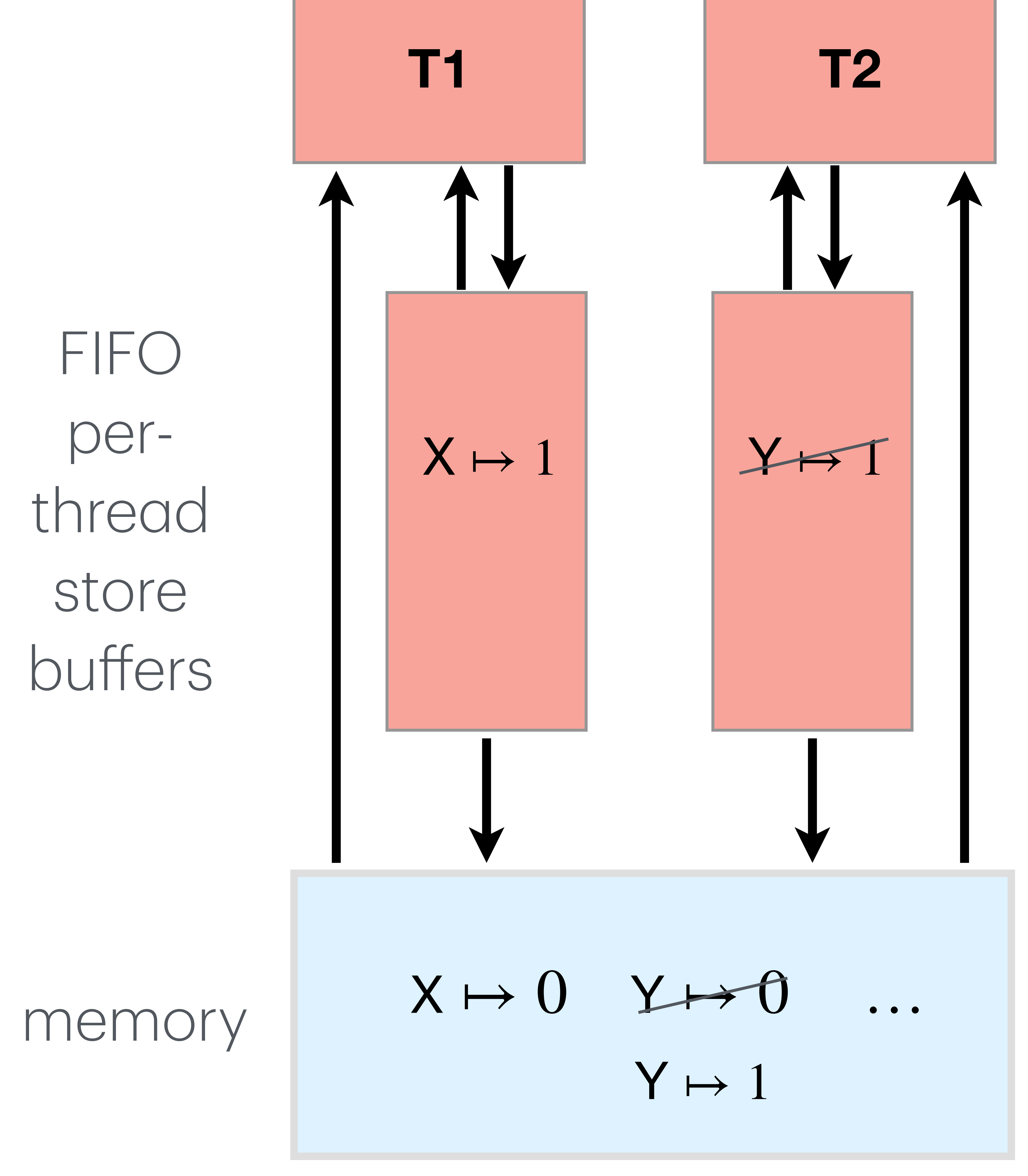
x86-TSO

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

```
X := 1              ‖  Y := 1
fence()             ‖  fence()
a := Y // 0         ‖  b := X // 0
```

FIFO per-thread store buffers

memory



Sewell, Sarkar, Owens, Zappa Nardelli, Myreen: **x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors.** Commun. ACM 53(7) 2010. https://doi.org/10.1145/1785414.1785443

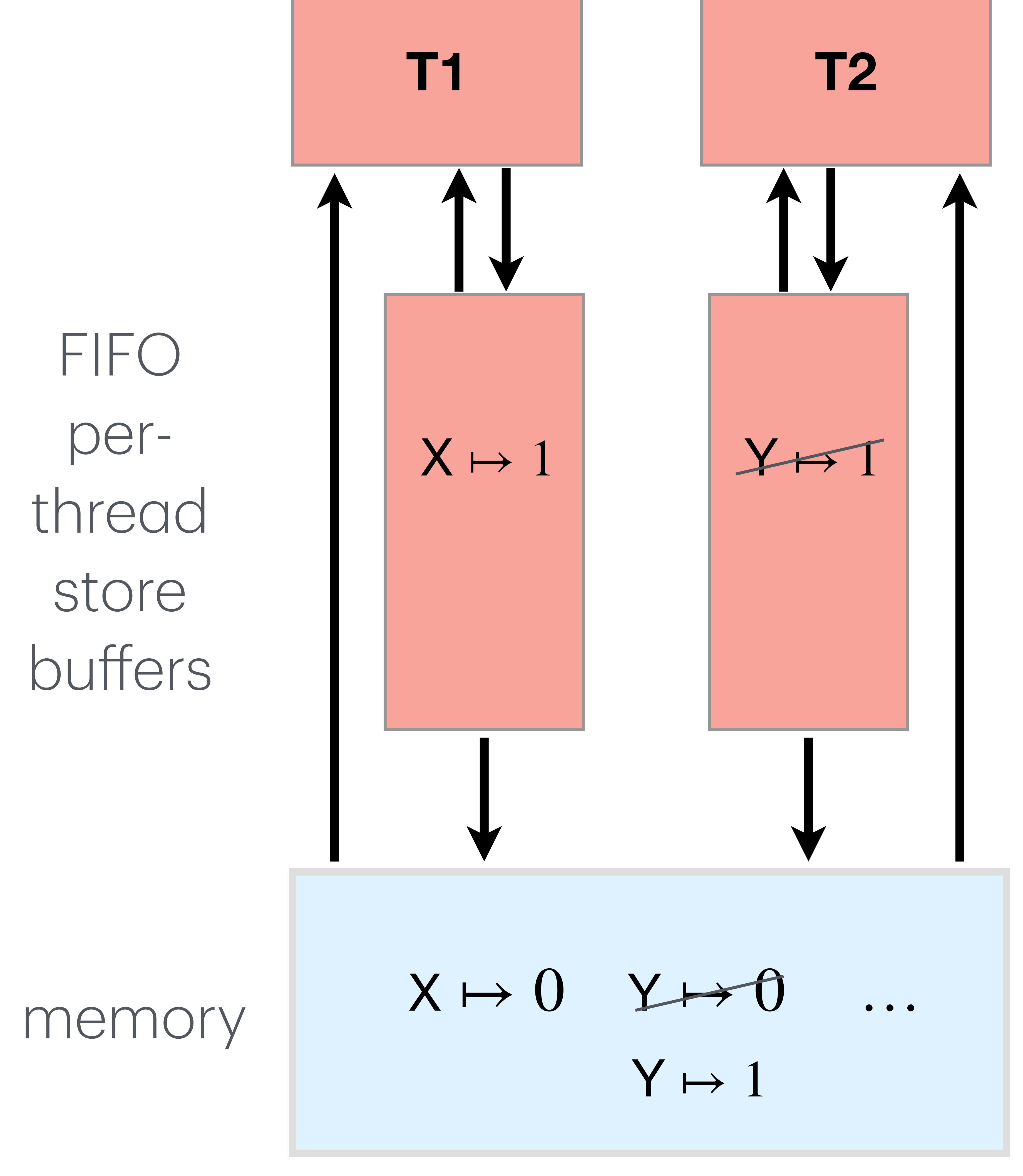
x86-TSO

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

```
X := 1            ‖   Y := 1
fence()           ‖   fence()
a := Y // 0       ‖   b := X // 0
```

FIFO per-thread store buffers

memory



Sewell, Sarkar, Owens, Zappa Nardelli, Myreen: **x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors.** Commun. ACM 53(7) 2010. https://doi.org/10.1145/1785414.1785443

WMM = out-of-order execution?

C1 / C2	Store	Load
Store	N	Y
Load	N	N

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

C1 / C2	Store	Load
Store	Y	Y
Load	Y	Y

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

possible reordering for *independent* accesses

L, Vafeiadis: **Explaining Relaxed Memory Models with Program Transformations.** FM 2016. https://doi.org/10.1007/978-3-319-48989-6_29

WMM ≠ out-of-order execution

`X := 1`	`a := X // 1`
	`lwsync`
	`b := Y // 0`
		`c := Y // 1`
		`lwsync`
		`d := X // 0`
			`Y := 1`

- Because of the `lwsync` fences, no intra-process reorderings are possible
- The threads may still observe the writes in different orders

Sarkar, Sewell, Alglave, Maranget, Williams: **Understanding POWER multiprocessors.** PLDI 2011. https://doi.org/10.1145/1993498.1993520

WMM = hardware models?

X := 1 Y := 1	a := X b := Y // 1 c := X // 0

compiler

X := 1 Y := 1	a := X b := Y // 1 c := a // 0

Read from untaken branch

```
a := X

Y := a
```

```
b := Y

if (b = 42) then
  c := 1
else
  c := 2
  b := 42

X := b
```

Can this program end with `c = 1` ?

Boehm, McKenney: **A Relaxed Guide to memory_order_relaxed.** 2020. https://open-std.org/JTC1/SC22/WG21/docs/papers/2020/p2055r0.pdf

Read from untaken branch

```
a := X

Y := a
```

||

```
b := Y

if (b = 42) then
  c := 1
else
  c := 2
  b := 42

X := b 42
```

(On the slide, the `b` in `X := b` is struck through in red and replaced by `42`.)

Can this program end with `c = 1` ?

Boehm, McKenney: **A Relaxed Guide to memory_order_relaxed.** 2020. https://open-std.org/JTC1/SC22/WG21/docs/papers/2020/p2055r0.pdf

Read from untaken branch

```
a := X

Y := a
```

```
b := Y

if (b = 42) then
  c := 1
else
  c := 2
  b := 42

X := b 42
```

(In the second thread, the `b` in `X := b` is crossed out and replaced with `42`; an arrow runs from `X := 42` up to `b := Y`.)

Can this program end with `c = 1` ?

Boehm, McKenney: **A Relaxed Guide to memory_order_relaxed.** 2020. https://open-std.org/JTC1/SC22/WG21/docs/papers/2020/p2055r0.pdf

Tricky combinations

- Repeated read elimination over a lock:

- Read hoisting (`t` is a fresh temporary):

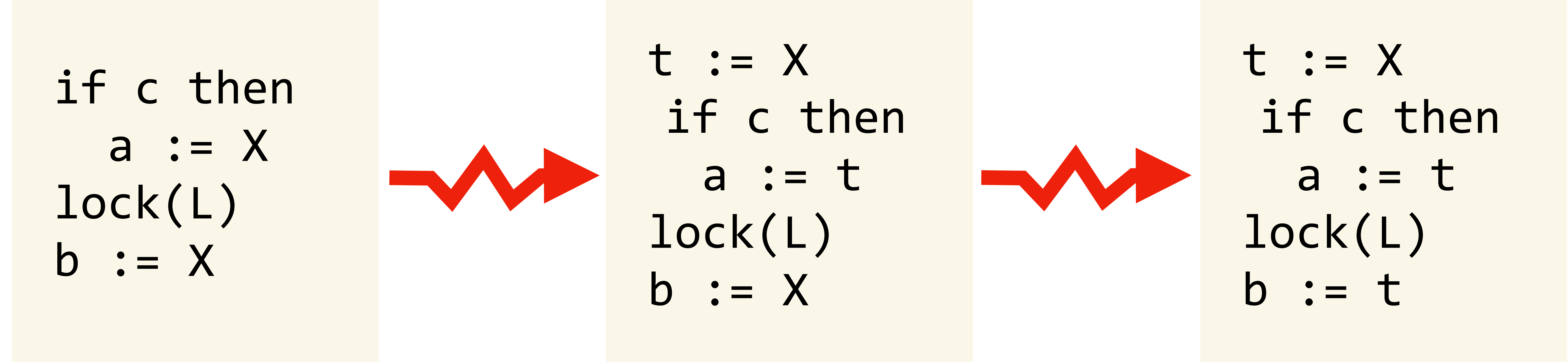
```
if c then
  a := X
```

⇝

```
t := X
if c then
  a := t
```

Allowing both is wrong!

- The combination of the two is unsafe:

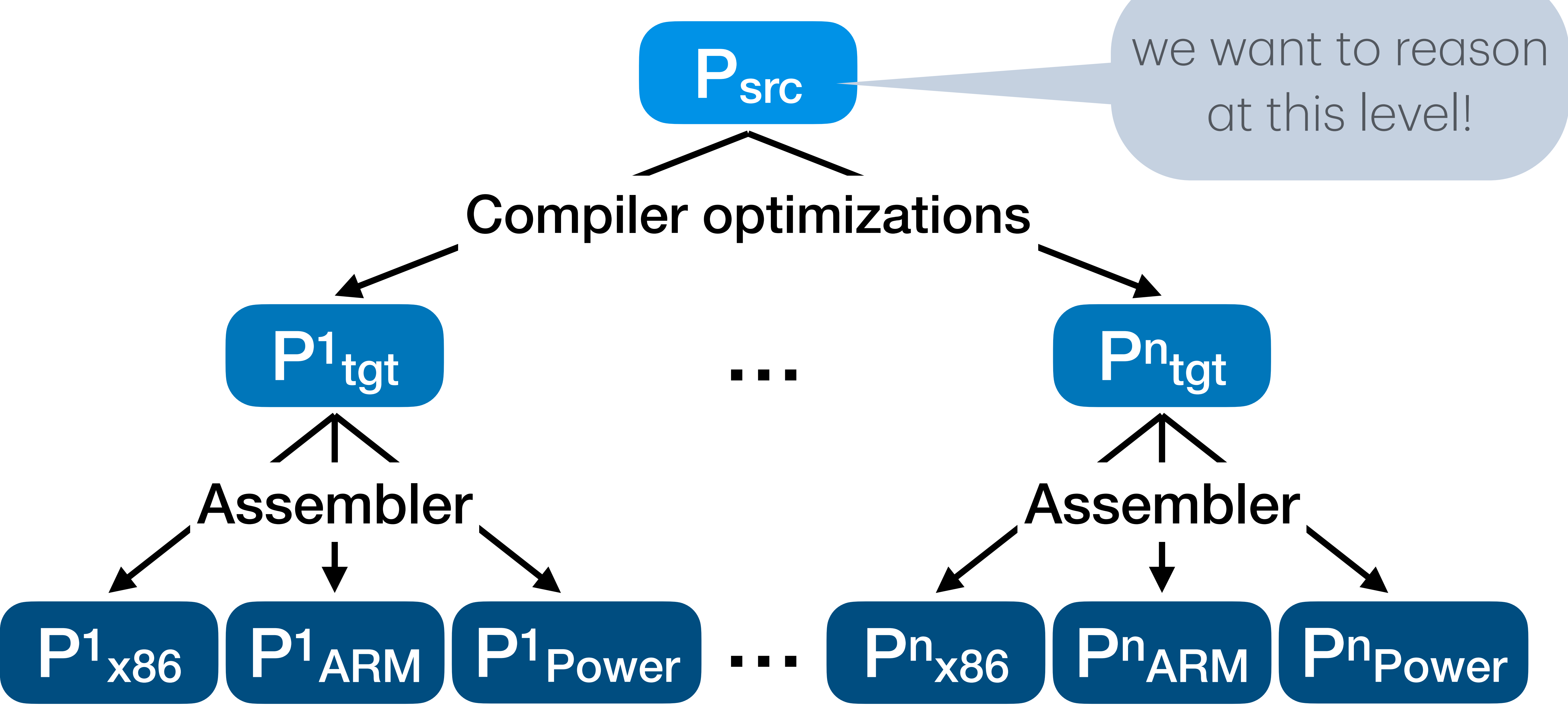


- When `c` is false , `X` is moved out of the critical region!

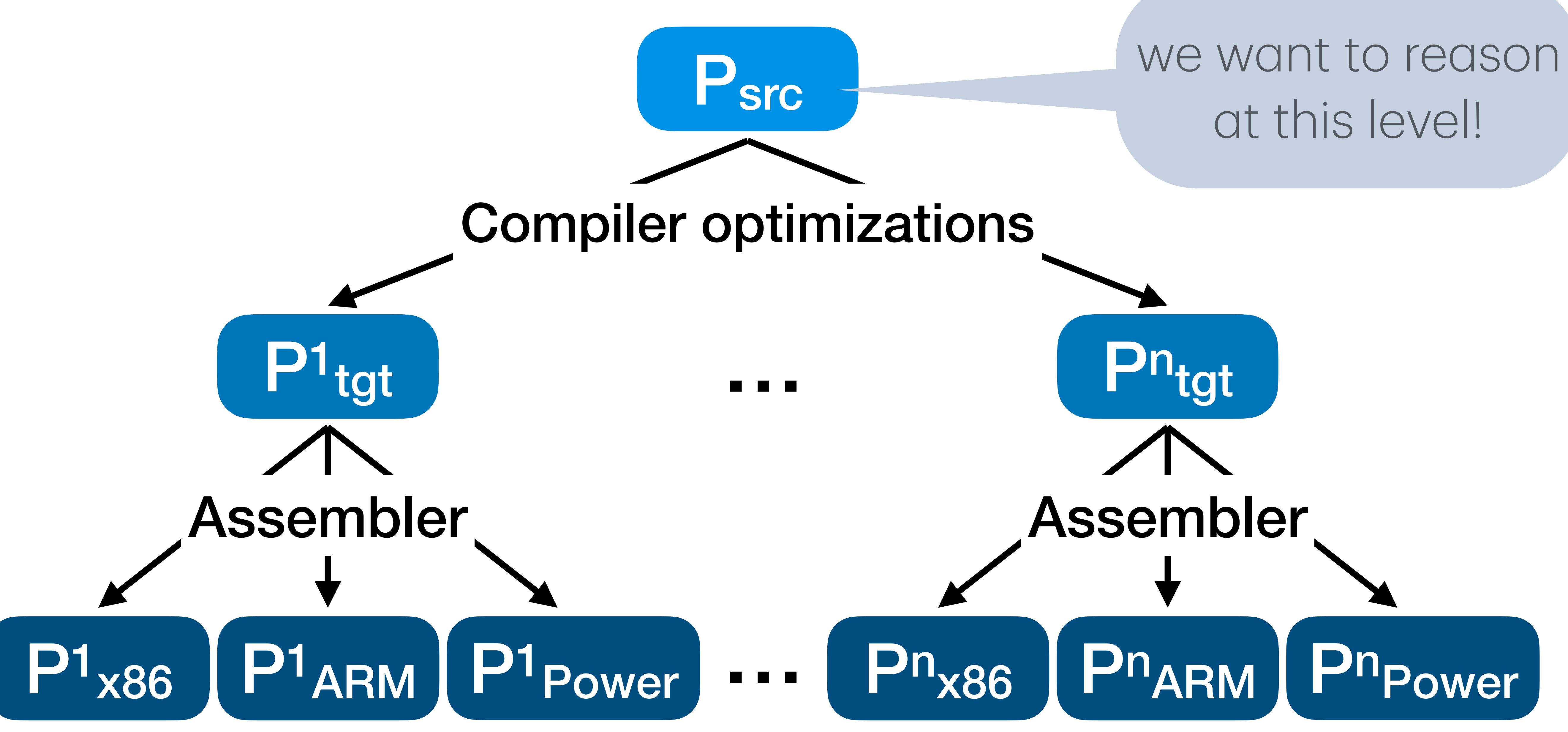
- We have to forbid one of the transformations:
  - C forbids load hoisting
  - LLVM forbids repeated read elimination over a lock

Chakraborty, Vafeiadis: **Validating optimizations of concurrent C/C++ programs.** CGO 2016. https://doi.org/10.1145/2854038.2854051

A WMM for a PL



A WMM for a PL



programmers

WMM

GCC

LLVM
COMPILER INFRASTRUCTURE

compilers

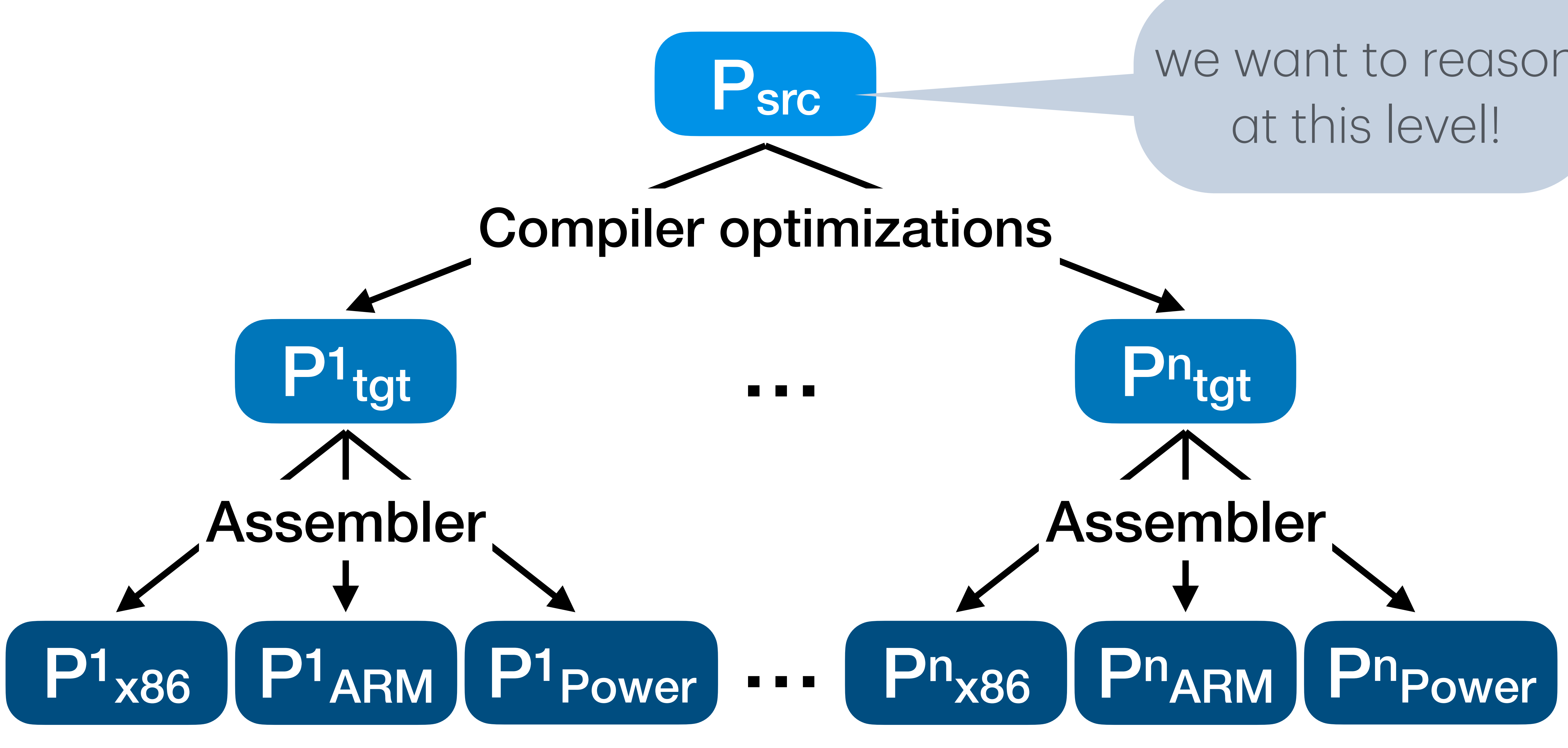
x86

Power

ARM

multicore architectures

A WMM for a PL



- C, C++
- Java
- OCaml
- JavaScript
- WebAssembly

- Linux kernel
- Rust
- LLVM
- ...

programmers

WMM

GCC

LLVM

COMPILER INFRASTRUCTURE

compilers

x86

Power

ARM

multicore architectures

Solving the memory model problem will require an ambitious and cross-disciplinary research direction.

BY SARITA V. ADVE AND HANS-J. BOEHM

Memory Models: A Case for Rethinking Parallel Languages and Hardware

Commun. ACM 53, 8 (August 2010).
https://doi.org/10.1145/1582716.1582718

The Problem of Programming Language Concurrency Semantics

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell

University of Cambridge

"Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still *does not have a credible proposal for the concurrency semantics* of any general-purpose high-level language that includes high performance shared-memory concurrency primitives. This is a *major open problem* for programming language semantics."

ESOP 2015. https://doi.org/10.1007/978-3-662-46669-8_12

Embracing weak consistency

- Not only a threat, but also an opportunity:
  - More scalable algorithms
  - Many (most?) concurrent idioms/algorithms do not need SC
  - Better understanding of our algorithms
  - Better understanding of concurrency
  - Local reasoning and more scalable verification
  - Open research problems!

The C11 memory model

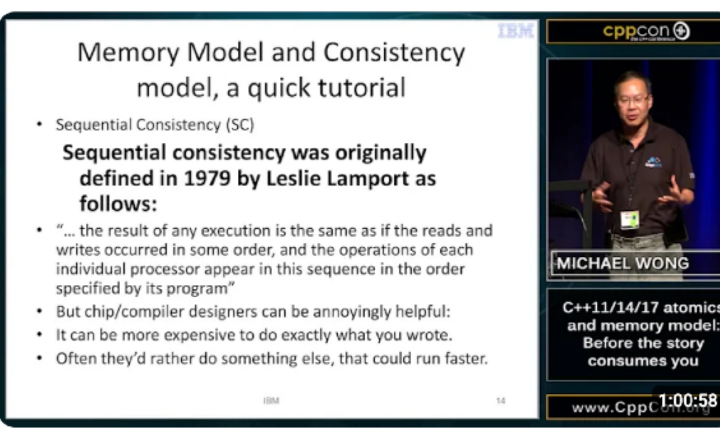
The C/C++11 memory model

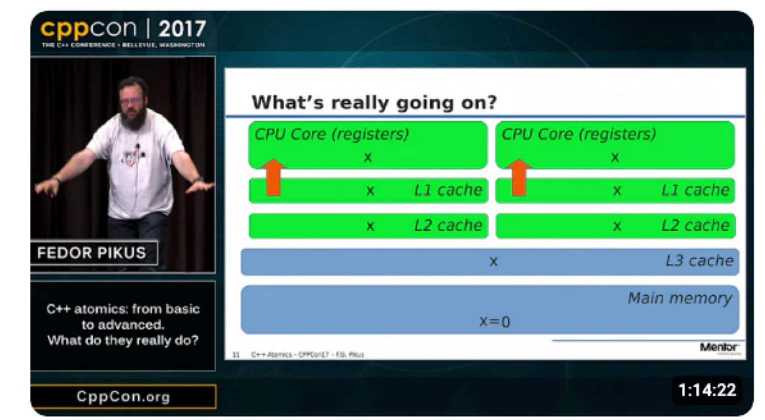
- In C/C++11 threads were made a part of the language specification
- A careful and sophisticated *declarative* weak memory model was published:
  - a result of several years of effort (starting around 2004)
  - building on the experience with the Java memory model
- Main design principles:
  - Tell *non-expert programmers* to avoid data races and provide strong semantics for them
    - Leave the semantics of data races completely undefined (**“catch-fire”**)
    - This way we can allow more flexible implementations and simpler model

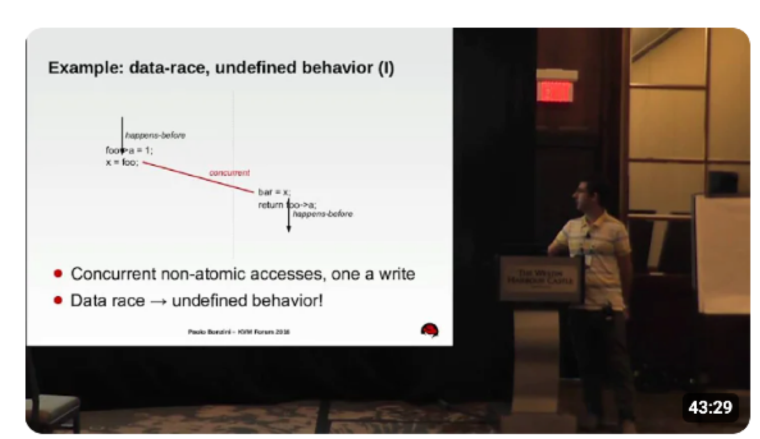
  - Give *experts* a way to write very carefully crafted, but portable, synchronization code that approaches the performance of assembly code

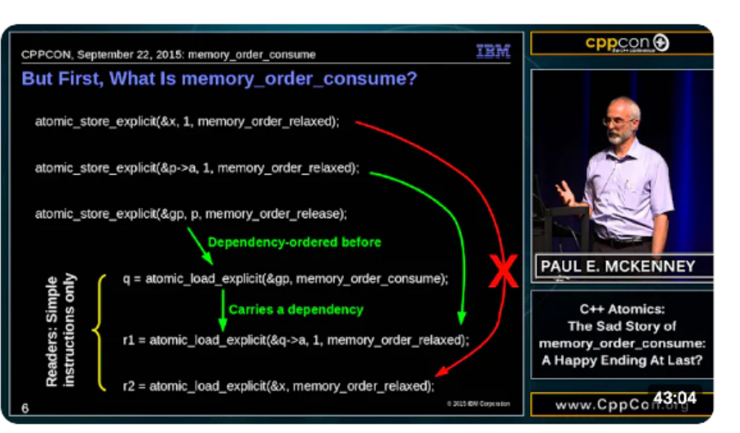
Some resources

- For language lawyers:
  - http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf
- A popularized history:
  - Boehm, Adve: **You Don't Know Jack about Shared Variables or Memory Models.** Commun. ACM 55.2 (Feb. 2012). https://doi.org/10.1145/2076450.2076465
- Formal treatment:
  - Batty, Owens, Sarkar, Sewell, Weber: **Mathematizing C++ Concurrency.** POPL 2011. http://doi.acm.org/10.1145/1926385.1926394
  - L, Vafeiadis, Kang, Hur, Dreyer: **Repairing Sequential Consistency in C/C++11.** PLDI 2017. https://doi.org/10.1145/3140587.3062352









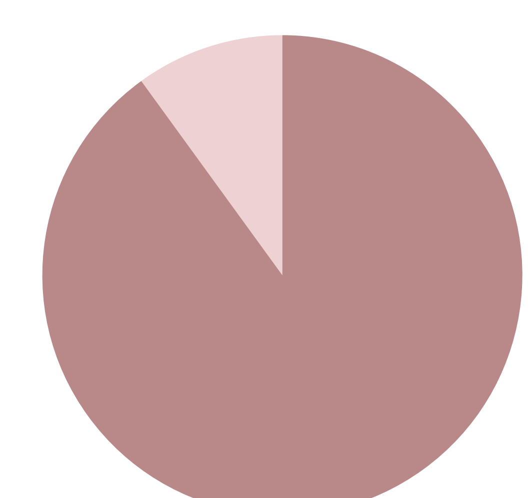
Why focus on C11?

- The world is programmed in C/C++.
- C11 is a prototype PL memory model: a solid starting point for other languages: LLVM, Java 9, WebAssembly, Rust, JavaScript…
- Architecture vendors aim to efficiently implement C11
- One of the most well-studied weak memory models: correctness, programmability guarantees, algorithms, verification,…

Main ingredients

non-expert mode

- Non-atomic memory accesses (reads/writes):
  - ordinary accesses for data manipulations
  - the majority of accesses in a typical program
  - insensitive to access granularity
- Locks
  - used to avoid data-races



---

expert mode

- Atomic memory accesses (reads/writes/RMWs)
  - used for synchronization
- Fences for fine-tuned synchronization patterns

Syntax examples (atomics)

- In C:
  - annotate the type, and then all accesses default to SC memory order:

```
Atomic(Node *) top;
```

  - Annotate an access:

```
t = atomic_load_explicit(top, memory_order_acquire);
```

- CAS in C++:

```
atomic_compare_exchange_weak_explicit(&head, &new_node->next, new_node,
            memory_order_release, memory_order_relaxed);
```

Examples with non-atomics and locks

```
lock(L)            ||  lock(L)
X := 1             ||  a := X // 2
unlock(L)          ||  unlock(L)
```

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

```
a := X // 1        ||  b := Y // 1
if a = 1 then      ||  if b = 1 then
  Y := 1           ||    X := 1
```

```
X := 1        ||  lock(L)
lock(L)       ||  a := Y // 1
Y := 1        ||  unlock(L)
unlock(L)     ||  if (a = 1) then
              ||    b := X // 0
```

Which of these programs are race-free? How are data races defined?

What are the guarantees for race-free programs?

The full model...

$$
\begin{aligned}
[\![-]\!] &: \mathsf{CExp} \rightarrow \mathbb{P}(\langle res : \mathsf{Val} \cup \{\bot\}, \mathcal{A} : \mathbb{P}(\mathsf{AName}), \mathsf{lab} : \mathcal{A} \rightarrow \mathsf{Act}, \mathsf{sb} : \mathbb{P}(\mathcal{A} \times \mathcal{A}), fst : \mathcal{A}, lst : \mathcal{A}\rangle) \\
[\![v]\!] &\stackrel{\text{def}}{=} \{\langle v, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{skip}\} \\
[\![\mathbf{alloc}()]\!] &\stackrel{\text{def}}{=} \{\langle \ell, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \ell \in \mathsf{Loc} \wedge \mathsf{lab}(a) = \mathrm{A}(\ell)\} \\
[\![[v]_Z := v']\!] &\stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{W}_Z(v, v')\} \\
[\![[v]_Z]\!] &\stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge \mathsf{lab}(a) = \mathrm{R}_Z(v, v')\} \\
[\![\mathbf{CAS}_{X,Y}(v, v_o, v_n)]\!] &\stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge v' \neq v_o \wedge \mathsf{lab}(a) = \mathrm{R}_Y(v, v')\} \\
&\cup \{\langle v_o, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{RMW}_X(v, v_o, v_n)\} \\
[\![\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2]\!] &\stackrel{\text{def}}{=} \{\langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \mid \langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!]\} \\
&\cup \{\langle res_2, \mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{lab}_1 \cup \mathsf{lab}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(lst_1, fst_2)\}, fst_1, lst_2\rangle \mid \\
&\quad \langle v_1, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!] \wedge \langle res_2, \mathcal{A}_2, \mathsf{lab}_2, \mathsf{sb}_2, fst_2, lst_2\rangle \in [\![E_2[v_1/x]]\!]\} \\
[\![\mathbf{repeat}\ E\ \mathbf{end}]\!] &\stackrel{\text{def}}{=} \{\langle res_N, \biguplus_{i\in[1..N]} \mathcal{A}_i, \bigcup_{i\in[1..N]} \mathsf{lab}_i, \bigcup_{i\in[1..N]} \mathsf{sb}_i \cup \{(lst_1, fst_2), \ldots, (lst_{N-1}, fst_N)\}, fst_1, lst_N\rangle \mid \\
&\quad \forall i.\ \langle res_i, \mathcal{A}_i, \mathsf{lab}_i, \mathsf{sb}_i, fst_i, lst_i\rangle \in [\![E]\!] \wedge (i \neq N \implies res_i = 0) \wedge res_N \neq 0\} \\
[\![E_1 \| E_2]\!] &\stackrel{\text{def}}{=} \{\langle \mathsf{combine}(res_1, res_2), \mathcal{A}_1 \uplus \mathcal{A}_2 \uplus \{a_{\text{fork}}, a_{\text{join}}\}, \mathsf{lab}_1 \cup \mathsf{lab}_2 \cup \{a_{\text{fork}} \mapsto \mathrm{skip}, a_{\text{join}} \mapsto \mathrm{skip}\}, \\
&\quad \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(a_{\text{fork}}, fst_1), (a_{\text{fork}}, fst_2), (lst_1, a_{\text{join}}), (lst_2, a_{\text{join}})\}, a_{\text{fork}}, a_{\text{join}}\rangle \mid \\
&\quad \langle res_1, \mathcal{A}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!] \wedge \langle res_2, \mathcal{A}_2, \mathsf{sb}_2, fst_2, lst_2\rangle \in [\![E_2]\!] \wedge a_{\text{fork}}, a_{\text{join}} \in \mathsf{AName}\}
\end{aligned}
$$

Figure 2. Semantics of closed program expressions.

$$
\begin{aligned}
&\nexists x.\ \mathsf{hb}(x, x) && \text{(IrreflexiveHB)} \\
&\forall \ell.\ \mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{iswrite}_\ell(a)\}, \mathsf{mo}) \wedge \mathsf{hb}_\ell \subseteq \mathsf{mo} && \text{(ConsistentMO)} \\
&\mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{isSeqCst}(a)\}, \mathsf{sc}) \wedge \mathsf{hb}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} \wedge \mathsf{mo}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} && \text{(ConsistentSC)} \\
&\forall b.\ \mathsf{rf}(b) \neq \bot \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a, b) && \text{(ConsistentRFdom)} \\
&\forall a, b.\ \mathsf{rf}(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \wedge \neg\mathsf{hb}(b, a) && \text{(ConsistentRF)} \\
&\forall a, b.\ \mathsf{rf}(b) = a \wedge (\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}) \implies \mathsf{hb}(a, b) && \text{(ConsistentRFna)} \\
&\forall a, b.\ \mathsf{rf}(b) = a \wedge \mathsf{isSeqCst}(b) \implies \mathsf{isc}(a, b) \vee \neg\mathsf{isSeqCst}(a) \wedge (\forall x.\ \mathsf{isc}(x, b) \Rightarrow \neg\mathsf{hb}(a, x)) && \text{(RestrSCReads)} \\
&\nexists a, b.\ \mathsf{hb}(a, b) \wedge \mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) && \text{(CoherentRR)} \\
&\nexists a, b.\ \mathsf{hb}(a, b) \wedge \mathsf{mo}(\mathsf{rf}(b), a) \wedge \mathsf{iswrite}(a) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) && \text{(CoherentWR)} \\
&\nexists a, b.\ \mathsf{hb}(a, b) \wedge \mathsf{mo}(b, \mathsf{rf}(a)) \wedge \mathsf{iswrite}(b) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) && \text{(CoherentRW)} \\
&\forall a.\ \mathsf{isrmw}(a) \wedge \mathsf{rf}(a) \neq \bot \implies \mathsf{mo}(\mathsf{rf}(a), a) \wedge \nexists c.\ \mathsf{mo}(\mathsf{rf}(a), c) \wedge \mathsf{mo}(c, a) && \text{(AtomicRMW)} \\
&\forall a, b, \ell.\ \mathsf{lab}(a) = \mathsf{lab}(b) = \mathrm{A}(\ell) \implies a = b && \text{(ConsistentAlloc)}
\end{aligned}
$$

$$
\begin{aligned}
\text{where } \mathsf{iswrite}_{\ell,v}(a) &\stackrel{\text{def}}{=} \exists X, v_{\text{old}}.\ \mathsf{lab}(a) \in \{\mathrm{W}_X(\ell, v), \mathrm{RMW}_X(\ell, v_{\text{old}}, v)\} \qquad \mathsf{iswrite}_\ell(a) \stackrel{\text{def}}{=} \exists v.\ \mathsf{iswrite}_{\ell,v}(a) \\
\mathsf{isread}_{\ell,v}(a) &\stackrel{\text{def}}{=} \exists X, v_{\text{new}}.\ \mathsf{lab}(a) \in \{\mathrm{R}_X(\ell, v), \mathrm{RMW}_X(\ell, v, v_{\text{new}})\} \qquad \text{etc.} \\
\mathsf{rsElem}(a, b) &\stackrel{\text{def}}{=} \mathsf{sameThread}(a, b) \vee \mathsf{isrmw}(b) \\
\mathsf{rseq}(a) &\stackrel{\text{def}}{=} \{a\} \cup \{b \mid \mathsf{rsElem}(a, b) \wedge \mathsf{mo}(a, b) \wedge (\forall c.\ \mathsf{mo}(a, c) \wedge \mathsf{mo}(c, b) \Rightarrow \mathsf{rsElem}(a, c))\} \\
\mathsf{sw} &\stackrel{\text{def}}{=} \{(a, b) \mid \mathsf{mode}(a) \in \{\mathrm{rel}, \mathrm{rel_acq}, \mathrm{sc}\} \wedge \mathsf{mode}(b) \in \{\mathrm{acq}, \mathrm{rel_acq}, \mathrm{sc}\} \wedge \mathsf{rf}(b) \in \mathsf{rseq}(a)\} \\
\mathsf{hb} &\stackrel{\text{def}}{=} (\mathsf{sb} \cup \mathsf{sw})^+ \\
\mathsf{hb}_\ell &\stackrel{\text{def}}{=} \{(a, b) \in \mathsf{hb} \mid \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)\} \\
X_{\mathsf{SeqCst}} &\stackrel{\text{def}}{=} \{(a, b) \in X \mid \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b)\} \\
\mathsf{isc}(a, b) &\stackrel{\text{def}}{=} \mathsf{iswrite}_{\mathsf{locs}(b)}(a) \wedge \mathsf{sc}(a, b) \wedge \nexists c.\ \mathsf{sc}(a, c) \wedge \mathsf{sc}(c, b) \wedge \mathsf{iswrite}_{\mathsf{locs}(b)}(c)
\end{aligned}
$$

Figure 3. Axioms satisfied by consistent C11 executions, $\mathsf{Consistent}(\mathcal{A}, \mathsf{lab}, \mathsf{sb}, \mathsf{rf}, \mathsf{mo}, \mathsf{sc})$.

$c : \mathrm{W}(\ell,1) \xrightarrow{\mathsf{rf}} a : \mathrm{R}(\ell,1)$; $d \xrightarrow{\mathsf{mo}} c$; $a \xrightarrow{\mathsf{hb}} b$; $d : \mathrm{W}(\ell,2) \xrightarrow{\mathsf{rf}} b : \mathrm{R}(\ell,2)$	$c : \mathrm{W}(\ell,2) \xrightarrow{\mathsf{mo}} a : \mathrm{W}(\ell,1)$; $c \xrightarrow{\mathsf{rf}} b$; $a \xrightarrow{\mathsf{hb}} b : \mathrm{R}(\ell,2)$	$c : \mathrm{W}(\ell,1) \xrightarrow{\mathsf{rf}} a : \mathrm{R}(\ell,1)$; $b \xrightarrow{\mathsf{mo}} c$; $a \xrightarrow{\mathsf{hb}} b : \mathrm{W}(\ell,2)$	$a \xrightarrow{\mathsf{rf}} b$ means $a = \mathsf{rf}(b)$; $a \xrightarrow{\mathsf{mo}} b$ means $\mathsf{mo}(a,b)$; $a \xrightarrow{\mathsf{hb}} b$ means $\mathsf{hb}(a,b)$
violates CoherentRR	violates CoherentWR	violates CoherentRW	

Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).

$$
\begin{aligned}
\mathtt{rb} &\triangleq \mathtt{rf}^{-1}; \mathtt{mo} && \textit{(reads-before)} \\
\mathtt{eco} &\triangleq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^+ && \textit{(extended coherence order)} \\
\mathtt{rs} &\triangleq [\mathtt{W}]; \mathtt{sb}|^?_{\mathtt{loc}}; [\mathtt{W}^{\sqsupseteq\mathtt{rlx}}]; (\mathtt{rf}; \mathtt{rmw})^* && \textit{(release sequence)} \\
\mathtt{sw} &\triangleq [\mathtt{E}^{\sqsupseteq\mathtt{rel}}]; ([\mathtt{F}]; \mathtt{sb})^?; \mathtt{rs}; \mathtt{rf}; \\
&\quad [\mathtt{R}^{\sqsupseteq\mathtt{rlx}}]; (\mathtt{sb}; [\mathtt{F}])^?; [\mathtt{E}^{\sqsupseteq\mathtt{acq}}] && \textit{(synchronizes with)} \\
\mathtt{hb} &\triangleq (\mathtt{sb} \cup \mathtt{sw})^+ && \textit{(happens-before)}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{sb}|_{\neq\mathtt{loc}} &\triangleq \mathtt{sb} \setminus \mathtt{sb}|_{\mathtt{loc}} \\
\mathtt{scb} &\triangleq \mathtt{sb} \cup \mathtt{sb}|_{\neq\mathtt{loc}}; \mathtt{hb}; \mathtt{sb}|_{\neq\mathtt{loc}} \cup \mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb} \\
\mathtt{psc}_{\text{base}} &\triangleq \left([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{hb}^?\right); \mathtt{scb}; \left([\mathtt{E}^{\mathtt{sc}}] \cup \mathtt{hb}^?; [\mathtt{F}^{\mathtt{sc}}]\right) \\
\mathtt{psc}_{\mathtt{F}} &\triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathtt{sc}}] \\
\mathtt{psc} &\triangleq \mathtt{psc}_{\text{base}} \cup \mathtt{psc}_{\mathtt{F}}
\end{aligned}
$$

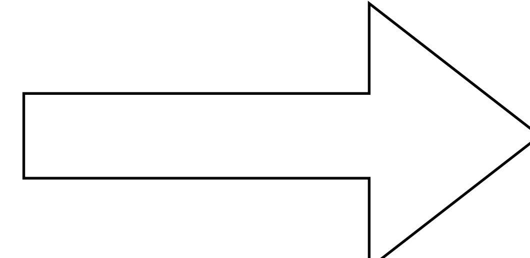
Definition 1. An execution G is called RC11-*consistent* if it is complete and the following hold:

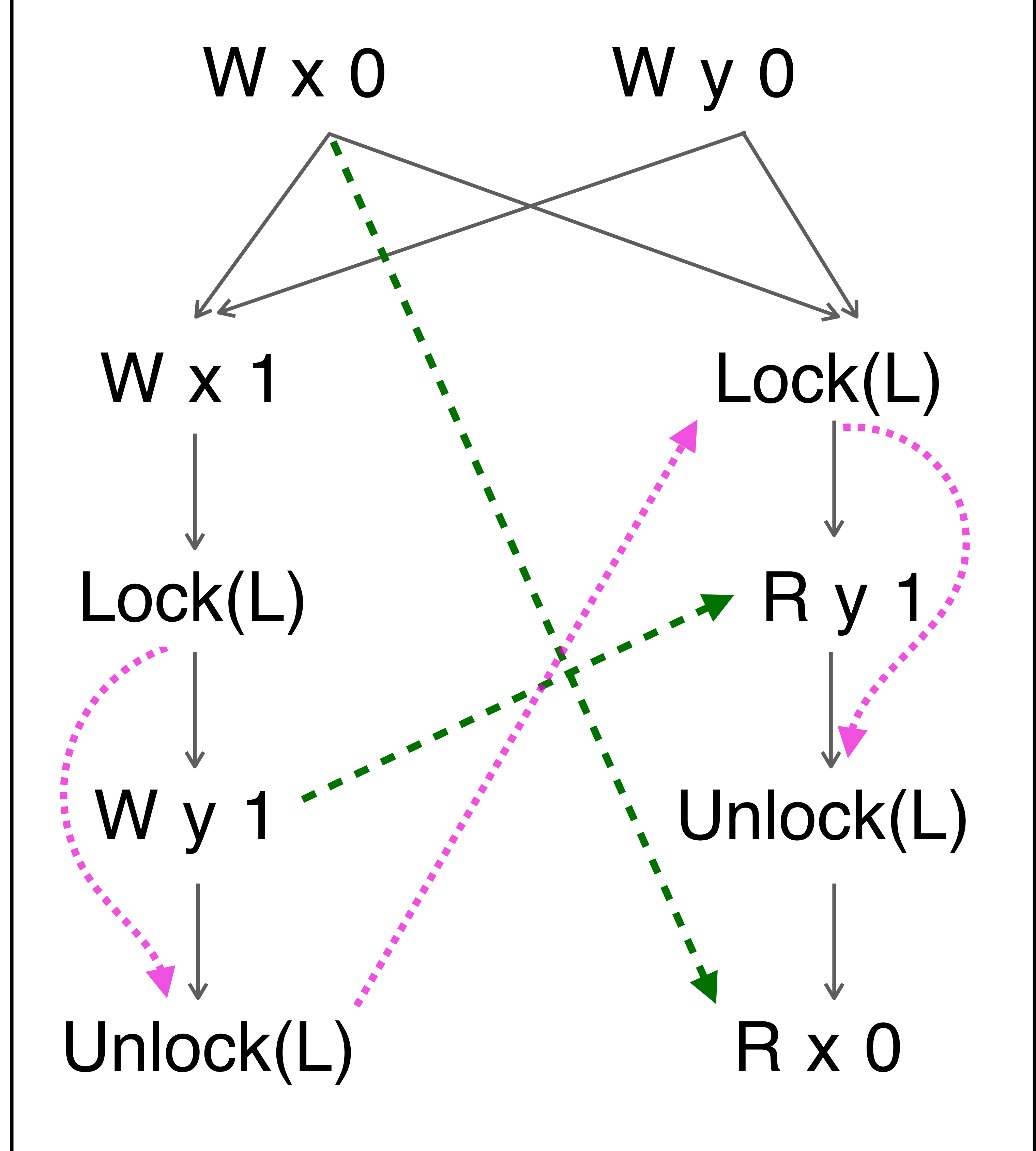
- $\mathtt{hb}; \mathtt{eco}^?$ is irreflexive. (COHERENCE)
- $\mathtt{rmw} \cap (\mathtt{rb}; \mathtt{mo}) = \emptyset$. (ATOMICITY)
- $\mathtt{psc}$ is acyclic. (SC)
- $\mathtt{sb} \cup \mathtt{rf}$ is acyclic. (NO-THIN-AIR)

Declarative memory models

- Possible program *behaviors* are represented by directed graphs

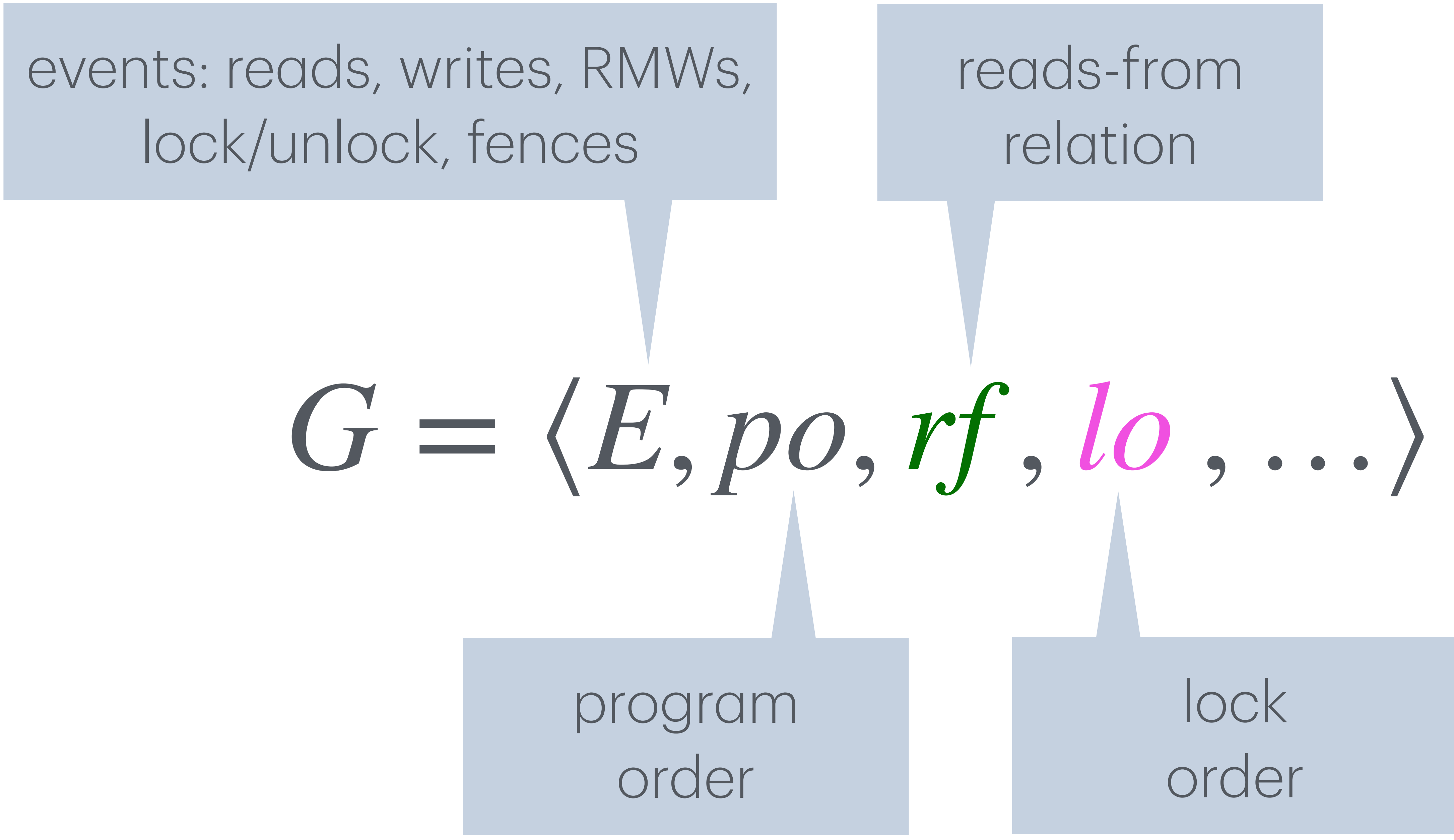
```
X := 1          || lock(L)
lock(L)         || a := Y // 1
Y := 1          || unlock(L)
unlock(L)       || if (a = 1) then
                ||   b := X // 0
```





- The model defines:
  - consistent execution graphs
  - racy execution graphs

Execution graphs



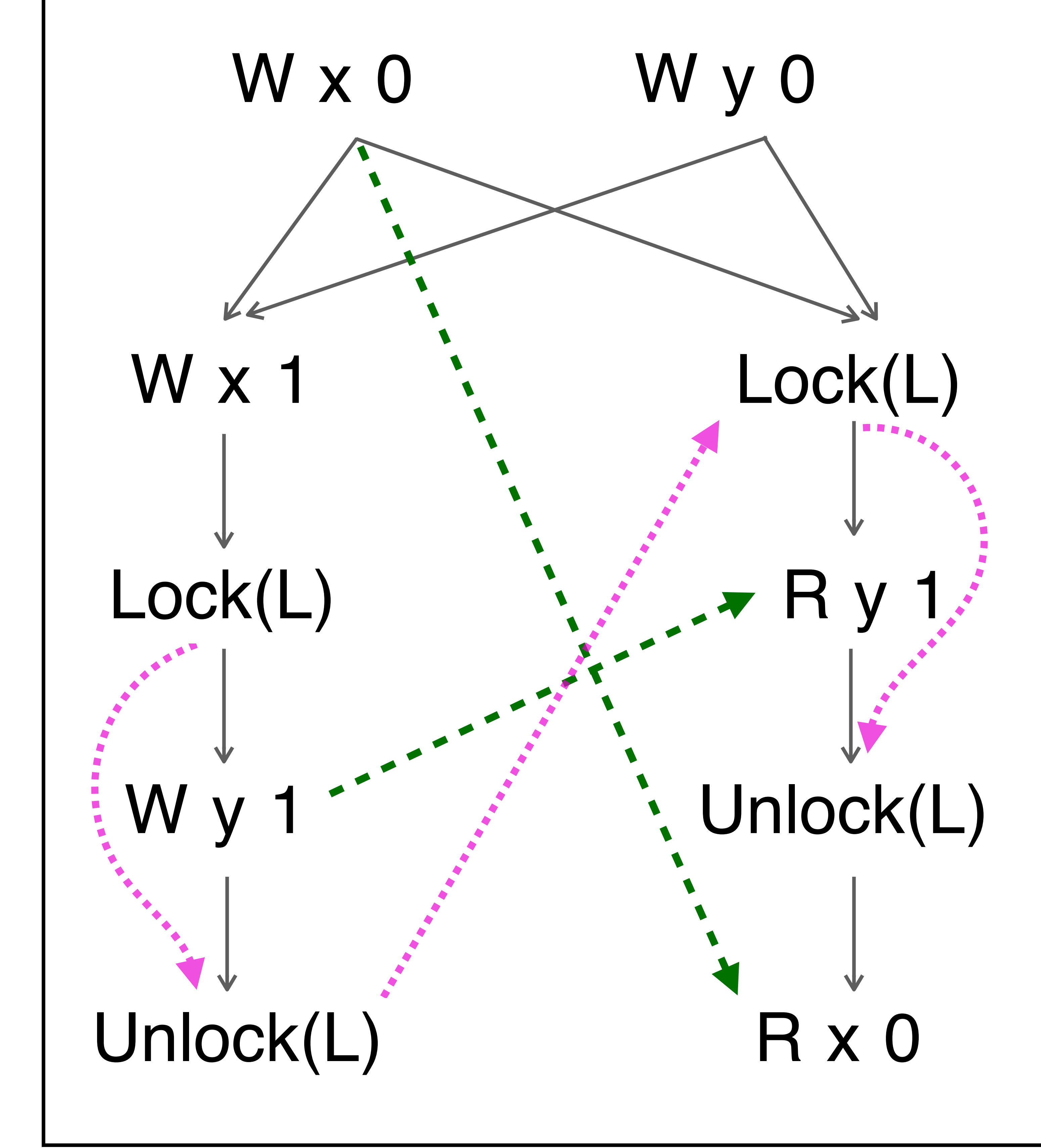
- There is a standard translation:

program ↦ set of candidate execution graphs

  - Read values are not constrained at this stage
  - Except for po, relations are existentially quantified

Well-formedness

- Program order *po*: partial order, per-thread total, initialization before everything
- reads-from *rf*: from a writing event to a reading event, value and location should match, every read reads from some write, an RMW cannot read from itself
- lock order *lo*: among lock and unlock events of the same lock, partial order, per-lock total, properly interleaved



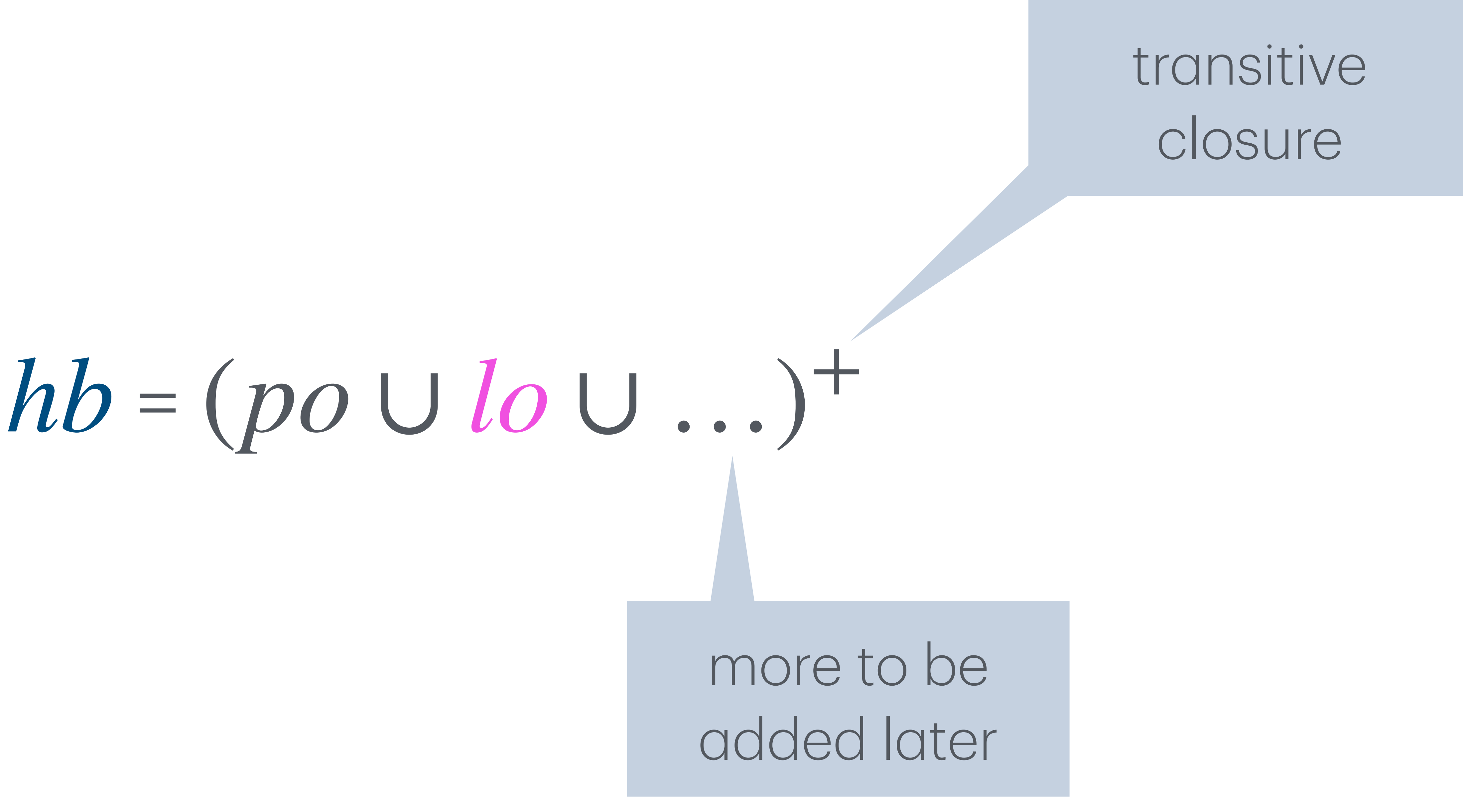
program-order *po*

reads-from *rf*

lock order *lo*

Happens-before

- The most central derived relation:

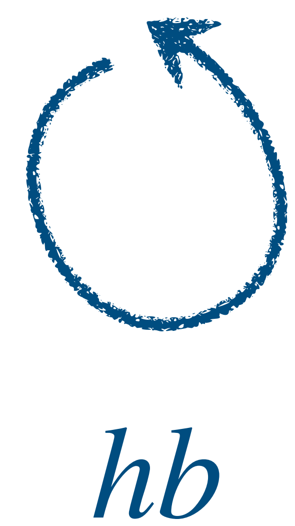


- Intuitively represents “knowledge”, “synchronization”, “causality"

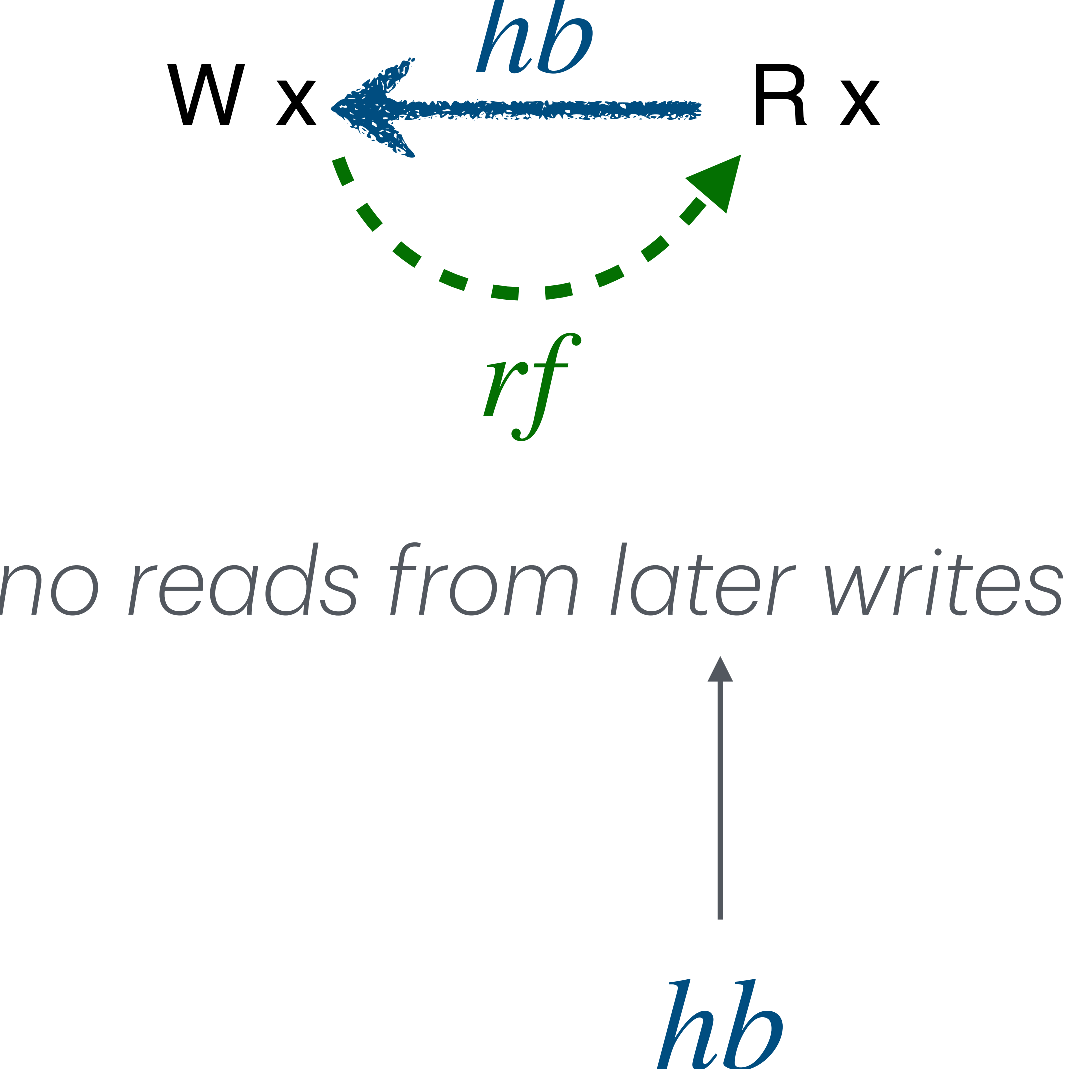
Execution-graph consistency

assuming only non-atomics & locks

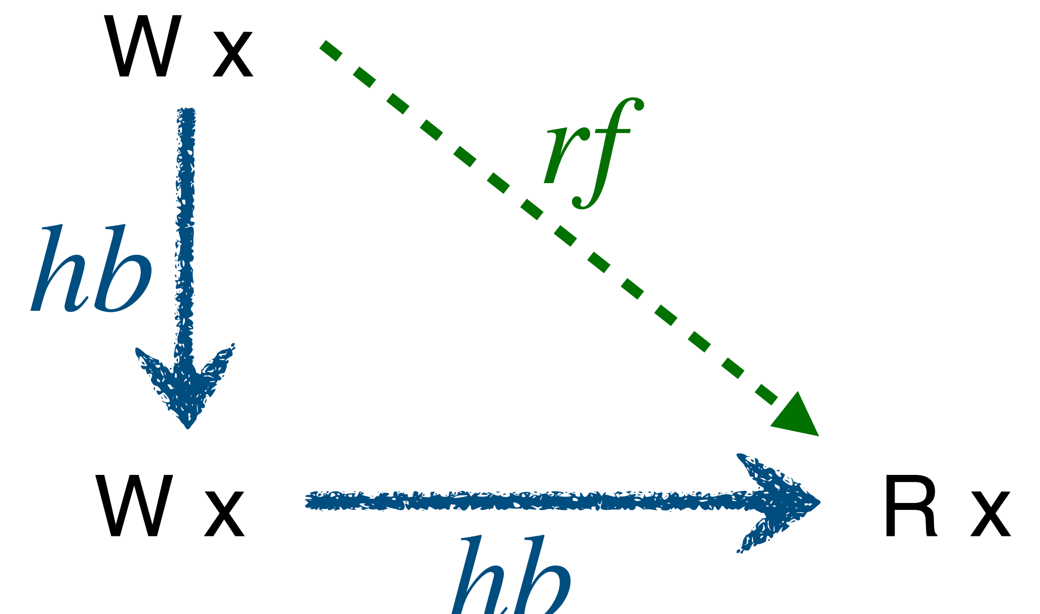
The following patterns should never occur:



hb *should be irreflexive*



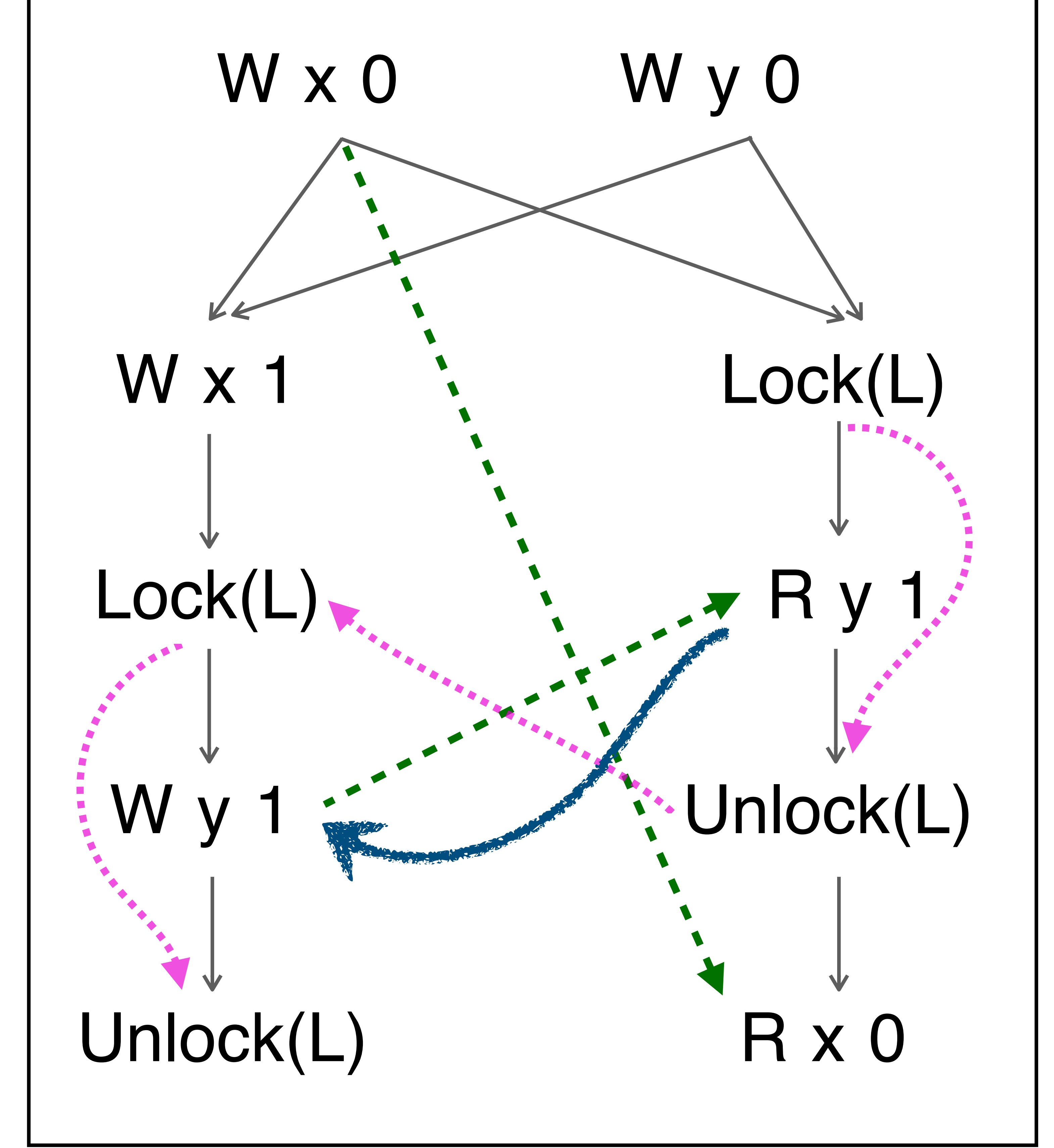
no reads from later writes



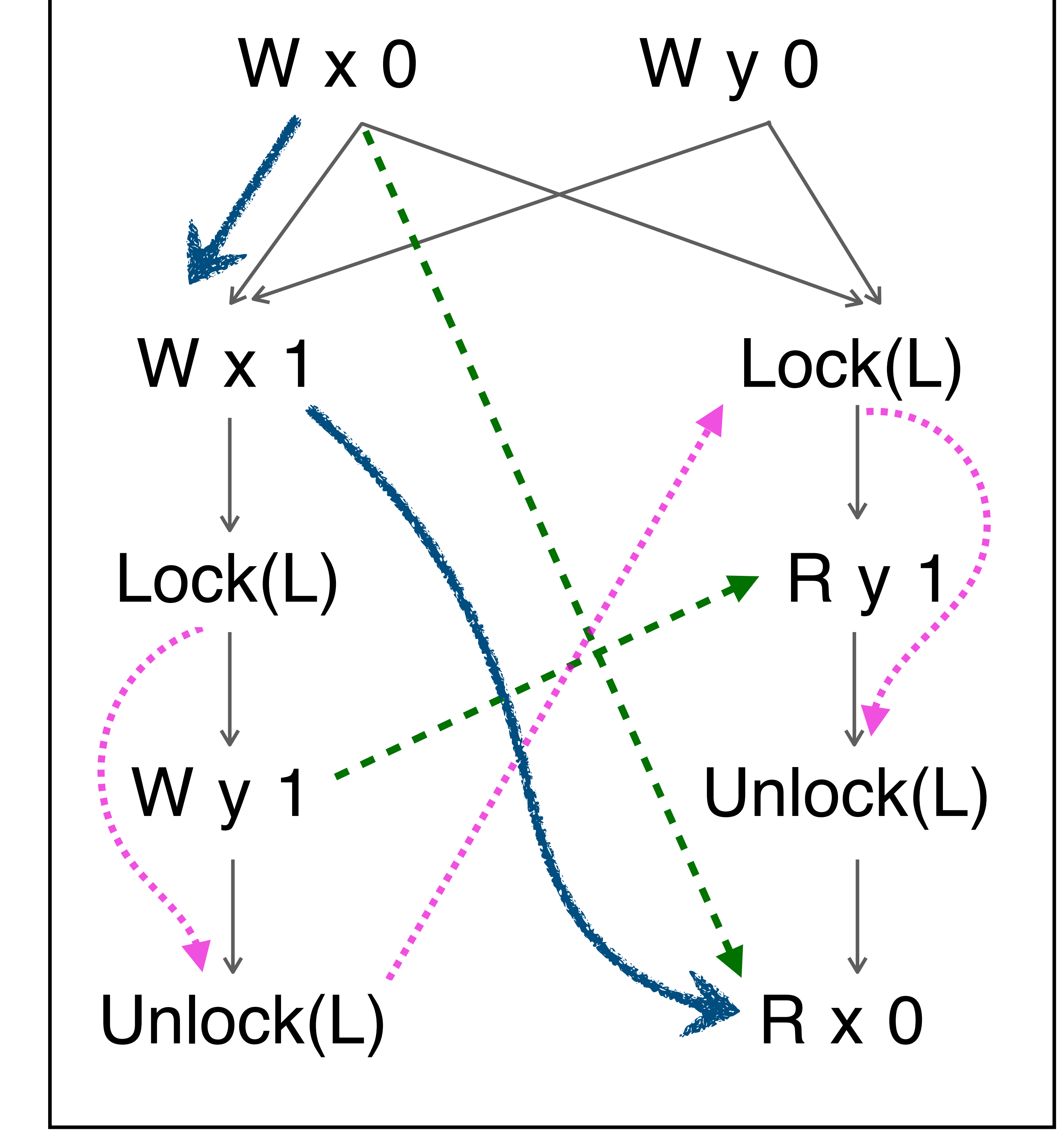
a thread may not read from a write
if it is aware of a later write to the same variable

hb

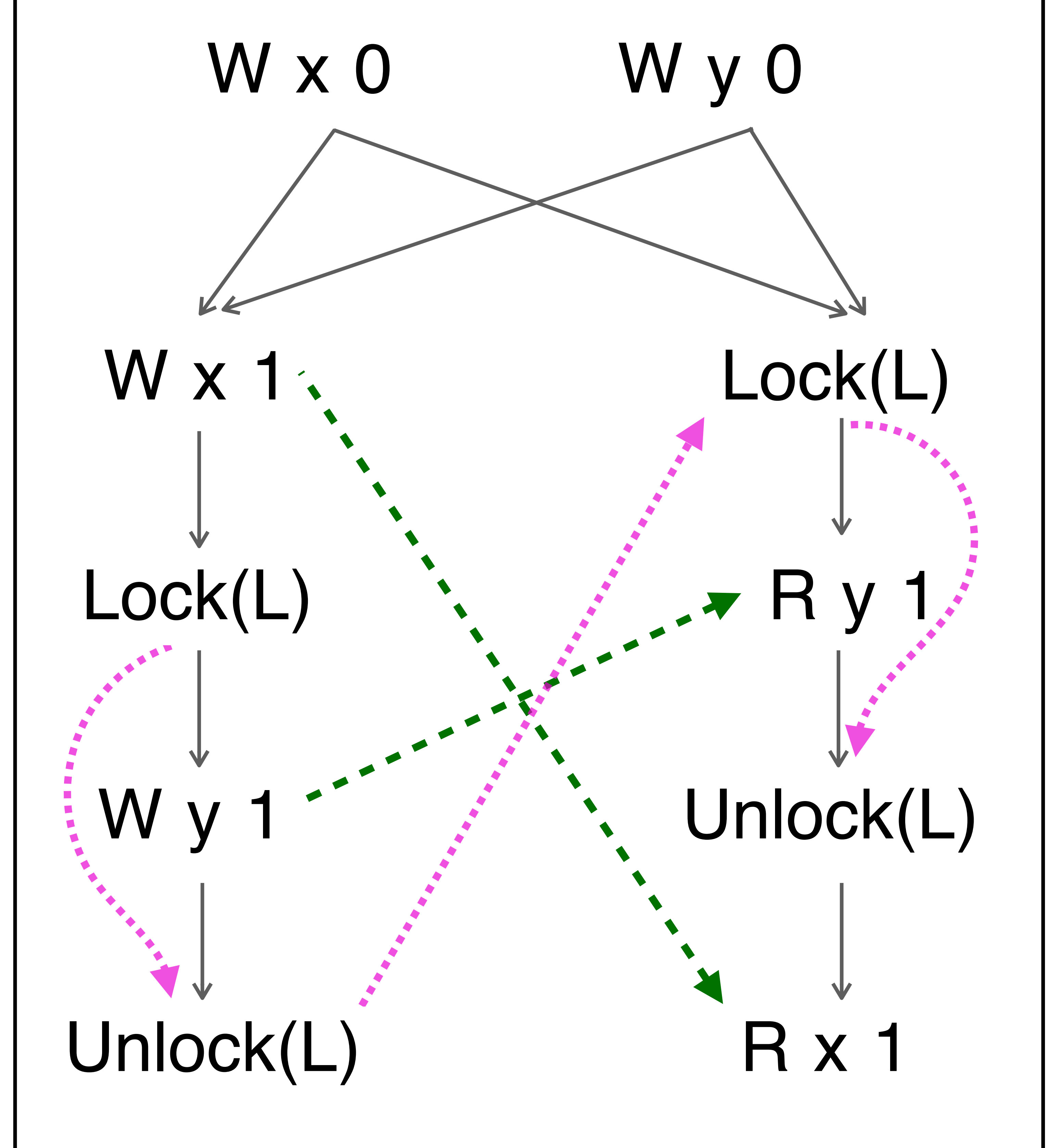
```
X := 1          || lock(L)
lock(L)         || a := Y
Y := 1          || unlock(L)
unlock(L)       || if (a = 1) then
                ||   b := X
```



inconsistent

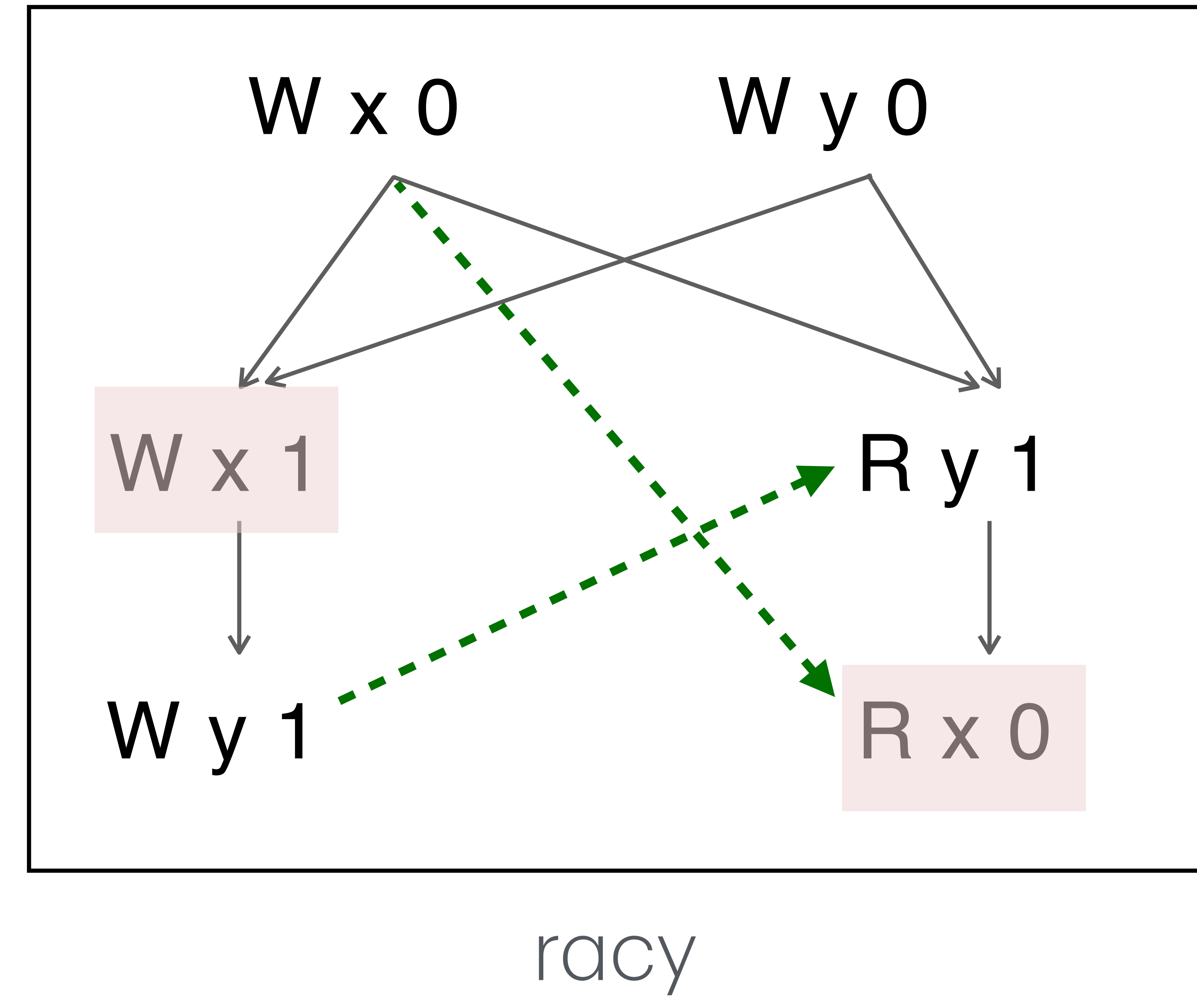


inconsistent

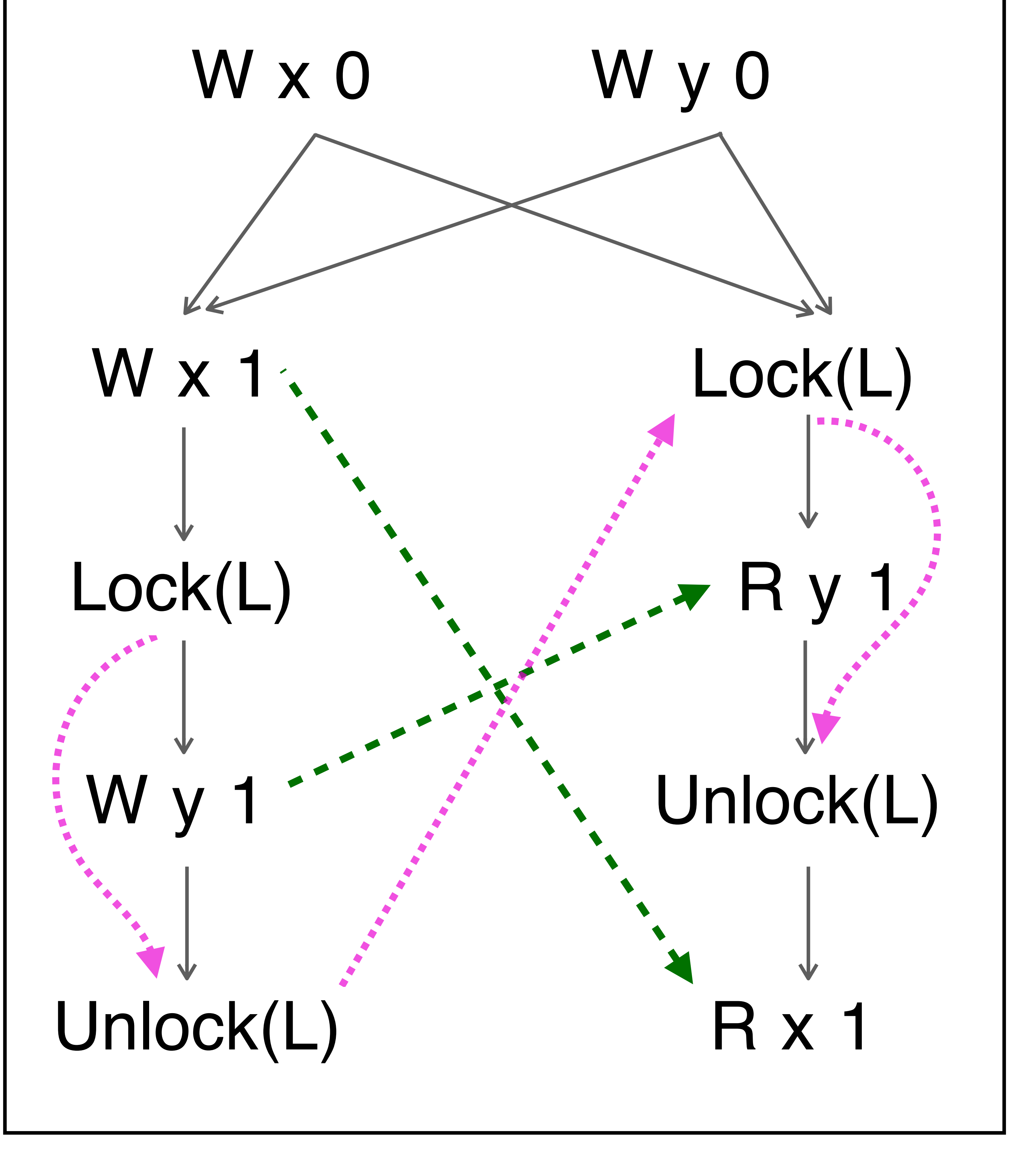


consistent

Data-races



racy



not racy

- Two events are *conflicting*:
  - *access* the same location
  - at least one is a write
- A *data-race* = two conflicting events:
  - at least one is non-atomic
  - unordered by ***hb***

Allowed behaviors

A behavior of a program is *allowed* if *one* of the following holds:

- It is obtained by some consistent execution graph of the program
- Some consistent execution graph of the program has a data race

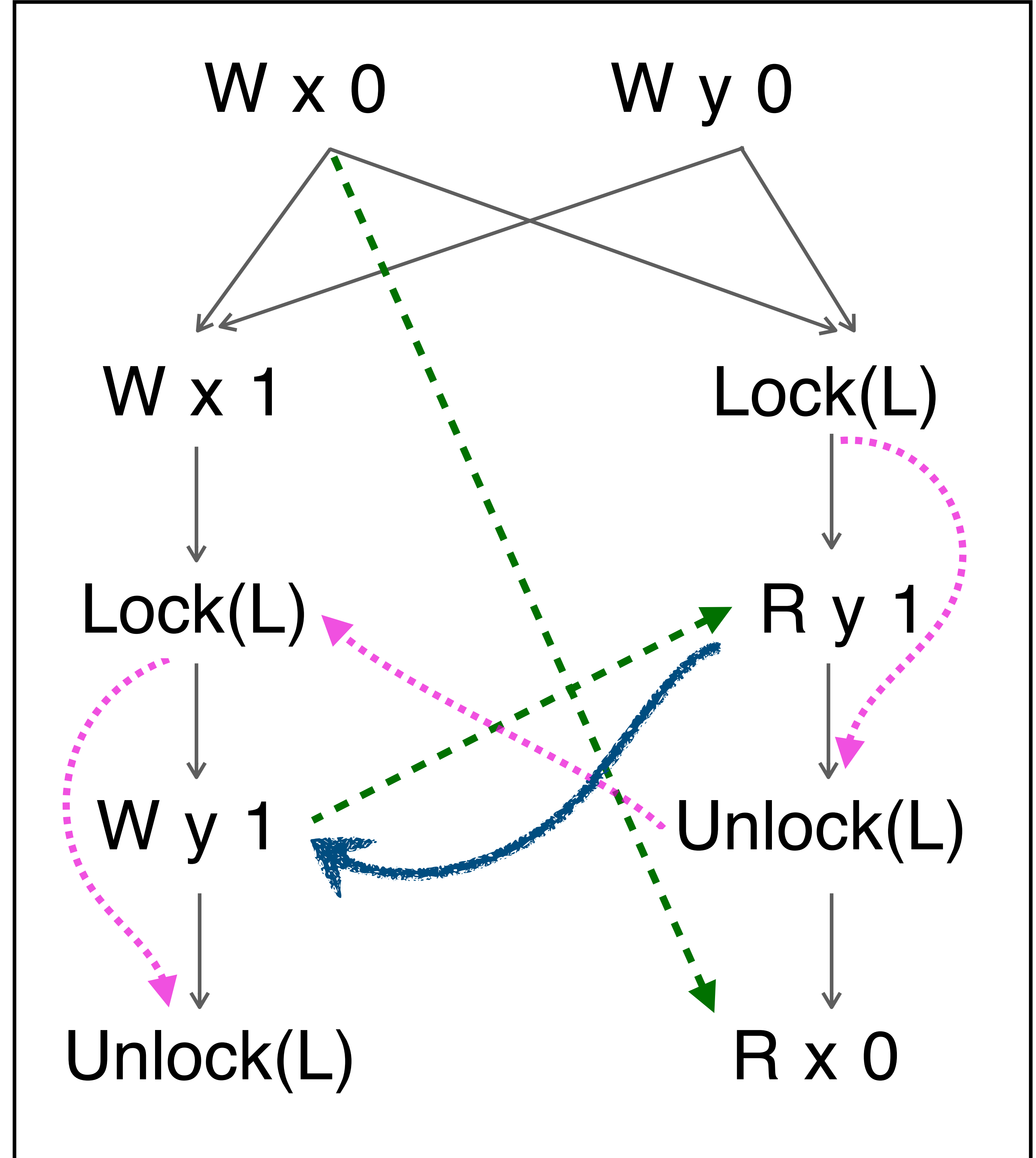
- Side note: *What is a behavior?*
  - Often taken to be the final values of the local variables
  - But, there are other options…

Example

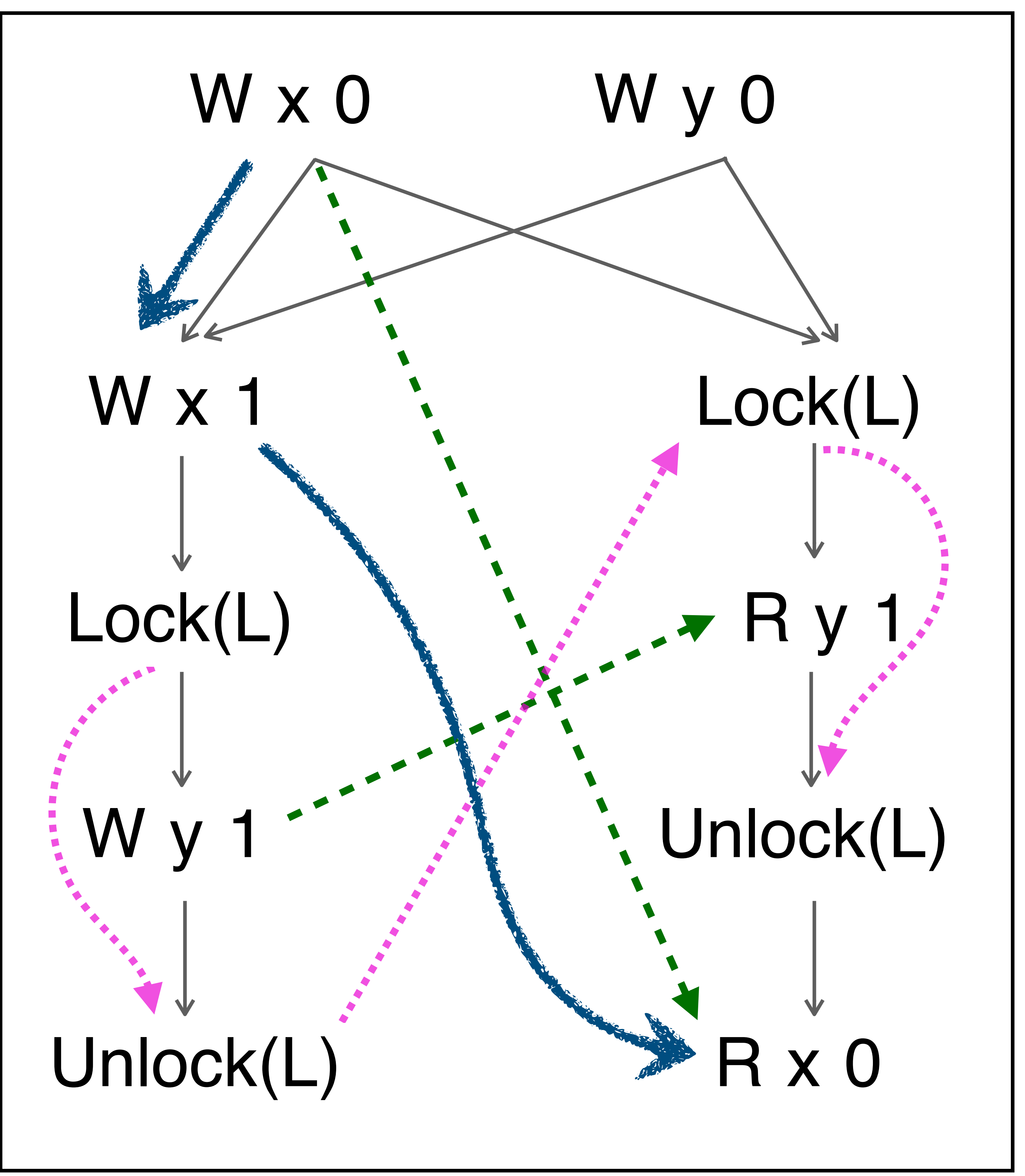
```
X := 1
lock(L)
Y := 1
unlock(L)
```

```
lock(L)
a := Y // 1
unlock(L)
if (a=1) then
  b := X // 0
```

behavior `a=1 & b=0` disallowed



inconsistent



inconsistent

+ no consistent graph of this program is racy

Atomic accesses

relaxed ⊏ release/acquire ⊏ sc

- All atomics guarantee coherence
  - accesses to each location are in a total order (that extends hb) where each read reads from the last write
- Release/Acquire enforce synchronization
  - via another case in the definition of hb
- SC accesses ensure a global total order among them

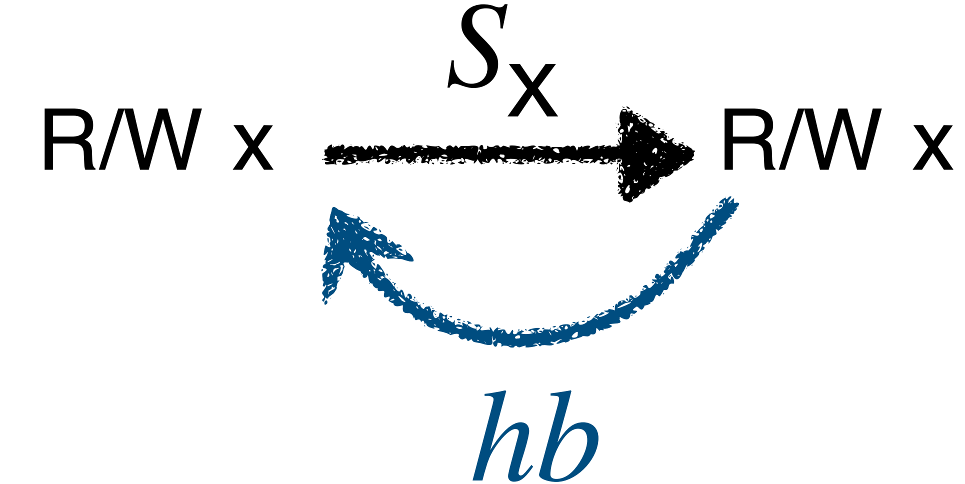
Coherence guarantees for atomics

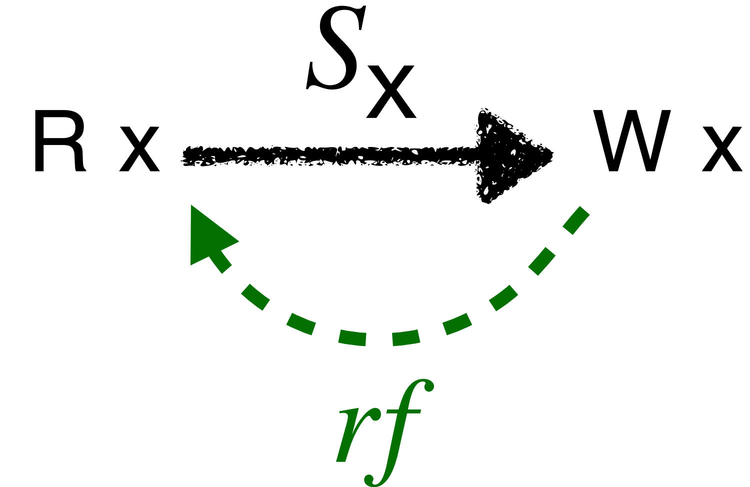
aka SC-per-location

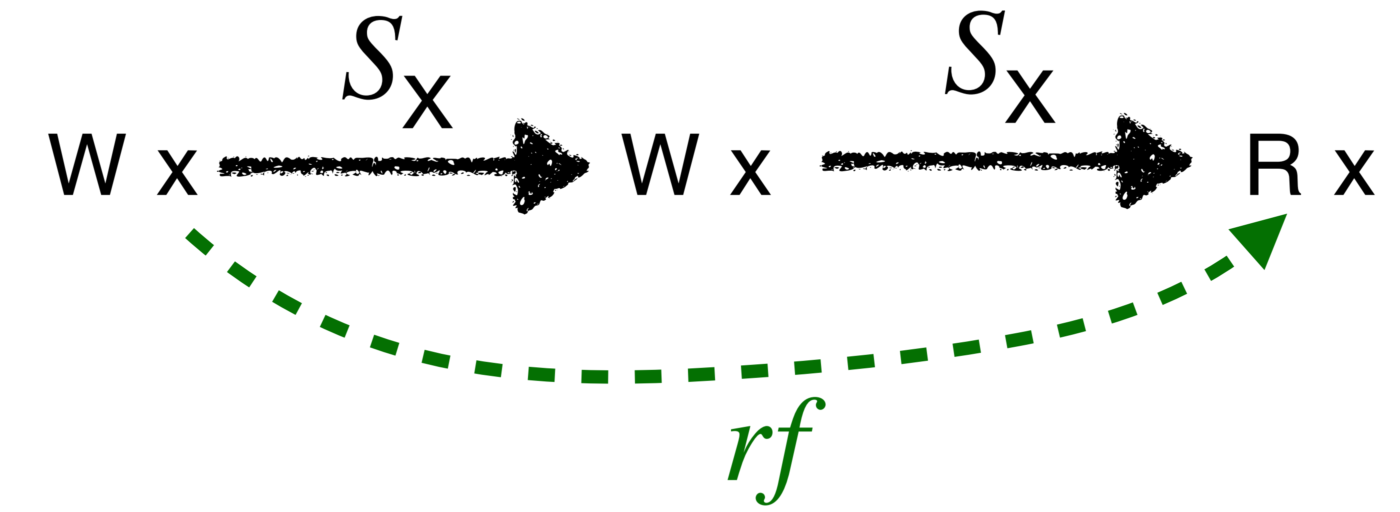
For every location X, there exists a relation S_X such that:

- S_X is a total order on all accesses to X
- S_X contains *hb* when restricted to accesses to X
- *rf* relates every read r from X to the S_X-maximal write that is S_X-before r

disallowed patterns:

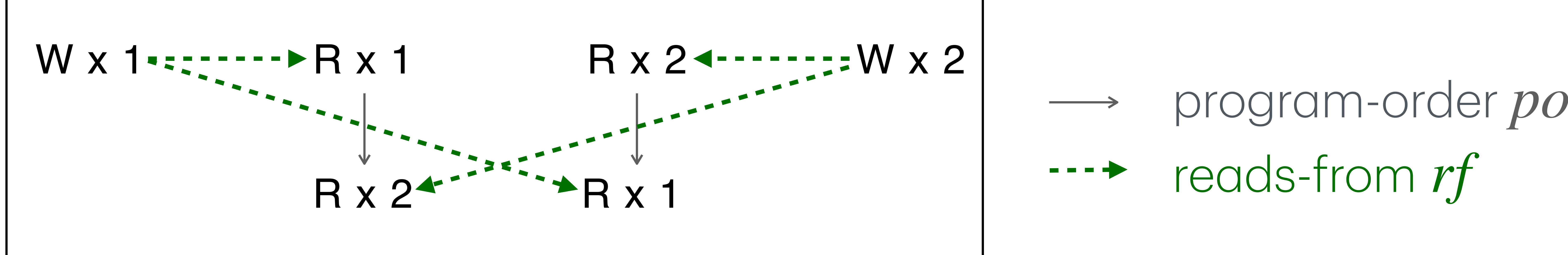






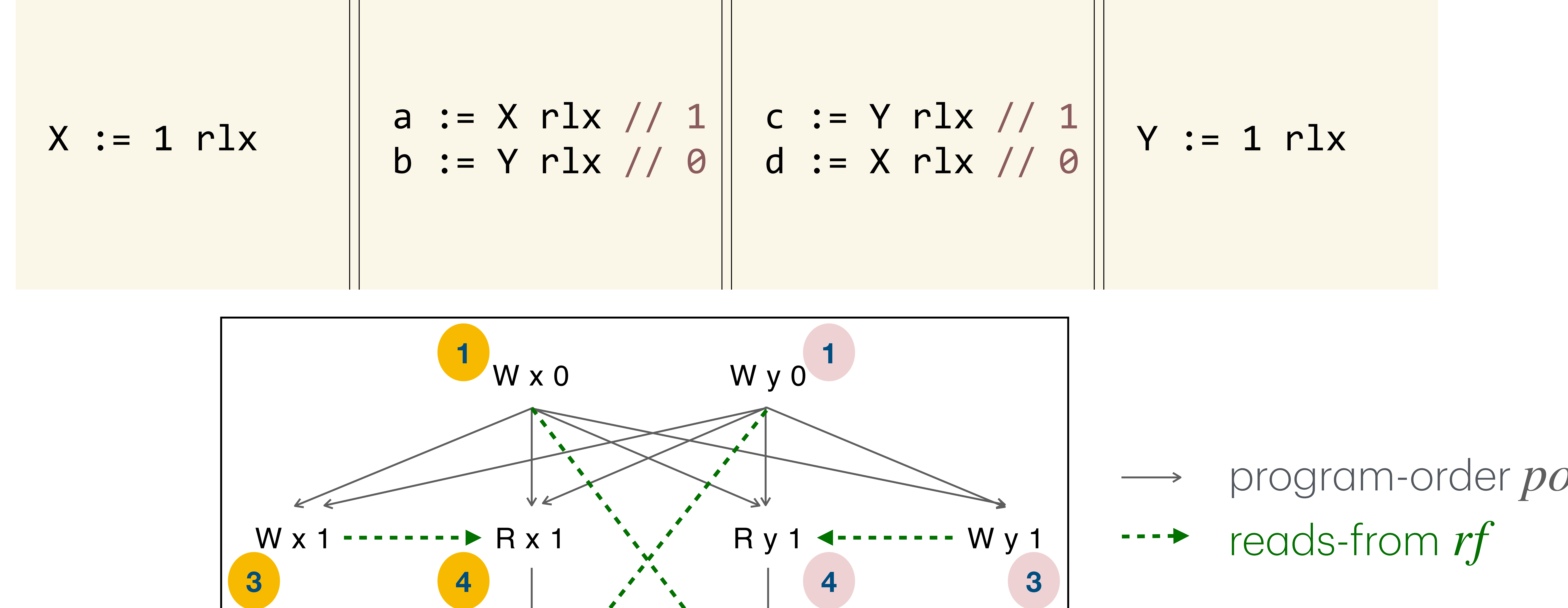
Example: Coherence

`X := 1 rlx`	`a := X rlx // 1` `b := X rlx // 2`	`c := X rlx // 2` `d := X rlx // 1`	`X := 2 rlx`



coherence forbids this behavior

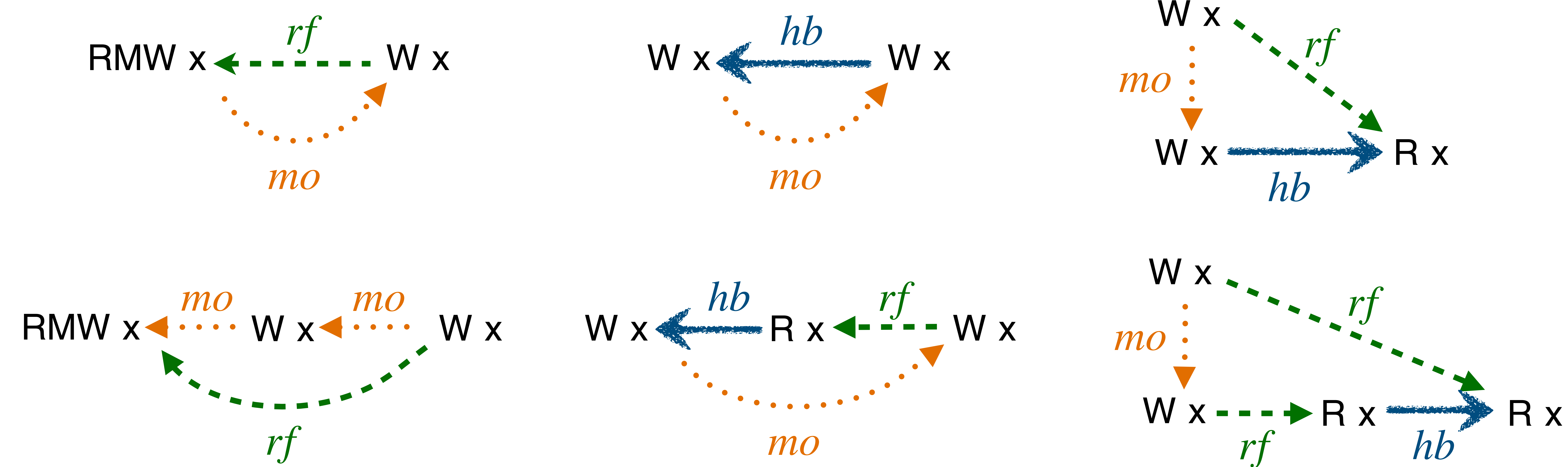
Example: IRIW (independent-reads-independent-writes)



coherence allows this behavior

Alternative formulation of coherence

- include *modification order* (aka coherence order) in execution graphs: $G = \langle E, po, rf, lo, mo, \ldots \rangle$
  - $mo = \bigcup_X mo_X$ where each mo_X is a total order on all **writes** to X
- forbid the following six patterns:

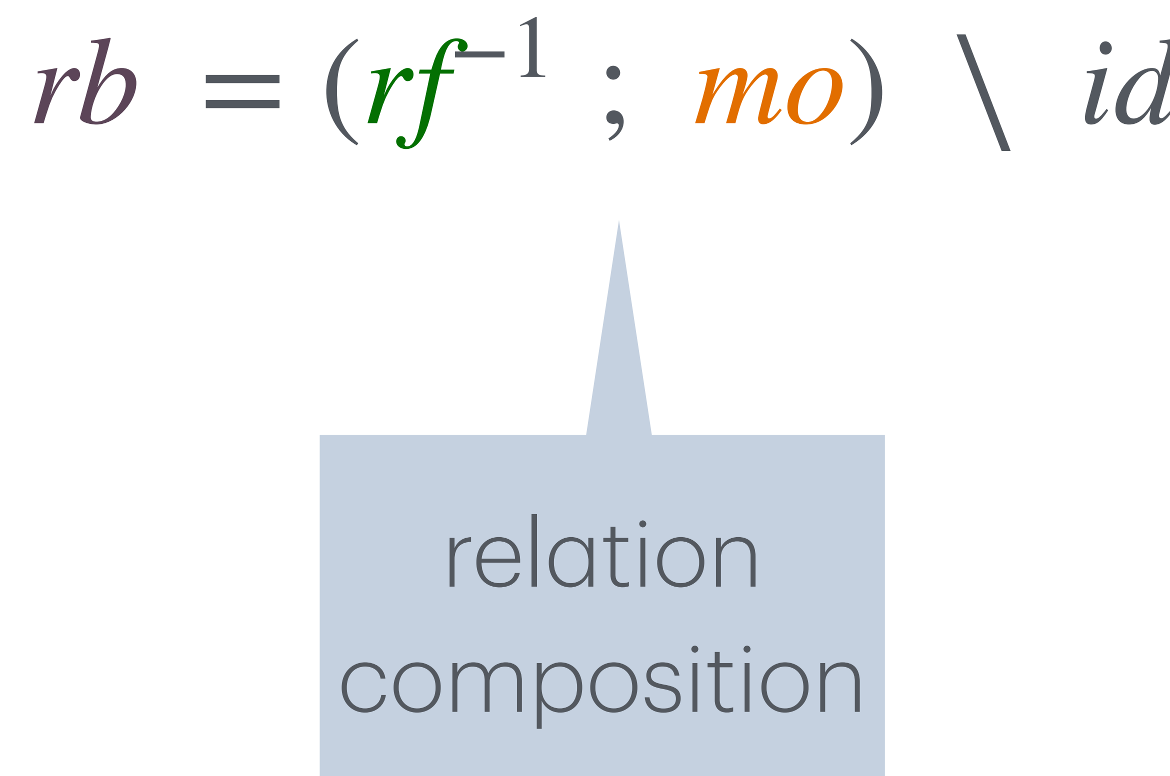


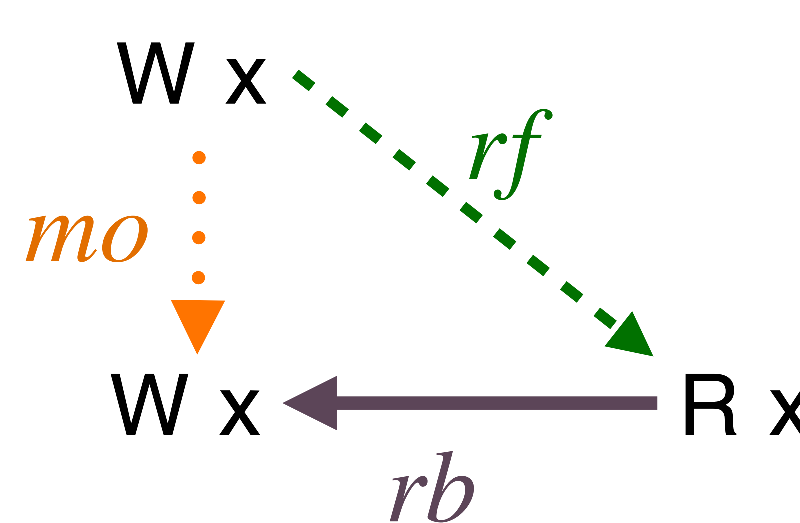
- This is equivalent to the previous formulation with a total order on all accesses to X

A more concise formulation

- Reads-before (aka from-read) relation:

$$rb = (rf^{-1}\ ;\ mo) \setminus id$$





- Coherence:

$$\text{acyclic}(hb|_{same-location} \cup rf \cup mo \cup rb)$$

- Compare to a standard declarative formulation of SC: acyclic($po \cup rf \cup mo \cup rb$)

Another concise formulation

- Extended coherence relation:

$$eco = (rf \cup mo \cup rb)^{+} = rf \cup mo \cup rb \cup (mo\,;rf) \cup (rb\,;rf)$$

- Coherence:

$$eco\,;hb^{?} \text{ is irreflexive}$$

- Compare to another standard declarative formulation of SC: acyclic($eco \cup hb$)

- There is also an alternative equivalent definition that avoids mo altogether:

L, Vafeiadis: **Owicki-Gries Reasoning for Weak Memory Models.** ICALP 2015. http://plv.mpi-sws.org/ogra/full-paper.pdf (appendix B)

Synchronization via atomic accesses

relaxed ⊏ release/acquire ⊏ sc

- Release/Acquire (and SC) accesses form “synchronization edges”:

$$sw = rf \cap (W^{\sqsupseteq \text{rel}} \times R^{\sqsupseteq \text{acq}})$$

$$hb = (po \cup lo \cup sw \cup \ldots)^{+}$$

- The full definition of sw is more involved, allowing more synchronization patterns:
  - using relaxed accesses + release/acquire fences
  - using “release sequences” (definition was changed in C++20)

Examples: MP (message passing)

1

```
          int Y = 0
          int X = 0
                  ||  a := X
Y := 42           ||  if (a=1) then
X := 1            ||    b := Y
```

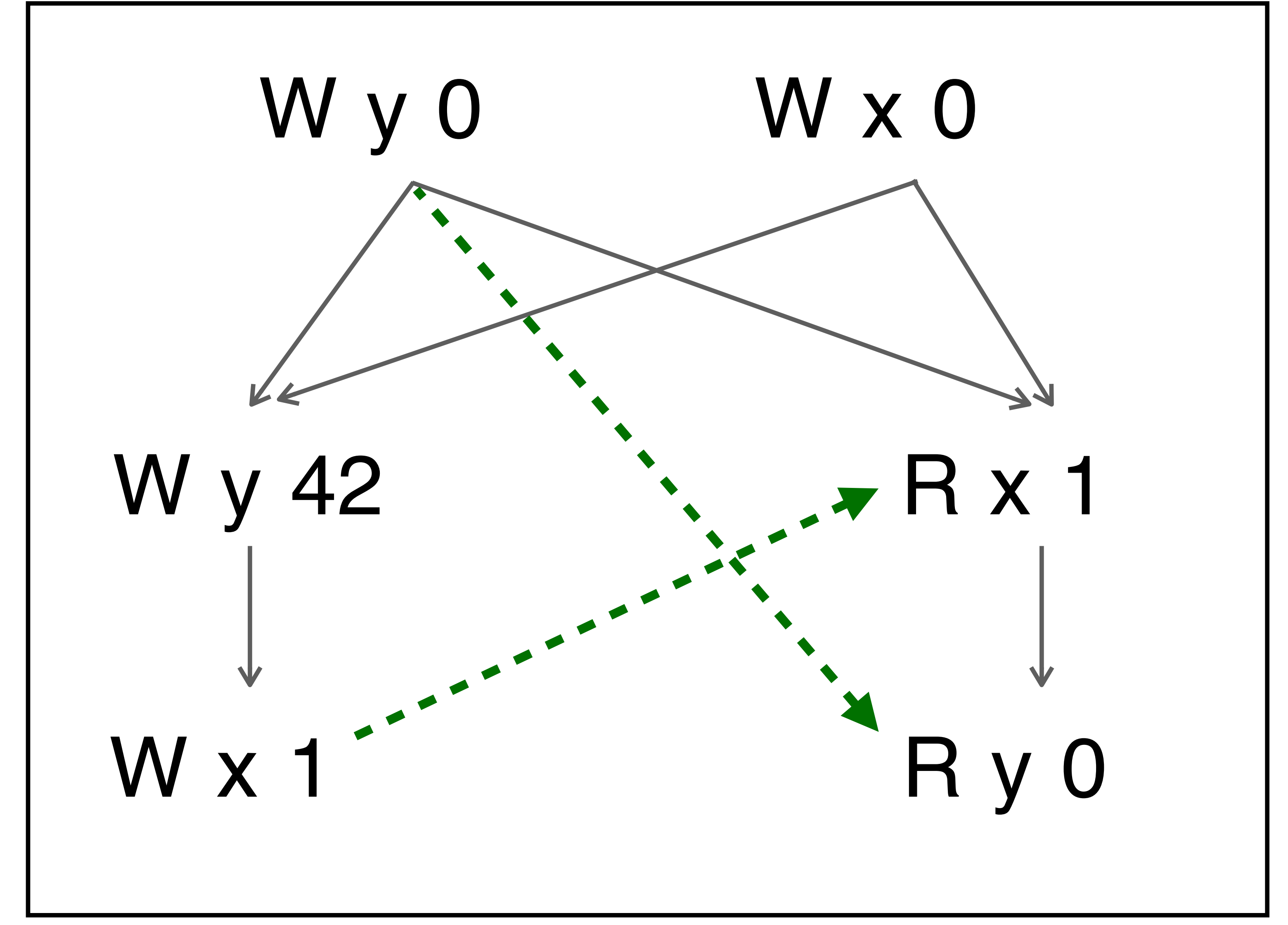
2

```
          int Y = 0
     atomic<int> X = 0
                  ||  a := X rlx
Y := 42           ||  if (a=1) then
X := 1 rlx        ||    b := Y
```

behavior `a=1 & b=0` allowed ?

3

```
          int Y = 0
     atomic<int> X = 0
                  ||  a := X acq
Y := 42           ||  if (a=1) then
X := 1 rel        ||    b := Y
```



Examples: MP (message passing)

1

```
int Y = 0
int X = 0

Y := 42     ||  a := X
X := 1      ||  if (a=1) then
            ||    b := Y
```

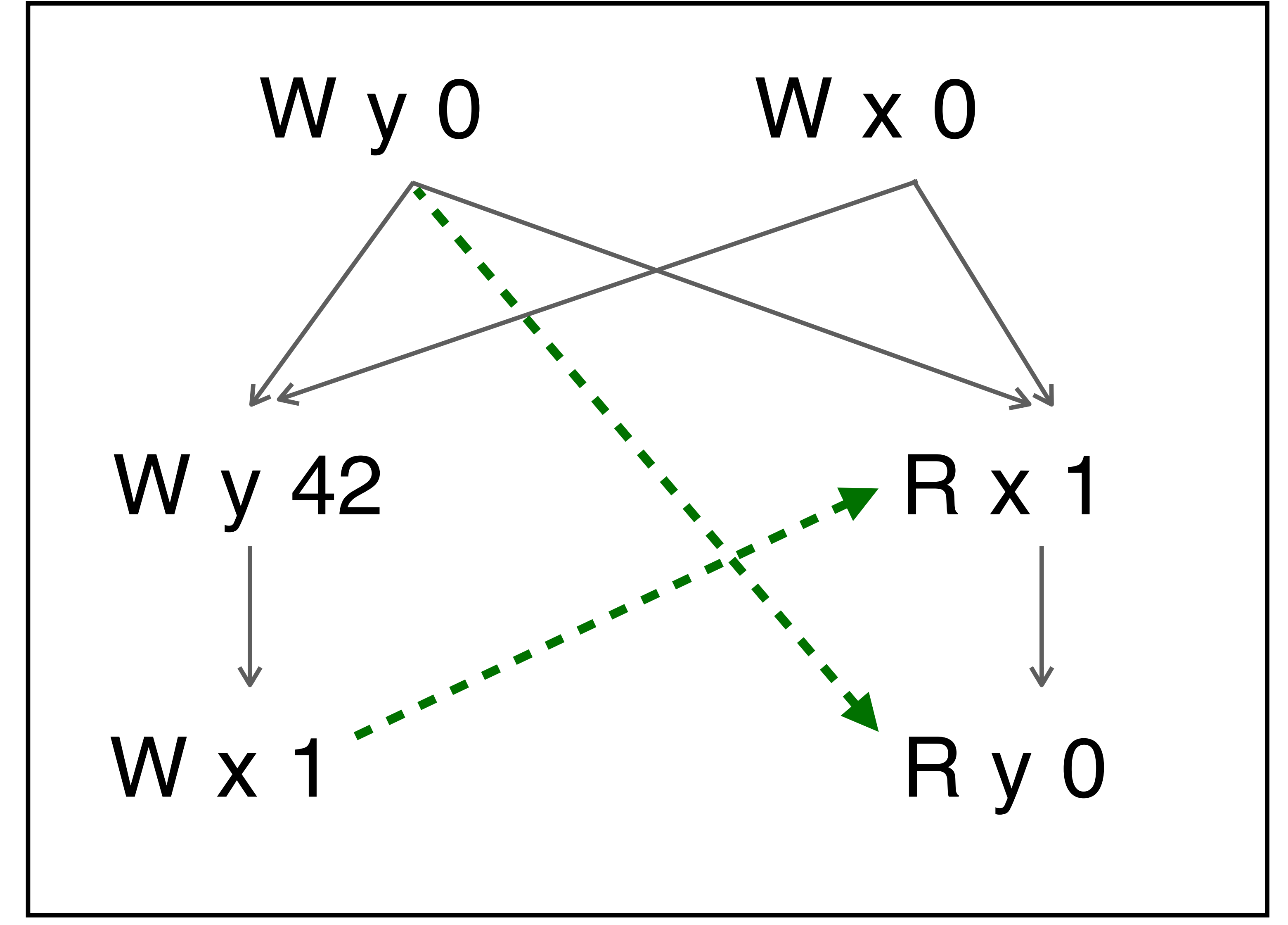
2

```
int Y = 0
atomic<int> X = 0

Y := 42     ||  a := X rlx
X := 1 rlx  ||  if (a=1) then
            ||    b := Y
```

behavior `a=1 & b=0` allowed ?

3

```
int Y = 0
atomic<int> X = 0

Y := 42     ||  a := X acq
X := 1 rel  ||  if (a=1) then
            ||    b := Y
```



Examples: MP (message passing)

1

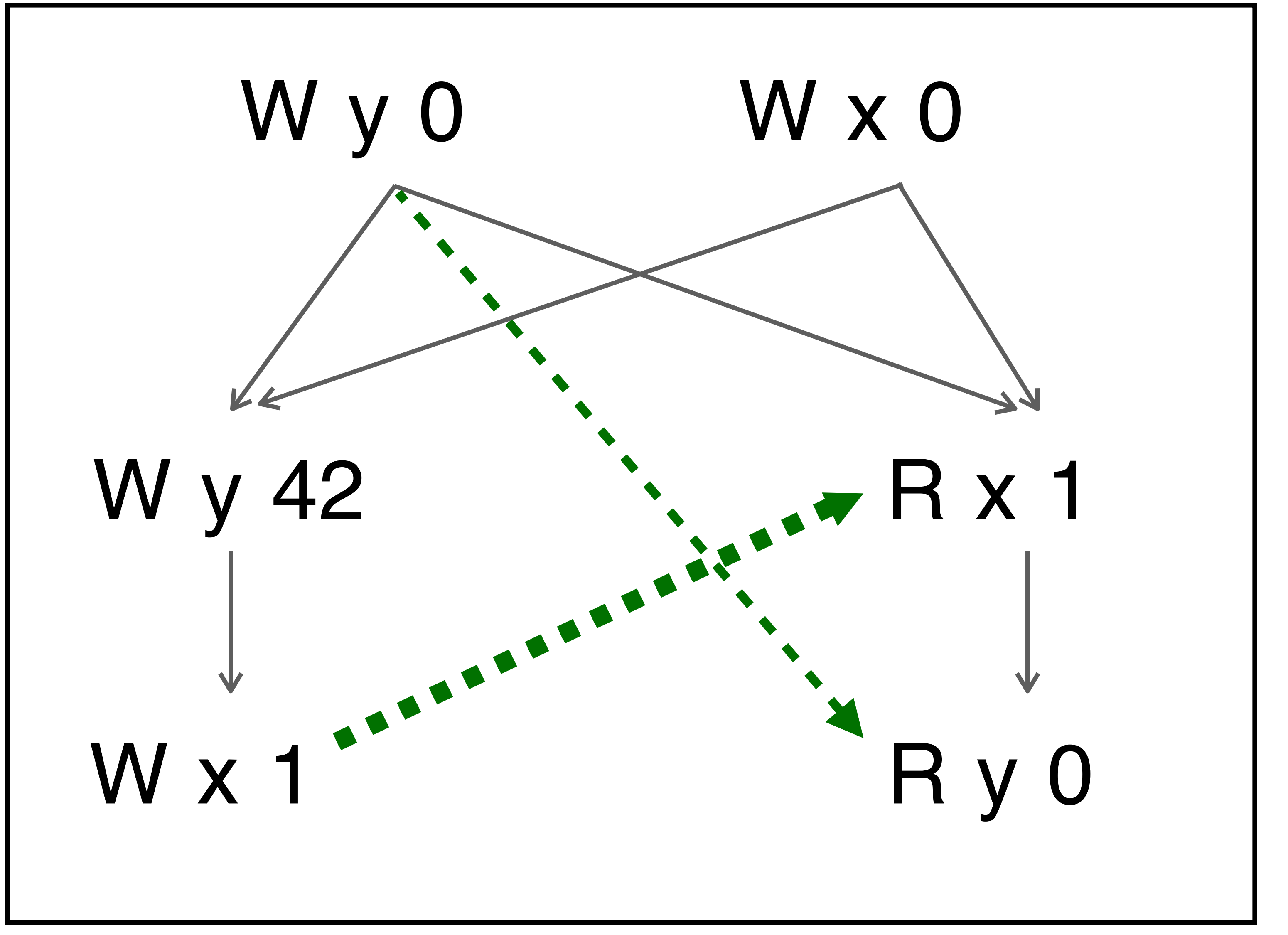
```
          int Y = 0
          int X = 0
                    ||  a := X
Y := 42             ||  if (a=1) then
X := 1              ||    b := Y
```

3

```
          int Y = 0
     atomic<int> X = 0
                    ||  a := X acq
Y := 42             ||  if (a=1) then
X := 1 rel          ||    b := Y
```

2

```
          int Y = 0
     atomic<int> X = 0
                    ||  a := X rlx
Y := 42             ||  if (a=1) then
X := 1 rlx          ||    b := Y
```



behavior `a=1 & b=0` allowed ?

Examples: MP (message passing)

1

```
int Y = 0
int X = 0

Y := 42        ||  a := X
X := 1         ||  if (a=1) then
               ||    b := Y
```

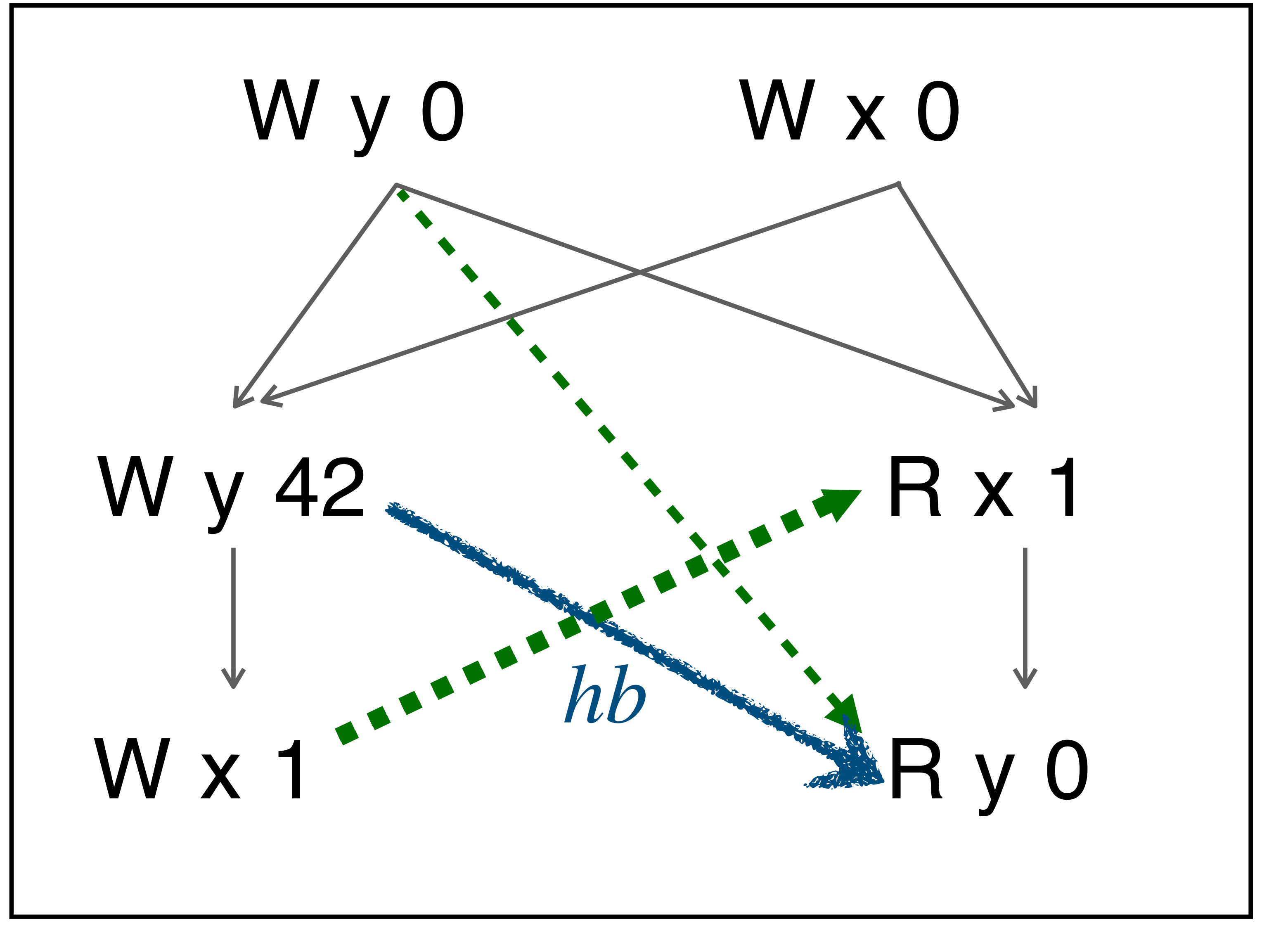
2

```
int Y = 0
atomic<int> X = 0

Y := 42        ||  a := X rlx
X := 1 rlx     ||  if (a=1) then
               ||    b := Y
```

behavior `a=1 & b=0` allowed ?

3

```
int Y = 0
atomic<int> X = 0

Y := 42        ||  a := X acq
X := 1 rel     ||  if (a=1) then
               ||    b := Y
```



SC accesses

relaxed ⊏ release/acquire ⊏ sc

- SC accesses can be used to provide sequentially consistent semantics when needed
- Roughly, there should exist a total order *sc* on all SC accesses in which every read reads from the last write
- The precise semantics is much more complicated

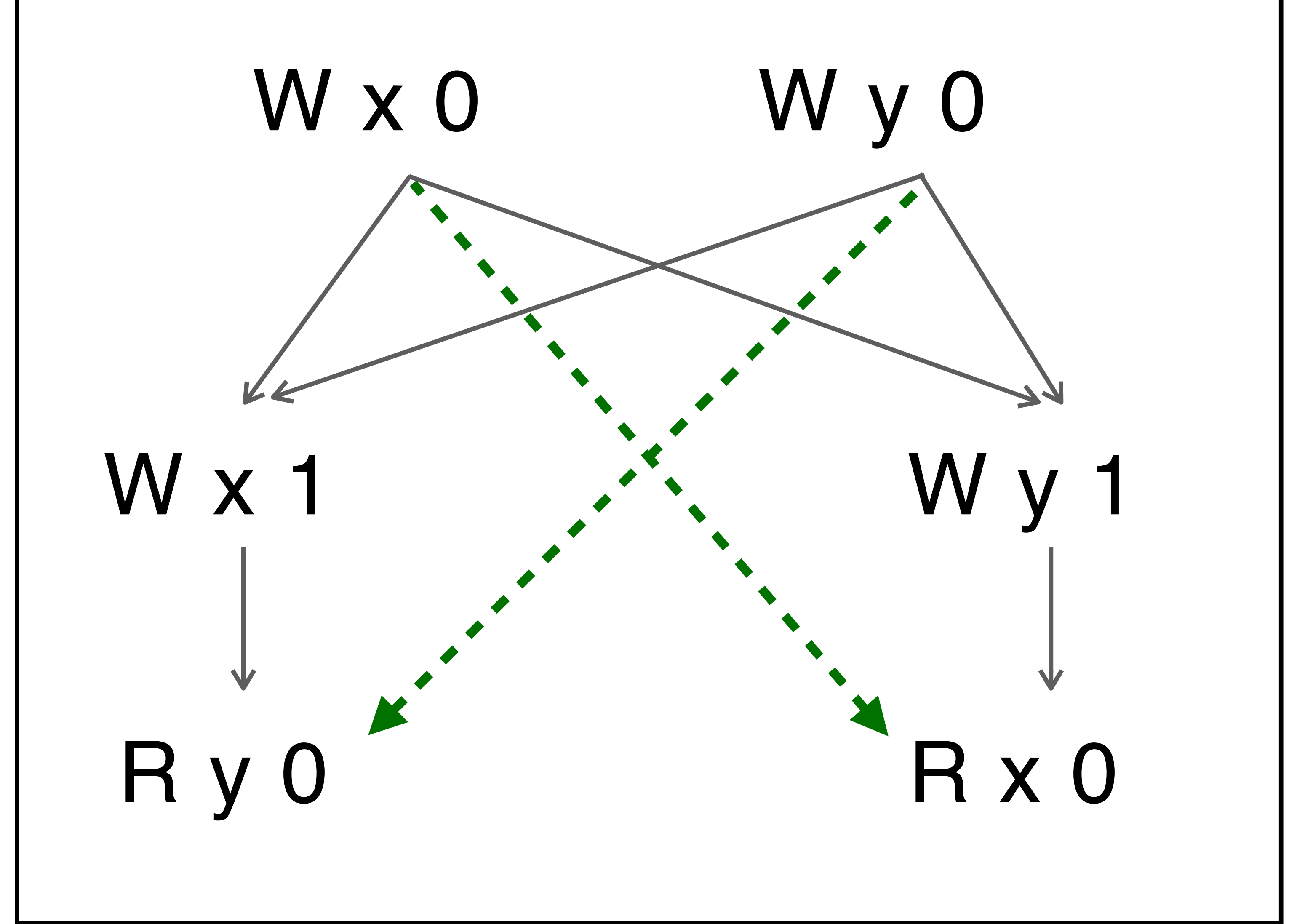
psc is acyclic.

$$
\begin{aligned}
\mathtt{sb}|_{\neq\mathtt{loc}} &\triangleq \mathtt{sb} \setminus \mathtt{sb}|_{\mathtt{loc}} \\
\mathtt{scb} &\triangleq \mathtt{sb} \cup \mathtt{sb}|_{\neq\mathtt{loc}}; \mathtt{hb}; \mathtt{sb}|_{\neq\mathtt{loc}} \cup \mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb} \\
\mathtt{psc}_{\mathrm{base}} &\triangleq \left([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathtt{hb}^{?}\right); \mathtt{scb}; \left([\mathtt{E}^{\mathtt{sc}}] \cup \mathtt{hb}^{?}; [\mathtt{F}^{\mathtt{sc}}]\right) \\
\mathtt{psc}_{\mathtt{F}} &\triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathtt{sc}}] \\
\mathtt{psc} &\triangleq \mathtt{psc}_{\mathrm{base}} \cup \mathtt{psc}_{\mathtt{F}}
\end{aligned}
$$

- It has been a rich source of bugs in the model, and it is currently under another revision...

Example: SB (store buffer)

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

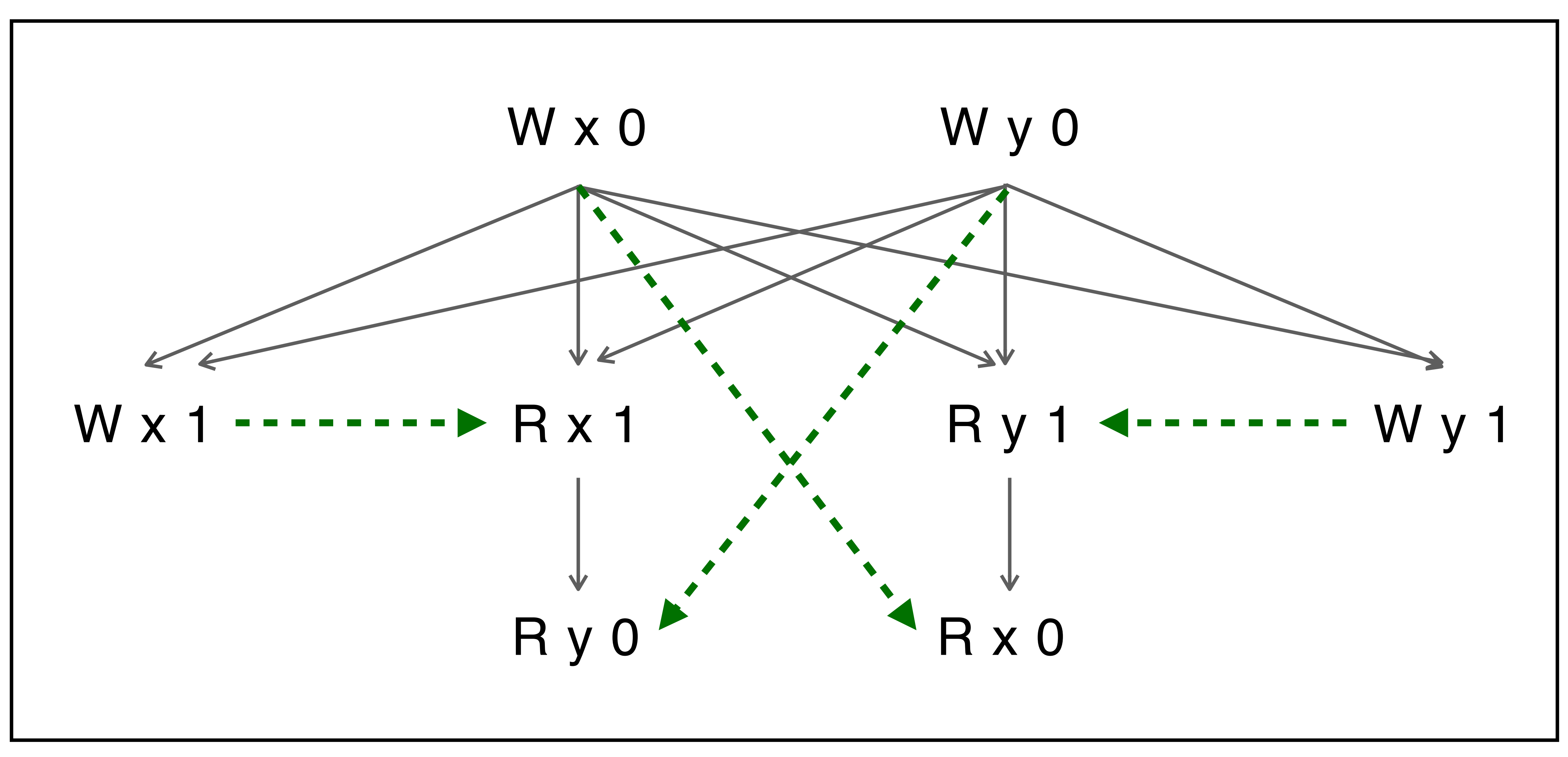


- Allowed with **`release`**/**`acquire`** atomic accesses
- Disallowed when all 4 accesses are **`sc`**

Example: IRIW (independent-reads-independent-writes)

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

- Allowed with **`release`**/**`acquire`** atomic accesses
- Disallowed when all 6 accesses are **`sc`**



Fixing SC accesses in C11

`X := 1 sc`	`a := X acq // 1` `b := Y sc  // 0`	`c := Y acq // 1` `d := X sc  // 0`	`Y := 1 sc`

- In the original C11 model this behavior was disallowed (the order *sc* had to agree with ***hb***)
- But it is allowed on POWER multicores after compilation mapping!
- The C/C++11 was weakened in order to solve this problem

L, Vafeiadis, Kang, Hur, Dreyer: **Repairing Sequential Consistency in C/C++11.** PLDI 2017. https://doi.org/10.1145/3140587.3062352

SC fences

```
atomic_thread_fence(memory_order_seq_cst)
```

- SC-fences provide another way to enforce SC semantics when needed
- Consistency essentially requires that there exists a total order sc_F on all SC fences in the graph that is a part of hb:

$$hb = (po \cup lo \cup sw \cup sc_F)^+$$

SC fences

- Weak behaviors can be forbidden by placing SC-fences:

X := 1 rlx SC-fence a := Y rlx // 0	Y := 1 rlx SC-fence b := X rlx // 0

X := 1 rlx	a := X rlx // 1 SC-fence b := Y rlx // 0	c := Y rlx // 1 SC-fence d := X rlx // 0	Y := 1 rlx

- SC-fences are often preferred by expert developers, making SC accesses rather useless…
- SC-fences can be encoded as release/acquire RMWs (e.g., `FADD(F,0)`) to a distinguished, otherwise unused location

Recap: The C11 memory model

- Catch-fire: races on non-atomics $\implies$ undefined behavior

- Relaxed atomics for racy (but non-synchronizing) accesses
- Atomics ensure coherence
- Locks and release/acquire atomics for synchronization
- SC atomics / fences for ensuring a global total order

relaxed $\sqsubset$ release/acquire $\sqsubset$ sc

The out-of-thin-air problem & RC11

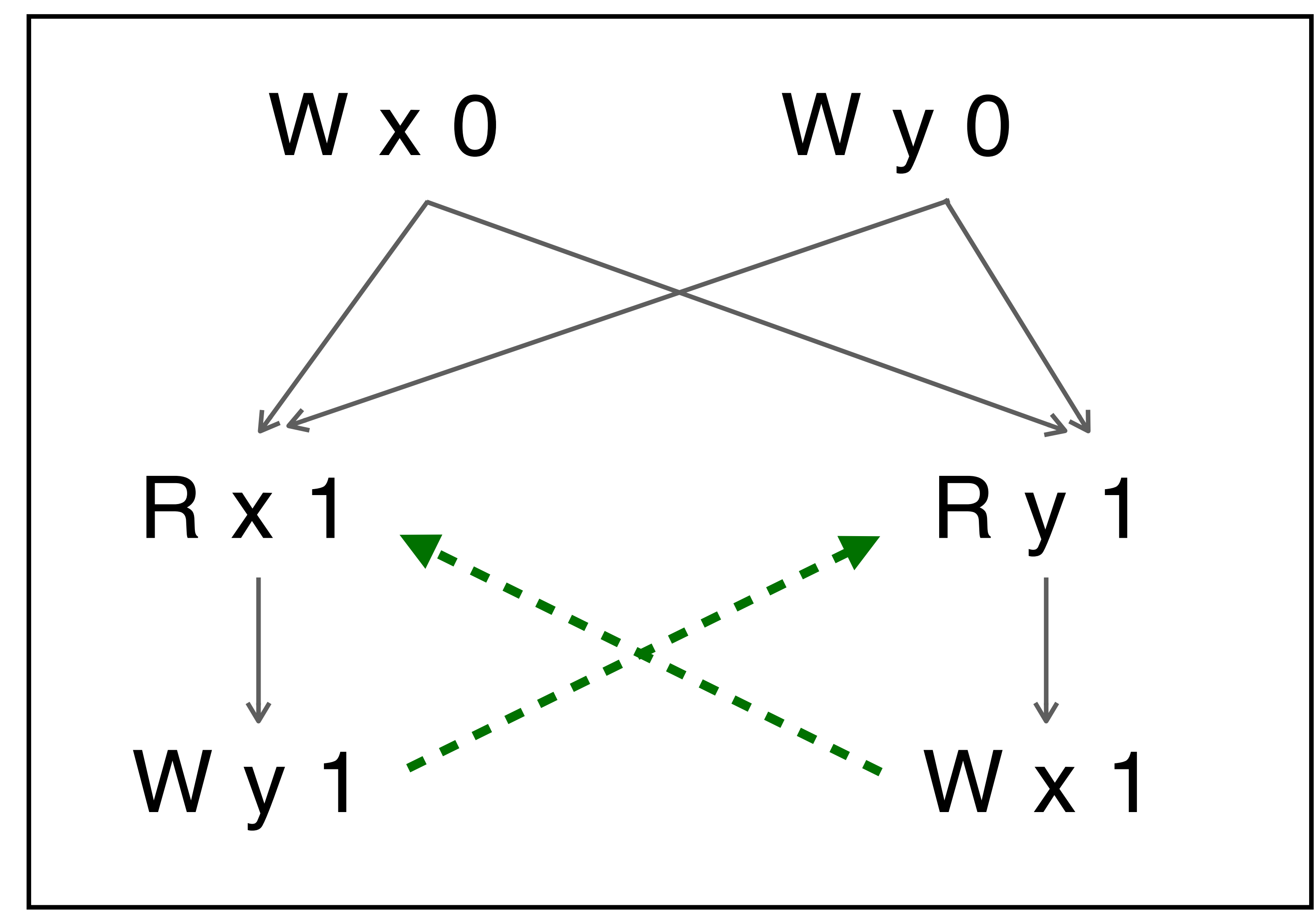
The out-of-thin-air problem

- The model presented so far is too weak.
- Values might appear **“out-of-thin-air”**!
- For the same reason, the DRF guarantee is broken (we will discuss later).

```
std::atomic<int> x = 0;
std::atomic<int> y = 0;

// Thread 1
int r1 = y.load(std::memory_order_relaxed);  // A
x.store(r1, std::memory_order_relaxed);  // B
// Thread 2
int r2 = x.load(std::memory_order_relaxed);  // C
y.store(r2, std::memory_order_relaxed);  // D
```

Example: LB (load buffer)

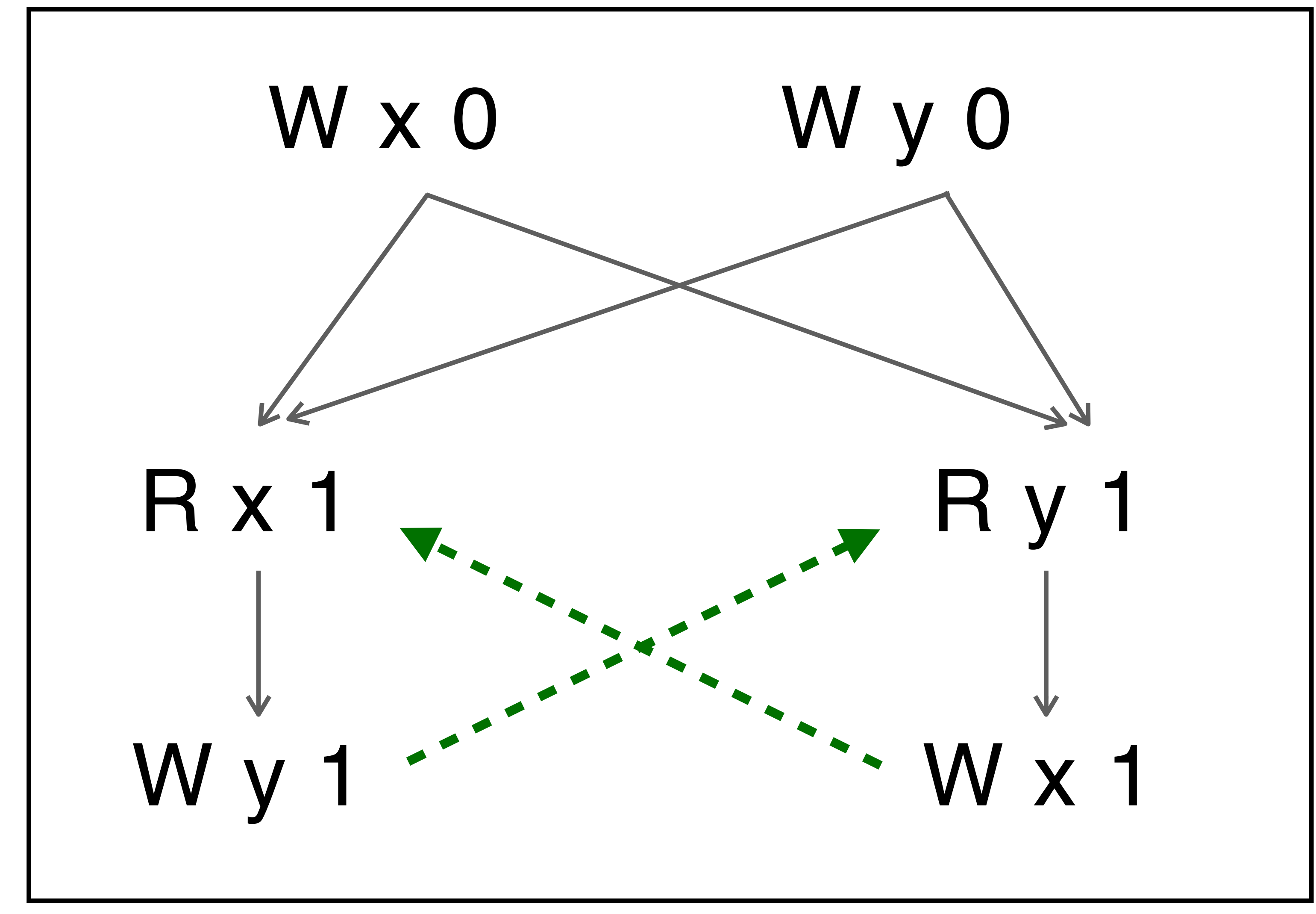
```
a := X rlx // 1     ||  b := Y rlx // 1
Y := 1 rlx          ||  X := 1 rlx
```



- C11 allows this behavior, for a good reason:
  - We want to compile relaxed accesses to plain machine accesses
  - Hardware models (POWER / ARM) allow it

Example: LB (load buffer)

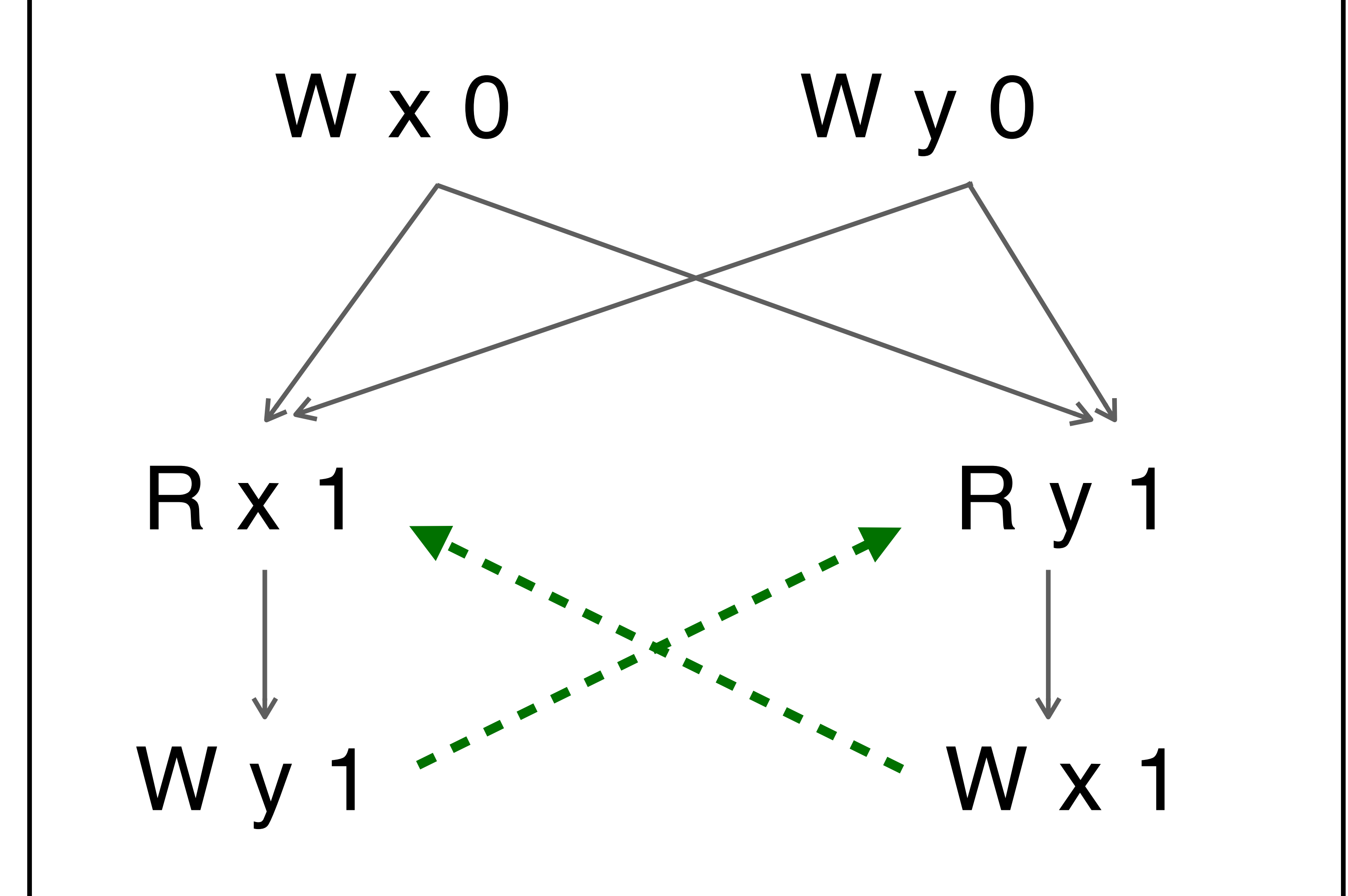
```
a := X rlx // 1     ||  b := Y rlx // 1
Y := a rlx          ||  X := b rlx
```



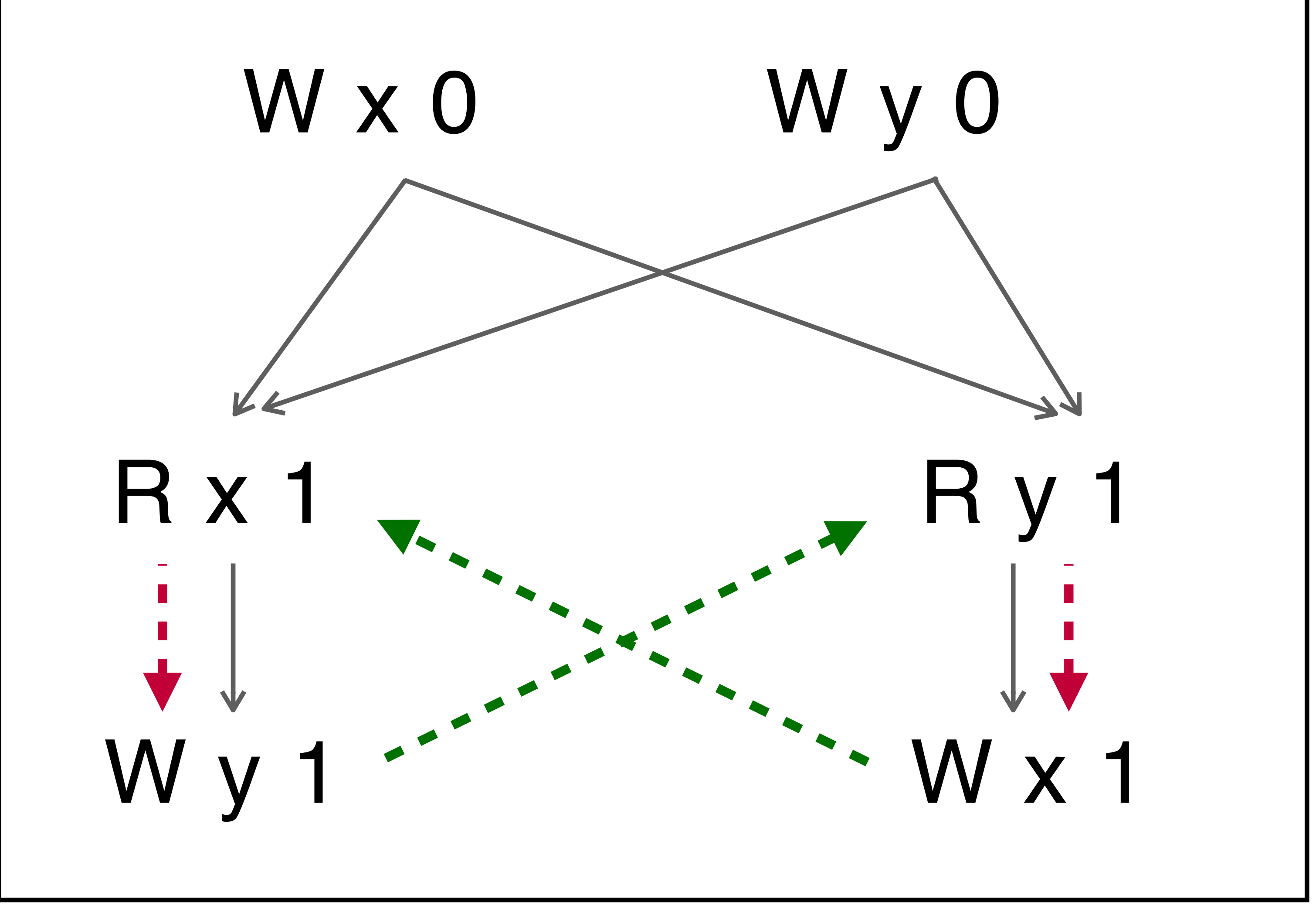
- But, it means that it also allows the above behavior
- The two behaviors are represented by the same execution graph!
- The value 1 appears ***"out-of-thin-air"***

The hardware solution

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




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



- Hardware models forbid: (*dep* ∪ *rf*) cycles

dependency *dep*

The hardware solution

- Hardware execution graph maintain *dependency relation* among events
- This is **not** a viable option for a PL since compilers may remove syntactic dependencies
- Devising a good "semantic" notion of dependency is an open challenge

```
a := X // 1        ||   b := Y // 1
Y := 1 + a - a     ||   X := 1 + b - b
```

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

The out-of-thin-air problem

- The C++14 standard states:

“Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.”

- But doesn’t give a sufficiently formal definition…

“Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still does not have a credible proposal for the concurrency semantics of any general-purpose high-level language that includes high performance shared-memory concurrency primitives. This is a major open problem for programming language semantics.”

Batty, Memarian, Nienhuis, Pichon-Pharabod, Sewell: **The Problem of Programming Language Concurrency Semantics.** ESOP 2015. https://doi.org/10.1007/978-3-662-46669-8_12

RC11: a conservative approach

- Disallow $(po \cup rf)$ cycles altogether.
  - Implementation cost: forbid RW-reodering for relaxed accesses
  - Importantly, reodering of non-atomic accesses is still sound!
  - Different strategies and their performance implications were investigated:

    Ou, Demsky: **Towards understanding the costs of avoiding out-of-thin-air results.** OOPSLA 2018. https://doi.org/10.1145/3276506
- The obtained model is called **RC11** ("repaired C11").

Boehm, Demsky: **Outlawing ghosts: avoiding out-of-thin-air results.** MSPC 2014. https://doi.org/10.1145/2618128.2618134

L, Vafeiadis, Kang, Hur, Dreyer: **Repairing Sequential Consistency in C/C++11.** PLDI 2017. https://doi.org/10.1145/3140587.3062352

Alternative proposals

- Solving the out-of-thin-air problem without changing the compilation schemes requires a major revision of the standard

- We cannot have a per-execution definition: validity of one execution depends on what happens in other executions

- Some prominent proposals:

  - Chakraborty, Vafeiadis. **Grounding thin-air reads with event structures.** POPL 2019. https://doi.org/10.1145/3290383

  - Jeffrey, Riely, Batty, Cooksey, Kaysin, Podkopaev. **The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency.** POPL 2022. https://doi.org/10.1145/3498716

  - Kang, Hur, L, Vafeiadis, Dreyer. **A promising semantics for relaxed-memory concurrency.** POPL 2017. https://doi.org/10.1145/3009837.3009850

RC11

- In the rest of this presentation, we mostly assume RC11: $(po \cup rf)$ is acyclic

- This model has been extensively studied in recent years:
  - acyclicity of $(po \cup rf)$ allows adaptations of existing techniques
  - we think about the system executing the program **“in-order”** on top of a non-standard memory system

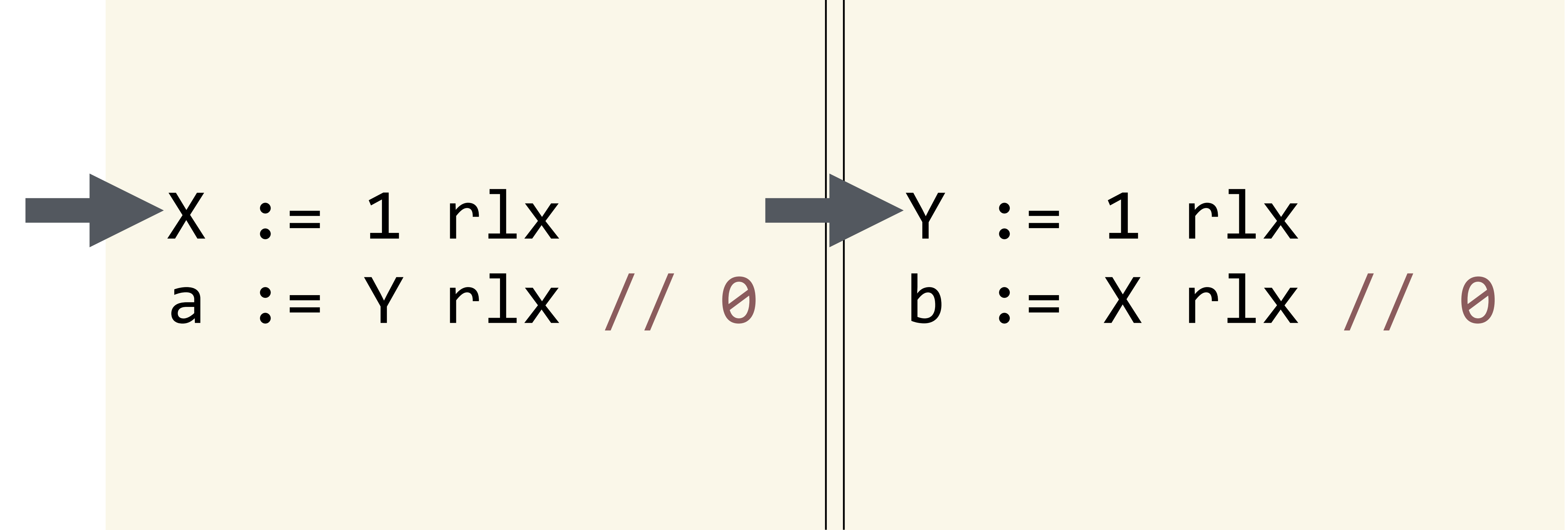
Operationalizing RC11

- State is the execution graph produced so far
- Non-deterministic choice where to read from (and where to place writes in the modification order)
- Consistency is checked at every step
- This memory system is synchronized with an "in-order" program semantics

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

Operationalizing RC11

- State is the execution graph produced so far
- Non-deterministic choice where to read from (and where to place writes in the modification order)
- Consistency is checked at every step
- This memory system is synchronized with an “in-order” program semantics



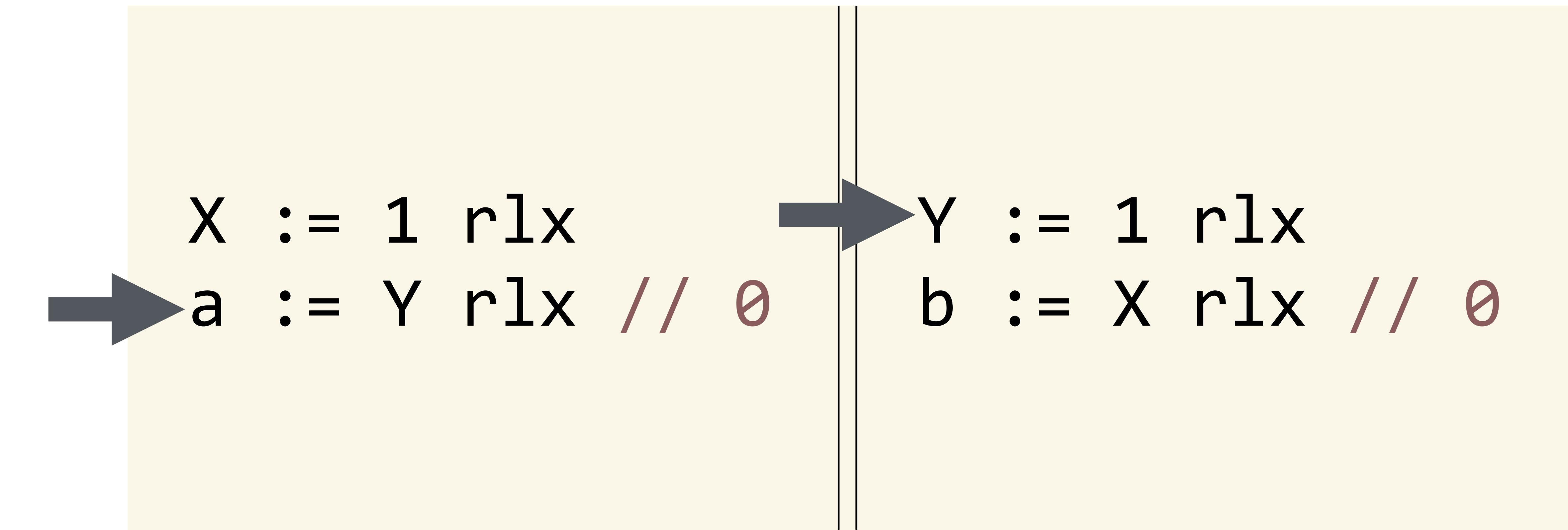
W x 0 W y 0

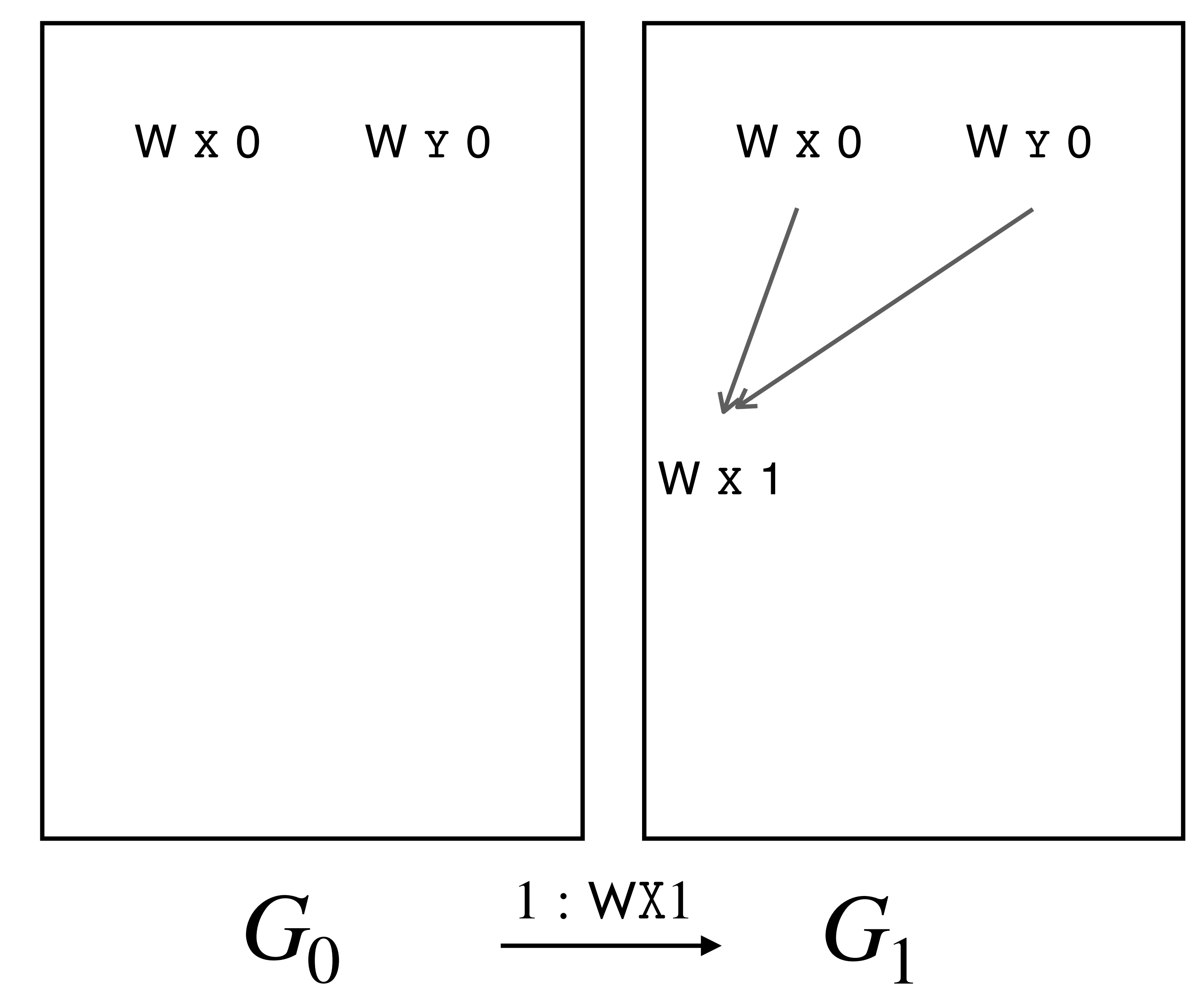
G_0

initial state

Operationalizing RC11

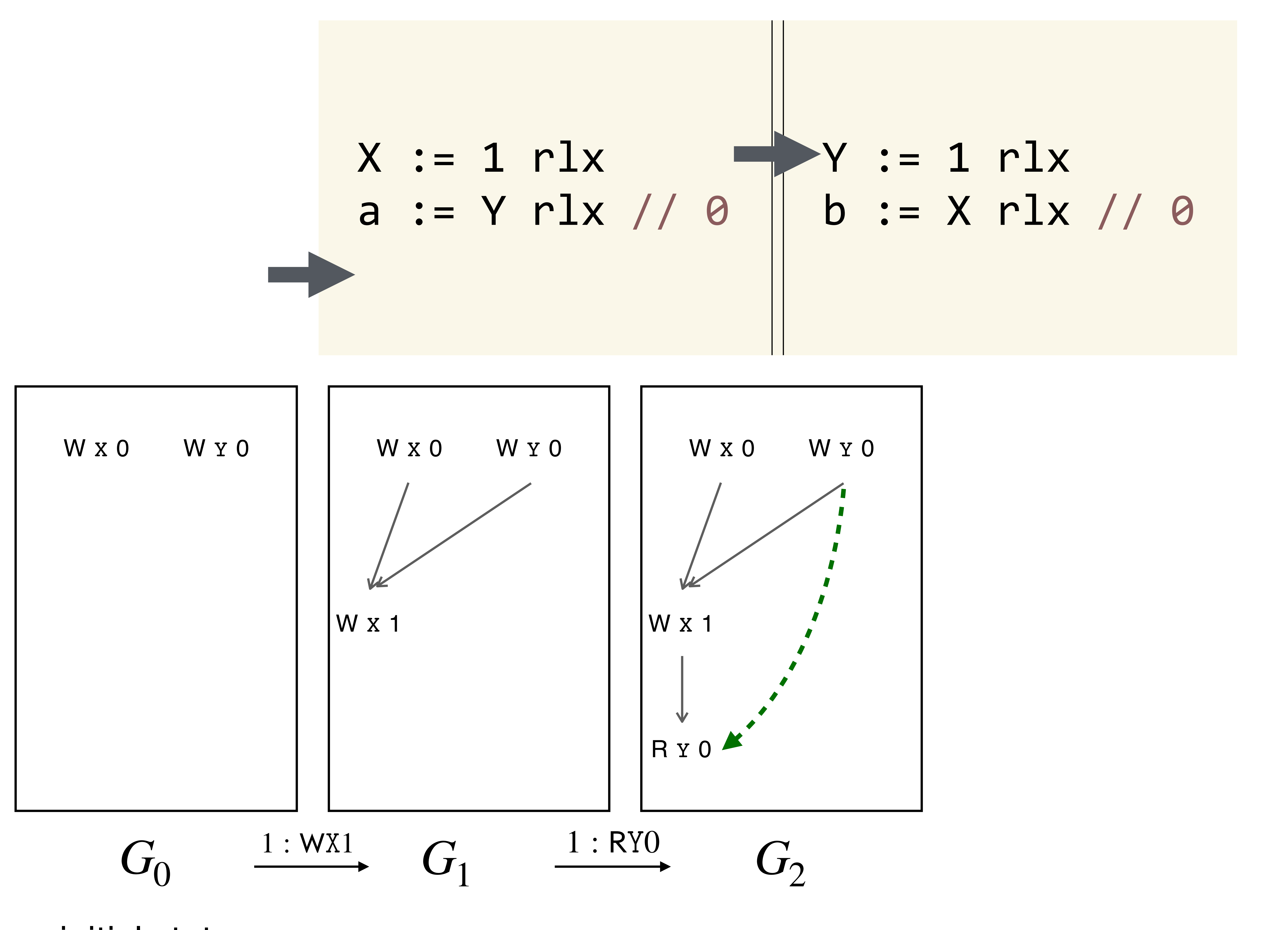
- State is the execution graph produced so far
- Non-deterministic choice where to read from (and where to place writes in the modification order)
- Consistency is checked at every step
- This memory system is synchronized with an “in-order” program semantics





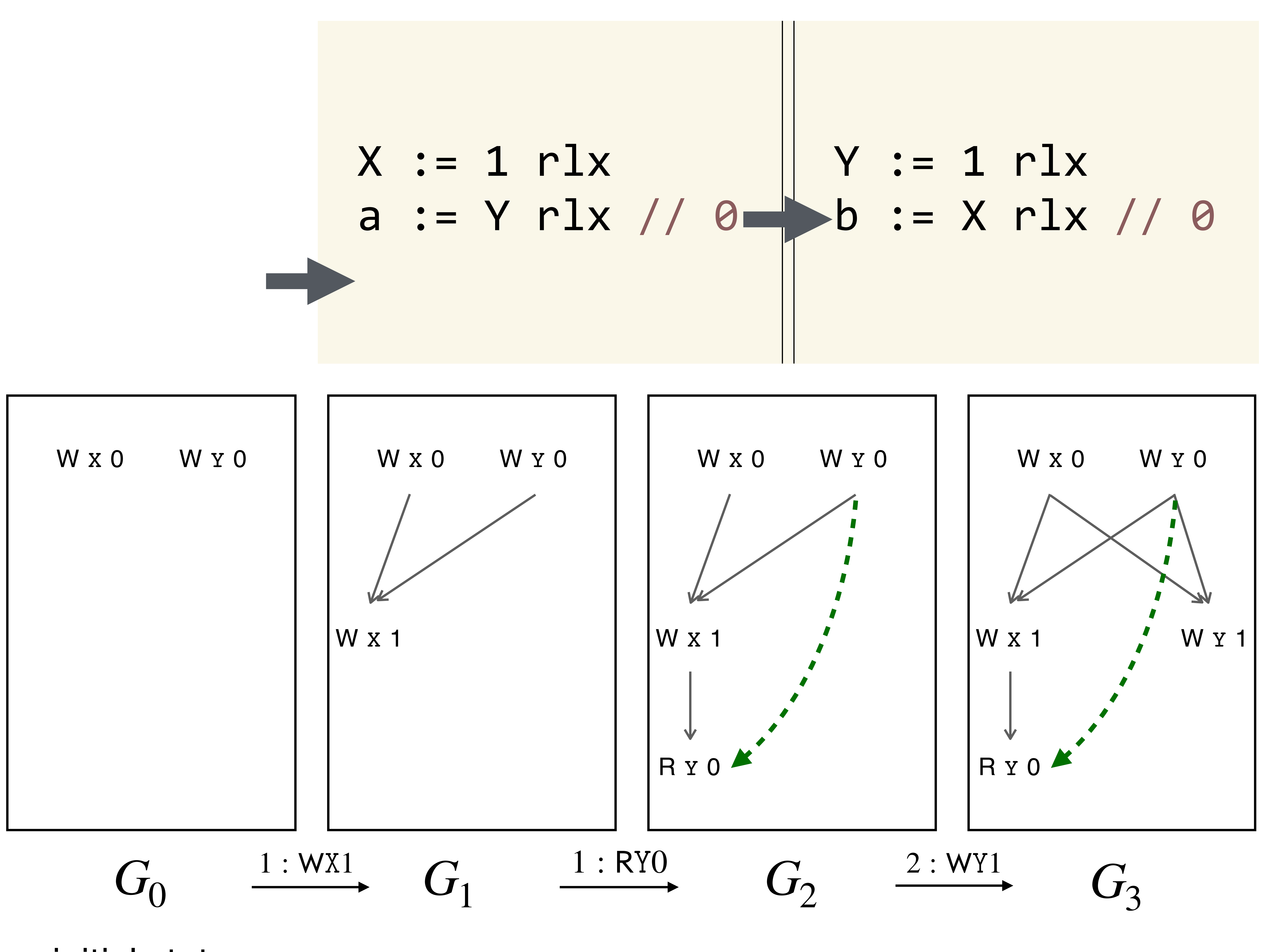
Operationalizing RC11

- State is the execution graph produced so far
- Non-deterministic choice where to read from (and where to place writes in the modification order)
- Consistency is checked at every step
- This memory system is synchronized with an "in-order" program semantics



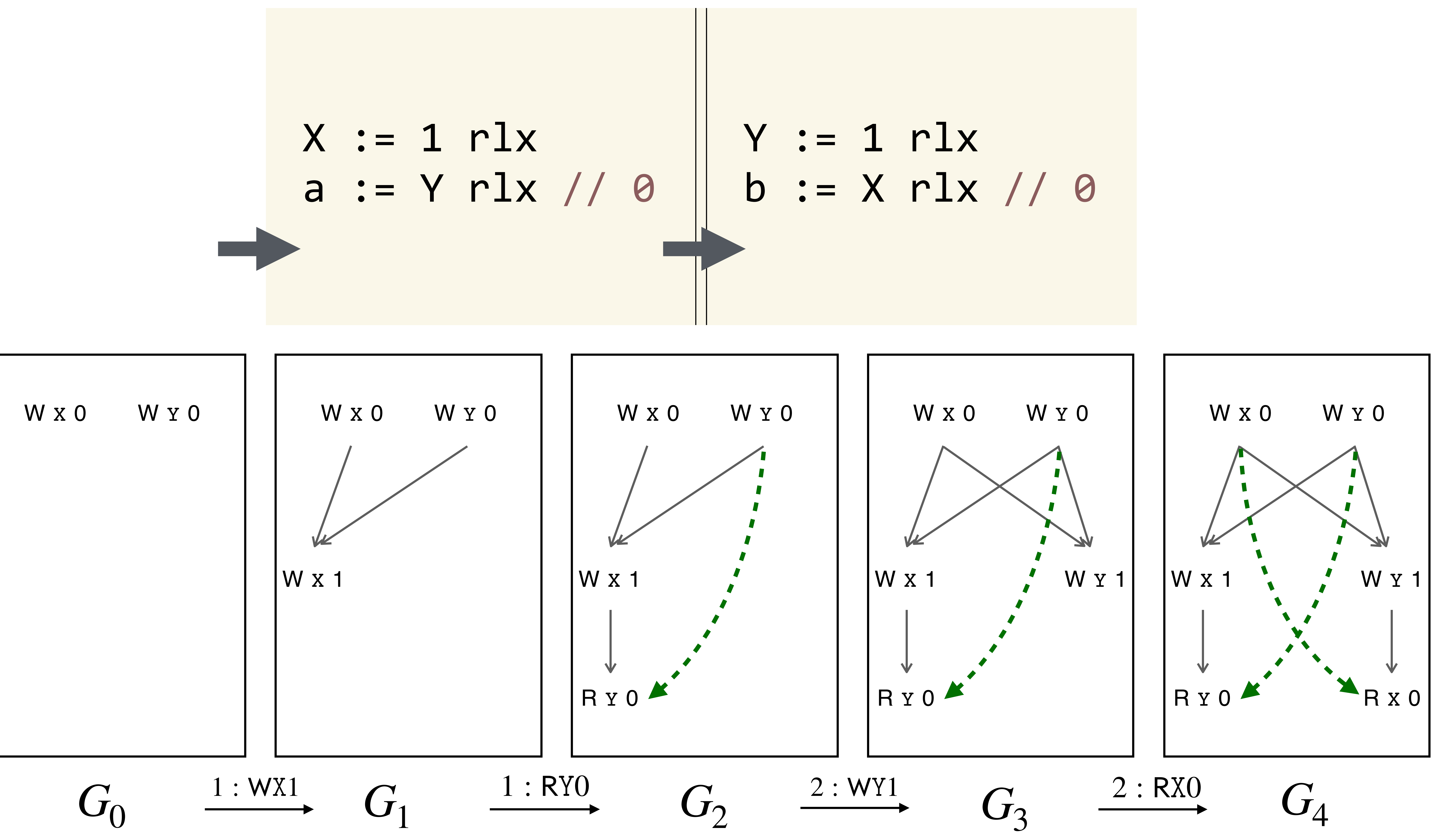
Operationalizing RC11

- State is the execution graph produced so far
- Non-deterministic choice where to read from (and where to place writes in the modification order)
- Consistency is checked at every step
- This memory system is synchronized with an "in-order" program semantics



Operationalizing RC11

- State is the execution graph produced so far
- Non-deterministic choice where to read from (and where to place writes in the modification order)
- Consistency is checked at every step
- This memory system is synchronized with an “in-order” program semantics



Operationalizing RC11

- Observation:
  - we don't need the full execution graph in the state
  - we only need the part that can affect consistency of later accesses
- This leads to more compact presentations (somewhat similar to distributed implementations)
- Note: The state space remains infinite (for program with loops)
  - important implications for algorithmic verification

- Next, we demonstrate this idea for the ***RA fragment***

relaxed ⊏ release/acquire ⊏ sc

The RA memory model

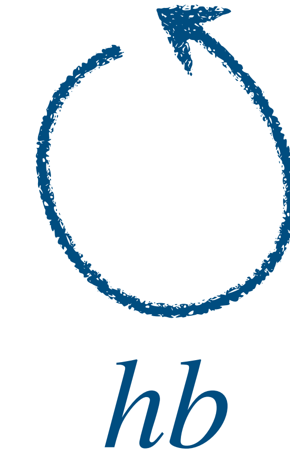
- An well-studied fragment of C11 is RA (intricate but not overwhelmingly detailed)
- Ensures causal consistency & coherence
- Supports "flag-based synchronization"

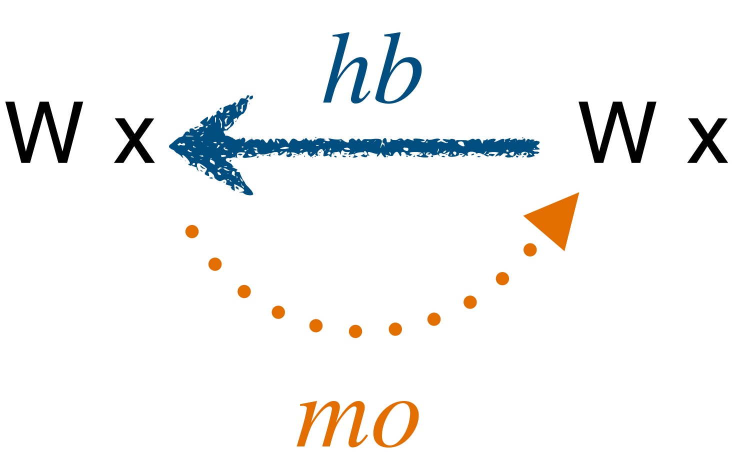
```
Y := 42          ||  a := X
X := 1           ||  if (a=1) then
                 ||    b := Y // 0
```

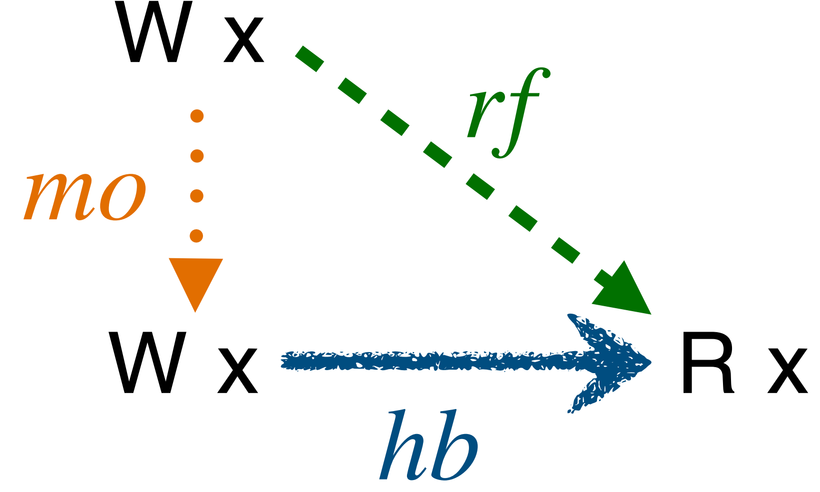
- Allows WR-reordering
- Threads can disagree about the order of writes: non-multi-copy-atomic
- Locks can be implemented using RMWs
- SC-fences can be encoded as RMWs to a distinguished otherwise unused location

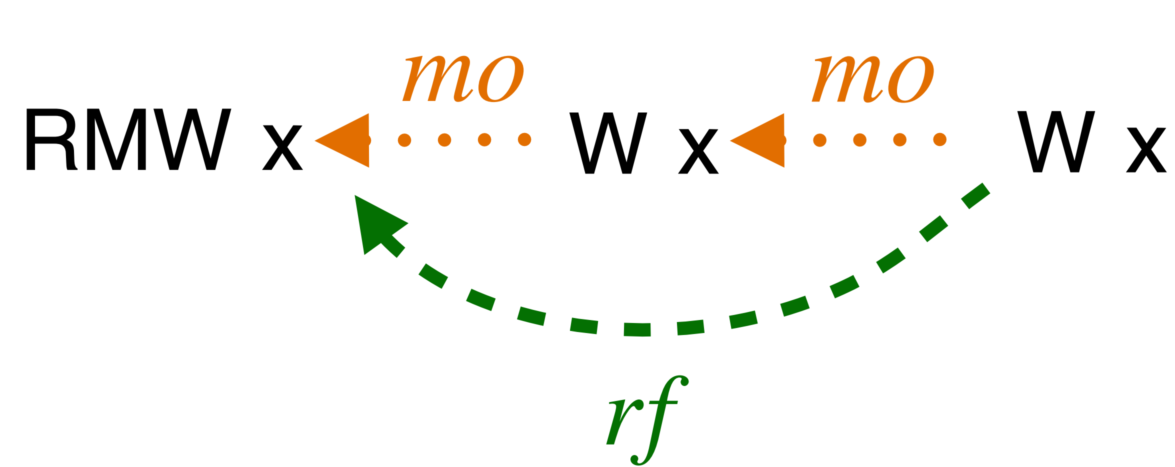
Declarative RA

- When restricting RC11 to only release/acquire accesses:
  - $hb = (po \cup rf)^+$
  - Four disallowed patterns:







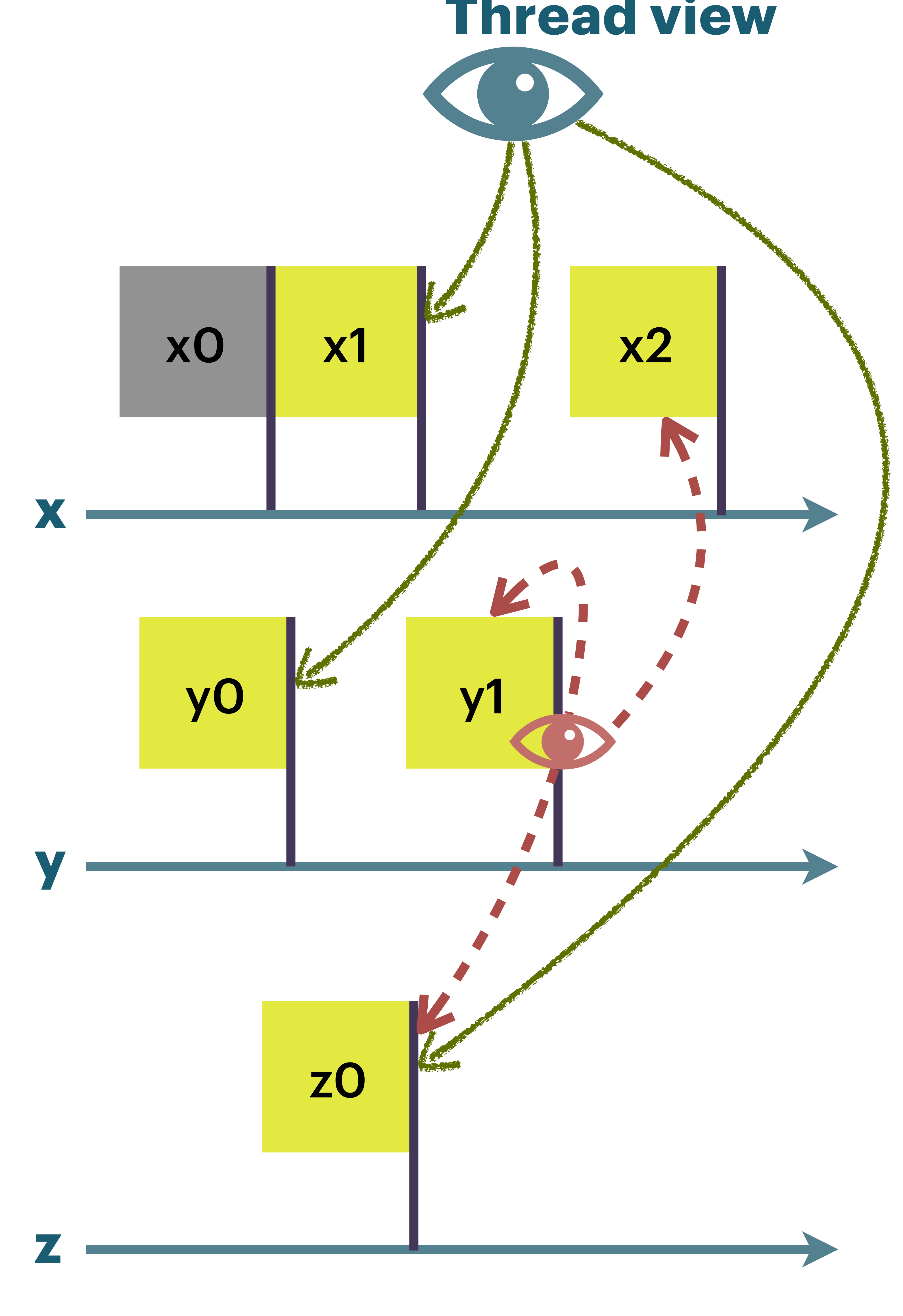


- Concise formulation: acyclic($hb|_{same-location} \cup mo \cup rb$)

Operational formulation

A view-based semantics

- Memory: Timeline per location (represents *mo*)
- Populated with immutable messages holding values
- Each view points to msgs on each timeline
- Threads have views — cannot read from “the past”
- Msgs have views for enforcing causal propagation
- Simulates the graph-based operational semantics



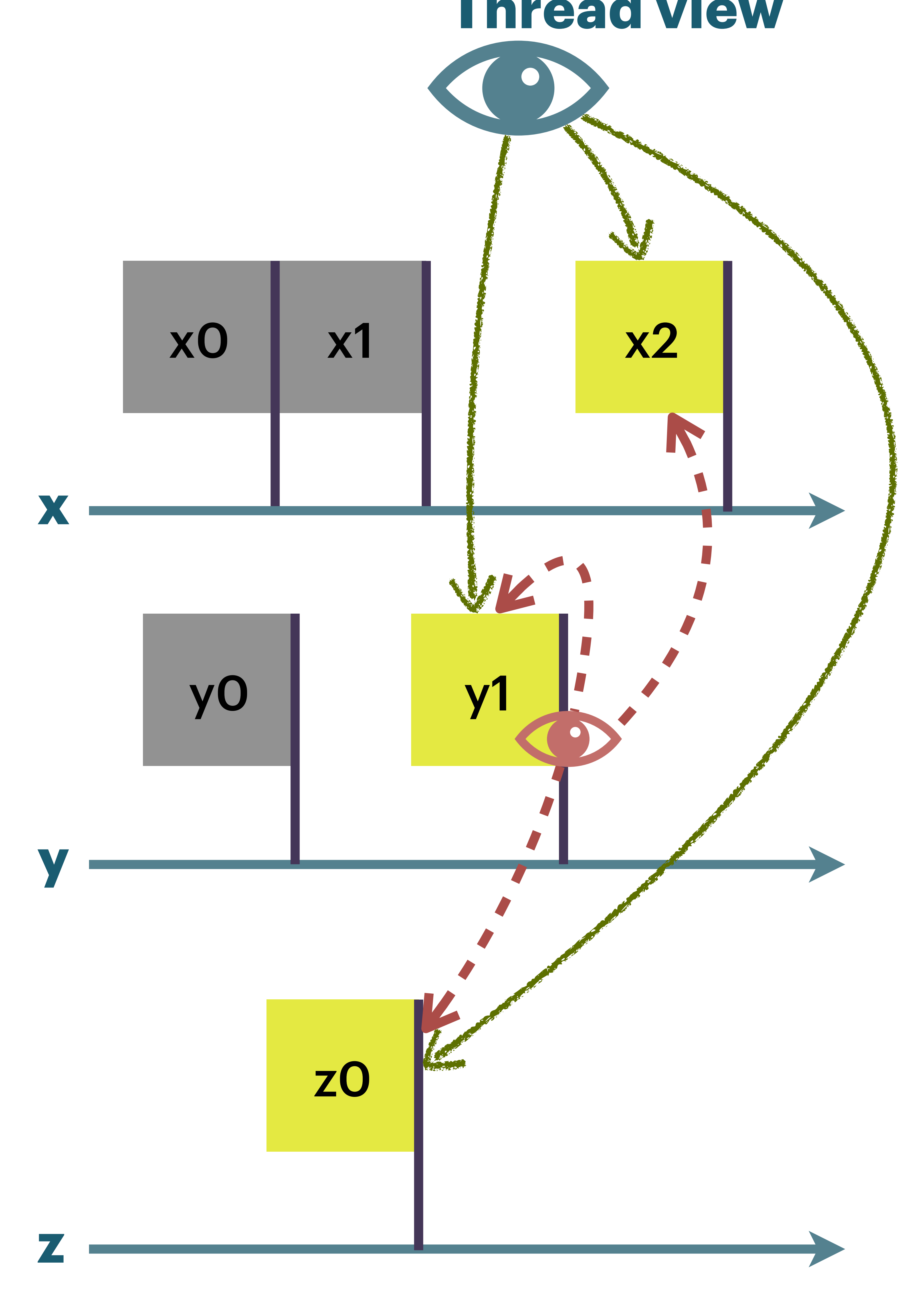
Kang, Hur, L, Vafeiadis, Dreyer: **A promising semantics for relaxed-memory concurrency.** POPL 2017. https://doi.org/10.1145/3009837.3009850

Dvir, Kammar, L: **A Denotational Approach to Release/Acquire Concurrency.** ESOP 2024. https://doi.org/10.1007/978-3-031-57267-8_5

Operational formulation

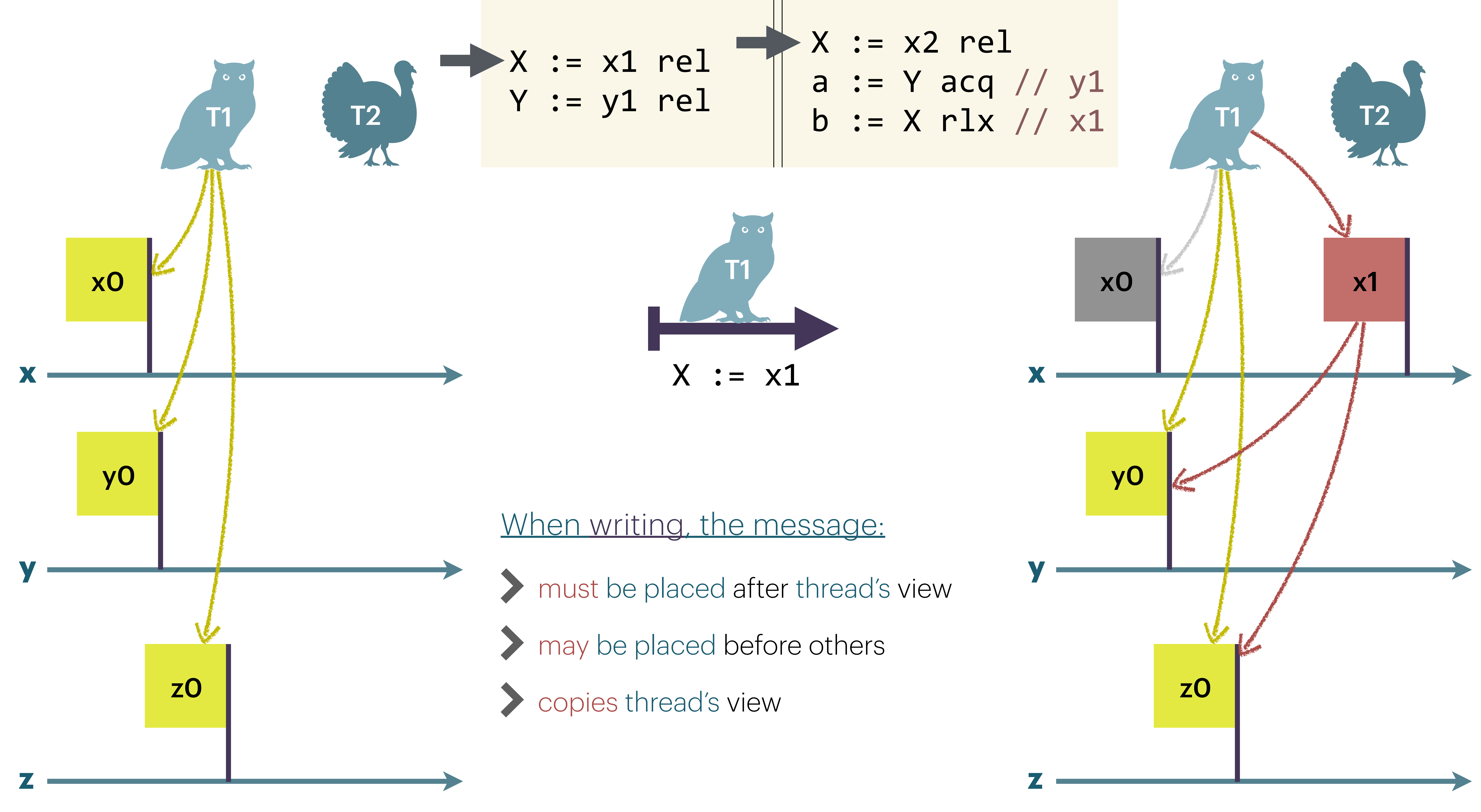
A view-based semantics

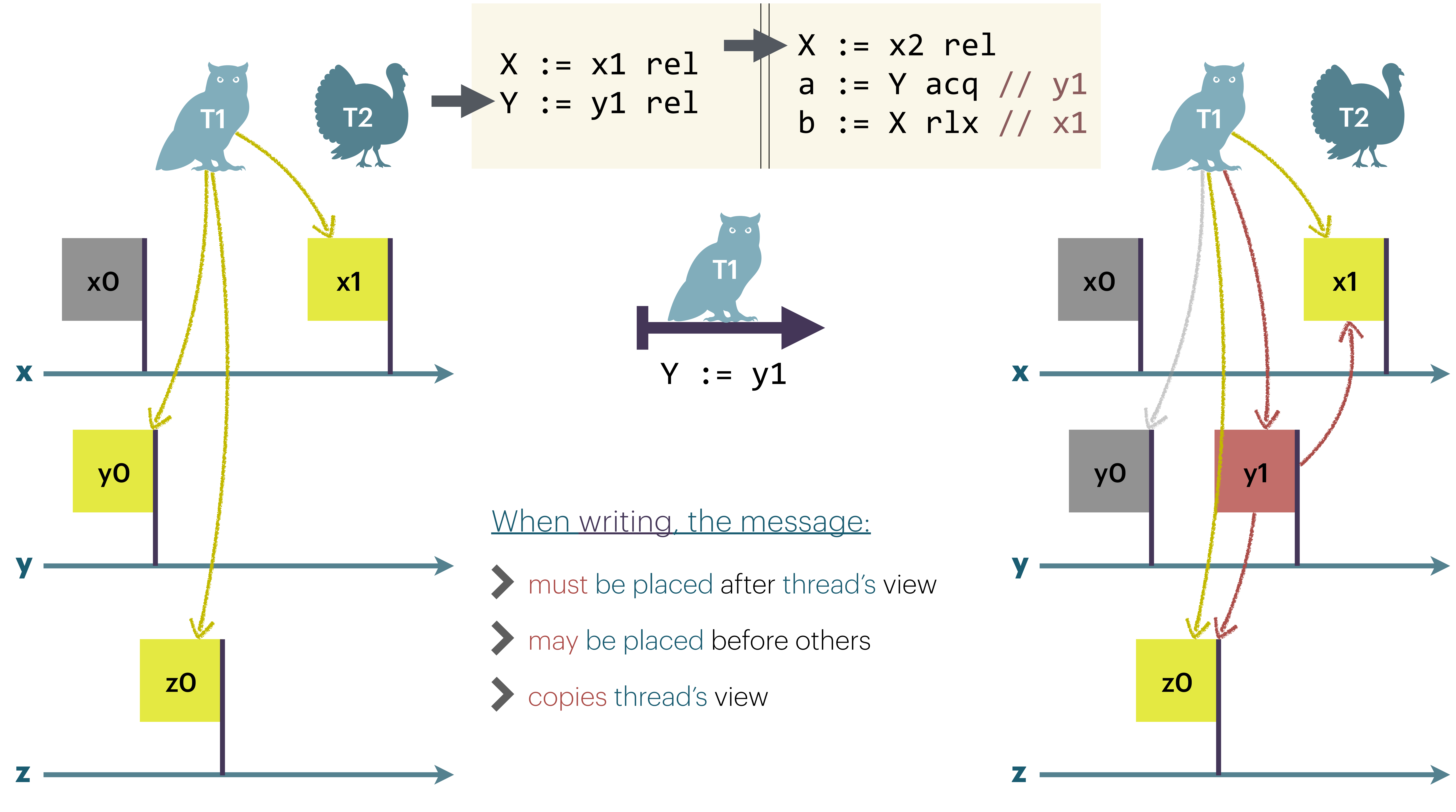
- Memory: Timeline per location (represents *mo*)
- Populated with immutable messages holding values
- Each view points to msgs on each timeline
- Threads have views — cannot read from “the past”
- Msgs have views for enforcing causal propagation
- Simulates the graph-based operational semantics



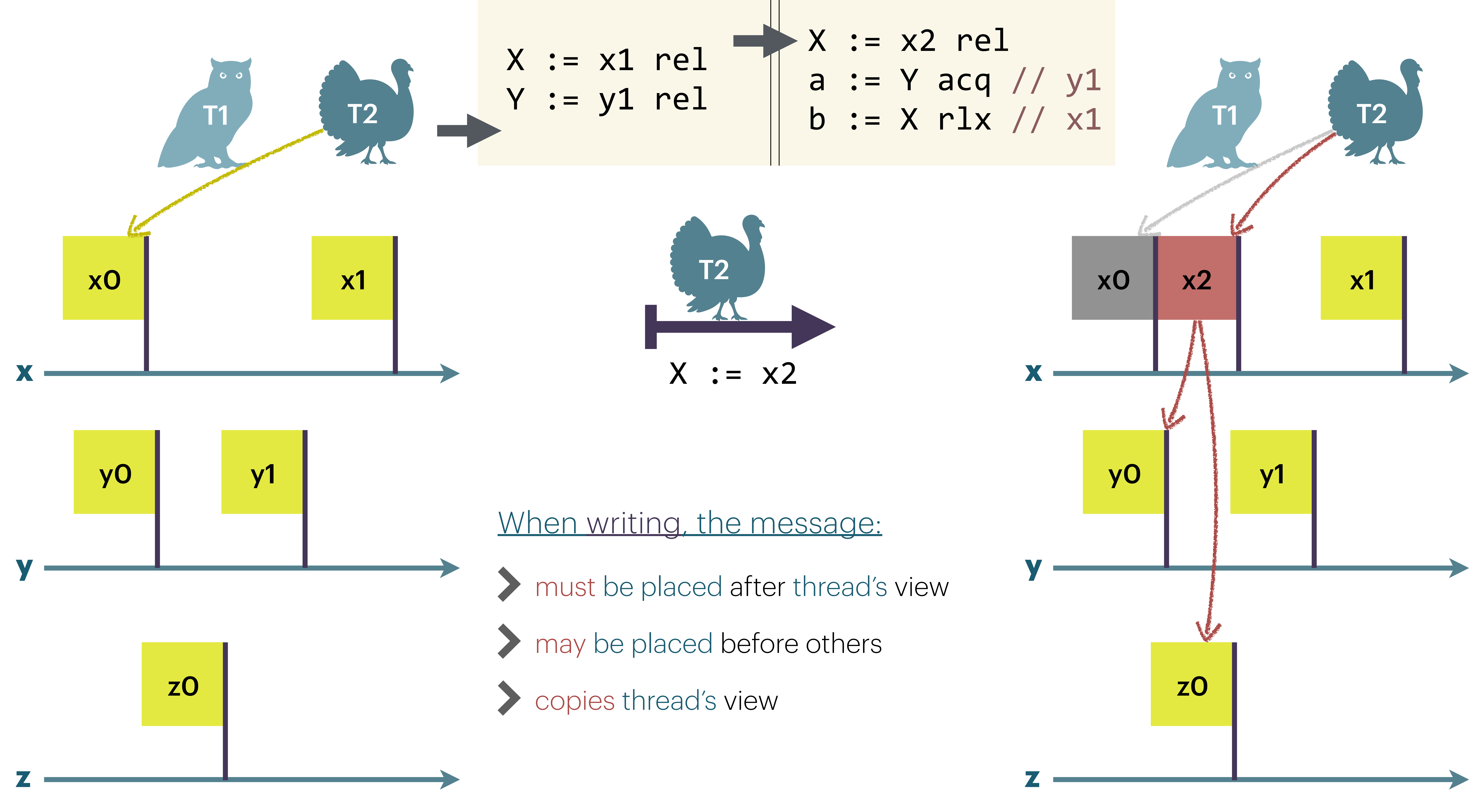
Kang, Hur, L, Vafeiadis, Dreyer: **A promising semantics for relaxed-memory concurrency.** POPL 2017. https://doi.org/10.1145/3009837.3009850

Dvir, Kammar, L: **A Denotational Approach to Release/Acquire Concurrency.** ESOP 2024. https://doi.org/10.1007/978-3-031-57267-8_5

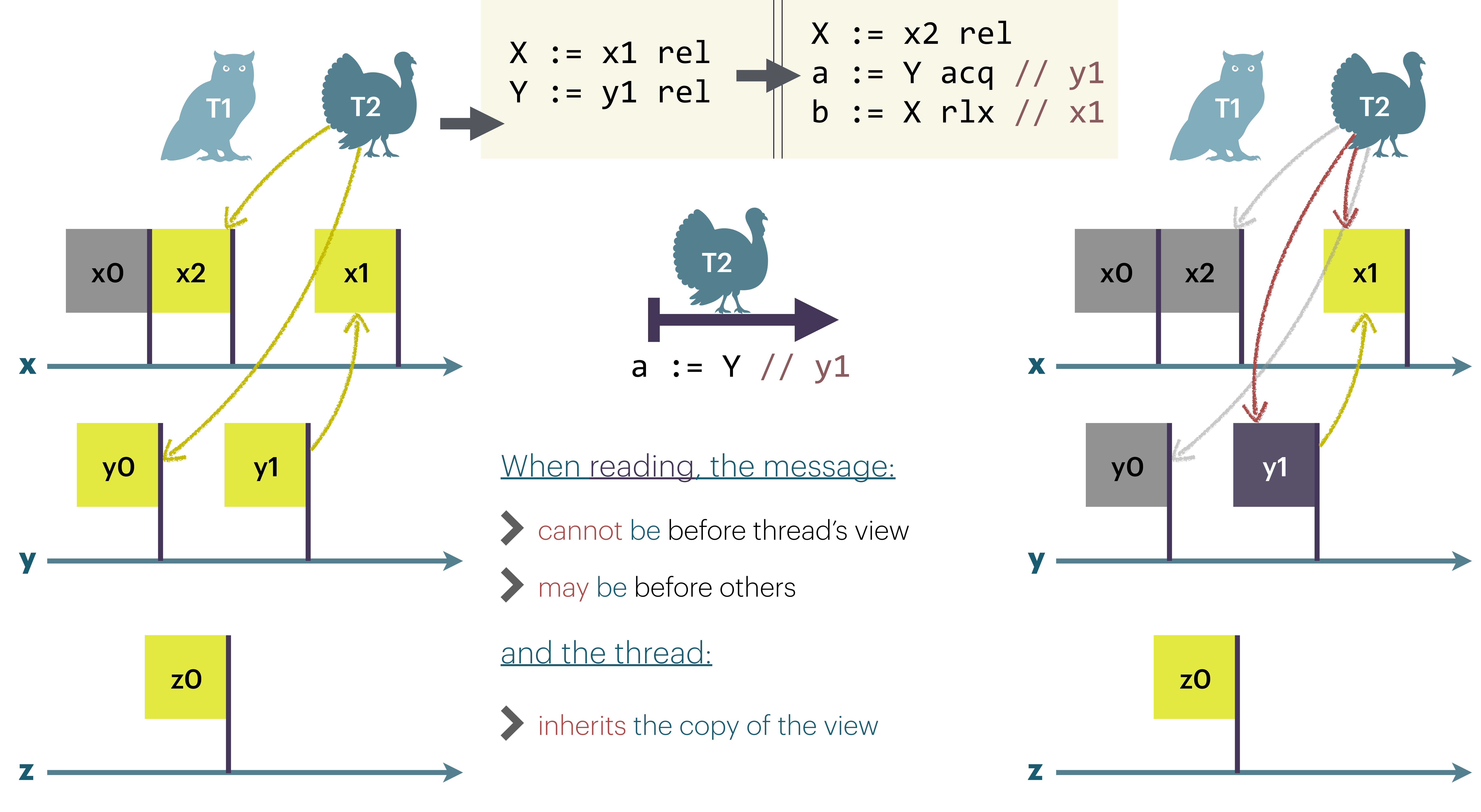

X := x1 rel
Y := y1 rel
X := x2 rel
a := Y acq // y1
b := X rlx // x1
T1
T2
x0
y0
z0
x
y
z
X := x1
x1
When writing, the message:
must be placed after thread's view
may be placed before others
copies thread's view

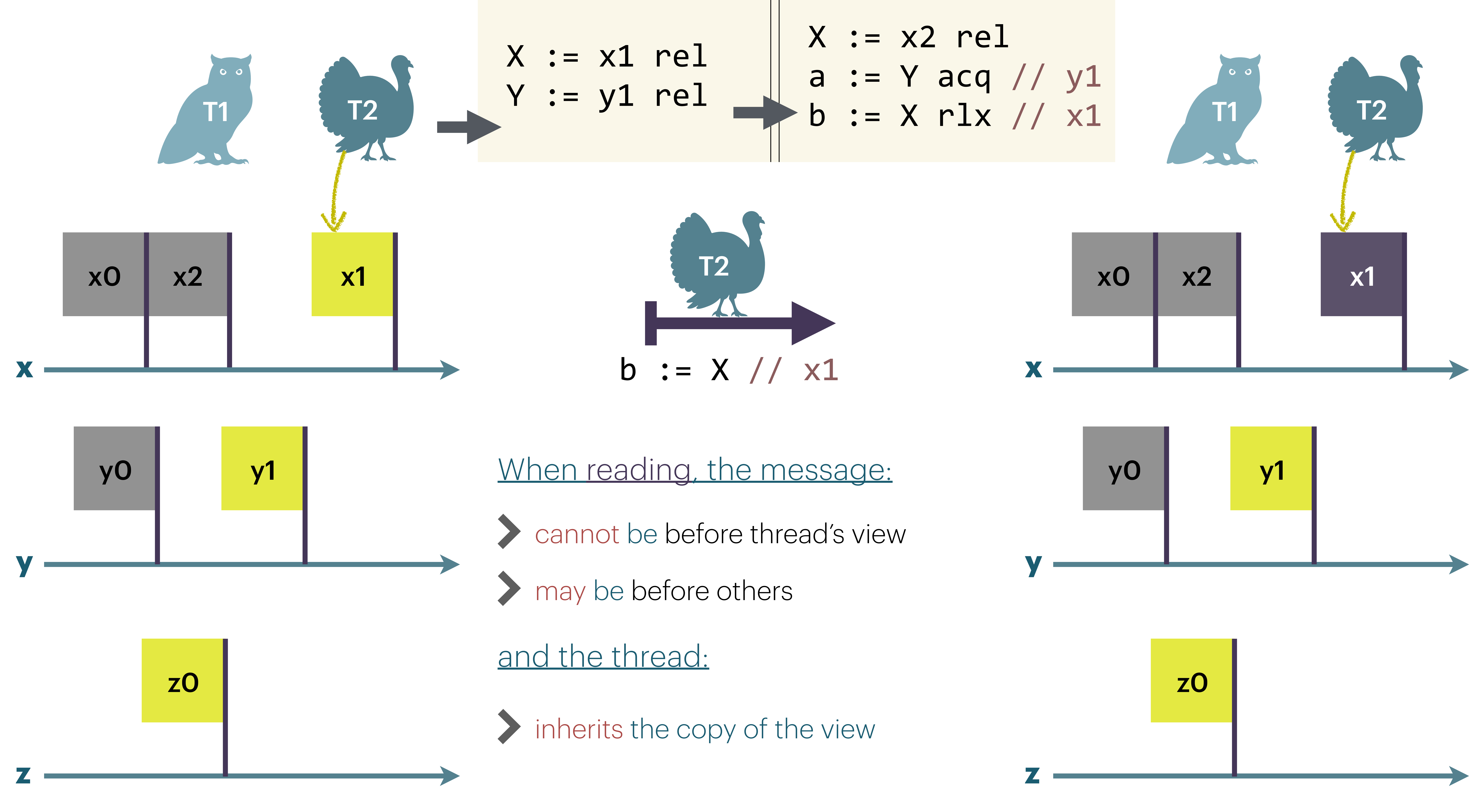


X := x1 rel
Y := y1 rel
X := x2 rel
a := Y acq // y1
b := X rlx // x1
T1
T2
x0
x1
y0
z0
x
y
z
T1
Y := y1
When writing, the message:
must be placed after thread's view
may be placed before others
copies thread's view
T1
T2
x0
x1
y0
y1
z0
x
y
z

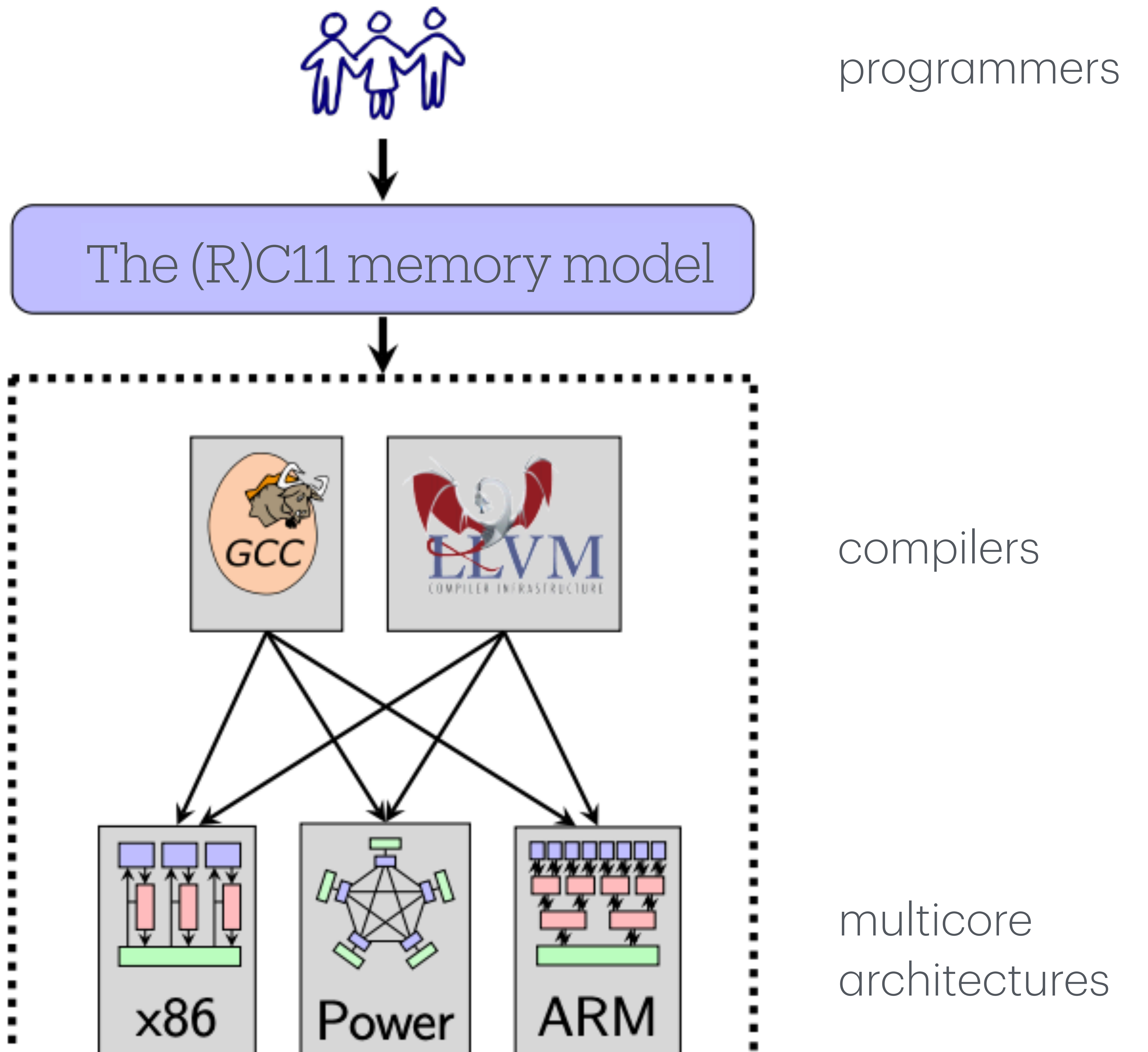


X := x1 rel
Y := y1 rel
X := x2 rel
a := Y acq // y1
b := X rlx // x1
T1
T2
x0
x1
x2
y0
y1
z0
x
y
z
X := x2
When writing, the message:
must be placed after thread's view
may be placed before others
copies thread's view


X := x1 rel
Y := y1 rel
X := x2 rel
a := Y acq // y1
b := X rlx // x1
T1
T2
x0
x2
x1
x
y0
y1
y
z0
z
a := Y // y1
When reading, the message:
cannot be before thread's view
may be before others
and the thread:
inherits the copy of the view


T1
T2
X := x1 rel
Y := y1 rel
X := x2 rel
a := Y acq // y1
b := X rlx // x1
T1
T2
x0
x2
x1
x
T2
b := X // x1
x0
x2
x1
x
y0
y1
y
When reading, the message:
cannot be before thread's view
may be before others
and the thread:
inherits the copy of the view
y0
y1
y
z0
z
z0
z

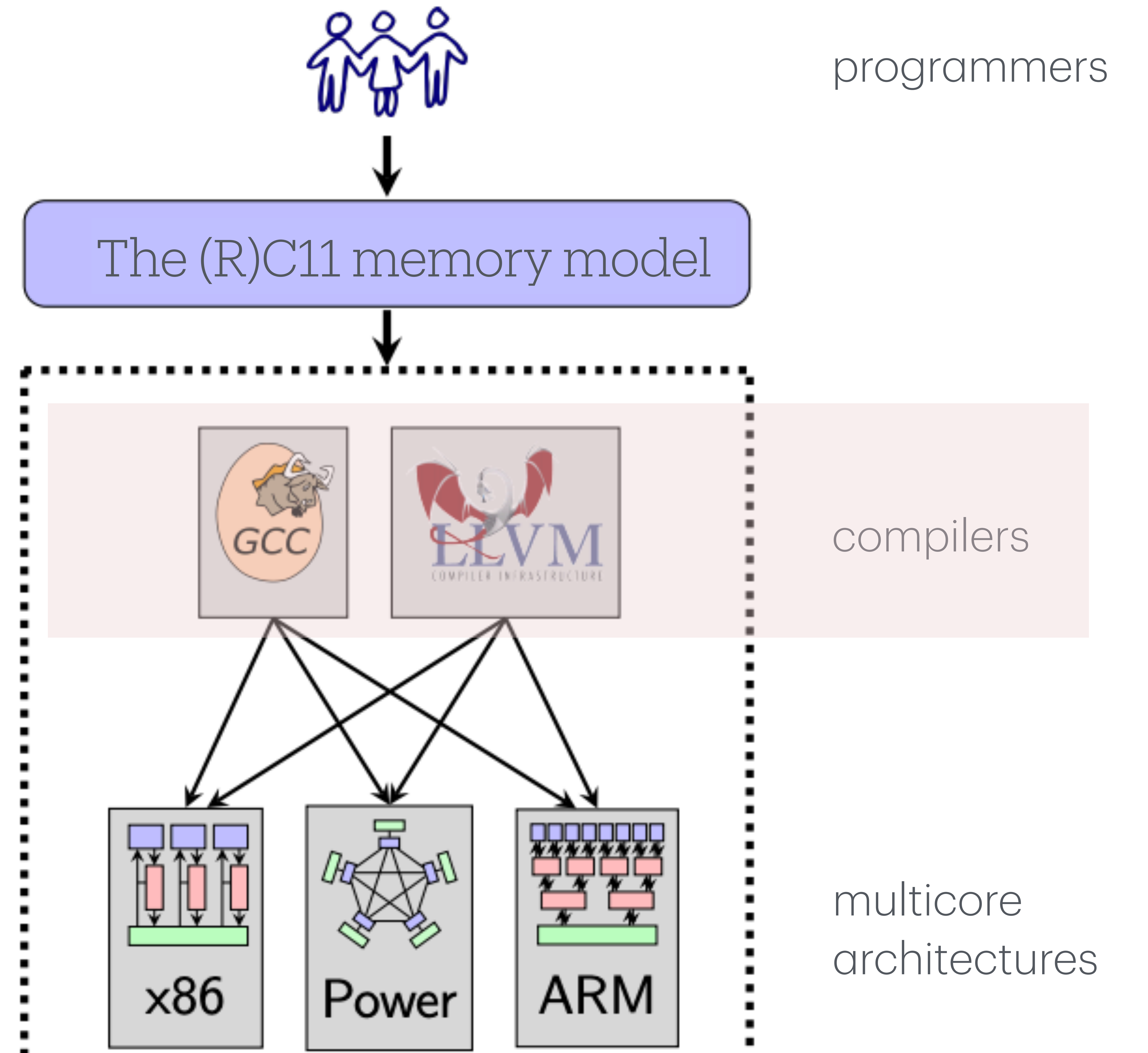
Implementability of (R)C11



programmers

compilers

multicore
architectures



programmers
The (R)C11 memory model
GCC
LLVM
compilers
x86
Power
ARM
multicore
architectures

Compiler optimizations

- A formal analysis of soundness of optimizations in C11:

  Vafeiadis, Balabonski, Chakraborty, Morisset, Zappa Nardelli: **Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it.** POPL 2015. https://doi.org/10.1145/2676726.2676995

- In RC11:

  L, Vafeiadis, Kang, Hur, Dreyer: **Repairing Sequential Consistency in C/C++11.** PLDI 2017. https://doi.org/10.1145/3140587.3062352

- A lot of focus on optimizations on atomics (eliminations and reorderings)

- Current compilers mostly optimize non-atomics

Transformation soundness

- Standard notion of soundness of local program transformations:

  $C_{src} \rightsquigarrow C_{tgt}$ if **Behaviors**$(P[C_{tgt}]) \subseteq$ **Behaviors**$(P[C_{src}])$ for every program context P

- For catch-fire semantics (as (R)C11), it implies:
  - if C_{src} is not racy, then C_{tgt} is not racy (the compiler must not introduce races)
  - if C_{src} is racy, then $C_{src} \rightsquigarrow C_{tgt}$ is sound for every C_{tgt} (the compiler may exploit races)

Allowed eliminations in RC11

$R^o; R^o \rightsquigarrow R^o$

$W^{sc}; R^{sc} \rightsquigarrow W^{sc}$

$W^o; W^o \rightsquigarrow W^o$

$W^o; R^{acq} \rightsquigarrow W^o$

together with access strengthening RC11 allows, e.g.:

WW ⇝ W: `X := 1 rel ; X := 2 rlx` ⇝ `X := 2 rel`

RR ⇝ R: `a := X acq ; b := X rlx` ⇝ `a := X acq ; b := a`

WR ⇝ W: `X := 1 rlx ; a := X acq` ⇝ `X := 1 rlx ; a := 1`

`X := 1 rlx ; a := X sc` ⇝ `X := 1 sc ; a := 1`

Allowed read-write reordering in RC11

thanks to "catch-fire"

`X ; Y ⇝ Y ; X`

X \ Y	$\mathtt{R}_y^{o_2}$	$\mathtt{W}_y^{o_2}$
$\mathtt{R}_x^{o_1}$	$o_1 \sqsubseteq \mathtt{rlx}$	$o_1, o_2 \sqsubseteq \mathtt{rlx} \wedge (o_1 = \mathtt{na} \vee o_2 = \mathtt{na})$
$\mathtt{W}_x^{o_1}$	$o_1 \neq \mathtt{sc} \vee o_2 \neq \mathtt{sc}$	$o_2 \sqsubseteq \mathtt{rlx}$

e.g.,

"roach motel"

RR: `a := X ; b := Y acq ⇝ b := Y acq ; a := X`

WW: `X := 1 rel ; Y := 2 ⇝ Y := 2 ; X := 1 rel`

WR: `X := 1 rel ; b := Y acq ⇝ b := Y acq ; X := 1 rel`

RW: `a := X ; Y := 1 ⇝ Y := 1 ; a := X`

Optimizing non-atomics

Thanks to catch-fire, non-atomics can be generally optimized as sequential code

`lock(L)`	`lock(L)`
`a := X`	`b := Y`
`Y := 1`	`X := b`
`unlock(L)`	`unlock(L)`

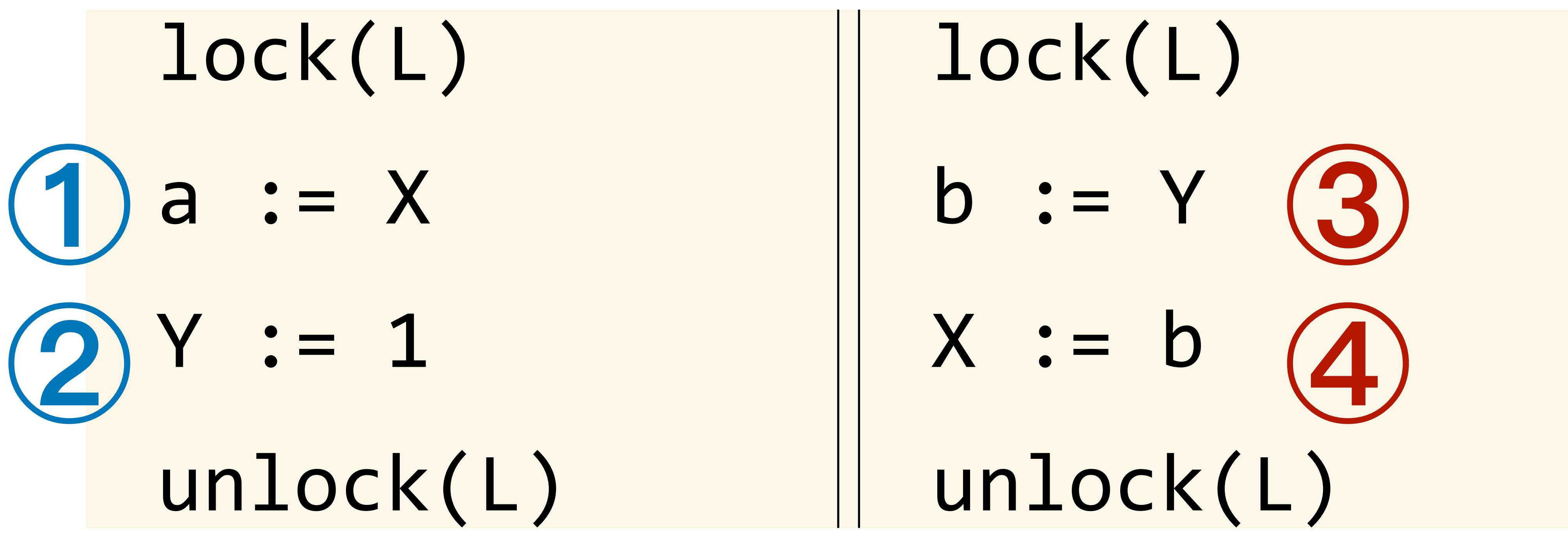
Optimizing non-atomics

Thanks to catch-fire, non-atomics can be generally optimized as sequential code

`lock(L)`	‖	`lock(L)`
① `a := X`	‖	`b := Y` ③
② `Y := 1`	‖	`X := b` ④
`unlock(L)`	‖	`unlock(L)`

Optimizing non-atomics

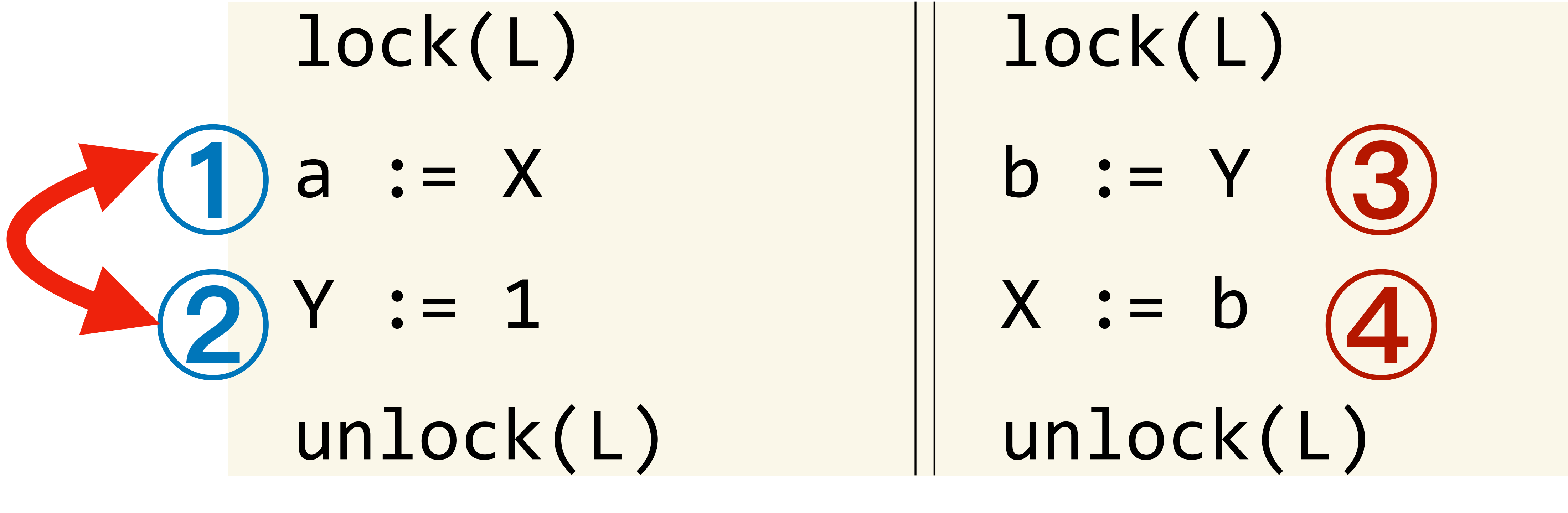
Thanks to catch-fire, non-atomics can be generally optimized as sequential code

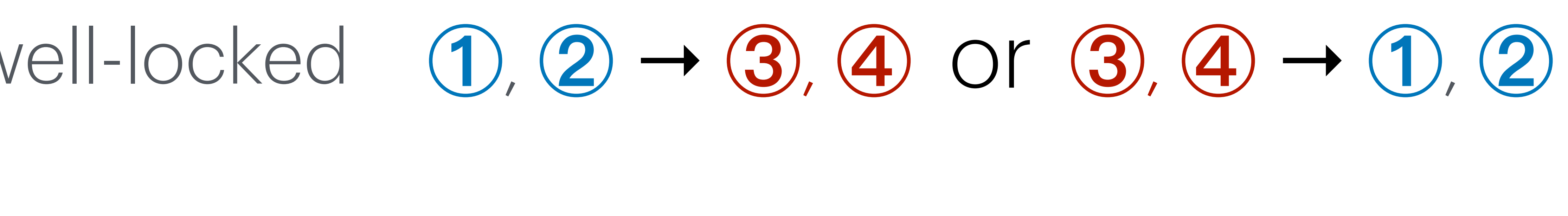


well-locked ①, ② → ③, ④ or ③, ④ → ①, ②

Optimizing non-atomics

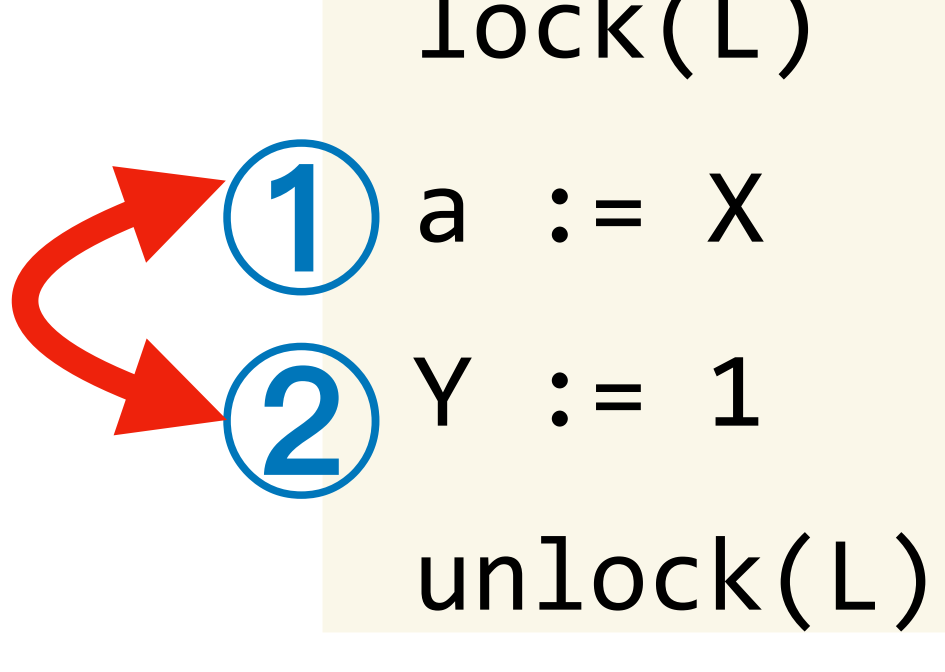
Thanks to catch-fire, non-atomics can be generally optimized as sequential code





Optimizing non-atomics

Thanks to catch-fire, non-atomics can be generally optimized as sequential code

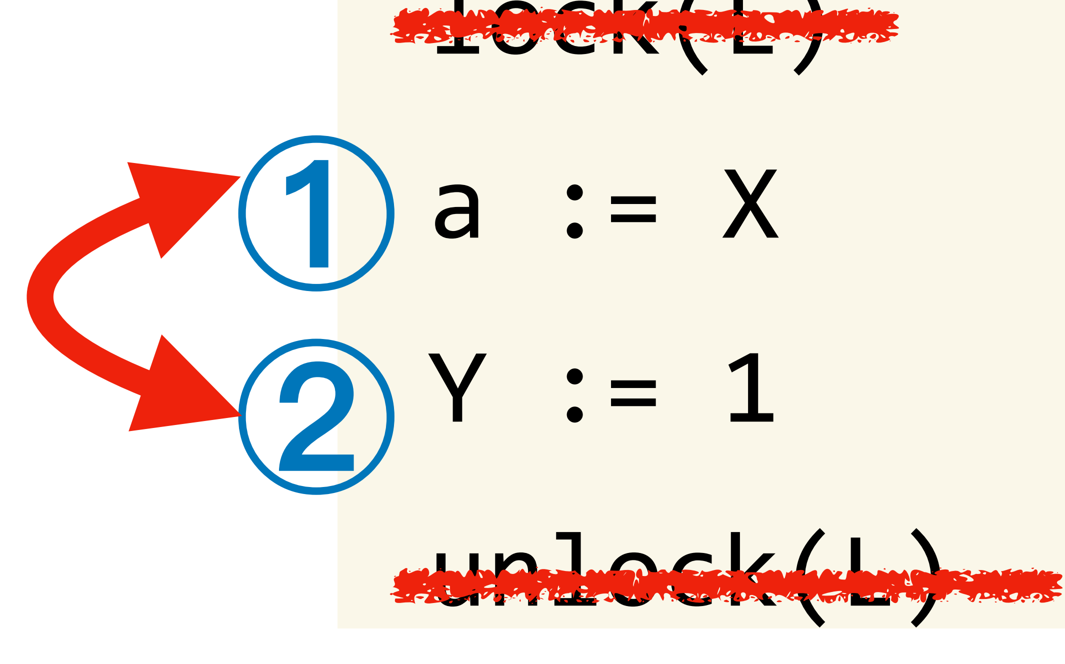


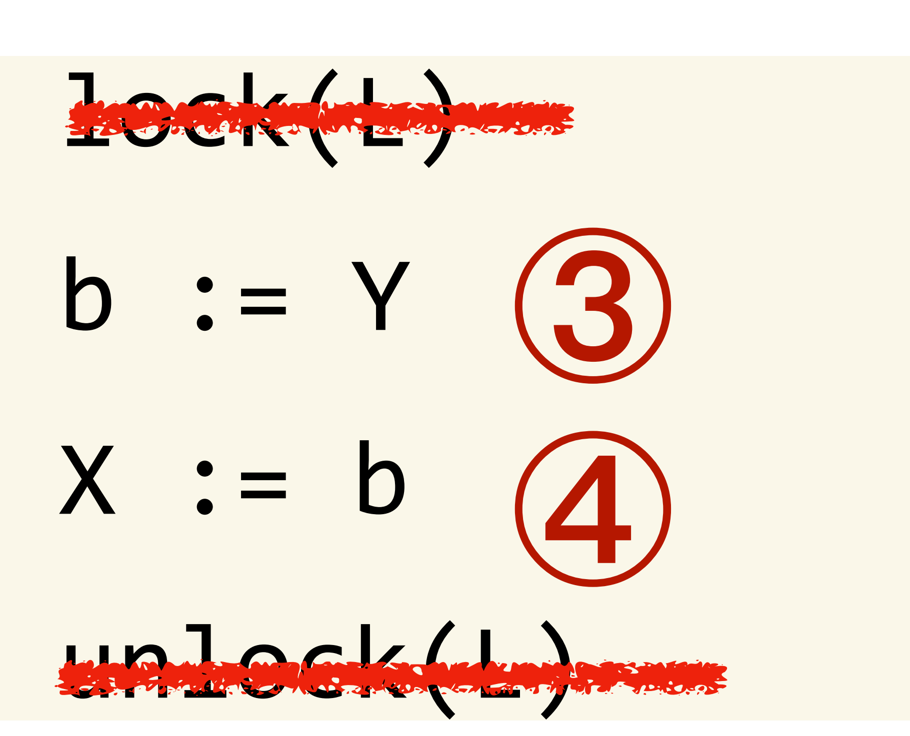
lock(L)	lock(L)
a := X	b := Y ③
Y := 1	X := b ④
unlock(L)	unlock(L)

well-locked ①, ② → ③, ④ or ③, ④ → ①, ②

Optimizing non-atomics

Thanks to catch-fire, non-atomics can be generally optimized as sequential code

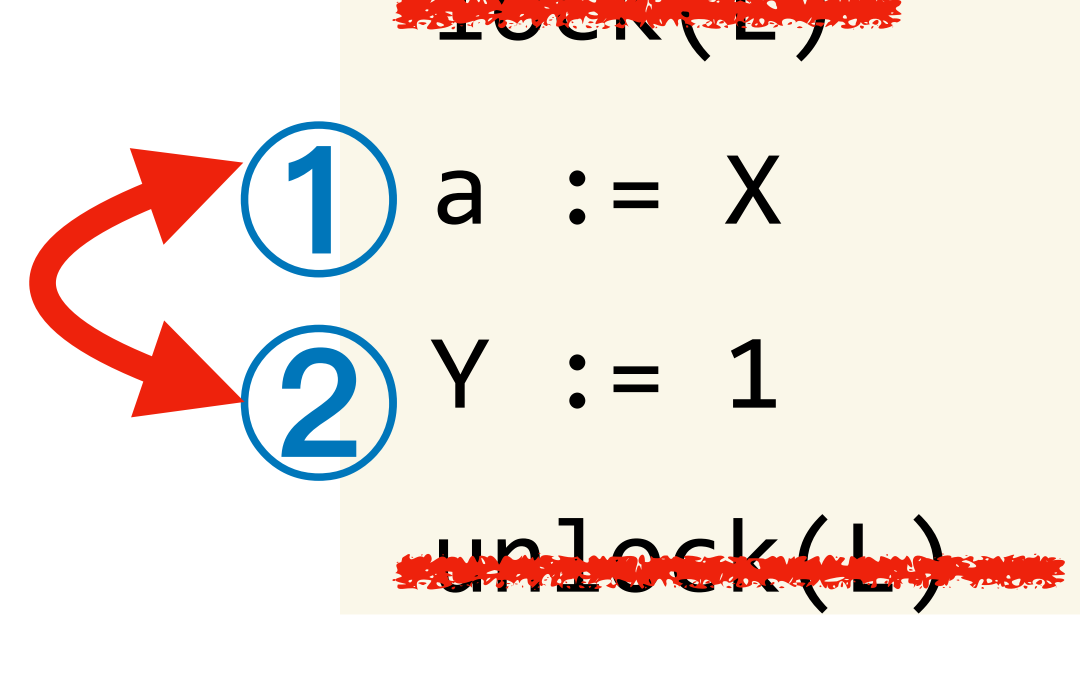




well-locked ①, ② → ③, ④ or ③, ④ → ①, ②

Optimizing non-atomics

Thanks to catch-fire, non-atomics can be generally optimized as sequential code



Thread 1		Thread 2	
	~~lock(L)~~	~~lock(L)~~	
①	a := X	b := Y	③
②	Y := 1	X := b	④
	~~unlock(L)~~	~~unlock(L)~~	

well-locked ①, ② → ③, ④ or ③, ④ → ①, ②

not well-locked

Optimizing non-atomics

Thanks to catch-fire, non-atomics can be generally optimized as sequential code

~~lock(L)~~	~~lock(L)~~
① a := X	b := Y ③
② Y := 1	X := b ④
~~unlock(L)~~	~~unlock(L)~~

well-locked ①, ② → ③, ④ or ③, ④ → ①, ②

Switching ① and ② makes no difference!

not well-locked e.g., ② → ③ → ④ → ①

Switching ① and ② makes a difference, but the program is wrong (racy)!

Load introduction is unsound

- But, irrelevant load introduction is unsound in catch-fire semantics!
- This is something compilers actually perform :(

Load introduction is unsound

- But, irrelevant load introduction is unsound in catch-fire semantics!
- This is something compilers actually perform :(

```
unsigned x, sum = 0;
foo(n, &x);
for (unsigned i = 0; i < n; i++)
  sum += x;
```

Load introduction is unsound

- But, irrelevant load introduction is unsound in catch-fire semantics!
- This is something compilers actually perform :(

```
unsigned x, sum = 0;
foo(n, &x);
for (unsigned i = 0; i < n; i++)
  sum += x;
```

GCC, LLVM

```
unsigned x, sum = 0;
foo(n, &x);

sum = x * n;
```

Load introduction is unsound

- But, irrelevant load introduction is unsound in catch-fire semantics!
- This is something compilers actually perform :(

```
n = 0
```

```
unsigned x, sum = 0;
foo(n, &x);
for (unsigned i = 0; i < n; i++)
  sum += x;
```

GCC, LLVM

```
unsigned x, sum = 0;
foo(n, &x);

sum = x * n;
```

Load introduction is unsound

- But, irrelevant load introduction is unsound in catch-fire semantics!
- This is something compilers actually perform :(

```
n = 0
```

```
unsigned x, sum = 0;
foo(n, &x);
for (unsigned i = 0; i < n; i++)
  sum += x;
```

GCC, LLVM

```
unsigned x, sum = 0;
foo(n, &x);

sum = x * n;
```

A load from x introduced!

Load introduction is unsound

- But, irrelevant load introduction is unsound in catch-fire semantics!
- This is something compilers actually perform :(

`n = 0`

```
unsigned x, sum = 0;
foo(n, &x);
for (unsigned i = 0; i < n; i++)
  sum += x;
```

GCC, LLVM

```
unsigned x, sum = 0;
foo(n, &x);

sum = x * n;
```

`n = 0` →
spawn a thread writing to `x`

A load from `x` introduced!

Load introduction is unsound

- But, irrelevant load introduction is unsound in catch-fire semantics!
- This is something compilers actually perform :(

n = 0

```
unsigned x, sum = 0;
foo(n, &x);
for (unsigned i = 0; i < n; i++)
  sum += x;
```

no data race

n = 0 →
spawn a thread writing to x

GCC, LLVM

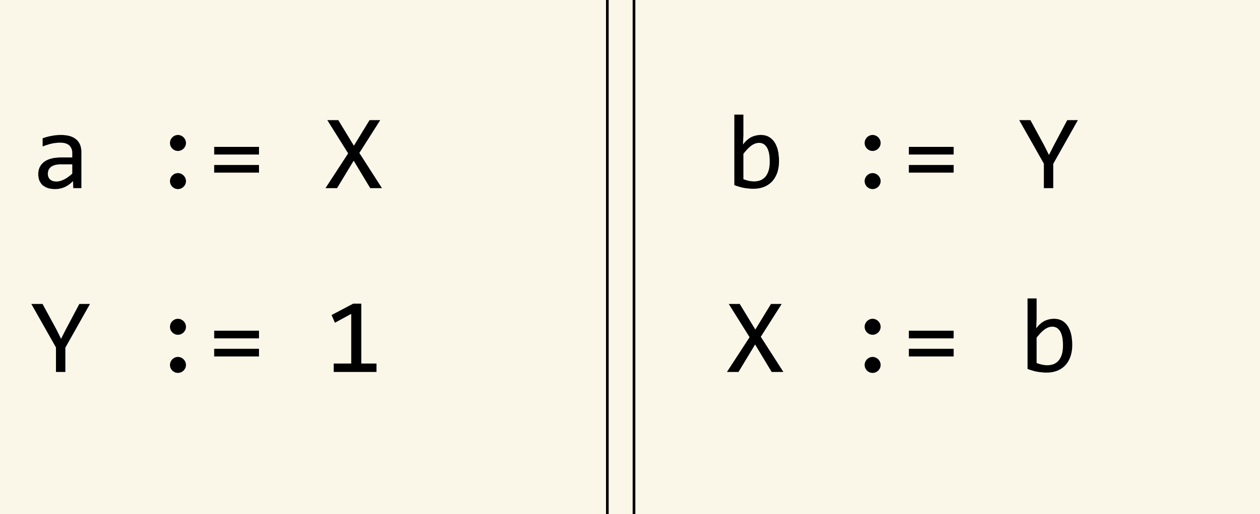
```
unsigned x, sum = 0;
foo(n, &x);

sum = x * n;
```

data race

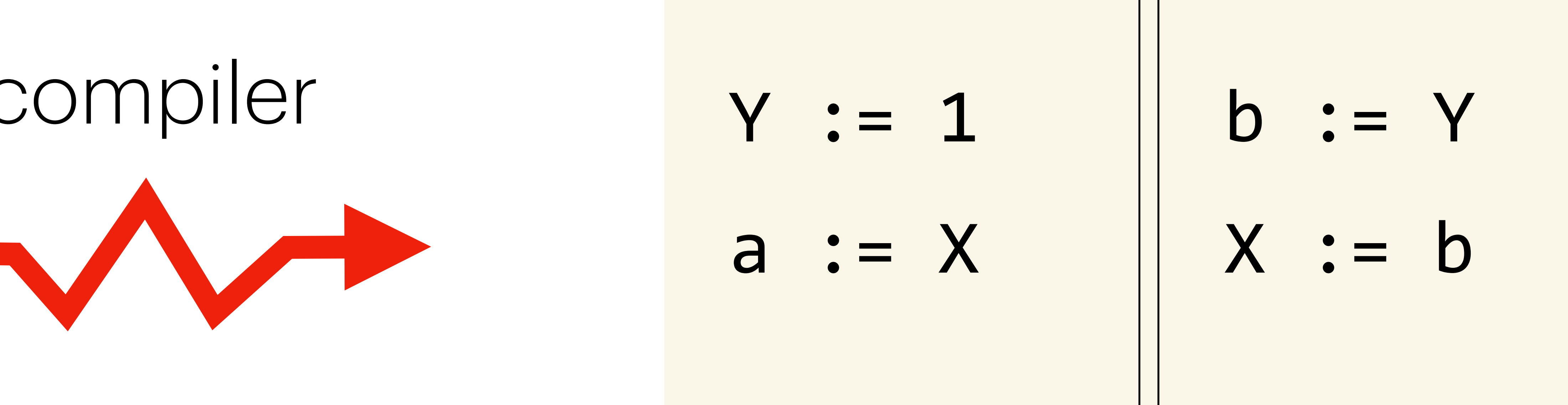
A load from x introduced!

Undefined value as a solution?

- Execute "in-order" and read "undefined value" for every race



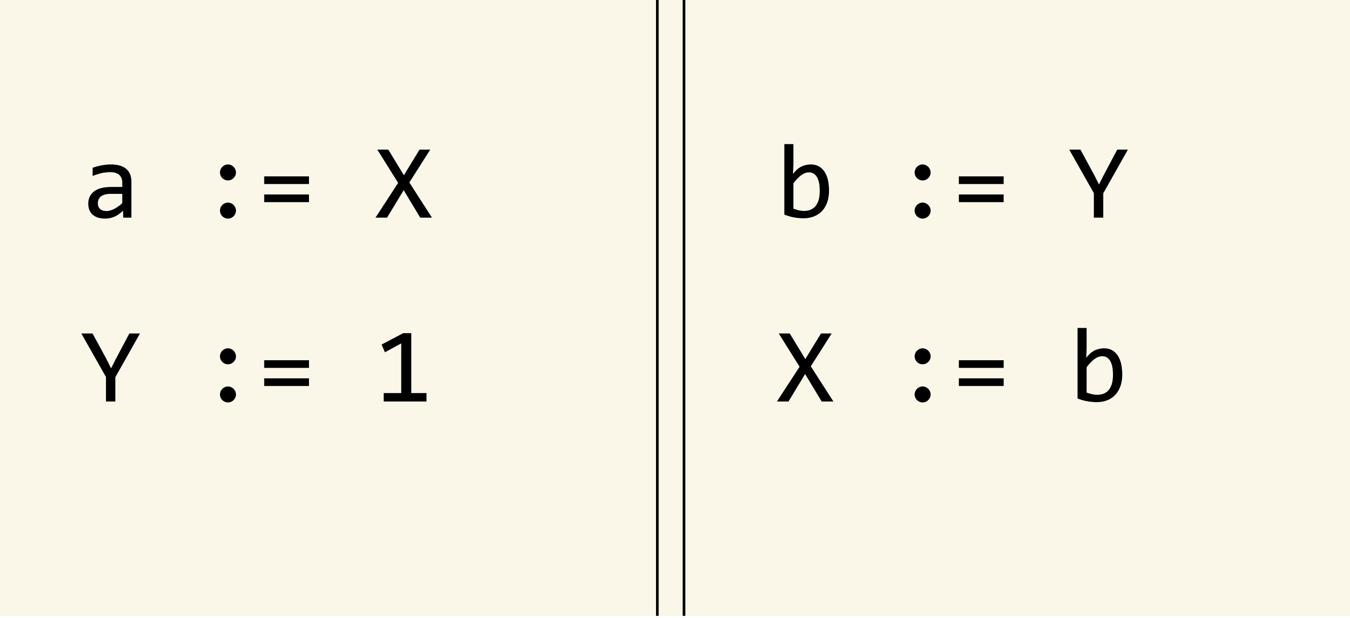
a = b = 1 disallowed

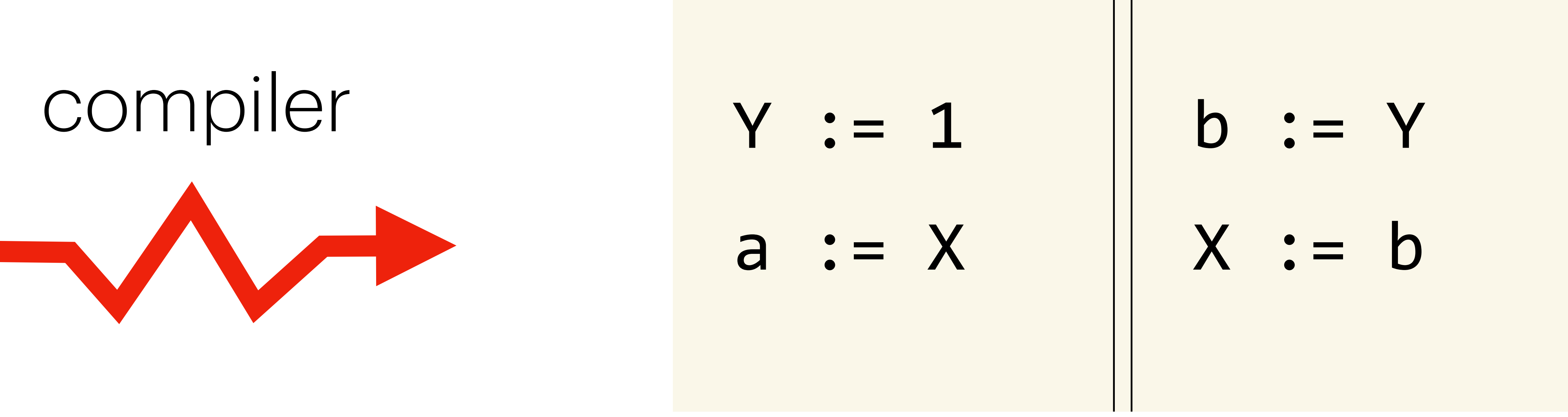


a = b = 1 allowed

Undefined value as a solution?

- Execute “in-order” and read “undefined value” for every race





RW reordering is unsound in this model

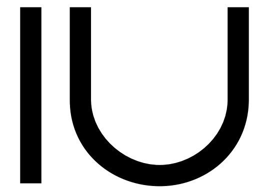
Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297

Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297

source (C/C++) model:
in-order



compiler IR model:
out-of-order

Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297

programming & reasoning

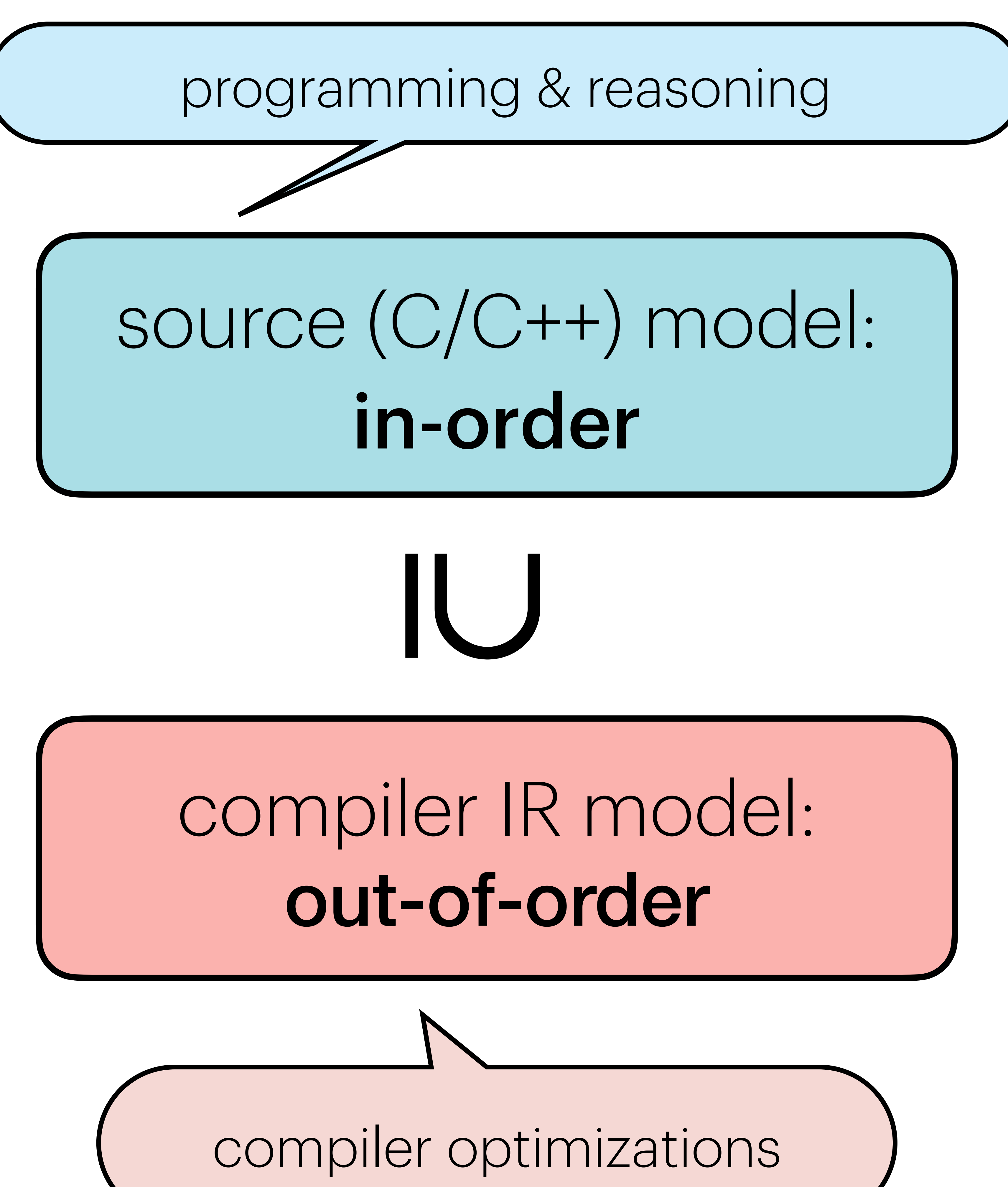
source (C/C++) model:
in-order

$\supseteq$

compiler IR model:
out-of-order

Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297



Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297

programming & reasoning

source (C/C++) model:
in-order

compiler IR model:
out-of-order

compiler optimizations

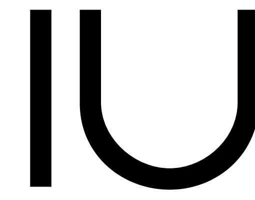
- In-order source model
  - Undefined behavior on WW & WR races
  - Based on RC11

Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297

programming & reasoning

source (C/C++) model:
in-order



compiler IR model:
out-of-order

compiler optimizations

- In-order source model
  - Undefined behavior on WW & WR races
  - Based on RC11
- Out-of-order IR model
  - Undefined behavior on WW races
  - Undefined value on WR races
  - Based on the "promising semantics"

Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297

programming & reasoning

source (C/C++) model:
in-order

compiler IR model:
out-of-order

compiler optimizations

- In-order source model
  - Undefined behavior on WW & WR races
  - Based on RC11
- Out-of-order IR model → RW reordering
  - Undefined behavior on WW races
  - Undefined value on WR races
  - Based on the "promising semantics"

Our Solution: An intermediate memory model

Lee, Cho, Margalit, Hur, L: **Putting Weak Memory in Order via a Promising Intermediate Representation.** PLDI 2023. https://doi.org/10.1145/3591297

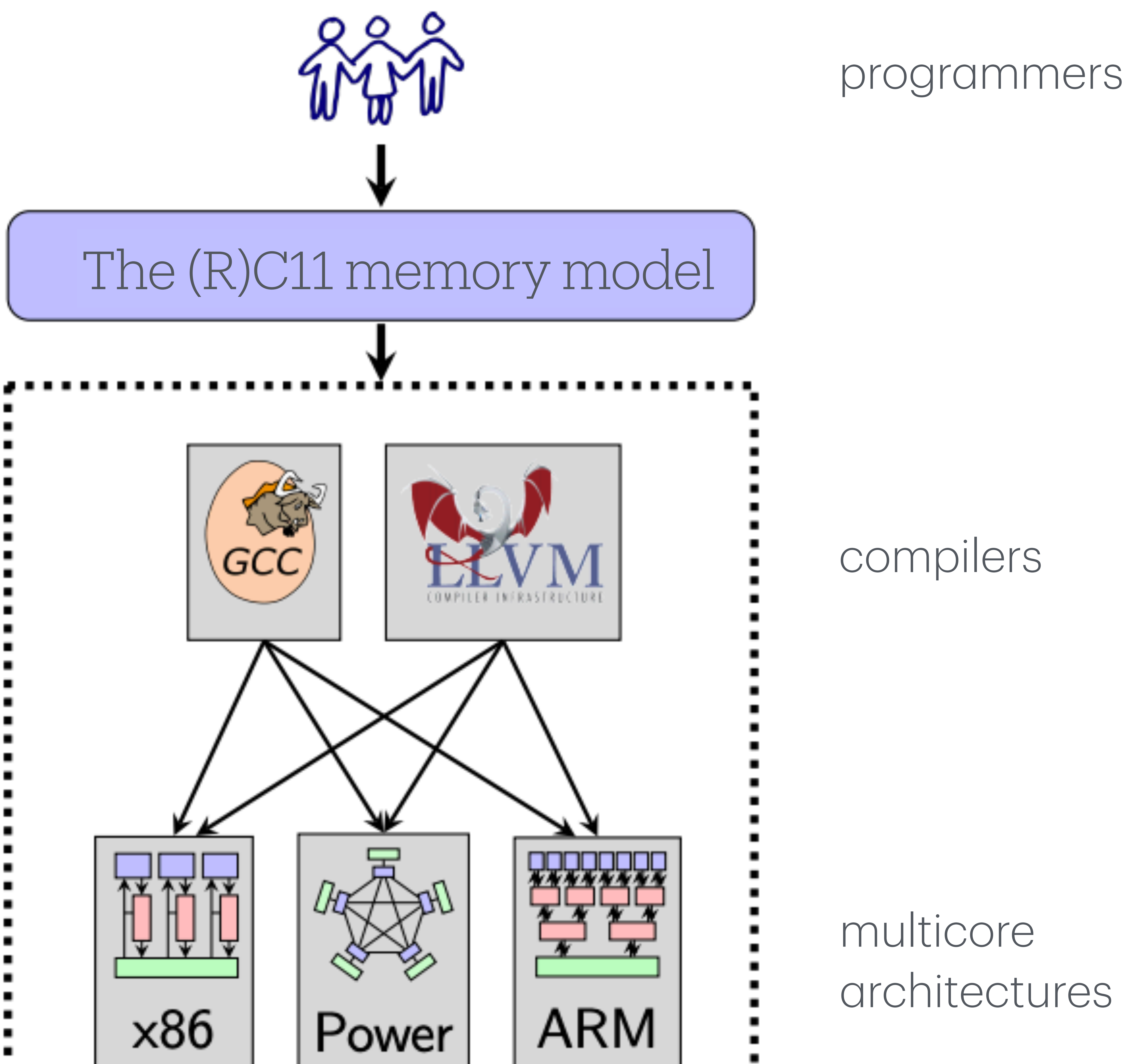
programming & reasoning

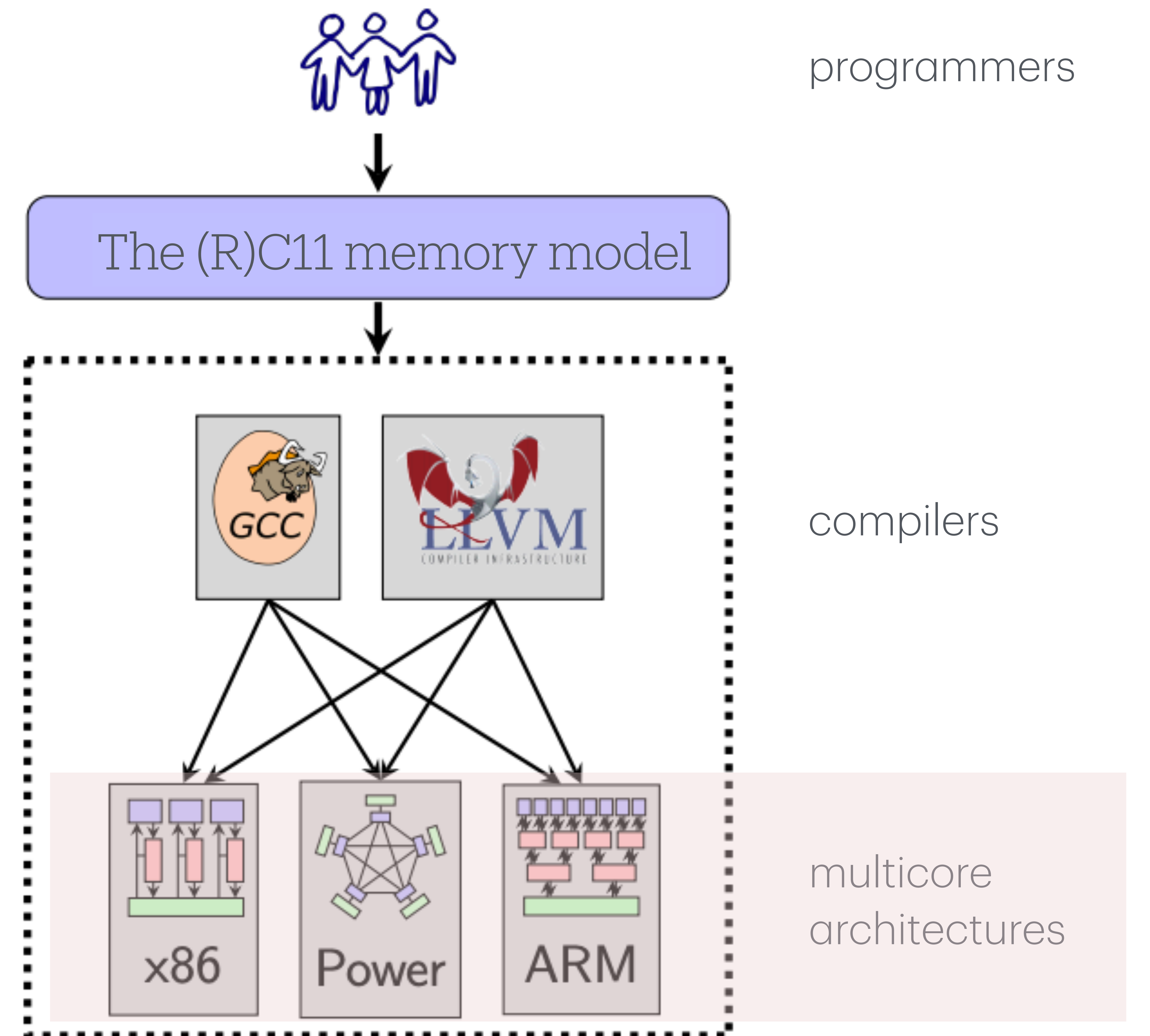
source (C/C++) model:
in-order

compiler IR model:
out-of-order

compiler optimizations

- In-order source model
  - Undefined behavior on WW & WR races
  - Based on RC11
- Out-of-order IR model (→ RW reordering)
  - Undefined behavior on WW races
  - Undefined value on WR races (→ load introduction)
  - Based on the "promising semantics"


programmers
The (R)C11 memory model
compilers
GCC
LLVM
COMPILER INFRASTRUCTURE
multicore
architectures
x86
Power
ARM



programmers

compilers

multicore
architectures

Implementability on multicore hardware

https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

$(\!\vert \mathtt{R} \vert\!)$	$\triangleq$ MOV (from memory)	$(\!\vert \mathtt{W}^{\sqsubseteq \mathtt{rel}} \vert\!)$	$\triangleq$ MOV (to memory)
$(\!\vert \mathtt{W}^{\mathtt{sc}} \vert\!)$	$\triangleq$ MOV;MFENCE	$(\!\vert \mathtt{RMW} \vert\!)$	$\triangleq$ CMPXCHG
$(\!\vert \mathtt{F}^{\neq \mathtt{sc}} \vert\!)$	$\triangleq$ No operation	$(\!\vert \mathtt{F}^{\mathtt{sc}} \vert\!)$	$\triangleq$ MFENCE

Figure 8. Compilation to TSO.

$(\!\vert \mathtt{R}^{\mathtt{na}} \vert\!)$	$\triangleq$ ld	$(\!\vert \mathtt{W}^{\mathtt{na}} \vert\!)$	$\triangleq$ st
$(\!\vert \mathtt{R}^{\mathtt{rlx}} \vert\!)$	$\triangleq$ ld;c	$(\!\vert \mathtt{W}^{\mathtt{rlx}} \vert\!)$	$\triangleq$ st
$(\!\vert \mathtt{R}^{\mathtt{acq}} \vert\!)$	$\triangleq$ ld;cmp;bc;isync	$(\!\vert \mathtt{W}^{\mathtt{rel}} \vert\!)$	$\triangleq$ lwsync;st
$(\!\vert \mathtt{F}^{\neq \mathtt{sc}} \vert\!)$	$\triangleq$ lwsync	$(\!\vert \mathtt{F}^{\mathtt{sc}} \vert\!)$	$\triangleq$ sync
$(\!\vert \mathtt{RMW}^{\mathtt{rlx}} \vert\!)$	$\triangleq$ L:lwarx;cmp;bc Le;stwcx.;bc L;Le:		
$(\!\vert \mathtt{RMW}^{\mathtt{acq}} \vert\!)$	$\triangleq$ $(\!\vert \mathtt{RMW}^{\mathtt{rlx}} \vert\!)$;isync		
$(\!\vert \mathtt{RMW}^{\mathtt{rel}} \vert\!)$	$\triangleq$ lwsync;$(\!\vert \mathtt{RMW}^{\mathtt{rlx}} \vert\!)$		
$(\!\vert \mathtt{RMW}^{\mathtt{acqrel}} \vert\!)$	$\triangleq$ lwsync;$(\!\vert \mathtt{RMW}^{\mathtt{rlx}} \vert\!)$;isync		

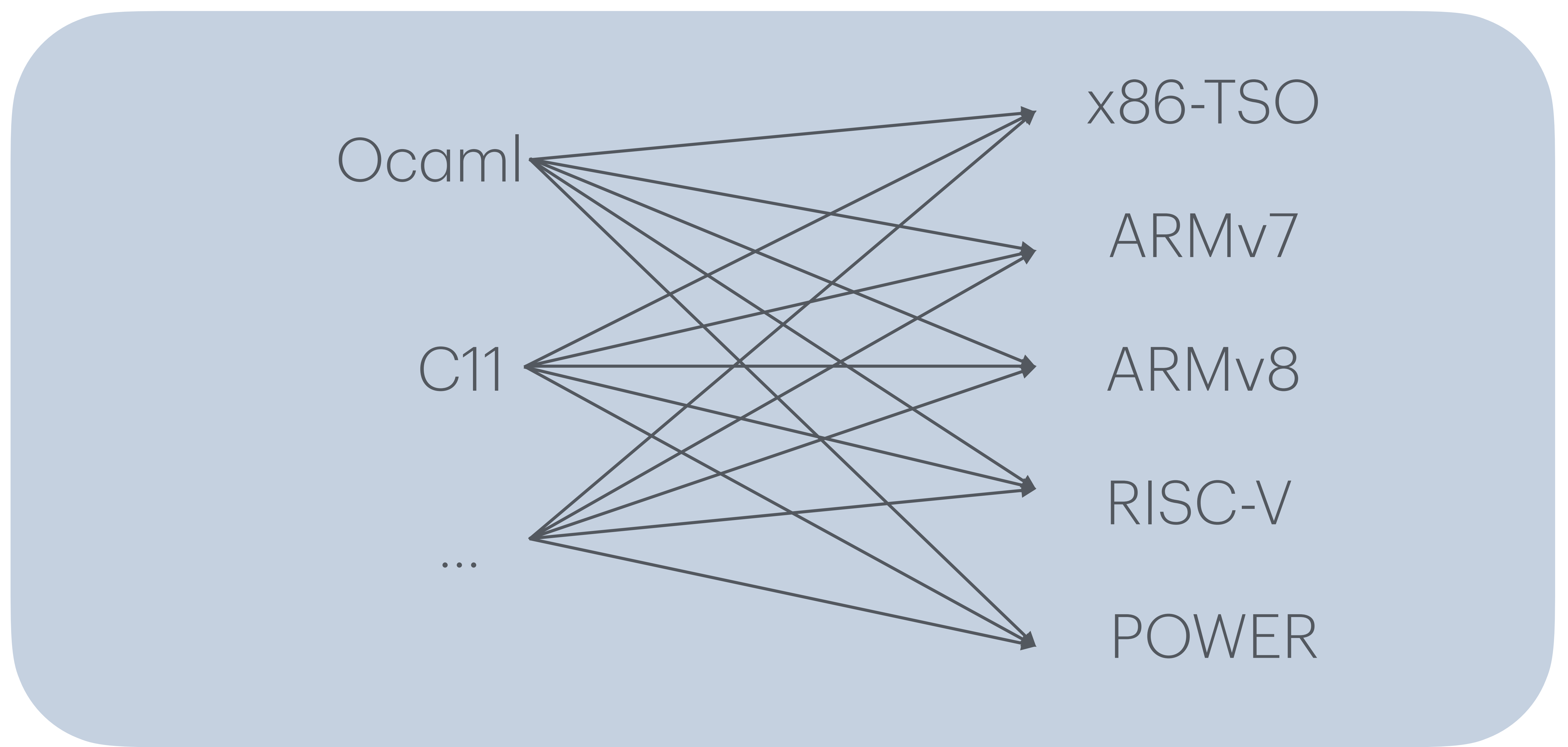
Figure 9. Compilation of non-SC primitives to Power.

Leading sync		Trailing sync	
$(\!\vert \mathtt{R}^{\mathtt{sc}} \vert\!)$	$\triangleq$ sync;$(\!\vert \mathtt{R}^{\mathtt{acq}} \vert\!)$	$(\!\vert \mathtt{R}^{\mathtt{sc}} \vert\!)$	$\triangleq$ ld;sync
$(\!\vert \mathtt{W}^{\mathtt{sc}} \vert\!)$	$\triangleq$ sync;st	$(\!\vert \mathtt{W}^{\mathtt{sc}} \vert\!)$	$\triangleq$ $(\!\vert \mathtt{W}^{\mathtt{rel}} \vert\!)$;sync
$(\!\vert \mathtt{RMW}^{\mathtt{sc}} \vert\!)$	$\triangleq$ sync;$(\!\vert \mathtt{RMW}^{\mathtt{acq}} \vert\!)$	$(\!\vert \mathtt{RMW}^{\mathtt{sc}} \vert\!)$	$\triangleq$ $(\!\vert \mathtt{RMW}^{\mathtt{rel}} \vert\!)$;sync

Figure 10. Compilations of SC accesses to Power.

Mapping correctness

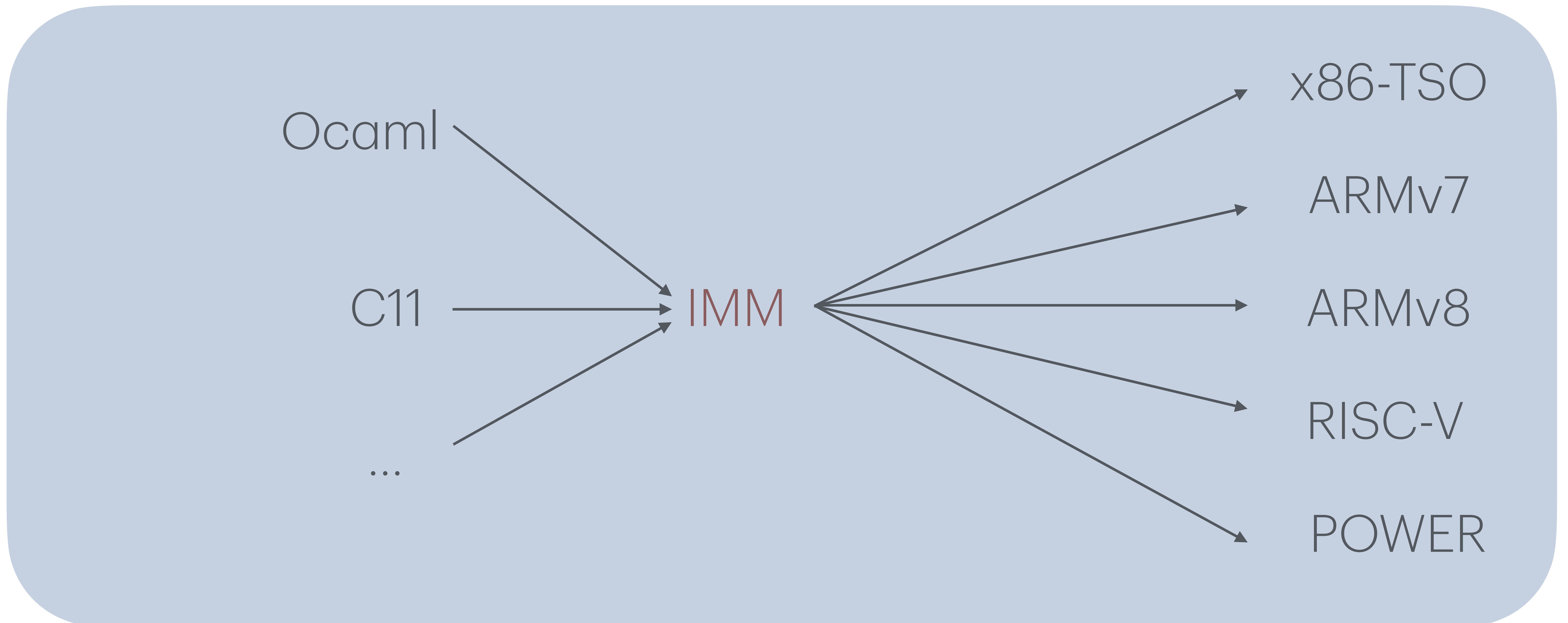
- $(\!| C |\!)$: mapping a program C to a given hardware
- A mapping is *correct* if $\text{Behaviors}_{PL}(C) \subseteq \text{Behaviors}_{Hardware}((\!| C |\!))$



An intermediate memory model

Podkopaev, L, Vafeiadis: **Bridging the gap between programming languages and hardware weak memory models.** POPL 2019. https://doi.org/10.1145/3290382

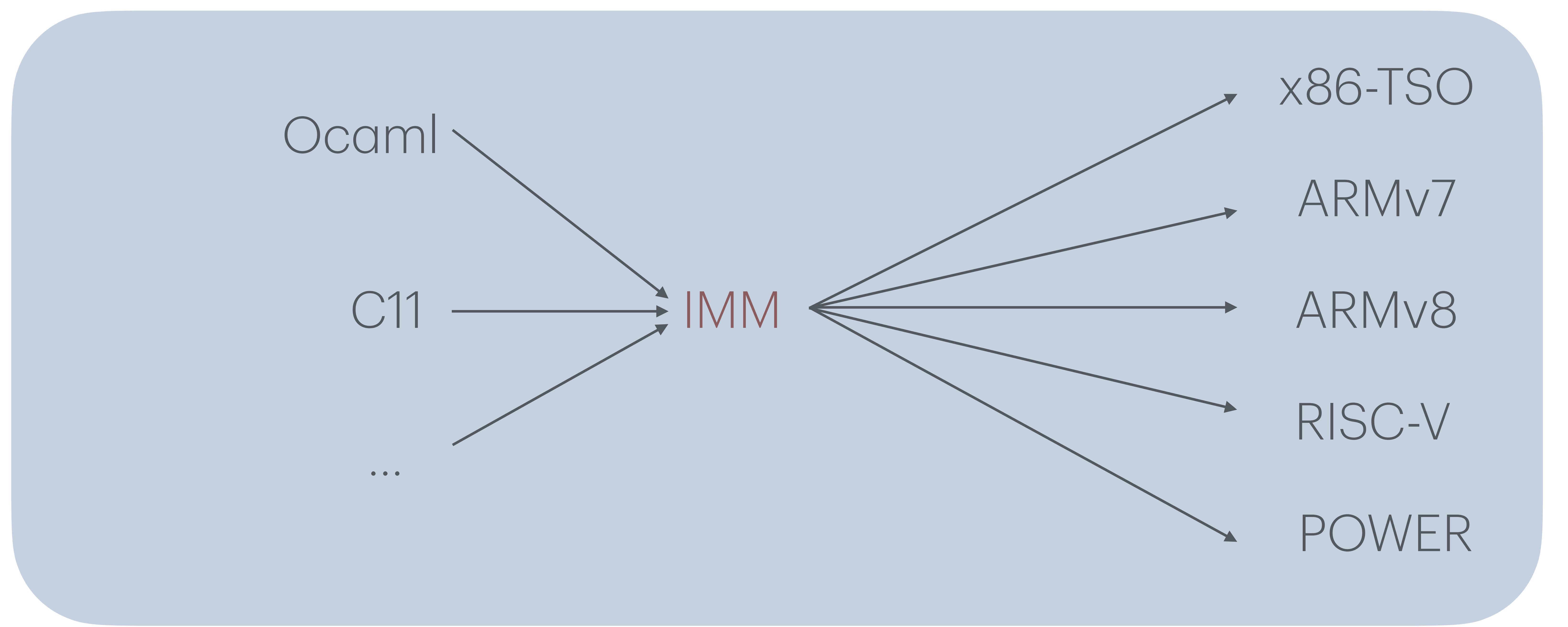
- IMM model as a common denominator of existing hardware weak memory models



An intermediate memory model

Podkopaev, L, Vafeiadis: **Bridging the gap between programming languages and hardware weak memory models.** POPL 2019. https://doi.org/10.1145/3290382

- IMM model as a common denominator of existing hardware weak memory models



Programming guarantees

- Data-race-freedom (DRF) theorems
- Library abstraction

Motivation for the DRF guarantee

- Weak memory models are complex
  - most programmers do not understand the underlying model

- We would like to provide a defensive programming discipline for non-experts:
  - ensures strong and more intuitive semantics
  - can be followed without understanding the full underlying weak memory model

- This was a main design goal for C11, let’s make it more formal…

The DRF guarantee

First attempt

```
X := 1
lock(L)
Y := 1
unlock(L)
```

```
lock(L)
a := Y
unlock(L)
if (a=1) then
  b := X
```

If a program P satisfies:

- has only non-atomics and locks
- is race-free

Then:

- P has only SC behaviors

no consistent execution graph
has conflicting events unordered by hb

The DRF guarantee

First attempt

```
X := 1
lock(L)
Y := 1
unlock(L)
```

```
lock(L)
a := Y
unlock(L)
if (a=1) then
  b := X
```

If a program P satisfies:

- has only non-atomics and locks
- is race-free

Then:

- P has only SC behaviors

no consistent execution graph
has conflicting events unordered by hb

- Is this good enough?

The DRF guarantee

First attempt

```
X := 1              ||   lock(L)
lock(L)             ||   a := Y
Y := 1              ||   unlock(L)
unlock(L)           ||   if (a=1) then
                    ||     b := X
```

If a program P satisfies:

- has only non-atomics and locks
- is race-free

Then:

- P has only SC behaviors

no consistent execution graph
has conflicting events unordered by *hb*

- Is this good enough?
  - Definition of races still requires to understand execution graphs, consistency, *hb*...

The DRF guarantee

First attempt

```
X := 1
lock(L)
Y := 1
unlock(L)
```

||

```
lock(L)
a := Y
unlock(L)
if (a=1) then
  b := X
```

If a program P satisfies:

- has only non-atomics and locks
- is race-free

no consistent execution graph
has conflicting events unordered by hb

Then:

- P has only SC behaviors

- Is this good enough?
  - Definition of races still requires to understand execution graphs, consistency, hb...
  - We also want to allow the use of atomics for avoiding races

The DRF guarantee

Second attempt

```
X := 1
lock(L)
Y := 1
unlock(L)
```

```
lock(L)
a := Y
unlock(L)
if (a=1) then
  b := X
```

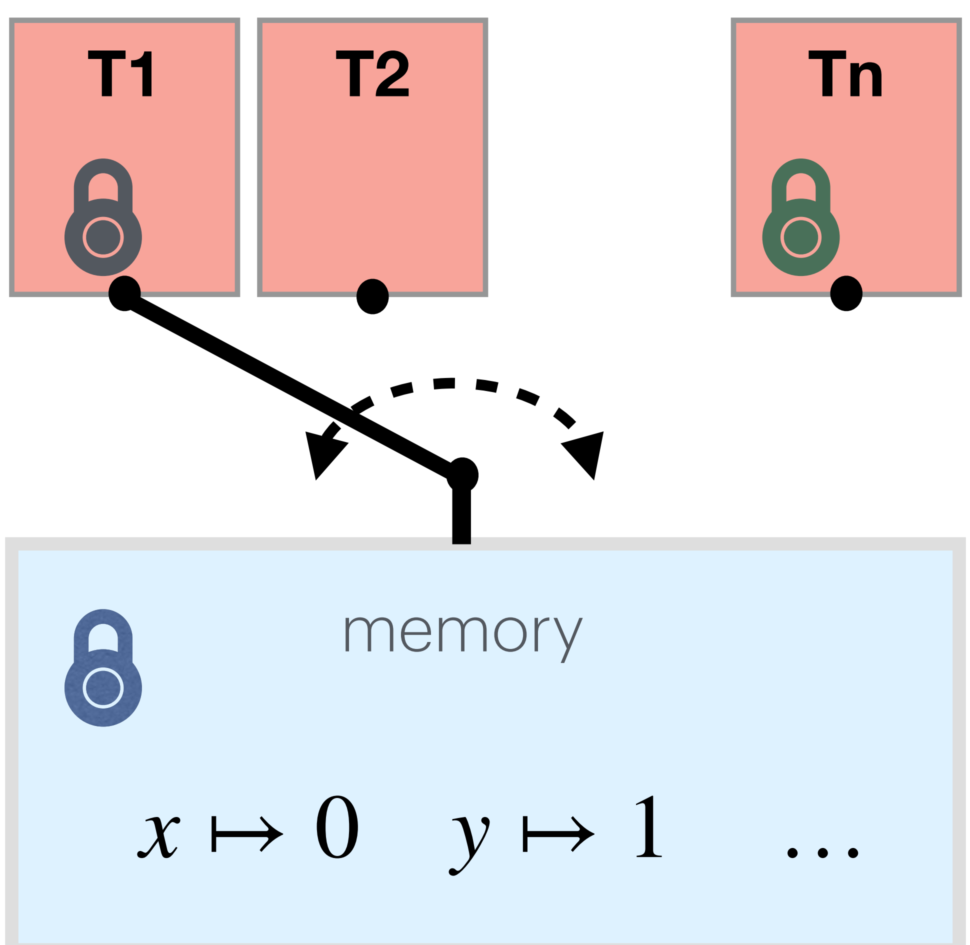
If a program P satisfies:

- has only non-atomics and locks
- is race-free under SC

Then:

- P has only SC behaviors

in every SC operational trace of P there are no consecutive conflicting accesses by different threads



- Is this good enough?
  - ~~Definition of races still requires to understand execution graphs, consistency, *hb*…~~
  - We also want to allow the use of atomics for avoiding races

The DRF guarantee

Final formulation

```
X := 1
Y := 1 sc
```

```
a := Y sc
if (a=1) then
  b := X
```

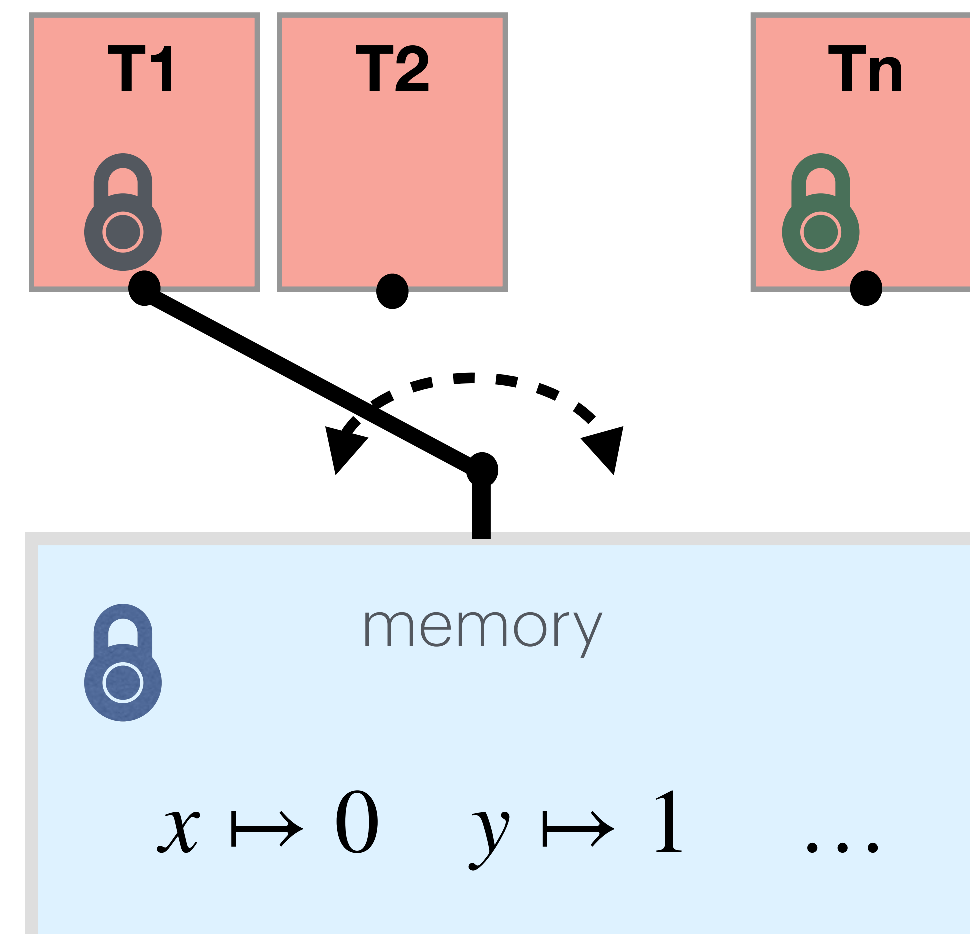
If a program P satisfies:

- ~~has only non-atomics and locks~~
- all races under SC semantics are on `sc` atomic accesses

Then:

- P has only SC behaviors

in every SC operational trace of P, all consecutive conflicting accesses by different threads are marked as `sc` in P



- Is this good enough?
  - ~~Definition of races still requires to understand execution graphs, consistency, *hb*...~~
  - ~~We also want to allow the use of atomics for avoiding races~~

Other DRF guarantees

- The assumption of the DRF guarantee is sometimes expensive to satisfy
- The conclusion is also very strong
- We would like to use another semantics instead of SC in the role of the strong semantics
- Let's see how it works for the Release/Acquire model (DRF-RA)

The DRF-RA guarantee

```
X := 1          ||  a := Y acq
Y := 1 rel      ||  if (a=1) then
                ||    b := X
```

If a program P satisfies:

- all races under RA semantics are on `rel/acq` atomic accesses

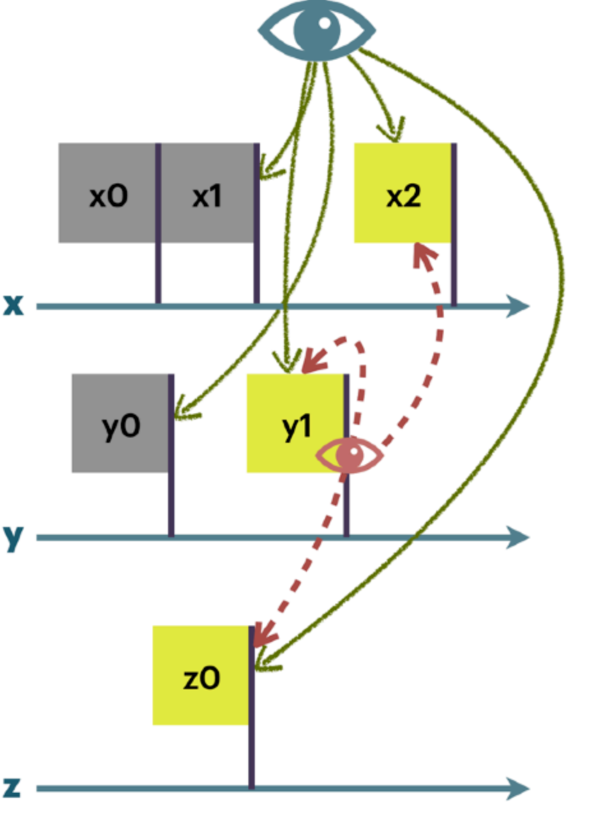
Then:

- P has only RA behaviors

in every RA-consistent execution graph of P, every pair of conflicting accesses unordered by *hb* are marked as `rel/acq` in P

There is also a formulation using the RA view-based semantics:

- A race = thread accesses X but not aware of the latest msg



Local DRF-SC guarantee

```
a := pop(S)        ||  b := pop(S)
X := a             ||  c := Y sc
Y := 1 sc          ||  if (c=1) then
                   ||    d := X
```

$Loc = \{\mathtt{X}, \mathtt{Y}\}$

- The assumptions above are global, which hinders modularity
- A local version can consider a set Loc of locations
- Let P[Loc := `sc`] denote the program P where all accesses to Loc are strengthened to `sc`

If:

- all races of P[Loc := `sc`] on locations in Loc under RC11 semantics are on accesses marked as `sc` in P

Then:

- every behavior of P is a behavior of P[Loc := `sc`]

Dolan, Sivaramakrishnan, Madhavapeddy: **Bounding data races in space and time.** PLDI 2018. https://doi.org/10.1145/3192366.3192421

Cho, Lee, Hur, L: **Modular data-race-freedom guarantees in the promising semantics.** PLDI 2021. https://doi.org/10.1145/3453483.3454082

Local DRF-RA guarantee

```
a := pop(S)        ||  b := pop(S)
X := a             ||  c := Y acq
Y := 1 rel         ||  if (c=1) then
                   ||    d := X
```

$Loc = \{\mathtt{X}, \mathtt{Y}\}$

- Let P[Loc := `ra`] denote the program P where all accesses to Loc are strengthened to `rel/acq`

If:

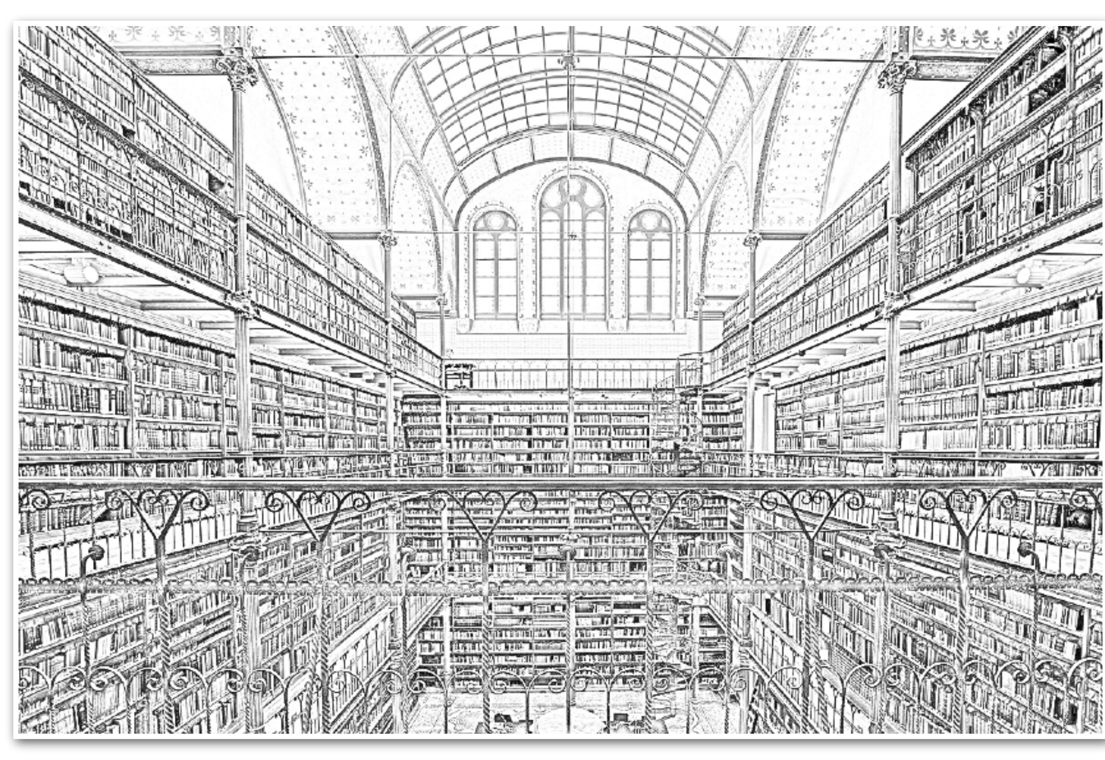
- all races of P[Loc := `ra`] on locations in Loc under RC11 semantics are on accesses marked as `rel/acq` in P

Then:

- every behavior of P is a behavior of P[Loc := `ra`]

Cho, Lee, Hur, L: **Modular data-race-freedom guarantees in the promising semantics.** PLDI 2021. https://doi.org/10.1145/3453483.3454082

Library abstraction



- Experts develop optimized concurrent objects implementations (aka *libraries*)
  - once and for all establish correctness w.r.t. their specifications
- Clients of these implementations reason about program behaviors assuming only the specifications
- Essential in programming, and even more critical in complicated concurrency models

This part is based on:

Singh, L: **An Operational Approach to Library Abstraction under Relaxed Memory Concurrency.** POPL 2023.
https://doi.org/10.1145/3571246

Code as specification

- Specification = *reference implementation*

- Simpler (and less efficient) than the implementation

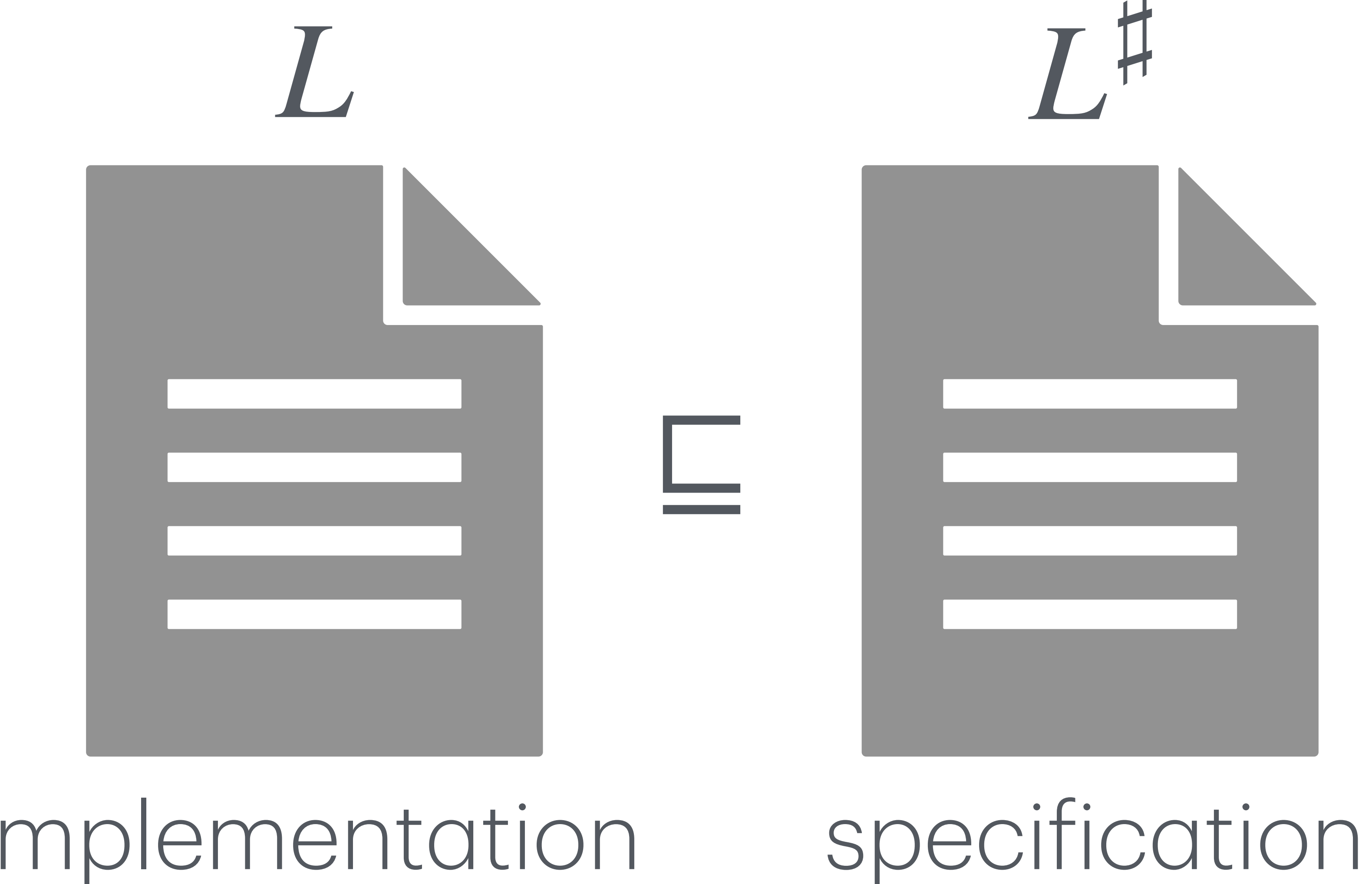
- Derive a reference implementation from a standard sequential specification $Spec$ (assuming SC):

  Take some sequential implementation of $Spec$ and wrap each method in an atomic block

  e.g., `enqueue(v) { … }` → `enqueue(v) { atomic { … } }`

  specification construct

Library correctness



- We aim to have *contextual refinement*:

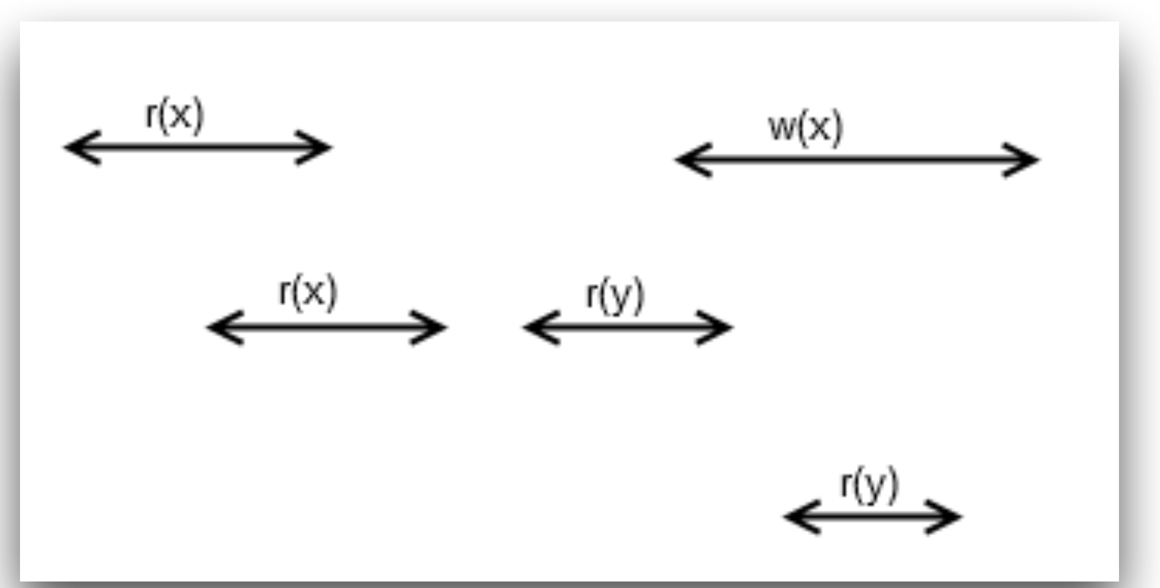
for every program P, Behaviors($P[L]$) $\subseteq$ Behaviors($P[L^\sharp]$)

- We assume that the client and the library use disjoint set of locations

- What correctness condition ensures contextual refinement?

Linearizability

as a library correctness condition under SC



For concurrent data-structures, under SC, linearizability ensures refinement:

Filipović, O'Hearn, Rinetzky, Yang: **Abstraction for concurrent objects.** Theoretical Computer Science 2010. https://doi.org/10.1016/j.tcs.2010.09.021

- If L is linearizable wrt a sequential specification $Spec$, then for every program P,
  Behaviors($P[L]$) ⊆ Behaviors($P[L^\sharp(Spec)]$) under SC

the reference implementation derived from $Spec$

- The converse direction also holds

Linearizability = history inclusion

Linearizability of L wrt $Spec$ holds iff Histories($MGC[L]$) $\subseteq$ Histories($MGC[L^\sharp(Spec)]$), where:

- $L^\sharp(Spec)$ is the reference implementation derived from the sequential specification $Spec$
- MGC denotes the most general client:
  concurrently and repeatedly call the methods of the library with arbitrary arguments
- History is a restriction of an operational trace to call/return

```
f(v) {
  X := v
  a := Y
  return(a)
}
```

example trace:

```
T1:call f(42)
T1:WX42
T2:call f(1)
T1:RY0
T1:return f 0
```

→

induced history:

```
T1:call f(42)
T2:call f(1)
T1:return f 0
```

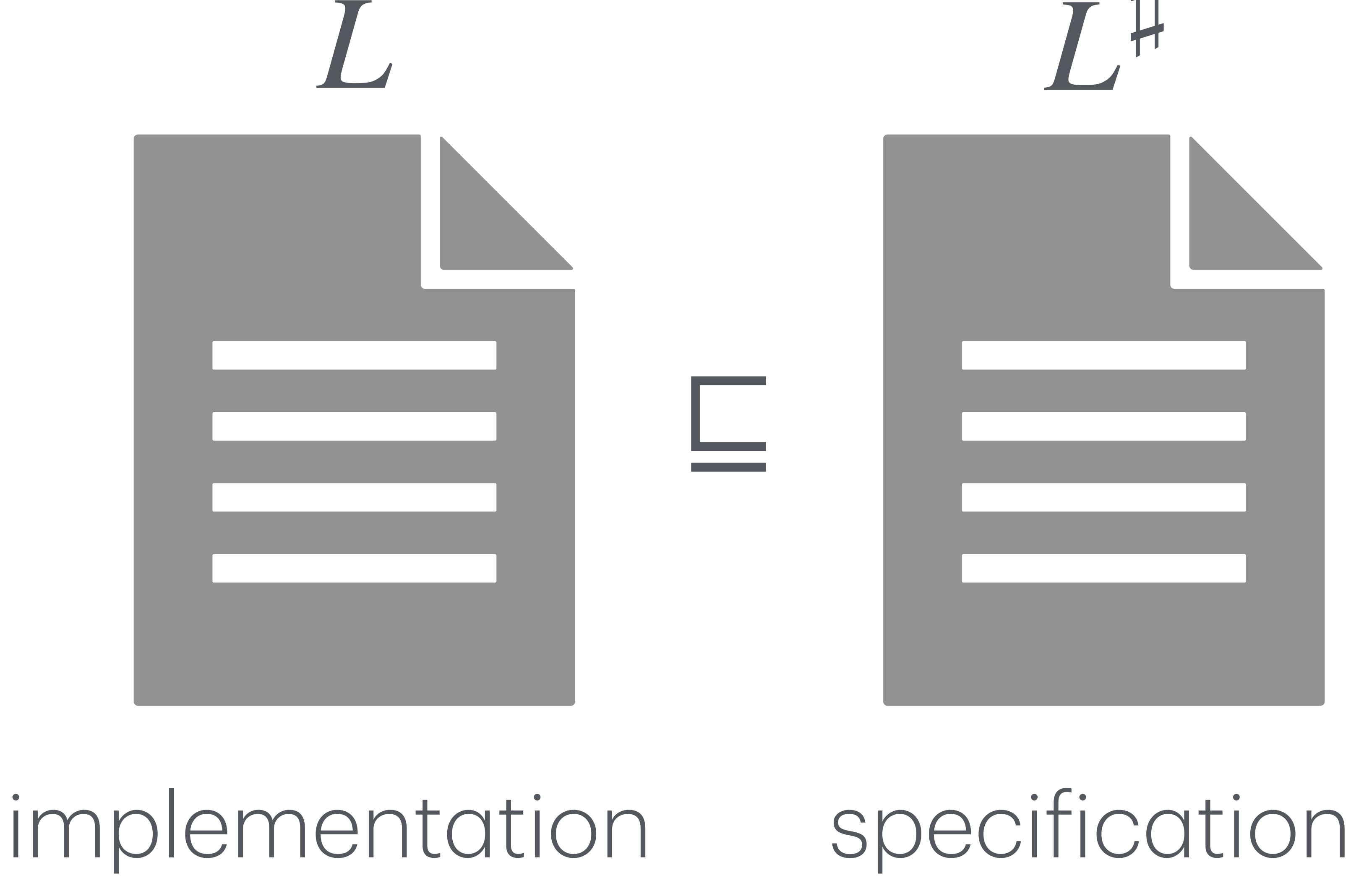
- Histories(P) denotes the set of histories induced by traces of program P

A more general abstraction theorem (for SC)

Theorem

If Histories($MGC[L]$) $\subseteq$ Histories($MGC[L^\sharp]$),
then for every program P, Behaviors($P[L]$) $\subseteq$ Behaviors($P[L^\sharp]$)

- Refinement via linearizability is a particular instance
- This theorem also allows non-atomic specifications



Example: SC assumption is essential!

Assumptions:

- `foo` and `bar` must be called at most once by different threads
- `bar` must be called after `foo` in the execution order

```
foo() {
 X := 1 rel
 return()
}
```

```
bar() {
 a := X acq
 return(a)
}
```

specification

$L^\sharp$

Example: SC assumption is essential!

Assumptions:

- `foo` and `bar` must be called at most once by different threads
- `bar` must be called after `foo` in the execution order

```
foo() {
 return()
}
```

```
foo() {
 X := 1 rel
 return()
}
```

```
bar() {
 pick a∈{0,1}
 return(a)
}
```

```
bar() {
 a := X acq
 return(a)
}
```

implementation

L

specification

$L^\sharp$

Example: SC assumption is essential!

Assumptions:

- `foo` and `bar` must be called at most once by different threads
- `bar` must be called after `foo` in the execution order

```
foo() {
 return()
}
```

```
bar() {
 pick a∈{0,1}
 return(a)
}
```

implementation

L

```
foo() {
 X := 1 rel
 return()
}
```

```
bar() {
 a := X acq
 return(a)
}
```

specification

$L^\sharp$

Histories($MGC[L]$)
$=$
Histories($MGC[L^\sharp]$)

```
T1:call foo()
T1:return foo
T2:call bar()
T2:return bar 0
```

```
T1:call foo()
T1:return foo
T2:call bar()
T2:return bar 1
```

Example: SC assumption is essential!

Assumptions:

- `foo` and `bar` must be called at most once by different threads
- `bar` must be called after `foo` in the execution order

```
Z := 1 rlx            ||  a := Y rlx
foo()                 ||  if (a=1) then
Y := 1 rlx            ||    c := bar() // 1
                      ||    d := Z rlx // 0
```

This behavior is:

- impossible with the specification $L^\sharp$
- but, possible with the implementation L!

```
foo() {
 return()
}

bar() {
 pick a∈{0,1}
 return(a)
}
```

implementation

L

```
foo() {
 X := 1 rel
 return()
}

bar() {
 a := X acq
 return(a)
}
```

specification

$L^\sharp$

Example: SC assumption is essential!

Assumptions:

- `foo` and `bar` must be called at most once by different threads
- `bar` must be called after `foo` in the execution order

```
foo() {
 return()
}
```

```
bar() {
 pick a∈{0,1}
 return(a)
}
```

implementation

L

```
foo() {
 X := 1 rel
 return()
}
```

```
bar() {
 a := X acq
 return(a)
}
```

specification

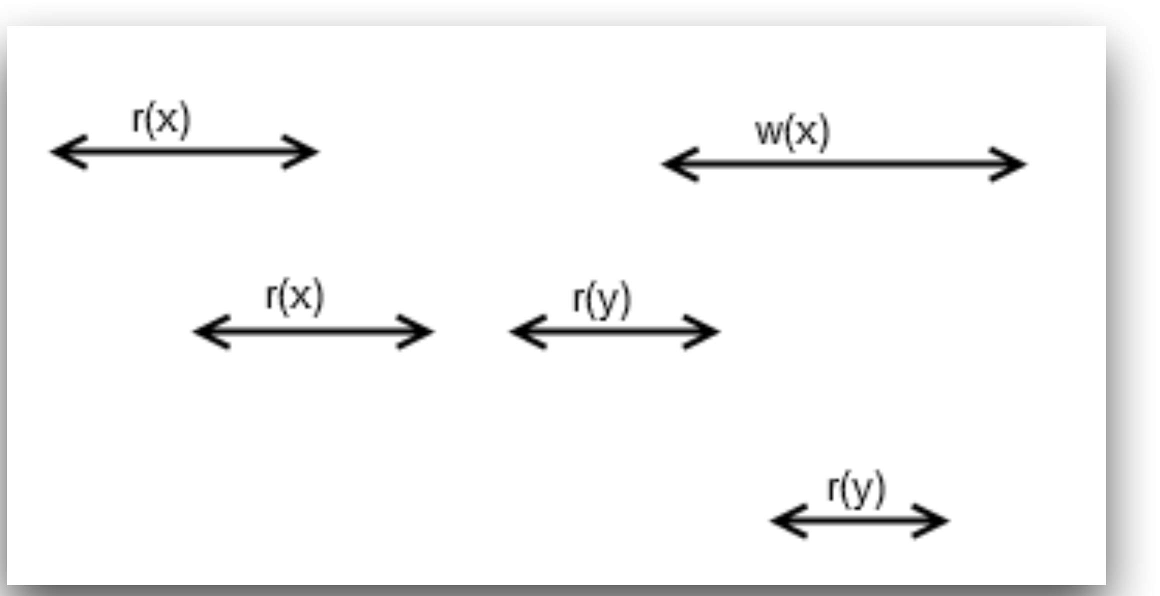
$L^\sharp$

`foo()`
`Y := 1 rel`
	`a := Y acq`
	`if (a=1) then`
	`  c := bar() // 0`

This behavior is:

- impossible with the specification $L^\sharp$
- but, possible with the implementation L!

What can we do about it?



- Under WMM client-library interaction is **not** fully captured by call/return histories
- We can work with partial orders (akin to execution graphs) rather than sequential histories:
  - Batty, Dodds, Gotsman: **Library abstraction for C/C++ concurrency.** POPL 2013. https://doi.org/10.1145/2429069.2429099
  - Doherty, Dongol, Wehrheim, Derrick: **Making Linearizability Compositional for Partially Ordered Executions.** IFM 2018. https://doi.org/10.1007/978-3-319-98938-9_7
- Or enrich sequential histories with more information:
  - Burckhardt, Gotsman, Musuvathi, Yang: **Concurrent Library Correctness on the TSO Memory Model.** ESOP 2012. https://doi.org/10.1007/978-3-642-28869-2_5
  - Khyzha, L: **Abstraction for Crash-Resilient Objects.** ESOP 2022. https://doi.org/10.1007/978-3-030-99336-8_10

Enriched histories for RC11

```
foo()
Y := 1 rel
```

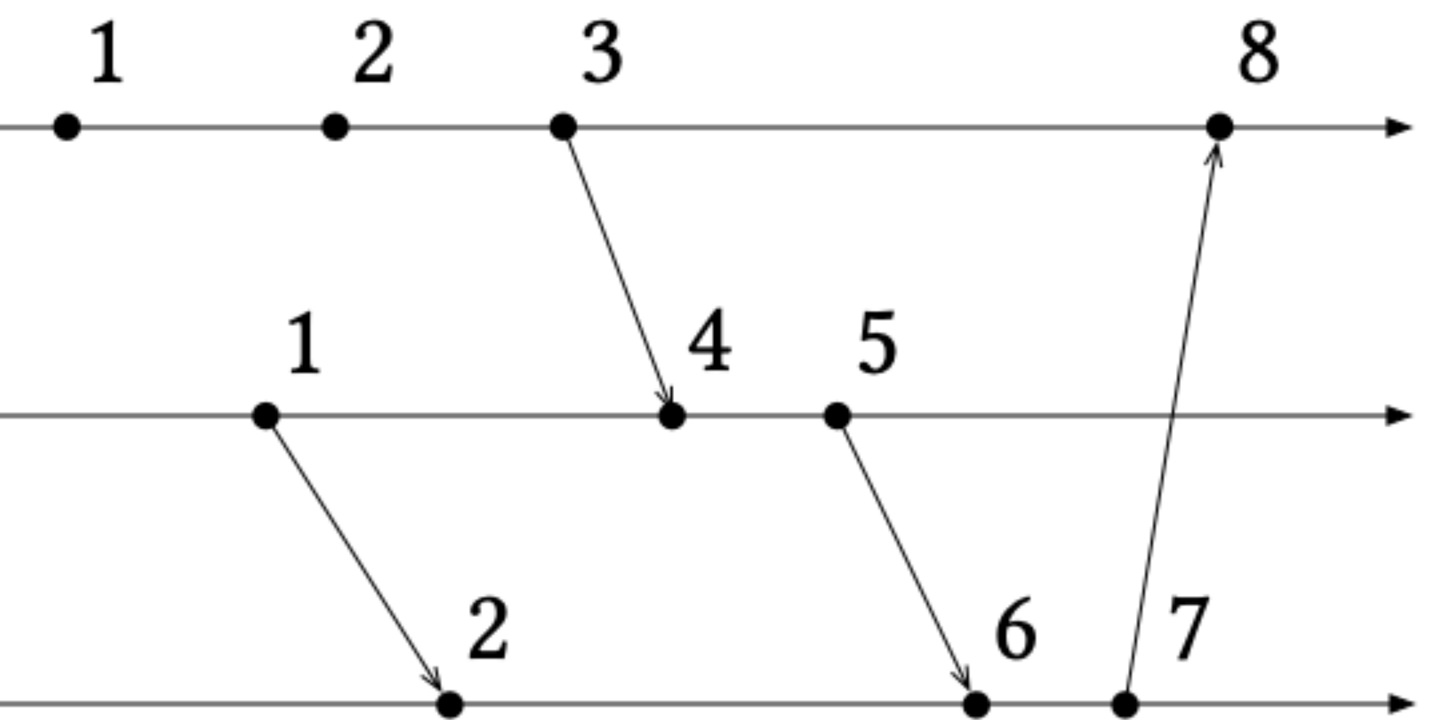
```
a := Y acq
if (a=1) then
  c := bar() // 0
```

- The read `Y=1` imposes *hb* order, so `T2` must be aware of `foo()`'s effect when `bar()` is called
- We will expose this in histories by including propagations of call/return

real-time order ⇏ happens before

A propagation semantics for RC11

- A novel operational semantics for (a fragment of) RC11
- Explicit point-to-point propagation transitions marking when an event of one thread becomes visible to another thread
- We include propagation of the call/return events in memory traces



Example: MP

`Y := 42 rel`
`X := 1 rel`
	`a := X acq`
	`if (a=1) then`
	`  b := Y acq // 0`

some possible traces:

```
T1:Wy42
T1:Wx1
T2:Rx0
```

```
T1:Wy42
T1:Wx1
T1→T2:Wy42
T2:Rx0
```

```
T1:Wy42
T1:Wx1
T1→T2:Wy42
T1→T2:Wx1
T2:Rx1
T2:Ry42
```

In the Release/Acquire fragment:

- propagation follows *hb*
- read from the *mo*-maximal write that was propagated to the thread

Example with function calls

```
foo() {
 X := 1 rel
 return()
}
```

```
bar() {
 a := X acq
 return(a)
}
```

```
foo()              ||  a := Y acq
Y := 1 rel         ||  if (a=1) then
                   ||    c := bar()
```

a possible trace

```
T1:call foo()
T1:Wx1
T1:return foo
T1:Wy1
T1→T2:call foo()
T1→T2:Wx1
T1→T2:return foo()
T1→T2:Wy1
T2:Ry1
T2:call bar()
T2:Rx1
T2:return bar 1
```

we include propagations of calls/returns in histories

Example

```
foo() {
 return()
}
```

```
bar() {
 pick a∈{0,1}
 return(a)
}
```

implementation

L

```
foo() {
 X := 1 rel
 return()
}
```

```
bar() {
 a := X acq
 return(a)
}
```

specification

$L^\sharp$

induced *enriched* histories of MGC

```
T1:call foo()
T1:return foo
T1→T2:call foo()
T1→T2:return foo()
T2:call bar()
T2:return bar 0
```

```
T1:call foo()
T1:return foo
T2:call bar()
T2:return bar 1
```

possible for L but not for $L^\sharp$!

Abstraction theorem for RC11

Theorem

If Histories($MGC[L]$) $\subseteq$ Histories($MGC[L^\sharp]$),
then for every program P, Behaviors($P[L]$) $\subseteq$ Behaviors($P[L^\sharp]$) **under SC**

Theorem

If PHistories($MGC[L]$) $\subseteq$ PHistories($MGC[L^\sharp]$),
then for every program P, Behaviors($P[L]$) $\subseteq$ Behaviors($P[L^\sharp]$) **under RC11**

where PHistories(P) denotes the set of enriched histories (with calls/returns/call propagations/return propagations) induced by traces of program P

Application: RCU

- Simple lock-based specification for basic Read-Copy-Update (RCU) primitives under RC11
  - unlike existing declarative ad-hoc specifications
  - RCU in client programs on RC11 can be understood via locks
- We used the the *FDR4 refinement checker* library correctness for a simple RCU implementation from:

Alglave, Maranget, McKenney, Parri, Stern: **Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel.** ASPLOS 2018. https://doi.org/10.1145/3173162.3177156

Restricted clients

Libraries often have “calling policies”
(e.g., single producer, consume only non-empty collections, …)

public member function

std::list::pop_front

```
void pop_front();
```

Delete first element

Removes the first element in the list container, effectively reducing its size by one.

Exception safety

If the container is not empty, the function never throws exceptions (no-throw guarantee).
Otherwise, it causes *undefined behavior*.

Restricted clients

```
push(1)
a := pop()
```

- We would like a stronger theorem:

Theorem

If PHistories($MGC_{poilcy}[L]$) $\subseteq$ PHistories($MGC_{poilcy}[L^\sharp]$),
then for every program P that adheres to the policy, Behaviors($P[L]$) $\subseteq$ Behaviors($P[L^\sharp]$) **under RC11**

- To show that P adheres to policy, should we use L or $L^\sharp$?
- We want it to be $L^\sharp$ so that the theorem can be applied without any knowledge of L !
- Circular argument? induction works!

Restricted clients

```
push(1)
a := pop()
```

- We would like a stronger theorem:

Theorem

If PHistories($MGC_{poilcy}[L]$) $\subseteq$ PHistories($MGC_{poilcy}[L^\sharp]$),
then for every program P that adheres to the policy, Behaviors($P[L]$) $\subseteq$ Behaviors($P[L^\sharp]$) **under RC11**

- To show that P adheres to policy, should we use L or $L^\sharp$?
- We want it to be $L^\sharp$ so that the theorem can be applied without any knowledge of L !
- Circular argument? induction works!

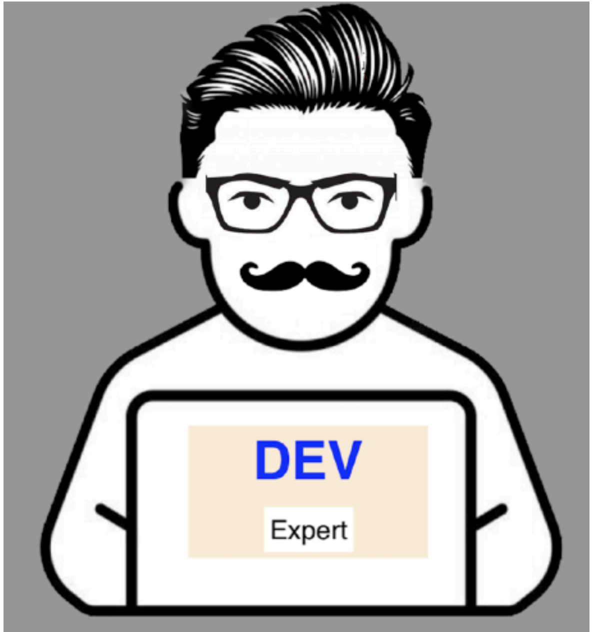
The final abstraction theorem

- If the following hold:

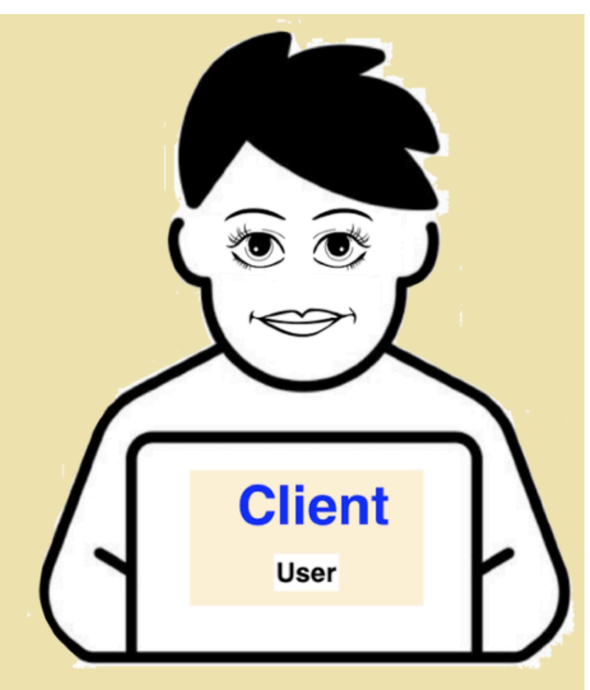
  1. PHistories($MGC_{poilcy}[L]$) ⊆ PHistories($MGC_{poilcy}[L^\sharp]$)
  2. $MGC_{poilcy}[L]$ is not racy
  3. PHistories($P[L^\sharp]$) ⊆ PHistories($MGC_{poilcy}[L^\sharp]$)
  4. $P[L^\sharp]$ is not racy

- Then: Behaviors($P[L]$) ⊆ Behaviors($P[L^\sharp]$)

developer obligations



client obligations



LDRF-RA via library abstraction

```
writeX(v) {
  X := v rel
  return()
  }
```

```
readX() {
  a := X acq
  return(a)
  }
```

Specification

$L^\sharp$

```
writeX(v) {
  X := v
  return()
  }
```

```
readX() {
  a := X
  return(a)
  }
```

Implementation

L

Loc = set of locations accesses solely by the library

MGC_{poilcy} = call methods in a way that avoids races between `writeX` and `readX`

developer obligation: ✓

client obligation:

PHistories($P[L^\sharp]$) ⊆ PHistories($MGC_{poilcy}[L^\sharp]$)

all races of P[Loc := `ra`] on locations in Loc under RC11 semantics are on accesses marked as `rel/acq` in P

conclusion:

Behaviors($P[L]$) ⊆ Behaviors($P[L^\sharp]$)

every behavior of P is a behavior of P[Loc := `ra`]

Library specification under WMM

Example

- Sequential specifications tell us nothing about the synchronization induced by the library

```
X := 42 rlx         ||  a := dequeue(q) // 1
enqueue(q,1)        ||  c := X rlx // 0
```

```
X := 1 rlx              ||  enqueue(q,1)
a := dequeue(q) // ⊥    ||  c := X rlx // 0
```

```
X := 1 rlx      ||  enqueue(q,2)     ||  b := dequeue(q) // 1
enqueue(q,1)    ||  A := X rlx // 0  ||  c := dequeue(q) // 2
```

- How to specify the different options?

Library specification under WMM

- Declarative library specifications with specialized synchronization relations:

  Raad, Doko, Rozic, L, Vafeiadis: **On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models.** POPL 2019. https://doi.org/10.1145/3290381

- An operational approach?
  - What can serve as reference implementation for different options?
  - A per-object lock gives us the strongest queue: real-time order ⇒ happens-before

RC11 library specification constructs

specification construct

- We propose partial locks: locks that induce **only intra-library** synchronization
- If we wrap a sequential implementation (using non-atomics) in a partial per-object lock, we obtain a queue that does not provide any synchronization to its clients

e.g., `enqueue(v) { … }` → `enqueue(v) {` $\texttt{lock}_{\texttt{lib}}$`(L) { … } }`

- By using release/acquire accesses in the specification we can express stronger queues

Verification under WMM

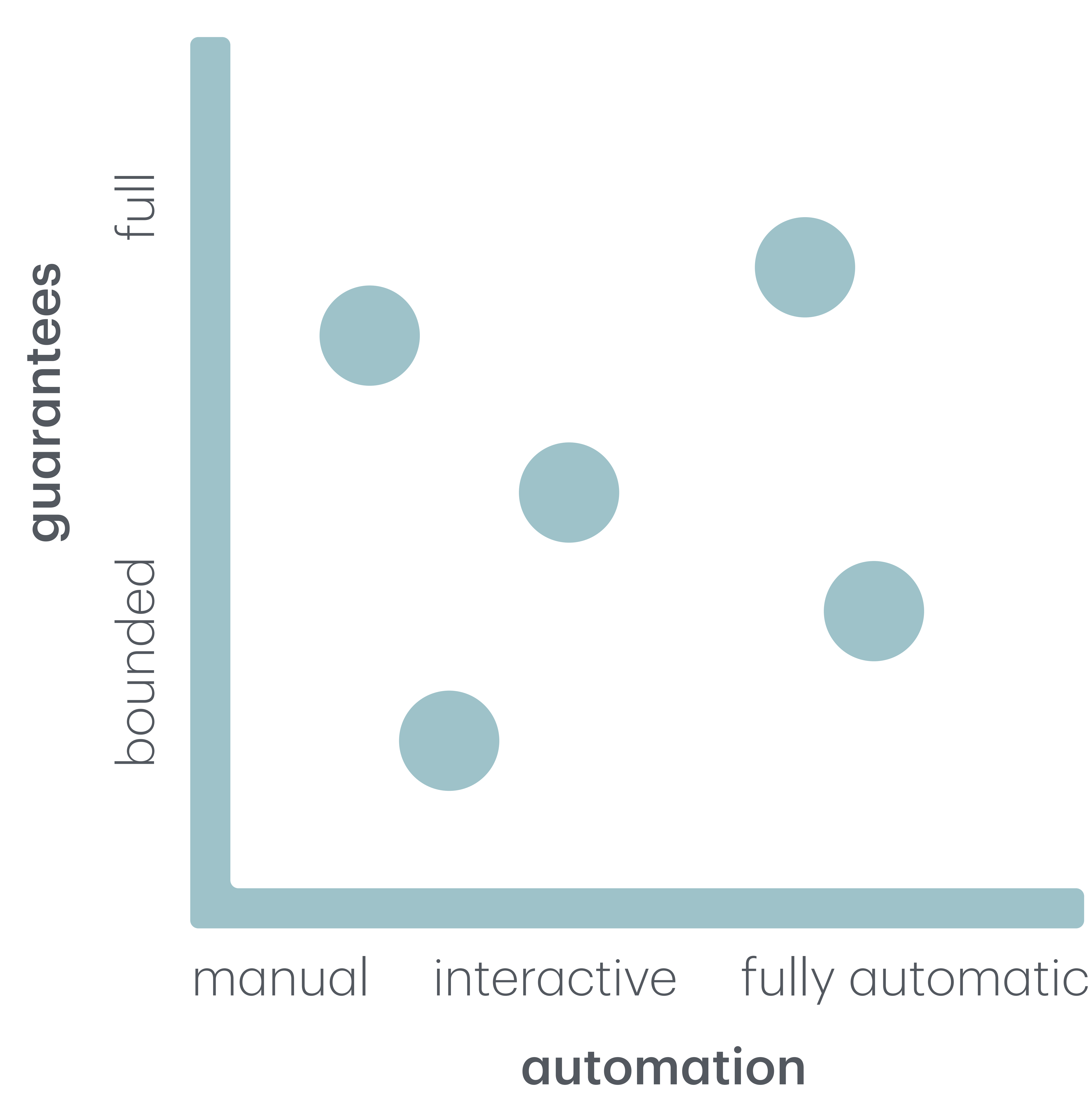
A short (and very partial) survey

Verification questions and approaches for WMM

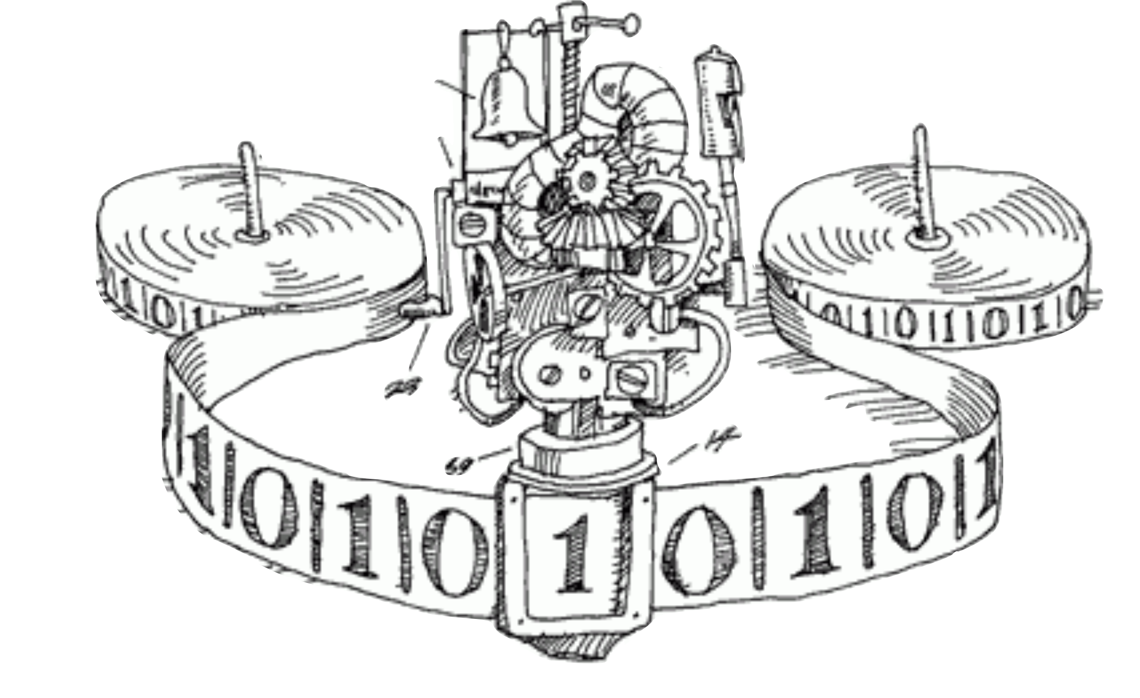
What do we verify?

- Program never crashes
- Provides mutual exclusion
- Correctly implements a concurrent data structure
- ...

1. Theoretical decidability
2. Model checking and testing
3. Program logics
4. Robustness



Theoretical verification



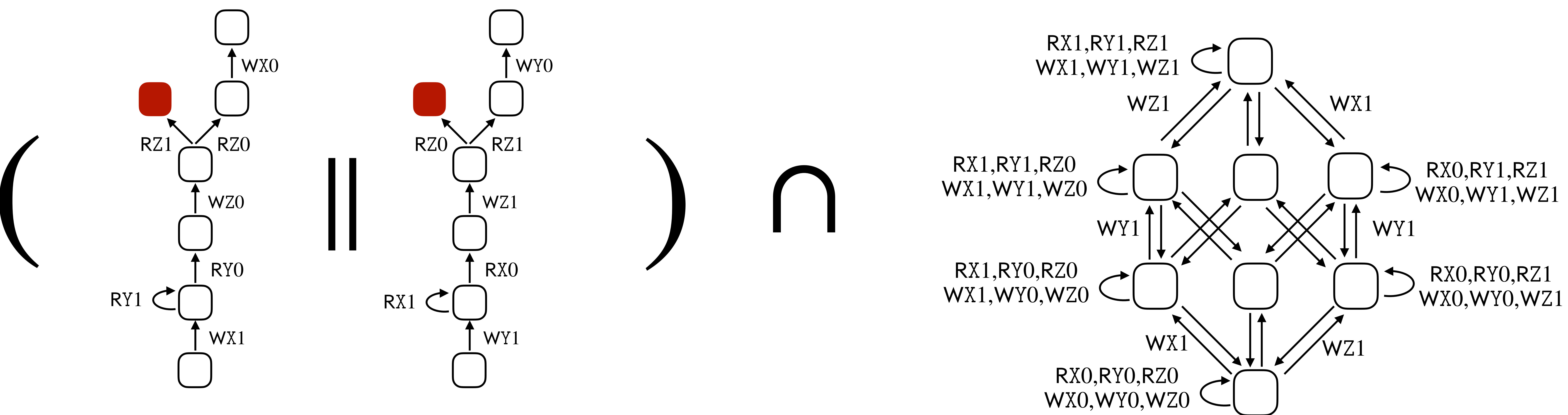
1

- Assume finite-state programs (but with loops!)
- There may still be infinitely many memory states
  (unbounded buffers, unbounded execution graphs)

- *Is state reachability is* ***decidable****? What is its complexity?*
- The answer depends on the underlying memory model

Theoretical verification under SC

1



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

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

Some results for weak memory modes

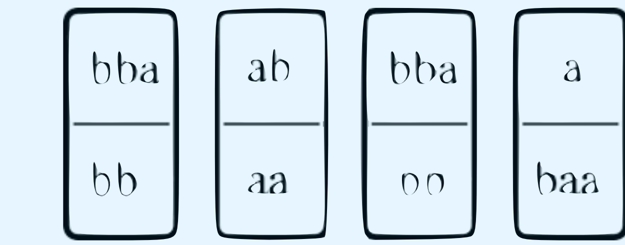
Reachability under x86-TSO is decidable:

- via a dual semantics (load-buffers instead of store buffers) that forms a WSTS

Abdulla, Atig, Bouajjani, Ngo: **A Load-Buffer Semantics for Total Store Ordering.** Log. Methods Comput. 2018. https://doi.org/10.23638/LMCS-14(1:9)2018

Reachability under RA is undecidable:

- reduction from Post correspondence problem

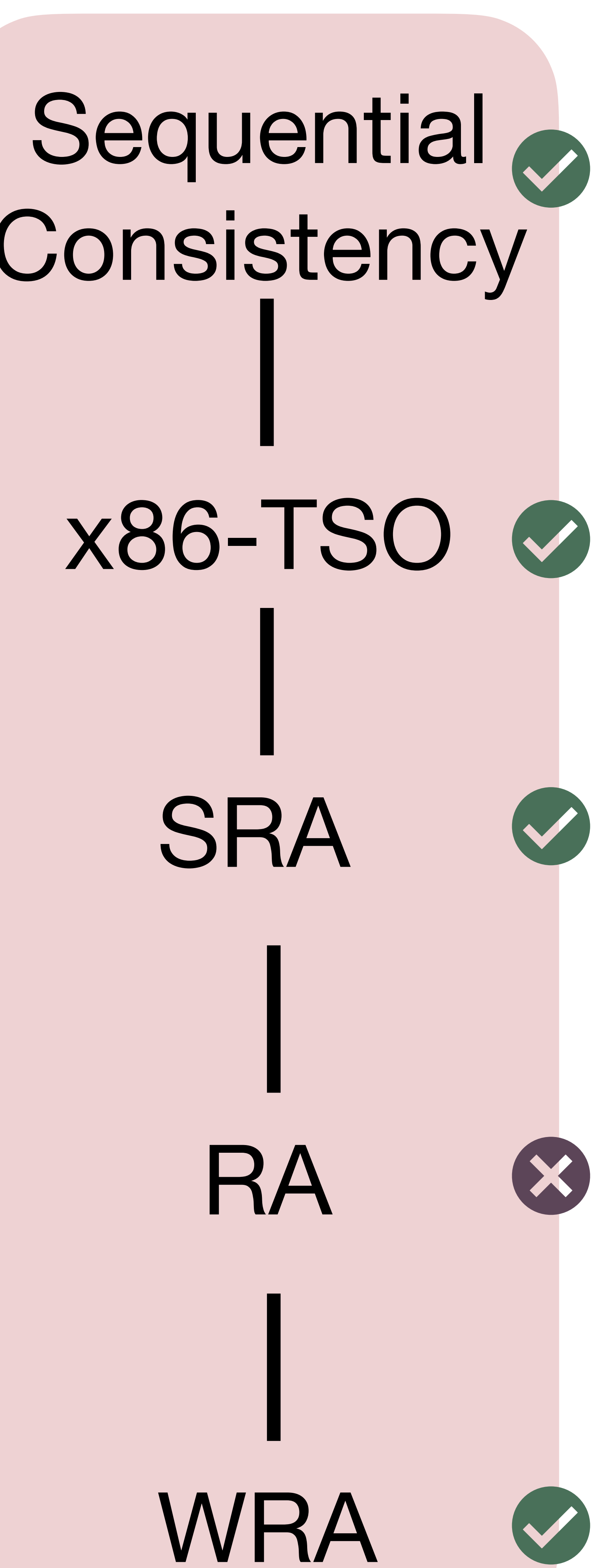


Abdulla, Arora, Atig, Krishna: **Verification of programs under the release-acquire semantics.** PLDI 2019. https://doi.org/10.1145/3314221.3314649

Reachability under WRA/SRA is decidable:

- via a potential-based semantics that forms a WSTS

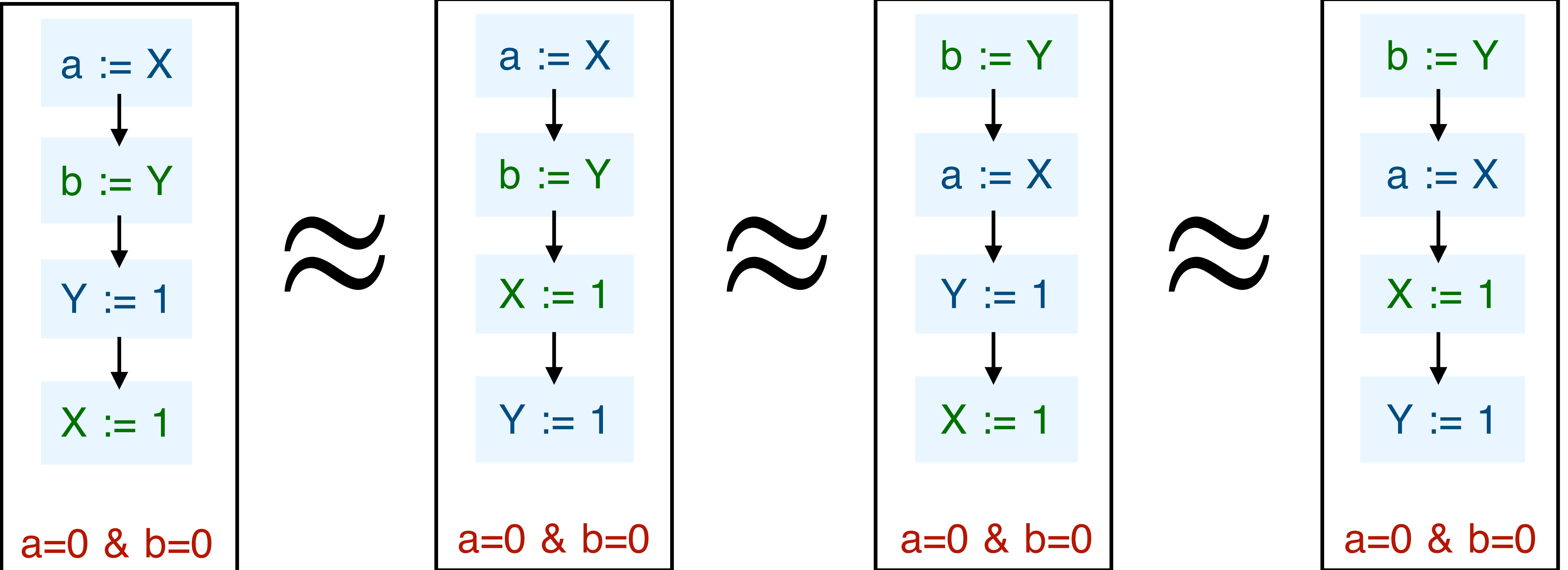
L, Boker: **What's Decidable About Causally Consistent Shared Memory?** ACM Trans. Program. Lang. Syst. 2022. https://doi.org/10.1145/3505273



Model checking

2

- Given a loop-free program (usually after loop unrolling), exhaustively verify that all its runs do not violate safety assertions
- Naively checking all traces is infeasible (for both time and memory)
- Remedies:
  - stateless verification: explore all executions without storing in memory the executions explored so far
  - partial order reduction: explore one candidate from each equivalence class

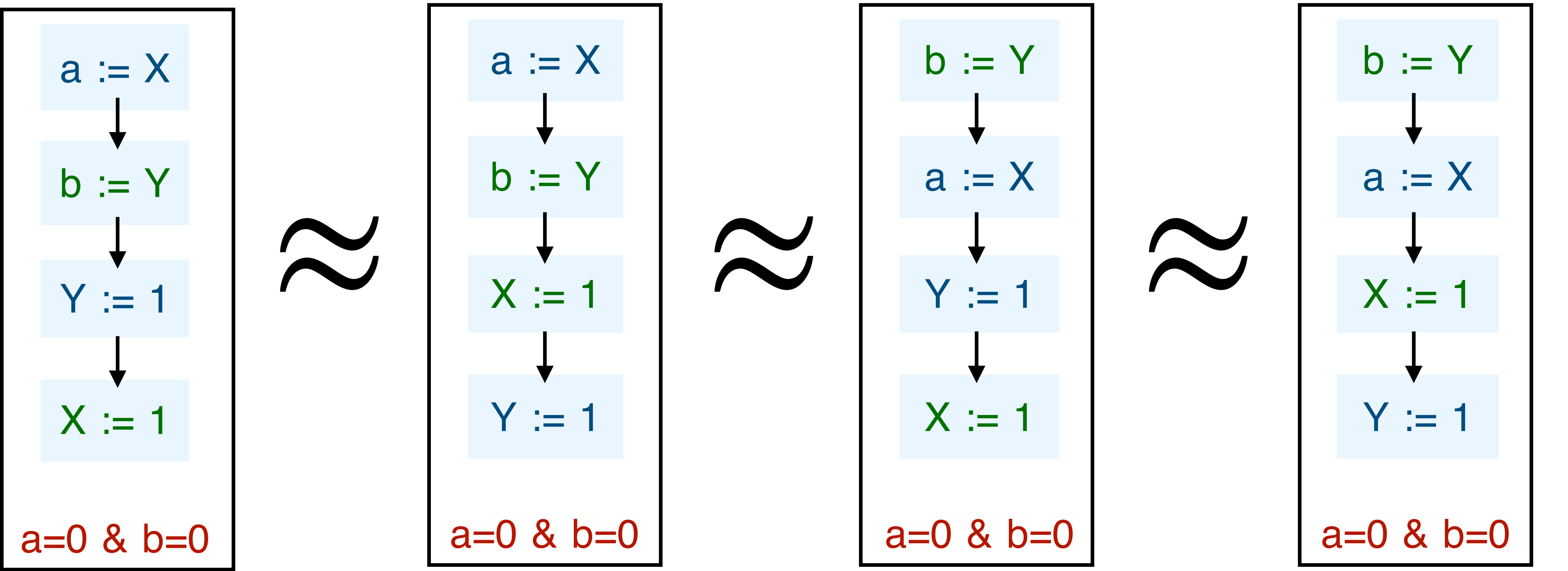


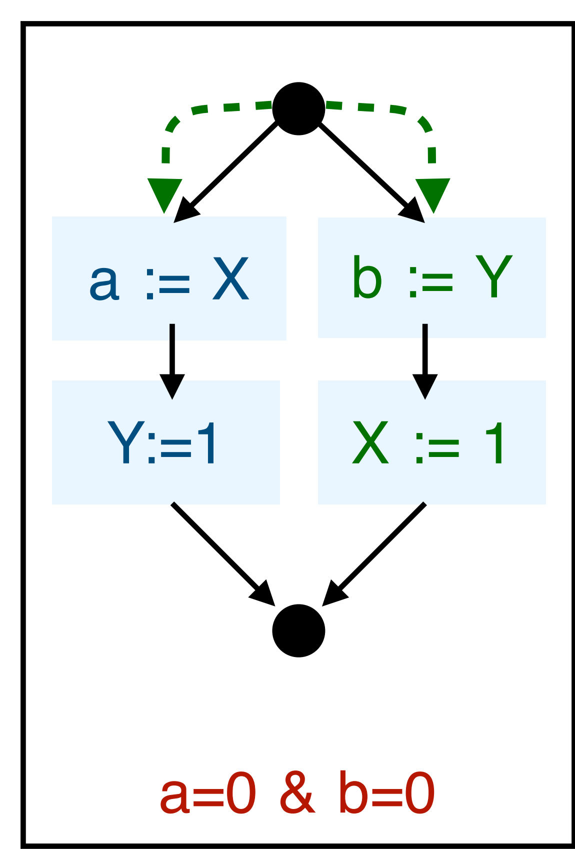
Abdulla, Aronis, Jonsson, Sagonas: **Optimal dynamic partial order reduction.** POPL 2014. https://doi.org/10.1145/2535838.2535845

Partial order reduction using execution graphs

2

- Explore consistent execution graphs rather than traces (also for SC!)
- Execution graphs track less redundant order and represent equivalence classes





Kokologiannakis, Raad, Vafeiadis: **Model checking for weakly consistent libraries.** PLDI 2019. https://doi.org/10.1145/3314221.3314609

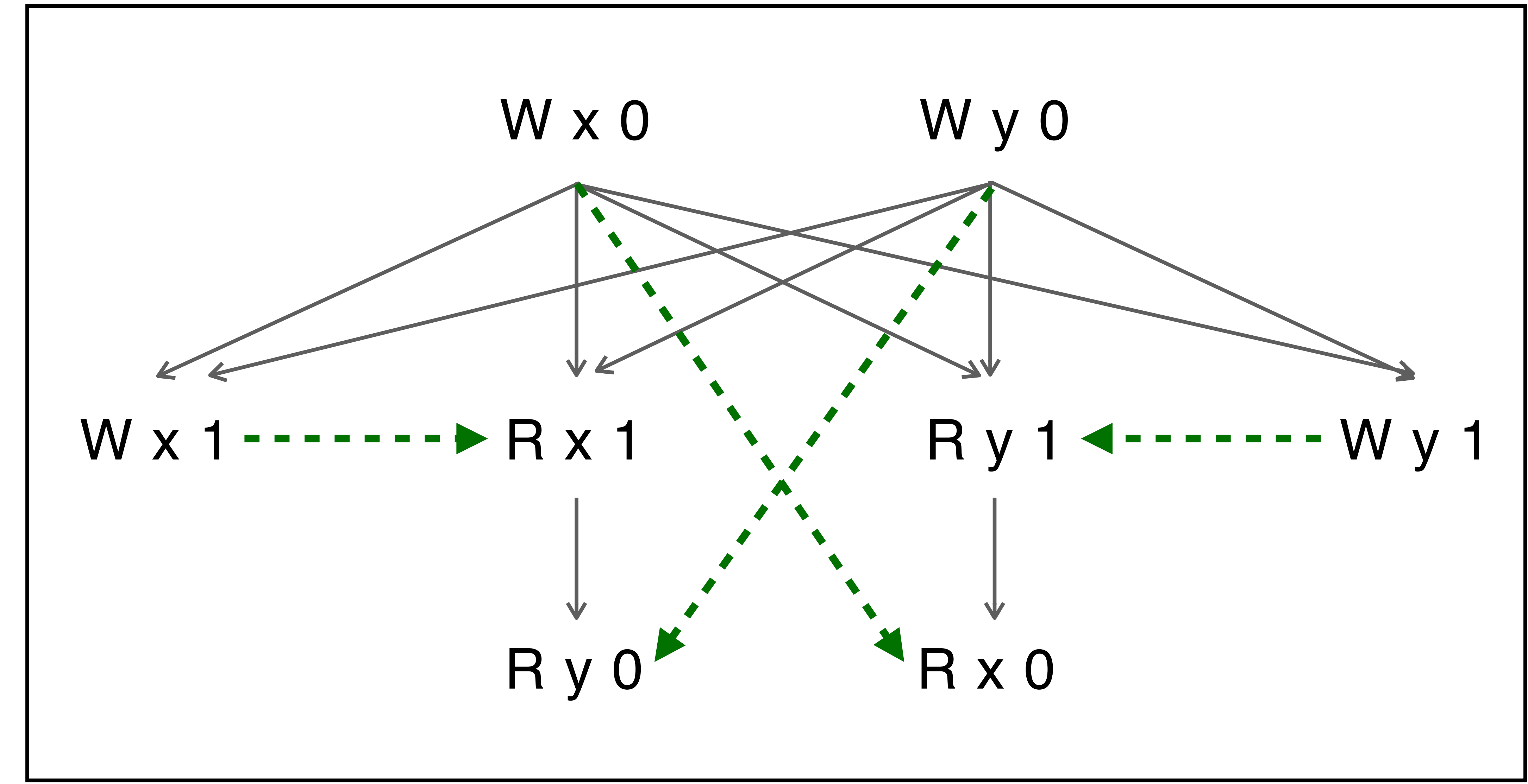
Kokologiannakis, Marmanis, Gladstein, Vafeiadis: **Truly stateless, optimal dynamic partial order reduction.** POPL 2021. https://doi.org/10.1145/3498711

Luo, Demsky: **C11Tester: a race detector for C/C++ atomics.** ASPLOS 2021. https://doi.org/10.1145/3445814.3446711.

A related problem

2

- Given an execution graph G check whether it is consistent under a memory model M
- Some weak memory models make this problem easier!
  - Given only program-order po and reads-from rf relations:
    - Checking for SC-consistency is NP-complete
    - Checking for RA-consistency is in PTIME



Chakraborty, Krishna, Mathur, Pavlogiannis: **How Hard Is Weak-Memory Testing?** POPL 2024. https://doi.org/10.1145/3632908

Program logics

3

- A (mostly) manual approach for syntax-guided verification
- Derivation rules that provide reasoning principles

$$\frac{\{P\}C_1\{Q\} \qquad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

$$\frac{\{P \land b\}C\{P\}}{\{P\}\text{while } b \text{ do } C\{P \land \neg b\}}$$

Thread 1 proof outline:

$\left\{\mathrm{Seen}(\pi, V_0) * \mathrm{Hist}(x, [(0, _, V_x)]) * V_x \sqsubseteq V_0 * \boxed{\mathrm{Inv}_y(V_0)}\right\}$

$x_{[\mathrm{na}]} := 37$

$\{\exists V_{37} \sqsupseteq V_0.\ \mathrm{Seen}(\pi, V_{37}) * \mathrm{Hist}(x, [(37, _, V_{37})])\}$

$\left\{\mathrm{Seen}(\pi, V_{37}) * \boxed{\mathrm{Inv}_x(V_{37})}\right\}$

open Inv_y:

$\{\mathrm{Seen}(\pi, V_{37}) * \exists h.\ \mathrm{Hist}(y, h) * \ldots\}$

$y_{[\mathrm{at}]} := 1$

$\left\{\exists V_1 \sqsupseteq V_{37}.\ \mathrm{Seen}(\pi, V_1) * \mathrm{Hist}(y, h \uplus [(1, _, V_1)]) * \boxed{\mathrm{Inv}_x(V_{37})}\right\}$

$\left\{\mathrm{Seen}(\pi, V_1) * \boxed{\mathrm{Inv}_y(V_0)}\right\}$

Thread 2 proof outline:

$\left\{\mathrm{Seen}(\pi, V_0) * \boxed{\mathrm{Inv}_y(V_0)} * \diamond\right\}$

repeat $y_{[\mathrm{at}]}$;

$\left\{\exists V_1, V_{37}, V_2.\ V_2 \sqsupseteq V_1 \sqsupseteq V_{37} * \mathrm{Seen}(\pi, V_2) * \boxed{\mathrm{Inv}_x(V_{37})} * \diamond\right\}$

$\{\mathrm{Seen}(\pi, V_2) * V_{37} \sqsubseteq V_2 * \mathrm{Hist}(x, [(37, _, V_{37})])\}$

$x_{[\mathrm{na}]}$

$\{z.\ \mathrm{Seen}(\pi, V_2) * z = 37 * \mathrm{Hist}(x, [(37, _, V_{37})])\}$

$\mathrm{Inv}_y(V_0) \triangleq \exists h.\ \mathrm{Hist}(y, h) * (0, _, V_0) \in h * \forall V_1, v_1 \neq 0.\ (v_1, _, V_1) \in h \Rightarrow \exists V_{37} \sqsubseteq V_1.\ \boxed{\mathrm{Inv}_x(V_{37})}$

$\mathrm{Inv}_x(V_{37}) \triangleq \diamond \lor \mathrm{Hist}(x, [(37, _, V_{37})])$

Owicki-Gries / rely-guarantee logics

3

$$\textbf{Init: } d := 0;\ f := 0;$$
$$\{f =_1 0 \land f =_2 0 \land d =_1 0 \land d =_2 0\}$$

Thread 1	**Thread 2**
$\{f \not\approx_2 1 \land d =_1 0\}$	$\{[f = 1](d =_2 5)\}$
$1 : d := 5;$	$3 : \textbf{do } r1 \leftarrow^{\mathsf{A}} f \textbf{ until } r1 = 1;$
$\{f \not\approx_2 1 \land d =_1 5\}$	$\{d =_2 5\}$
$2 : f :=^{\mathsf{R}} 1;$	$4 : r2 \leftarrow d;$
$\{true\}$	$\{r2 = 5\}$

$$\{r2 = 5\}$$

- SC-based reasoning is unsound
- Develop specialized assertions for expressing invariants on top of an operational presentation of the memory model

Dalvandi, Doherty, Dongol, Wehrheim: **Owicki-Gries Reasoning for C11 RAR.** ECOOP 2020. https://doi.org/10.4230/LIPIcs.ECOOP.2020.11

L, Dongol, Wehrheim: **Rely-Guarantee Reasoning for Causally Consistent Shared Memory.** CAV 2023. https://doi.org/10.1007/978-3-031-37706-8_11

Seperation logics

3

- Concurrent separation logic is designed to reason about DRF programs
- So it is trivially sound under models that satisfy the DRF guarantee
- Extensions allow reasoning about synchronization primitives of RC11
- In particular, ownership transfer is possible via `rel/acq` synchronization

Invariant: $\exists vs, G.\ \mathrm{Queue}(q, vs, G) * \mathrm{deqPerm}(\mathrm{size}(G.\mathrm{so})) * \mathrm{size}(G.\mathrm{so}) \leq 2 * \ldots$

T1

$\{\mathrm{SeenQueue}(q, \emptyset, \emptyset) * \exists V_1\}$

$\langle \mathrm{Queue}(q, _) * \ldots \rangle\ \mathbf{enq}(q, 41); \langle \mathrm{Queue}(q, _) * \ldots \rangle$

$\langle \mathrm{Queue}(q, _) * \ldots \rangle\ \mathbf{enq}(q, 42); \langle \mathrm{Queue}(q, _) * \ldots \rangle$

$\{\mathrm{SeenQueue}(q, G_1, \{e_1, e_2\}) * \ldots\}$

$\mathrm{flag} :=_{\mathtt{rel}} 1$

T2

$\{\mathrm{SeenQueue}(q, \emptyset, \emptyset) * \mathrm{deqPerm}(1) * \exists V_2\}$

$\langle \mathrm{Queue}(q, _) * \ldots \rangle\ \mathbf{deq}(q)$

$\langle \mathrm{Queue}(q, _) * \ldots \rangle$

$\{\mathrm{SeenQueue}(q, G_2, \{d_2\}) * \ldots\}$

T3

$\{\mathrm{SeenQueue}(q, \emptyset, \emptyset) * \mathrm{deqPerm}(1) * \exists V_3\}$

$\mathtt{while}\ ({*}^{\mathtt{acq}}\mathrm{flag} == 0)\{\};$

$\{\mathrm{SeenQueue}(q, G_1, \{e_1, e_2\}) * \mathrm{deqPerm}(1) * \exists V_3\}$

$\langle \mathrm{Queue}(q, _) * \ldots \rangle\ \mathbf{deq}(q)\ \langle \mathrm{Queue}(q, _) * \ldots \rangle$

$\{v.\ \mathrm{SeenQueue}(q, G_3, \{e_1, e_2, d_3\}) * v \in \{41, 42\}\}$

Vafeiadis, Narayan: **Relaxed separation logic: a program logic for C11 concurrency.** OOPSLA 2013. https://doi.org/10.1145/2509136.2509532

Dang, Jourdan, Kaiser, Dreyer: **RustBelt meets relaxed memory.** POPL 2020. https://doi.org/10.1145/3371102

Dang, Jung, Choi, Nguyen, Mansky, Kang, Dreyer: **Compass: strong and compositional library specifications in relaxed memory separation logic.** PLDI 2022. https://doi.org/10.1145/3519939.3523451

Robustness

4

- Many (useful) programs are *robust*:

    all program behaviors allowed by RC11 are in fact also allowed by SC

Robustness

4

- Many (useful) programs are *robust*:

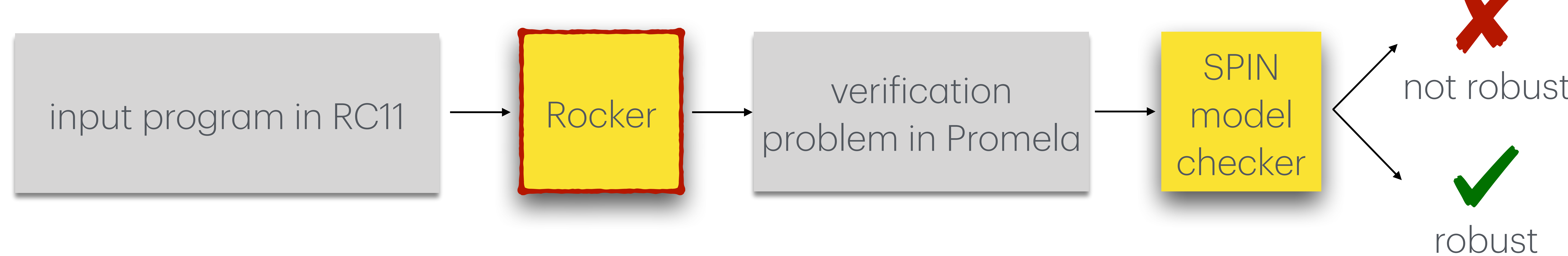
  all program behaviors allowed by RC11 are in fact also allowed by SC

verification under weak memory = verification under SC + robustness

Theorem

Execution-graph robustness against RC11 is PSPACE-complete

Idea: run an instrumented program under SC that monitors whether some step is allowed under RC11 but not under SC



L, Margalit: **Robustness against release/acquire semantics.** PLDI 2019. https://doi.org/10.1145/3314221.3314604

Margalit, L: **Verifying observational robustness against a c11-style memory model.** POPL 2021. https://doi.org/10.1145/3434285

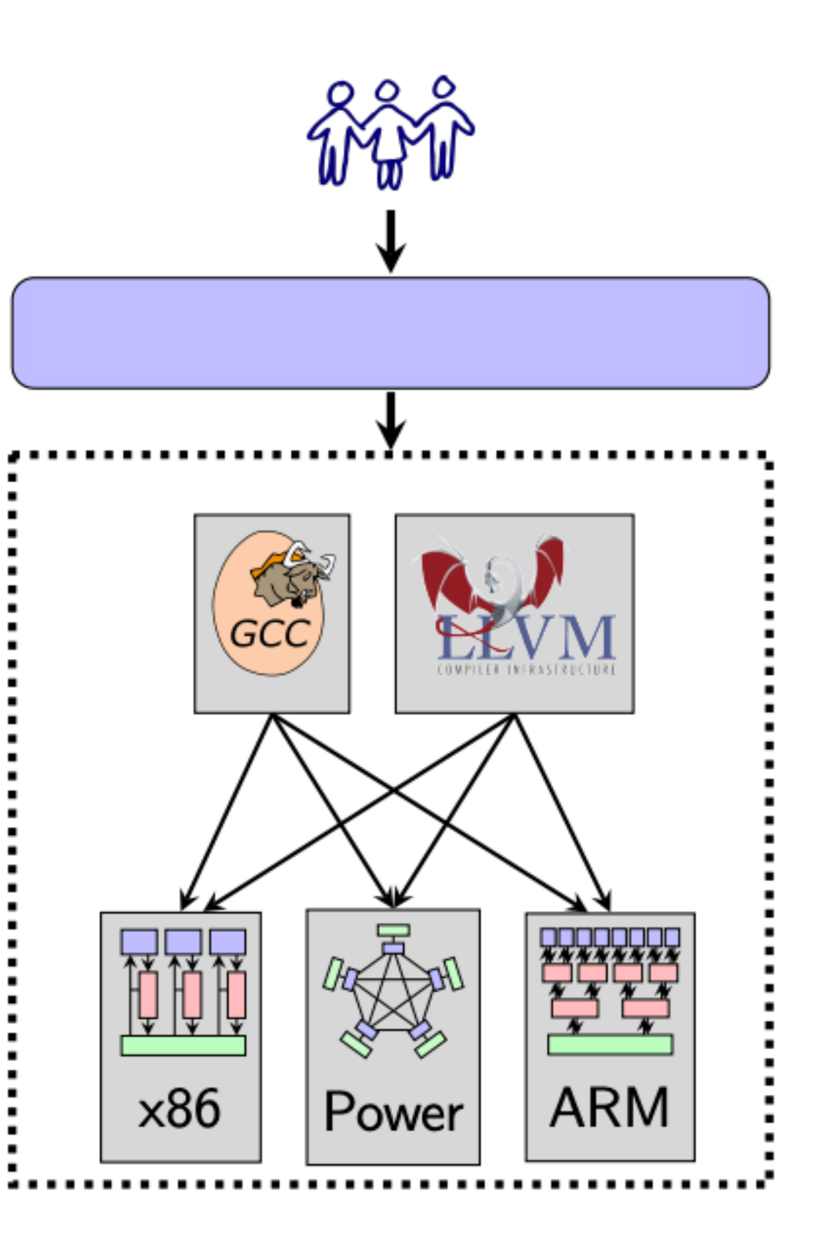
Conclusion

We talked about:

1. The C/C++11 memory model
2. The out-of-thin-air problem & RC11
3. Implementability of (R)C11: compiler optimizations and mapping to hardware
4. Programmability guarantees: DRF theorems, library abstraction
5. Verification

Weak memory models are not only a threat, but also an opportunity to better understand concurrency!

Conclusion



We talked about:

1. The C/C++11 memory model
2. The out-of-thin-air problem & RC11
3. Implementability of (R)C11: compiler optimizations and mapping to hardware
4. Programmability guarantees: DRF theorems, library abstraction
5. Verification

Weak memory models are not only a threat, but also an opportunity to better understand concurrency!