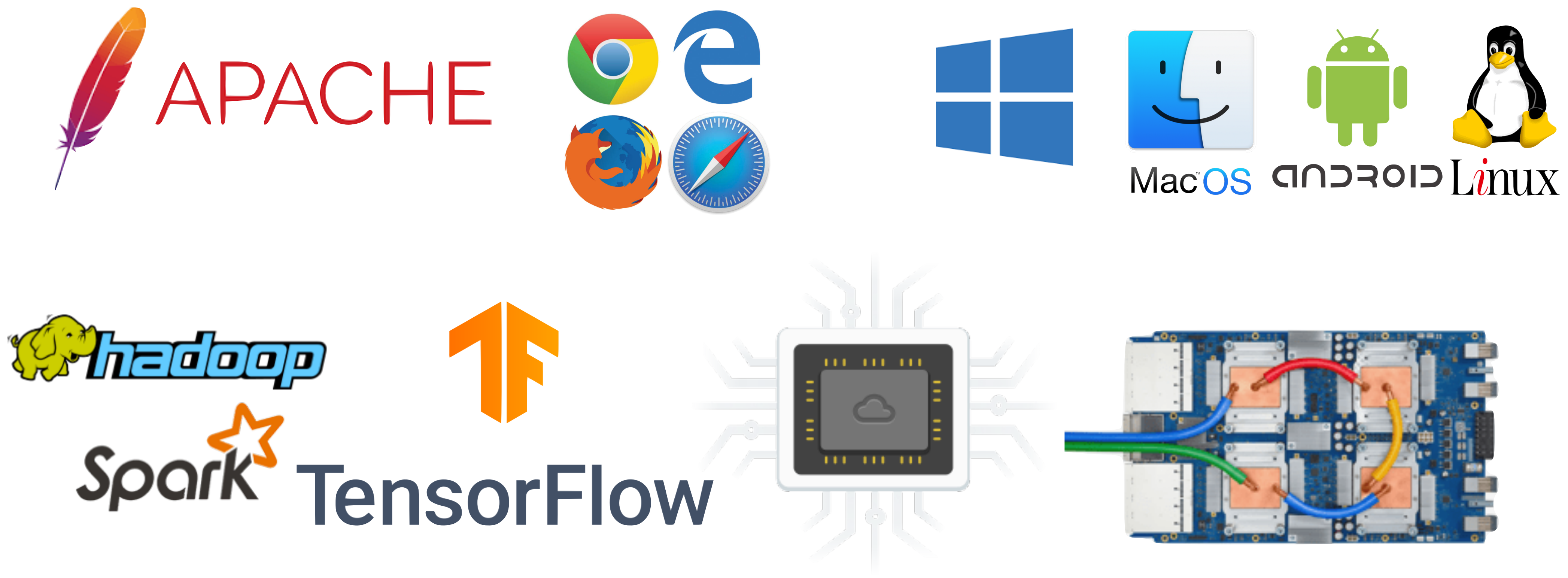
Dynamic Data Race Prediction

Umang Mathur
NUS National University of Singapore

Concurrency : Software and Challenges

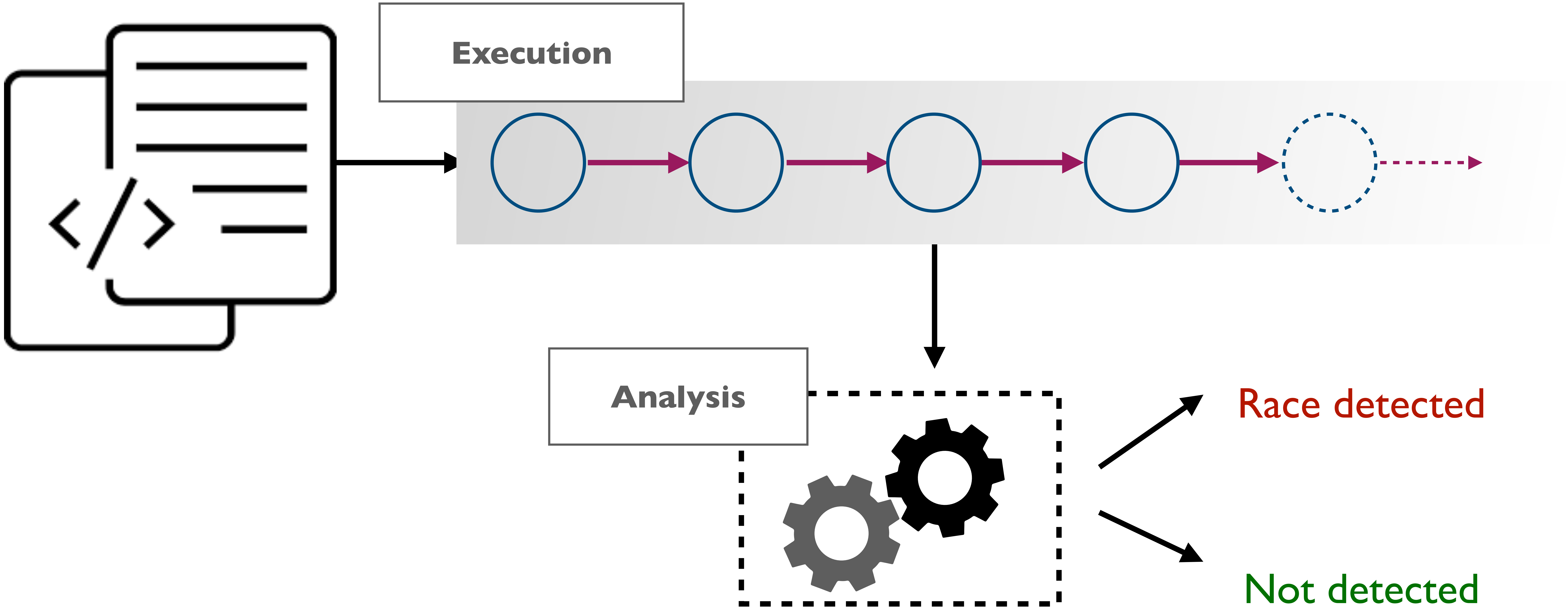


- Ubiquitous computing paradigm
  - Back-bone of big-data and AI revolution
- Challenging to develop concurrent software
  - Large interleaving space

- **Data Races**
  - are most common problems
  - manifest in production despite rigorous testing
  - hard to even reproduce!

A data race occurs when the program accesses a shared memory location from two different threads concurrently without any synchronisation

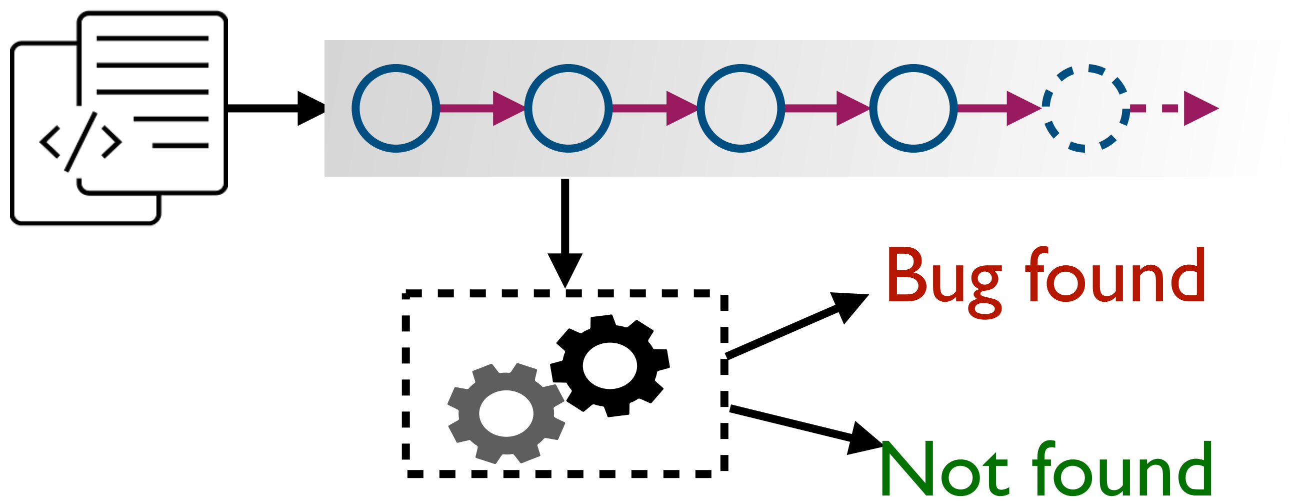
Dynamic Analysis for Detecting Data Races



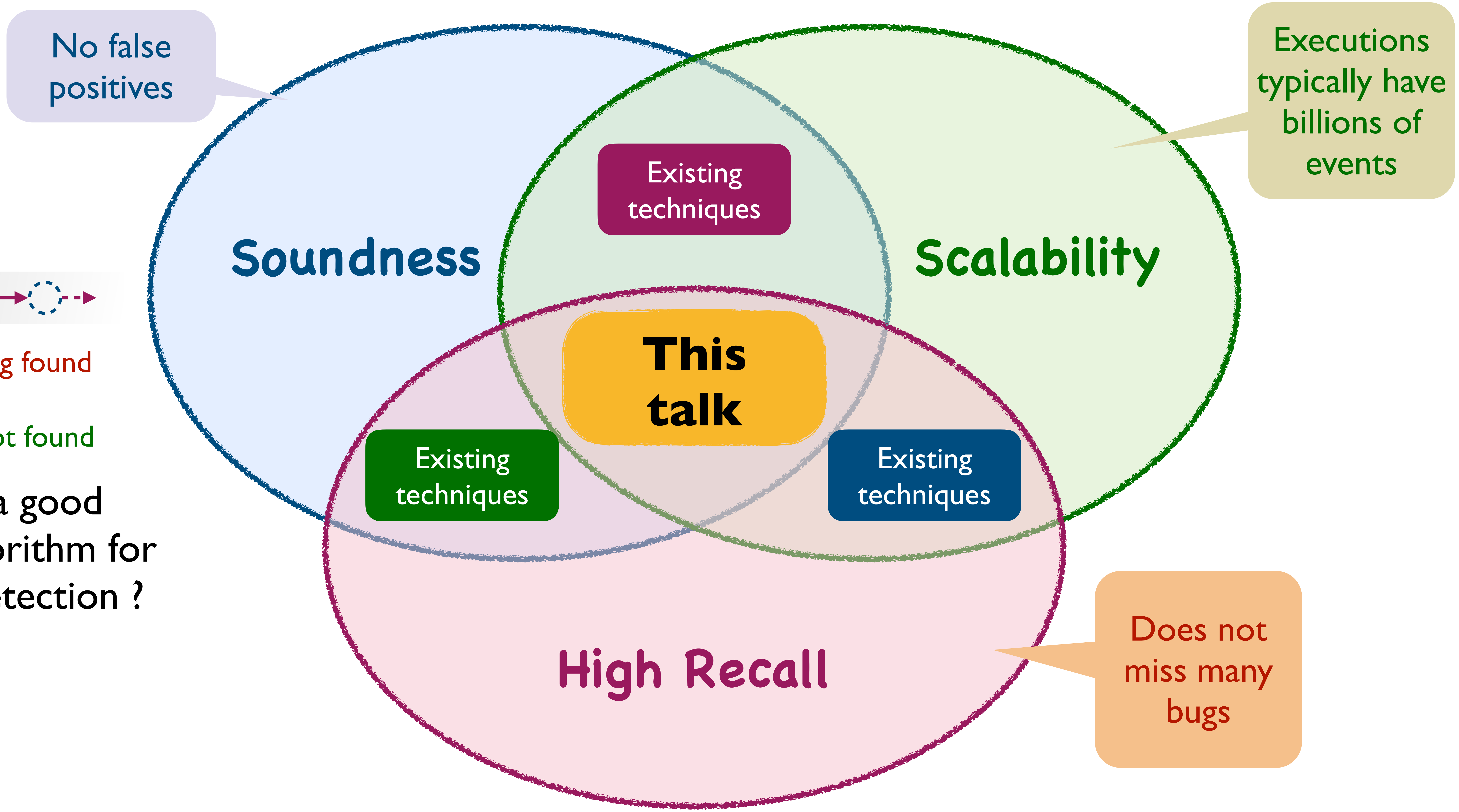
Race detected

Not detected

Algorithms for Data Race Detection



What constitutes a good Dynamic Analysis Algorithm for Concurrency Bug Detection ?



No false positives

Executions typically have billions of events

Soundness

Scalability

High Recall

Existing techniques

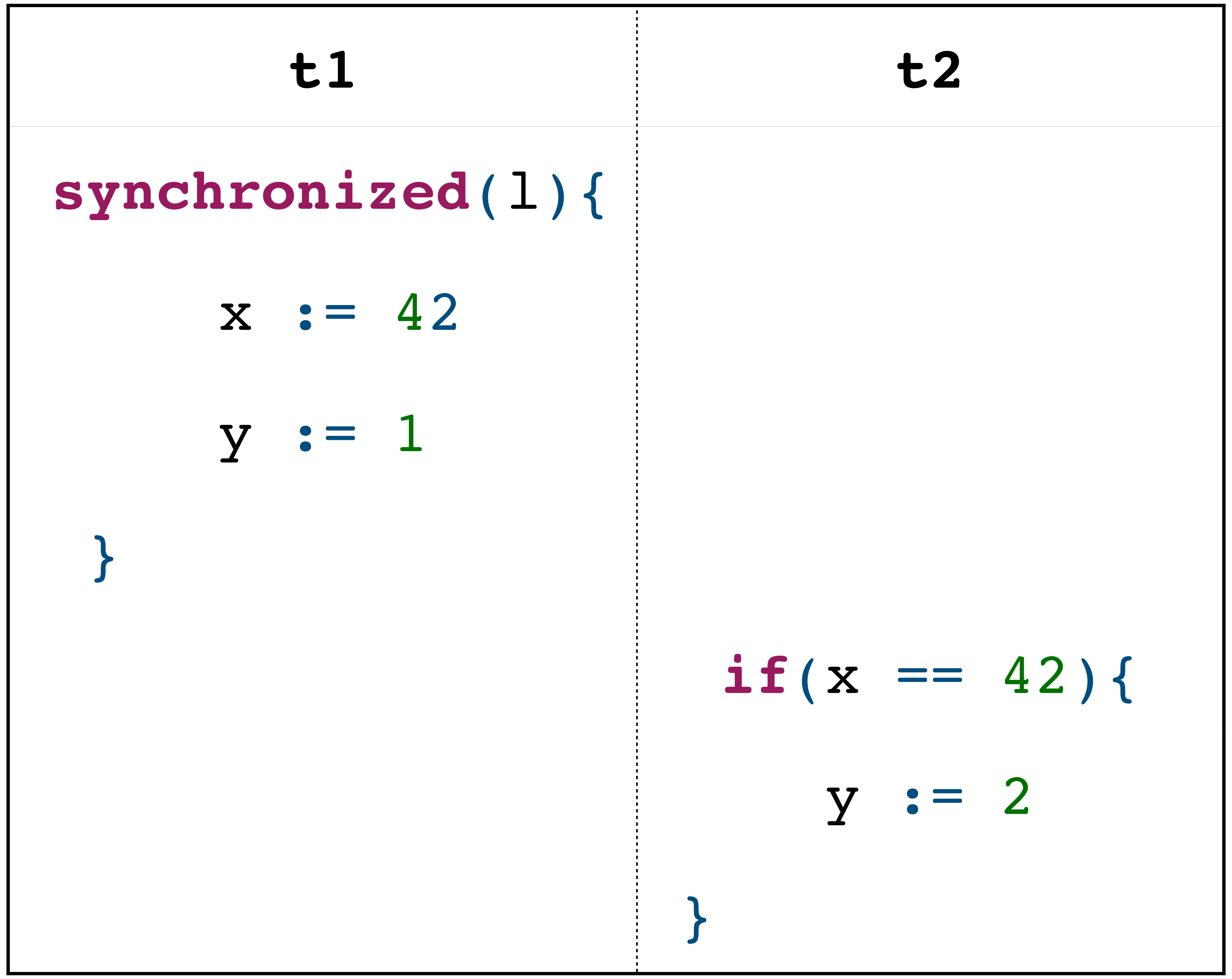
This talk

Existing techniques

Existing techniques

Does not miss many bugs

Concurrent Programs and Traces



t1	**t2**
`synchronized(l){`	
`x := 42`	
`y := 1`	
`}`	
	`if(x == 42){`
	`y := 2`
	`}`

Concurrent Program

- Threads
- Shared memory
- Locks for mutual exclusion
  - Critical sections cannot overlap

Concurrent Programs and Traces

```
t1                       t2

synchronized(l){
     x := 42
     y := 1
 }
                          if(x == 42){
                               y := 2
                          }
```

Concurrent Program

	t1	t2
1	acq(l)	
2	w(x)	
3	w(y)	
4		r(x)
5	rel(l)	
6		w(y)

Execution trace

Concurrent Programs and Traces

Event operations:

- Acquire and release of locks
- Access to memory locations

	t1	**t2**
1	`acq(l)`	
2	`w(x)`	
3	`w(y)`	
4		`r(x)`
5	`rel(l)`	
6		`w(y)`

Execution trace

Data Race Prediction : Fundamentals

Data Races

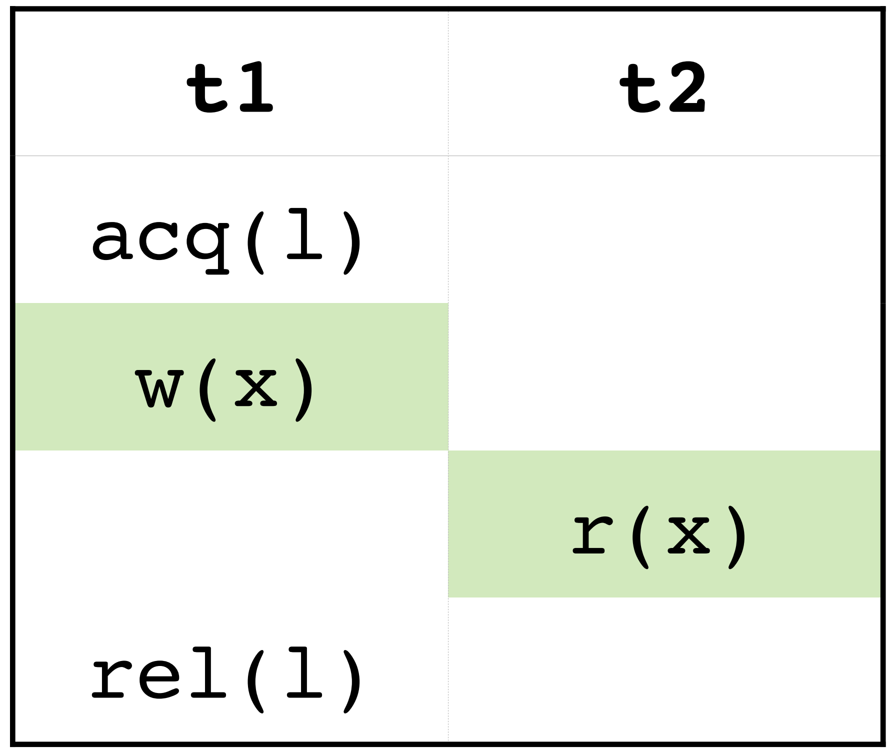
Data Race

An execution has data race if

- pair of conflicting events
- concurrent

1. Same memory location
2. Different threads
3. At least one write

Consecutive



t1	t2
acq(l)	
w(x)	
	r(x)
rel(l)	

Data Race Detection

(1) Execute program and observe trace

(2) Check if data race exists in the observed trace

Data Race Detection

Data Race

An execution has data race if

- pair of conflicting events
- consecutive

1. Same memory location
2. Different threads
3. At least one write

or concurrent

t1	**t2**
`acq(l)`	
`w(x)`	
	`r(x)`
`rel(l)`	

Data Race **Detection**

Prone to ***missing data races***:

- Executions are sensitive to thread scheduling
- Even multiple runs may not help

Can we do better?

Data Race **Prediction**

Data Race

An execution has data race if
- pair of conflicting events
- consecutive

1. Same memory location
2. Different threads
3. At least one write

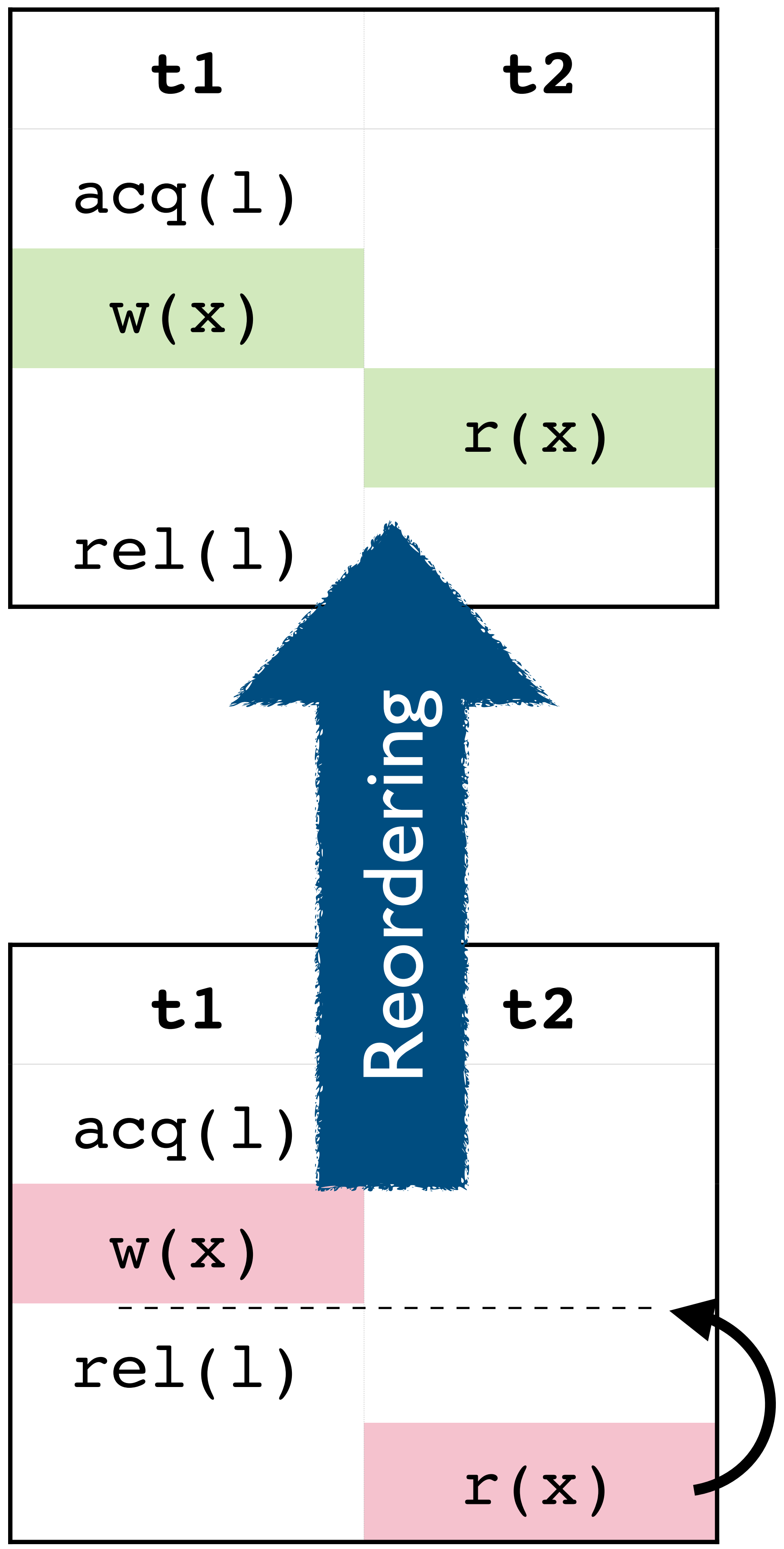
or concurrent

Data Race **Detection**

Prone to ***missing data races***:
- Executions are sensitive to thread scheduling
- Even multiple runs may not help

Data Race **Prediction**

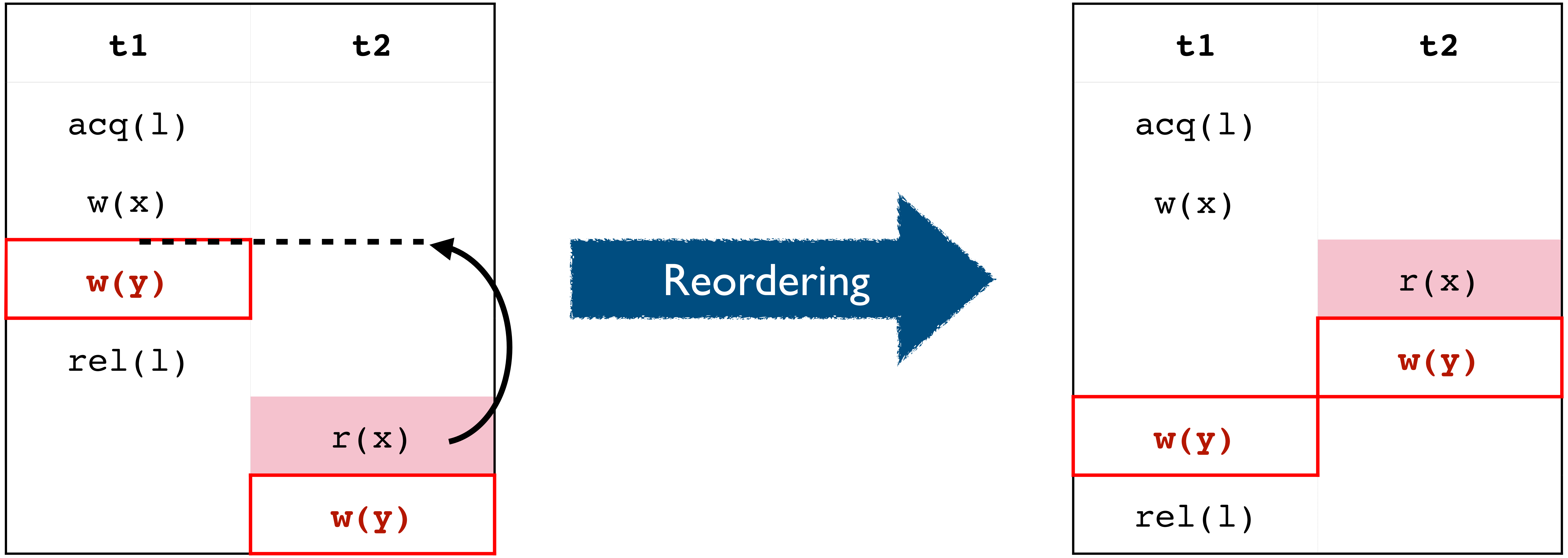
Reorder executions to expose data races



t1	**t2**
`acq(l)`	
`w(x)`	
	`r(x)`
`rel(l)`	

t1	**t2**
`acq(l)`	
`w(x)`	
`rel(l)`	
	`r(x)`

Which reorderings are allowed?

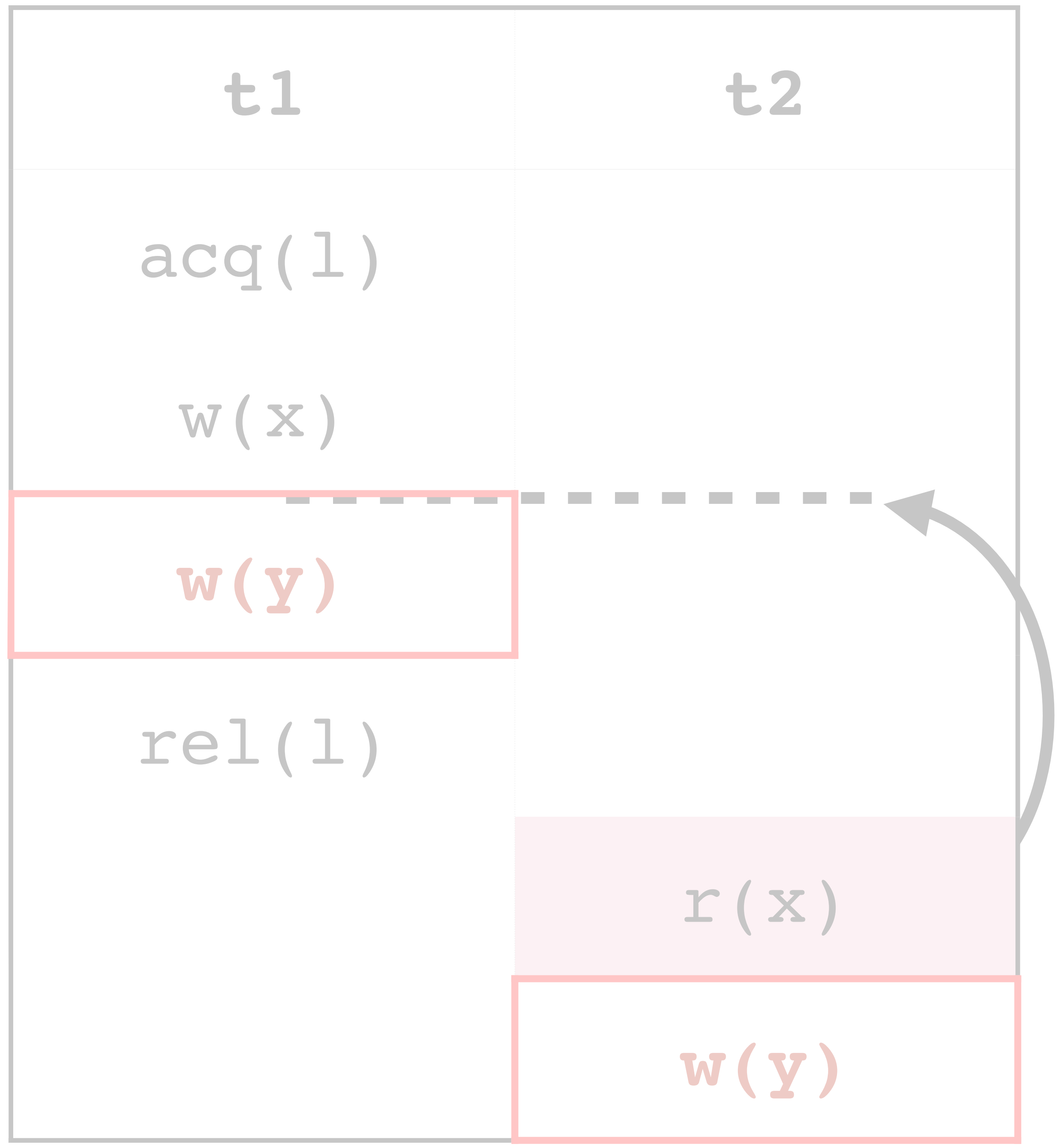


t1	t2
acq(l)	
w(x)	
w(y)	
rel(l)	
	r(x)
	w(y)

t1	t2
acq(l)	
w(x)	
	r(x)
	w(y)
w(y)	
rel(l)	

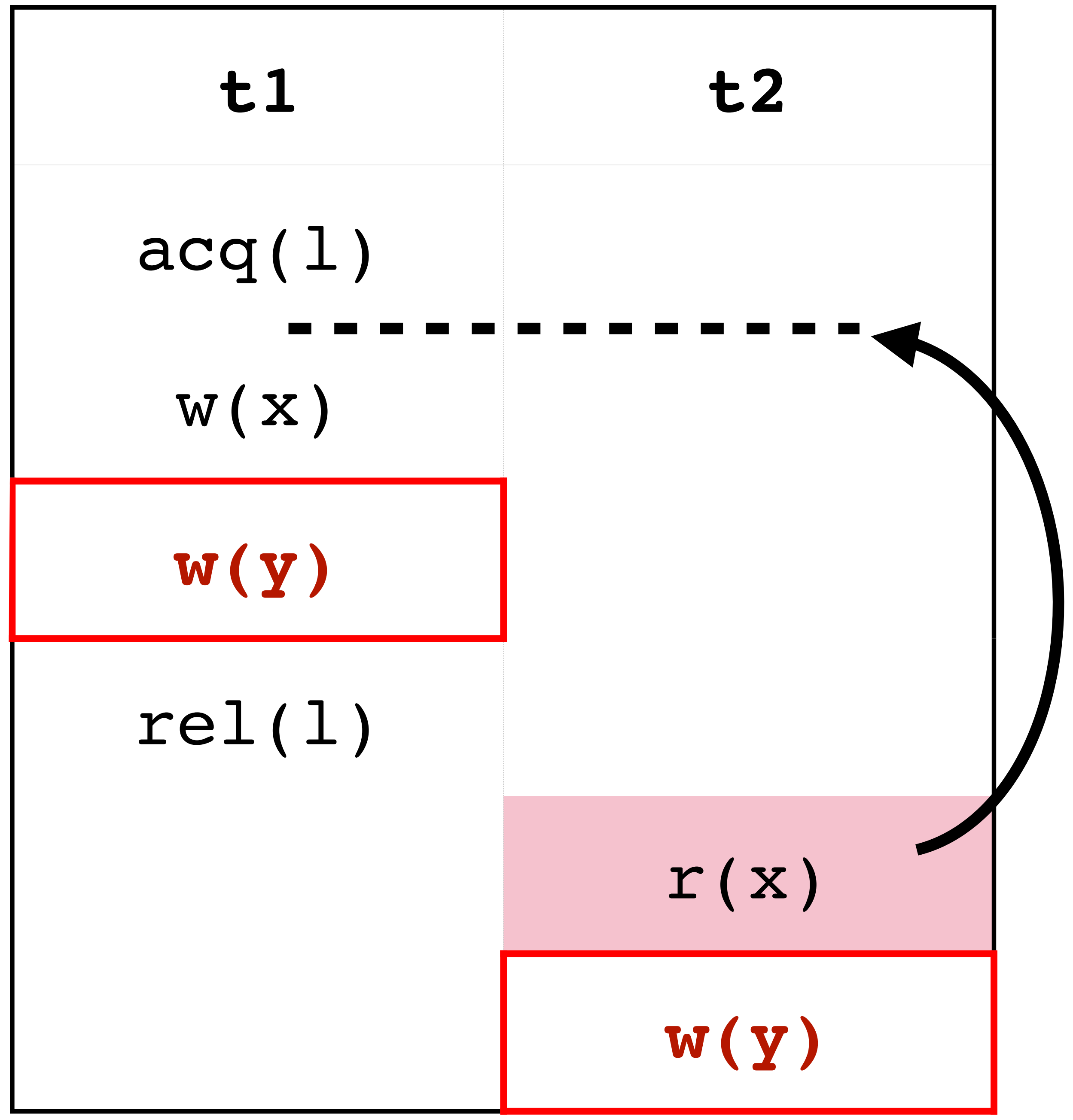
Reordering 1 ✓

Which reorderings are allowed?



t1	t2
acq(l)	
w(x)	
w(y)	
rel(l)	
	r(x)
	w(y)

Reordering 1 ✓



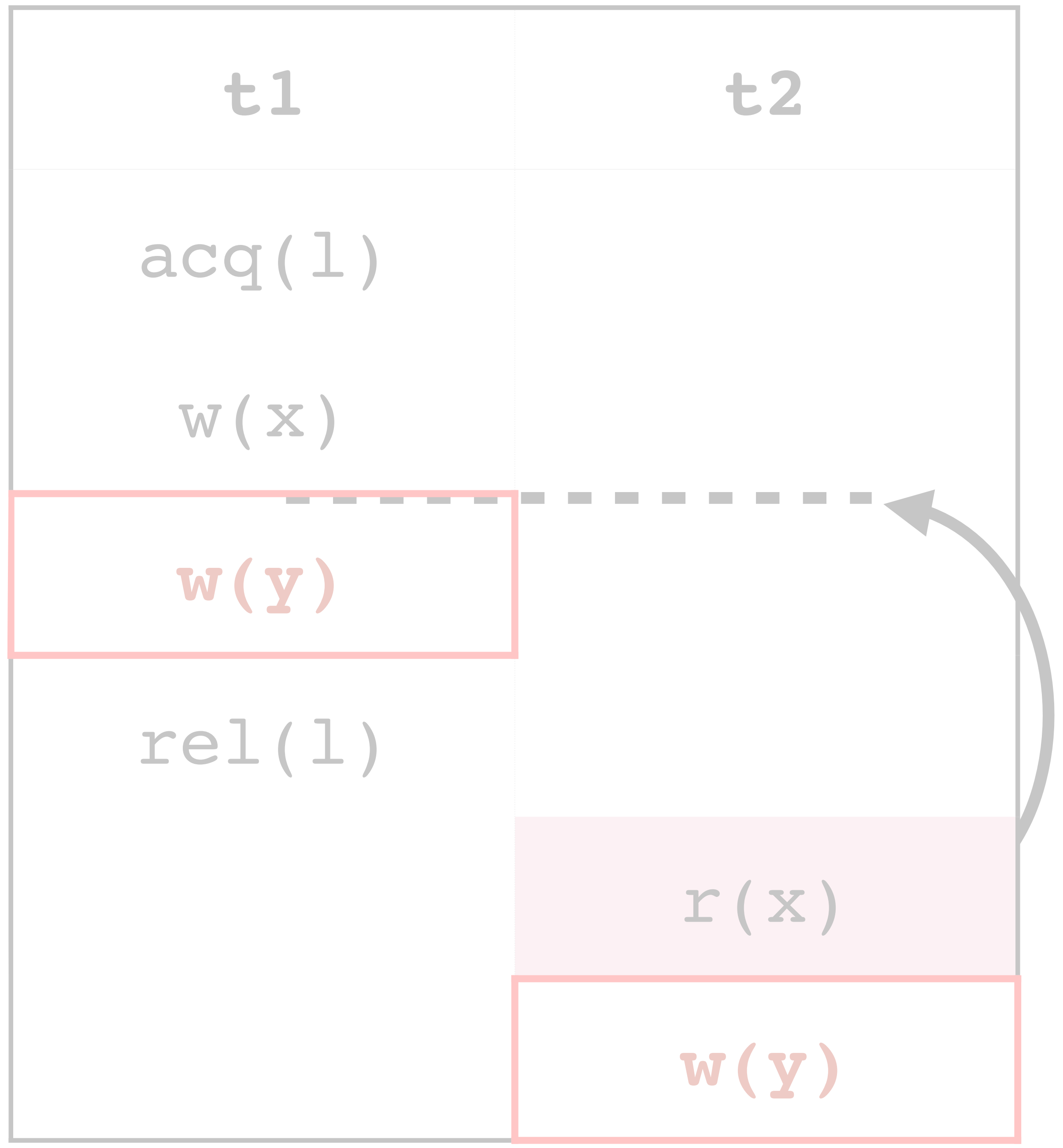
t1	t2
acq(l)	
w(x)	
w(y)	
rel(l)	
	r(x)
	w(y)

Reordering 2 ?

Source agnostic analysis: some reorderings may not be allowed in some programs

Which reorderings are allowed?

Initially, `x == 0` and `y == 0`

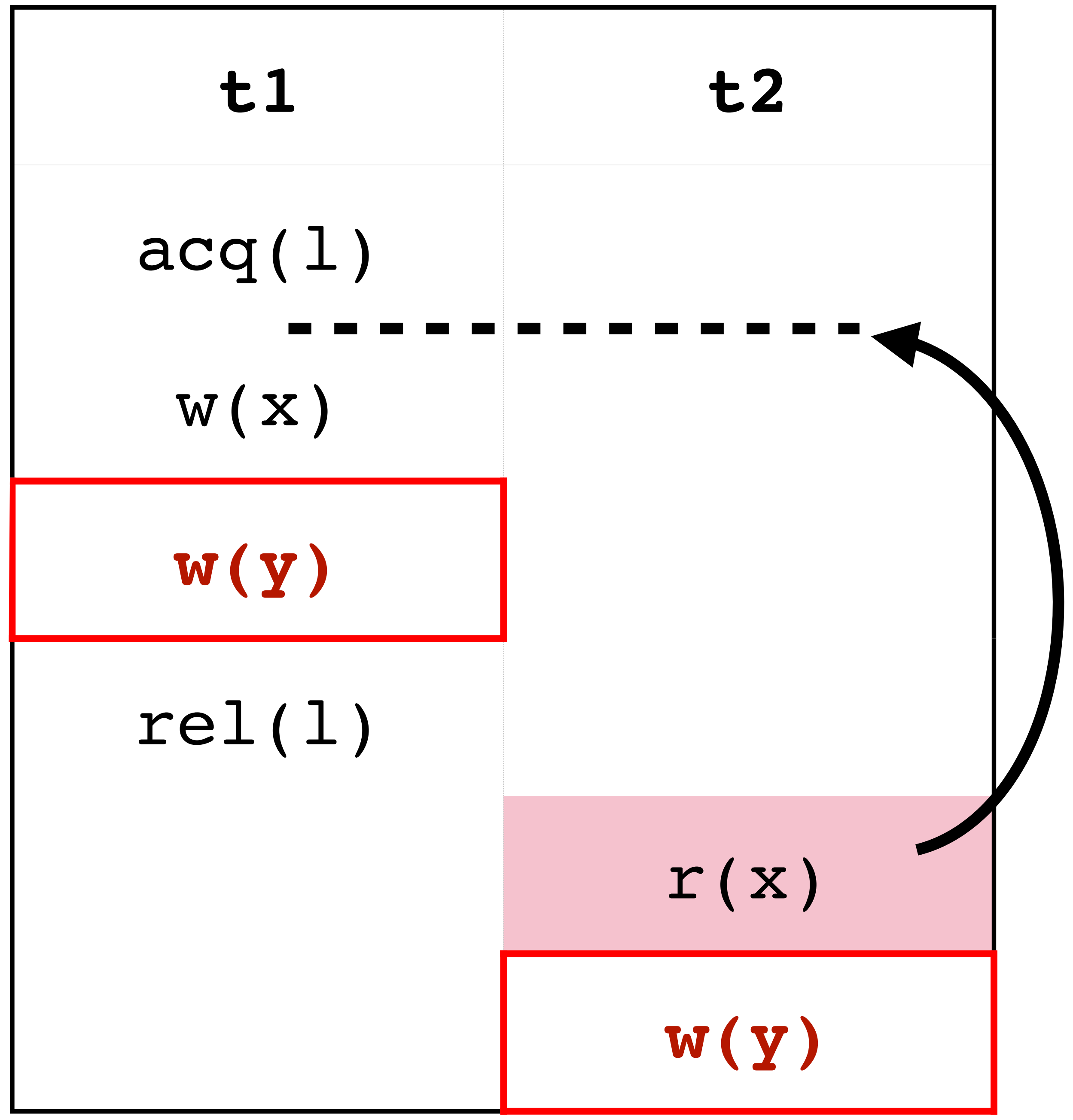


t1	t2
acq(l)	
w(x)	
w(y)	
rel(l)	
	r(x)
	w(y)

Reordering 1 ✔

```
t1                        t2
sync(l) {
      x := 42
      y := 1
 }
                            if(x == 42){
                                y := 2
                            }
```

Possible source program



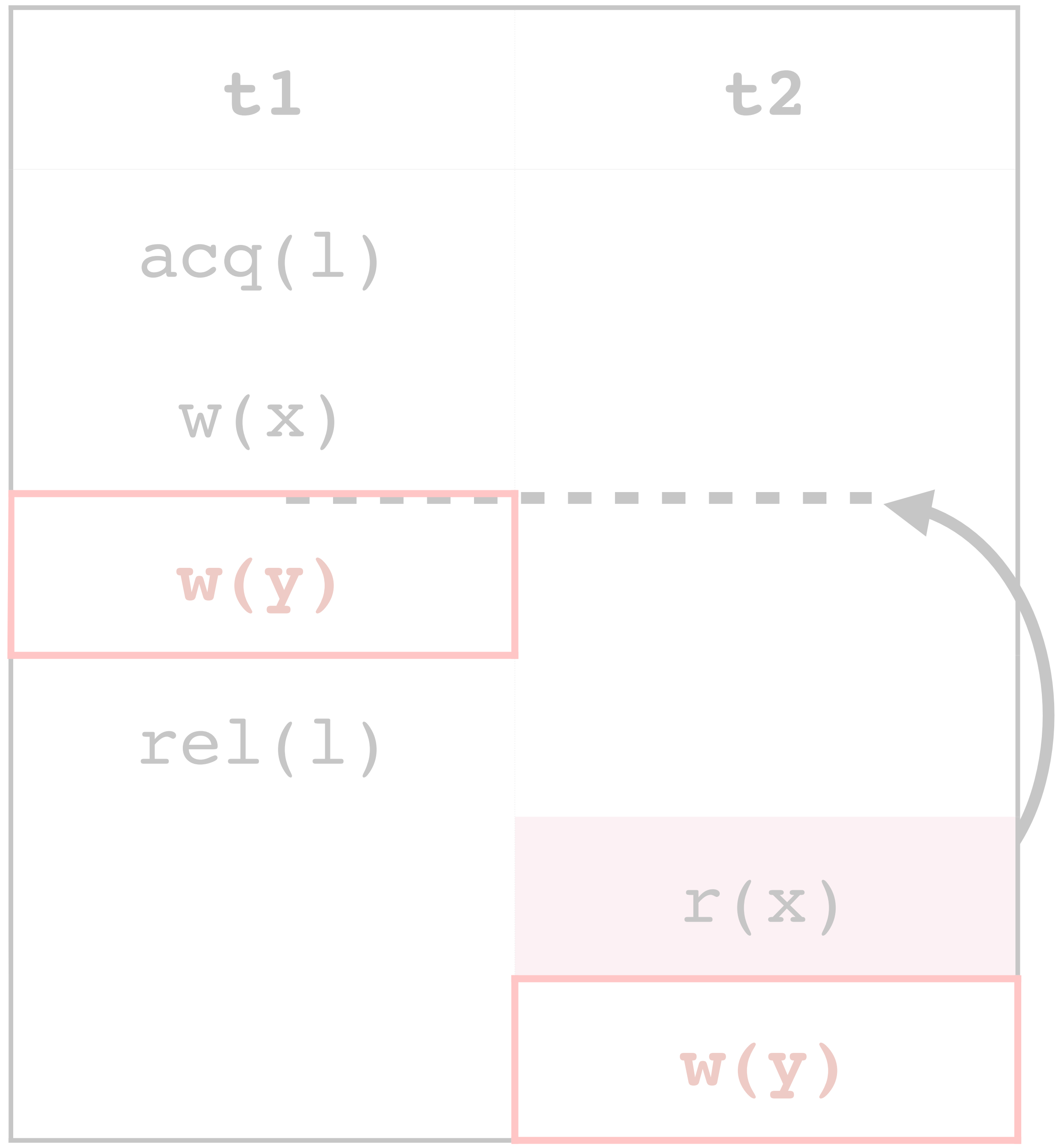
t1	t2
acq(l)	
w(x)	
w(y)	
rel(l)	
	r(x)
	w(y)

Reordering 2 ✗

Source agnostic analysis: some reorderings may not be allowed in some programs

Which reorderings are allowed?

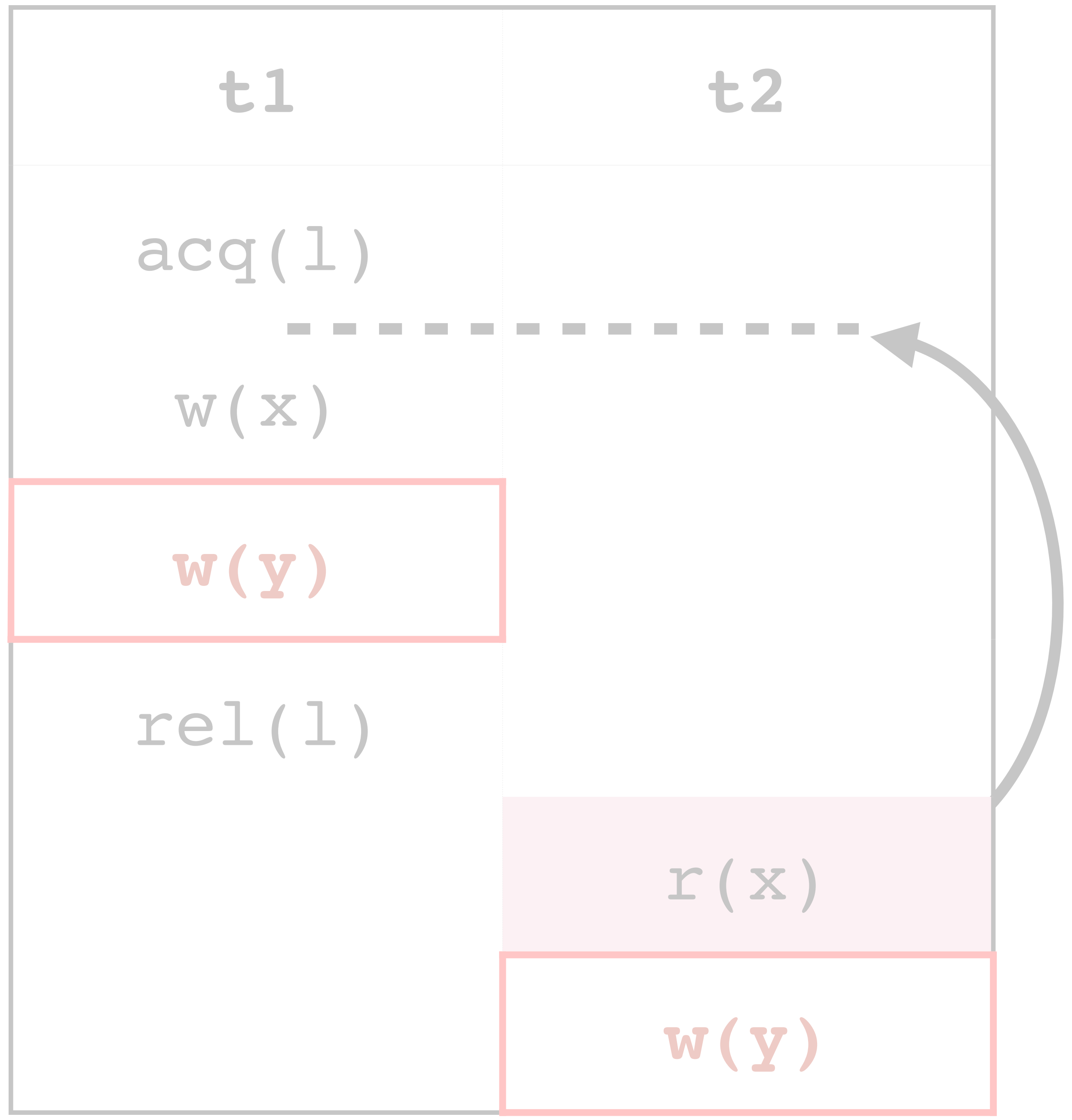
Initially, `x == 0` and `y == 0`



Reordering 1 ✔

```
t1                 t2
sync(l) {
     x := 42
     y := 1
 }
                    if(x == 42){
                        y := 2
                    }
```

Possible source program



Reordering 2 ✘

Source agnostic analysis: some reorderings are ***always*** allowed

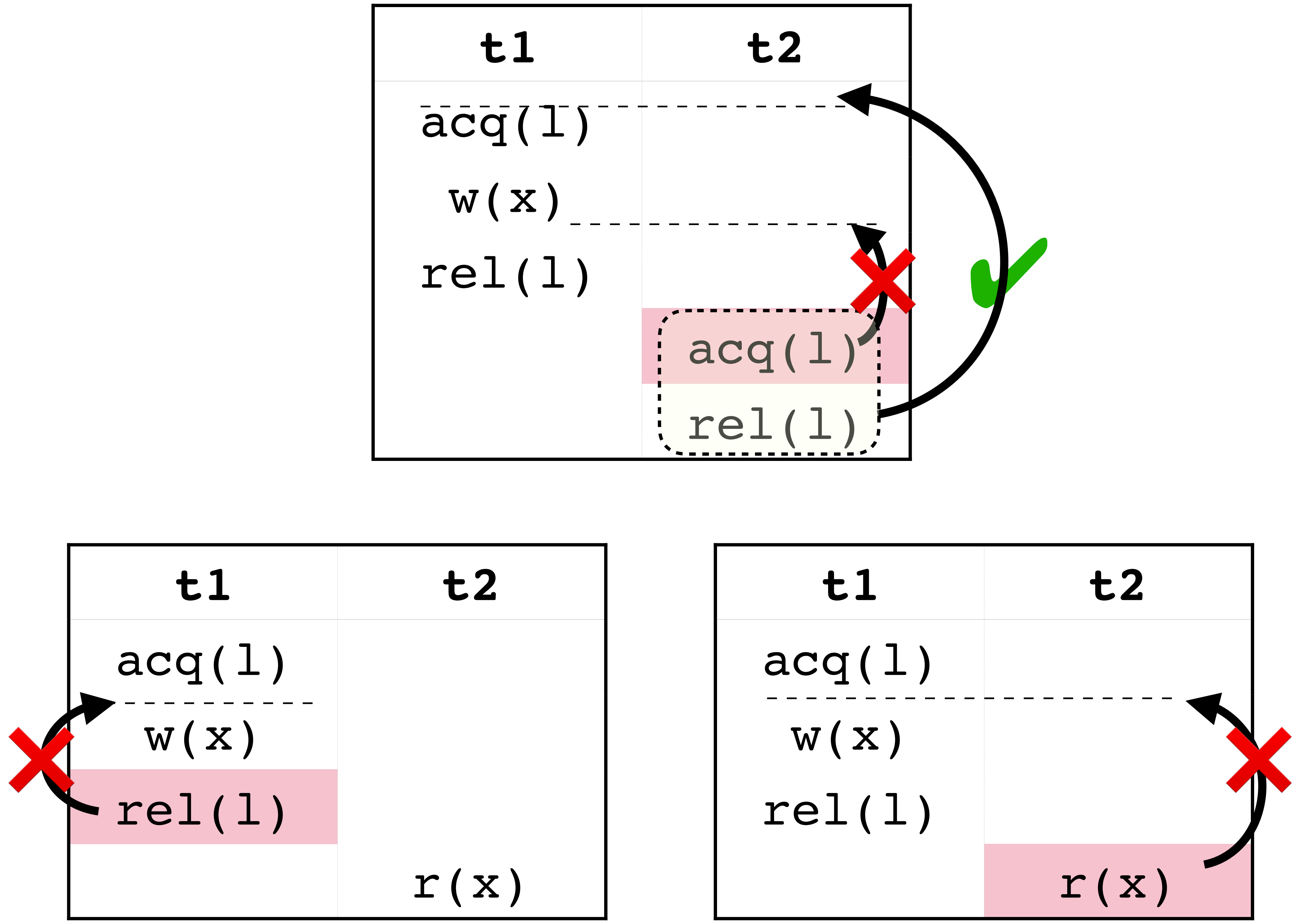
Which reorderings are allowed?

Any program that generates the observed execution must also generate the reordering

Correct reordering*

Reorderings must satisfy some properties -

1. Preserve lock semantics
   - critical sections on same lock don’t overlap
2. Preserve intra-thread ordering
3. Preserve control flow
   - Every read sees its original write



t1	t2
acq(l)	
w(x)	
rel(l)	
	acq(l)
	rel(l)

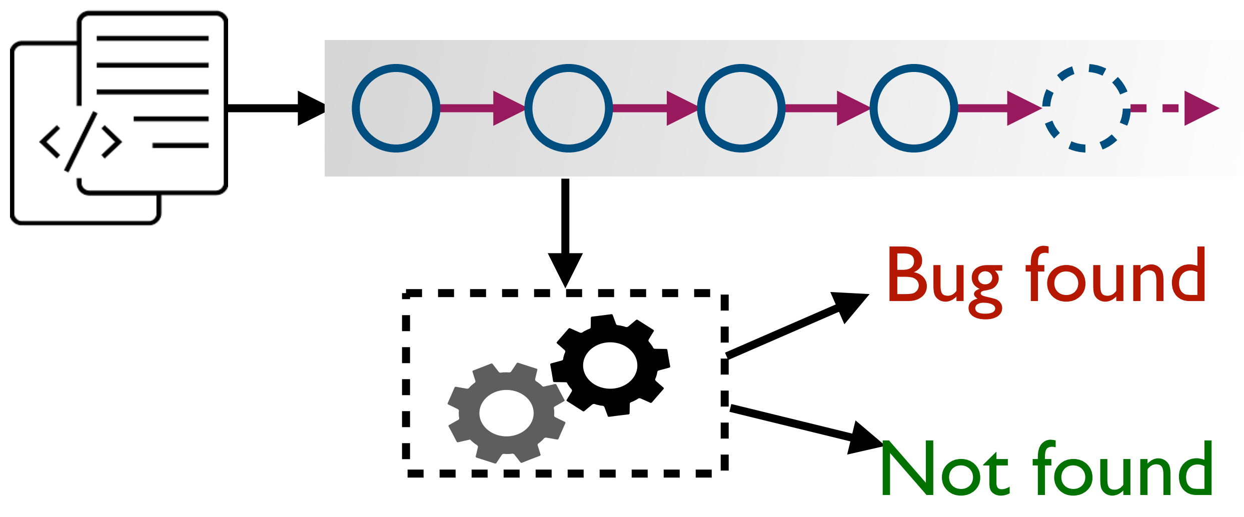
t1	t2
acq(l)	
w(x)	
rel(l)	
	r(x)

t1	t2
acq(l)	
w(x)	
rel(l)	
	r(x)

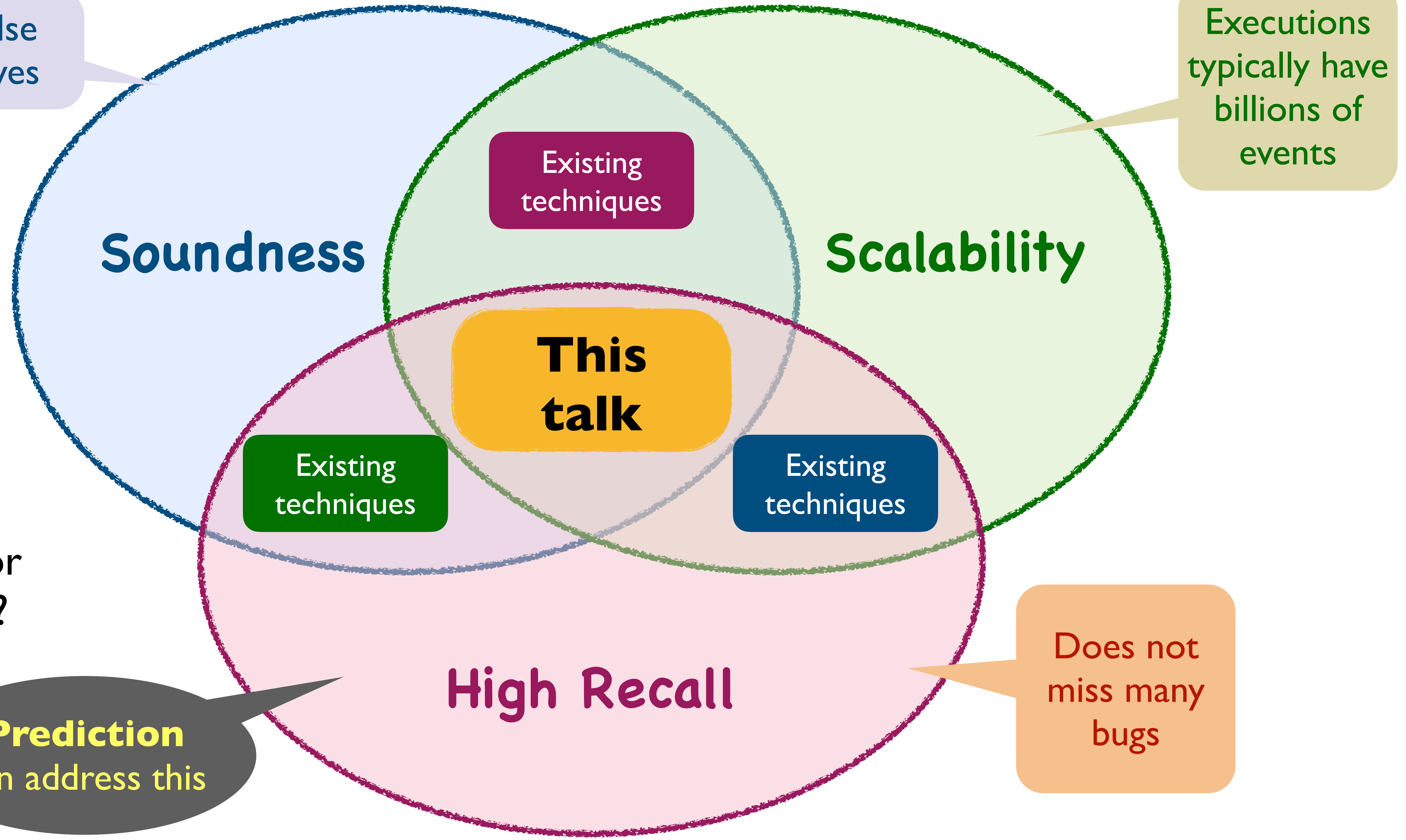
---

* Şerbănuţă et al, Maximal Causal Models for Sequentially Consistent Systems, RV 2012

Algorithms for Data Race Detection



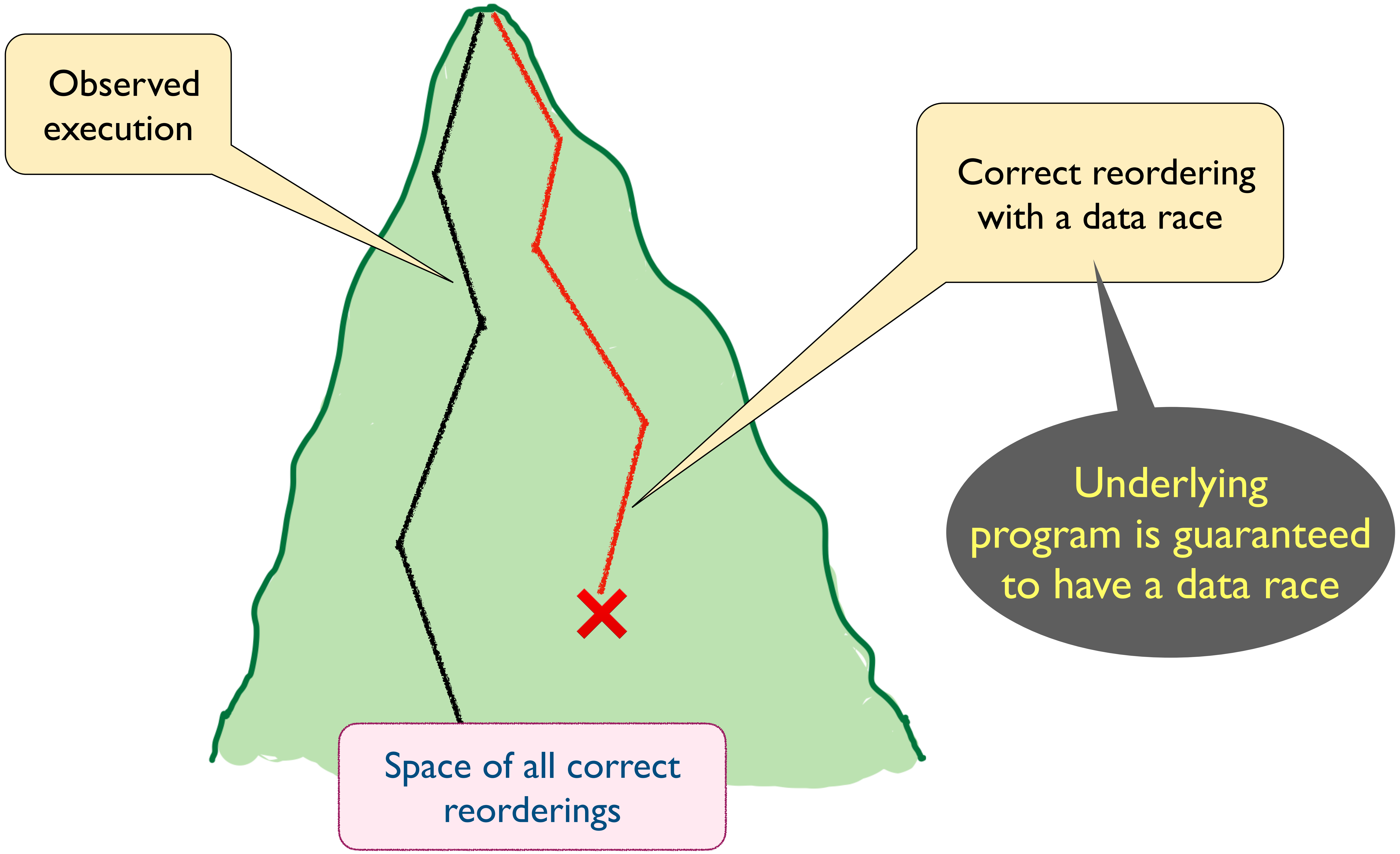
What constitutes a good Dynamic Analysis Algorithm for Concurrency Bug Detection ?



Data Race Prediction

Data Race Prediction

Given an execution σ, is there a **correct reordering** with a data race?



Observed execution

Correct reordering with a data race

Underlying program is guaranteed to have a data race

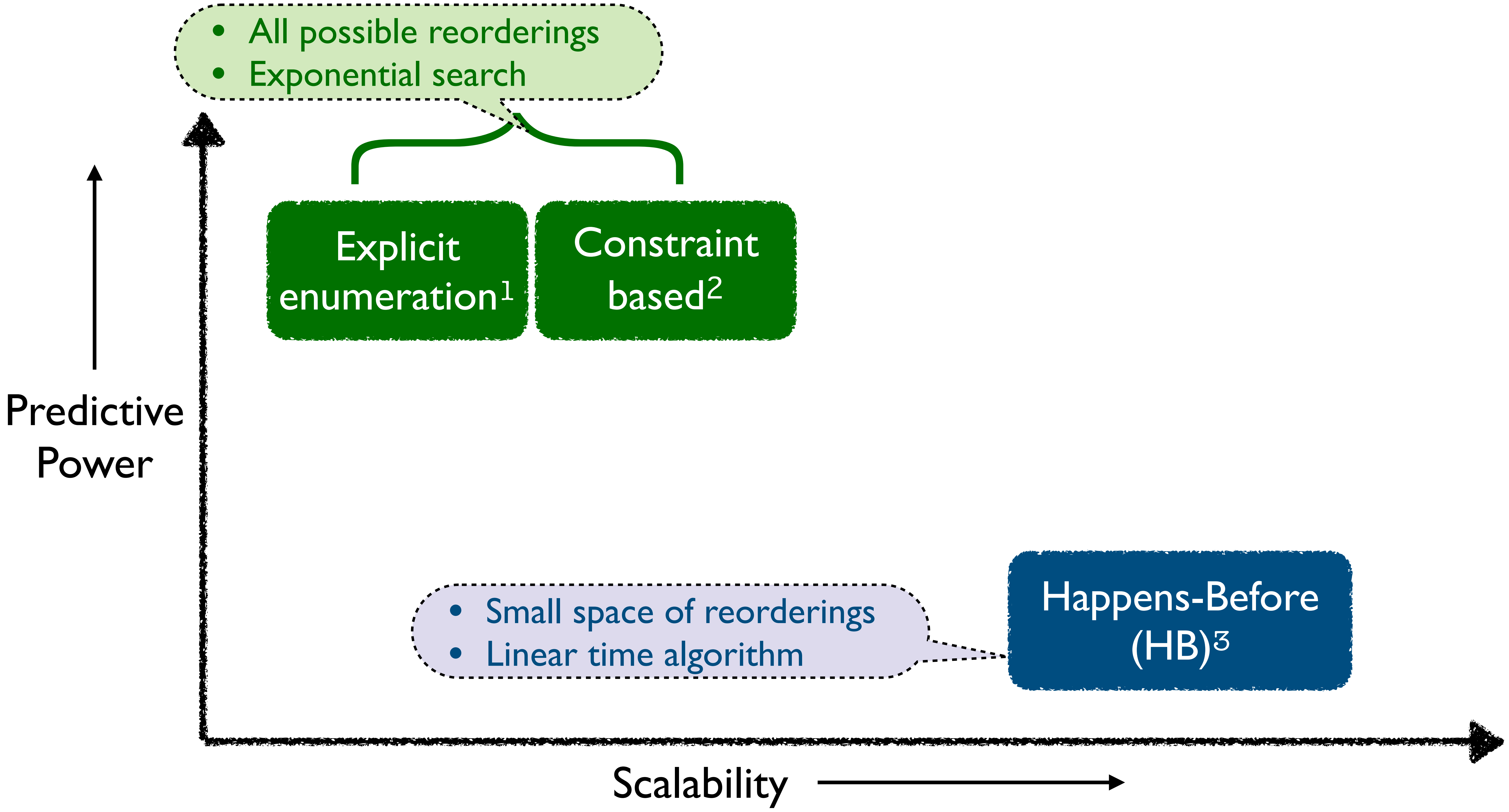
Space of all correct reorderings

Data Race Prediction : Prior Techniques

Data Race Prediction

Given an execution σ, is there a **correct reordering** with a data race?

Existing Techniques


- All possible reorderings
- Exponential search

Explicit enumeration[1]

Constraint based[2]

Predictive Power

- Small space of reorderings
- Linear time algorithm

Happens-Before (HB)[3]

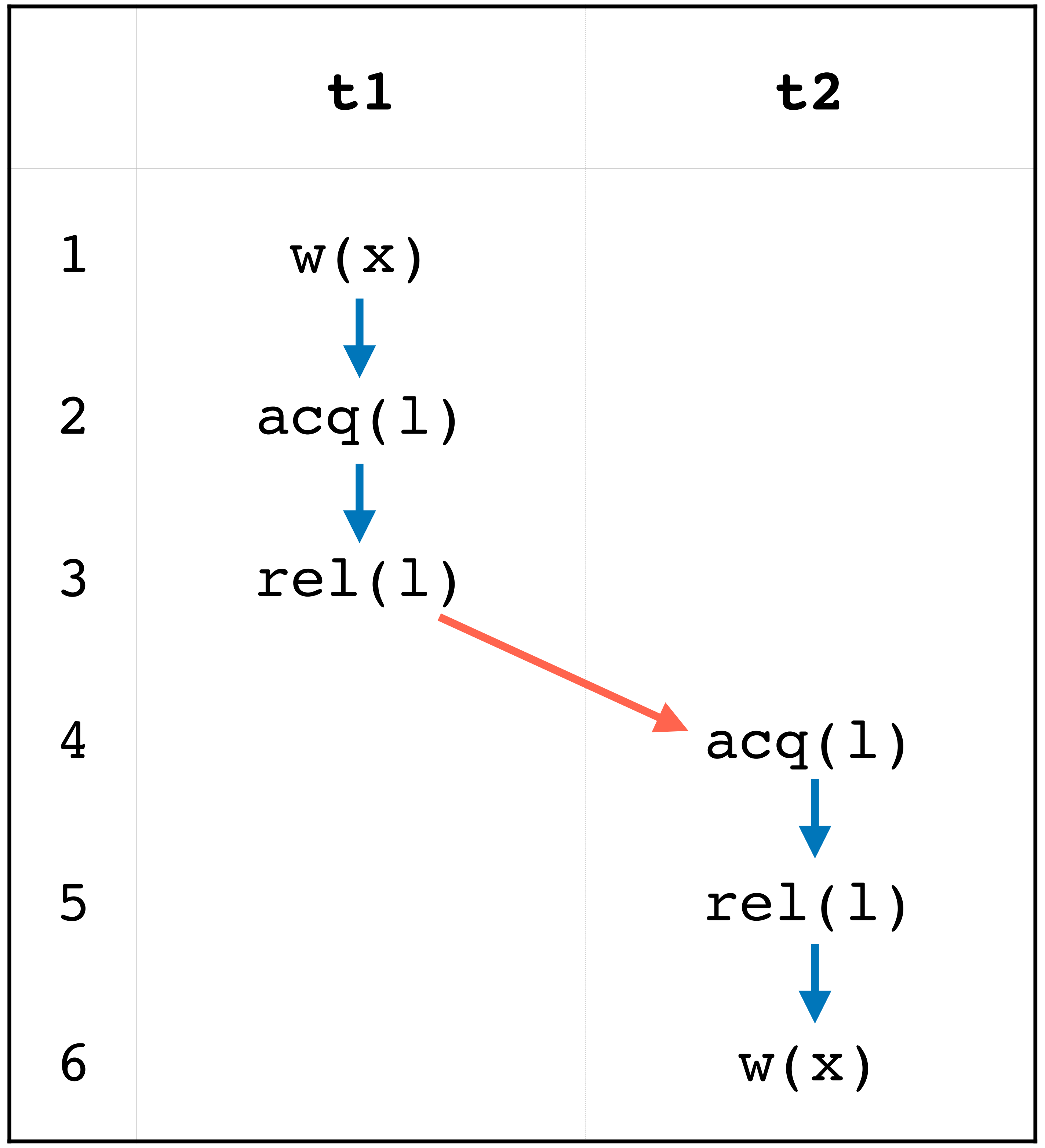
Scalability

---

1. Sen et. al., Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions, FMOODS 2005
2. Said et. al., Generating Data Race Witnesses by an SMT-Based Analysis, NFM 2011
3. Lamport, Time, clocks, and the ordering of events in a distributed system, CACM 1978

Happens-Before

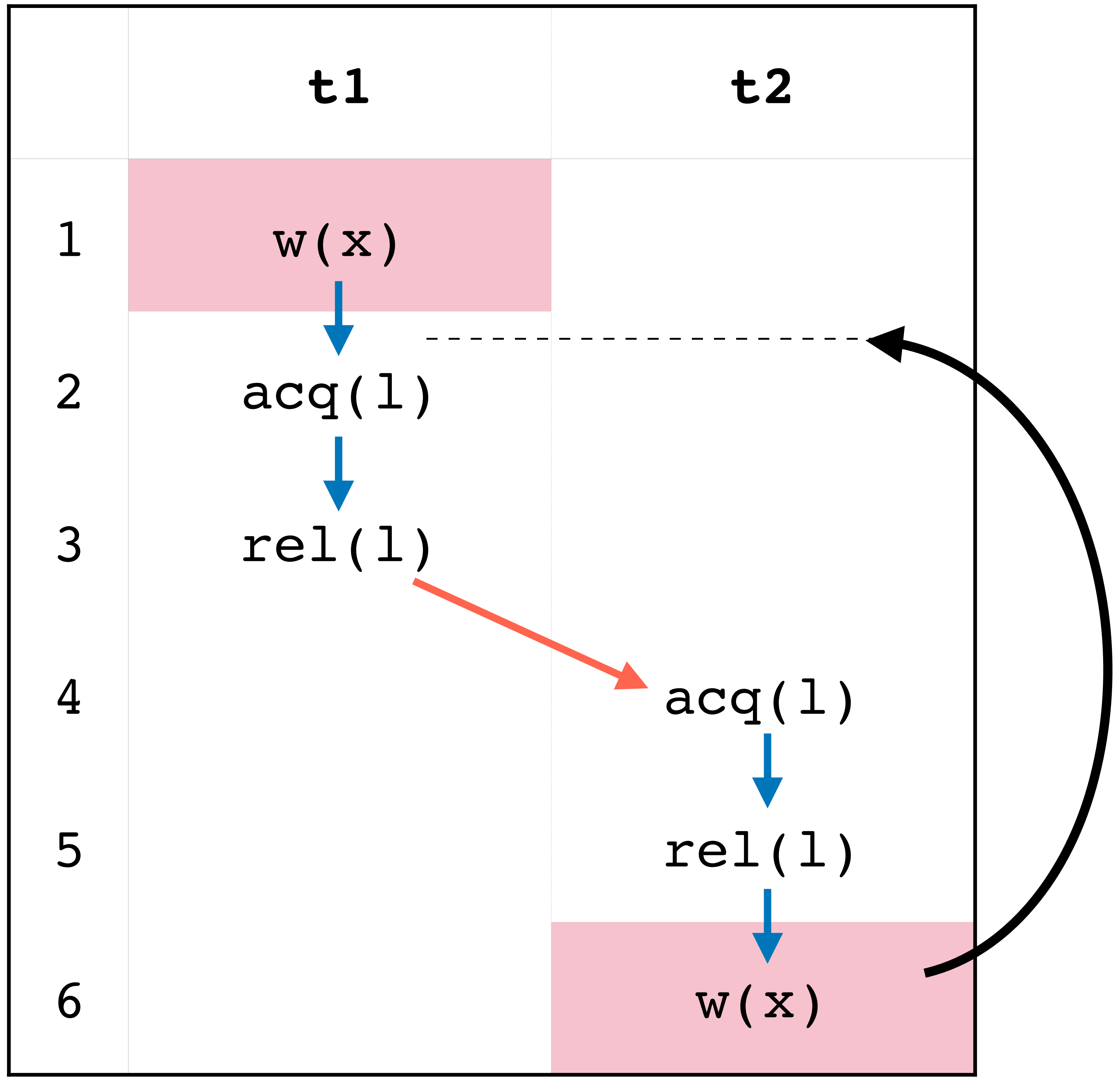
- $\leq_{HB}$ orders events of an execution σ as follows
  1. Intra-thread ordering
  2. Critical sections on the same lock are ordered as in σ
     - release of earlier to acquire of later
- Race if *conflicting* events are not ordered
- Sound - no false alarms
- Race detection algorithm
  - Linear time
  - One pass streaming (does not store the trace)



	t1	t2
1	w(x)	
2	acq(l)	
3	rel(l)	
4		acq(l)
5		rel(l)
6		w(x)

Happens-Before

- $\leq_{HB}$ orders events of an execution σ as follows
  1. Intra-thread ordering
  2. Critical sections on the same lock are ordered as in σ
     - release of earlier to acquire of later
- Race if *conflicting* events are not ordered
- Sound - no false alarms
- Race detection algorithm
  - Linear time
  - One pass streaming (does not store the trace)

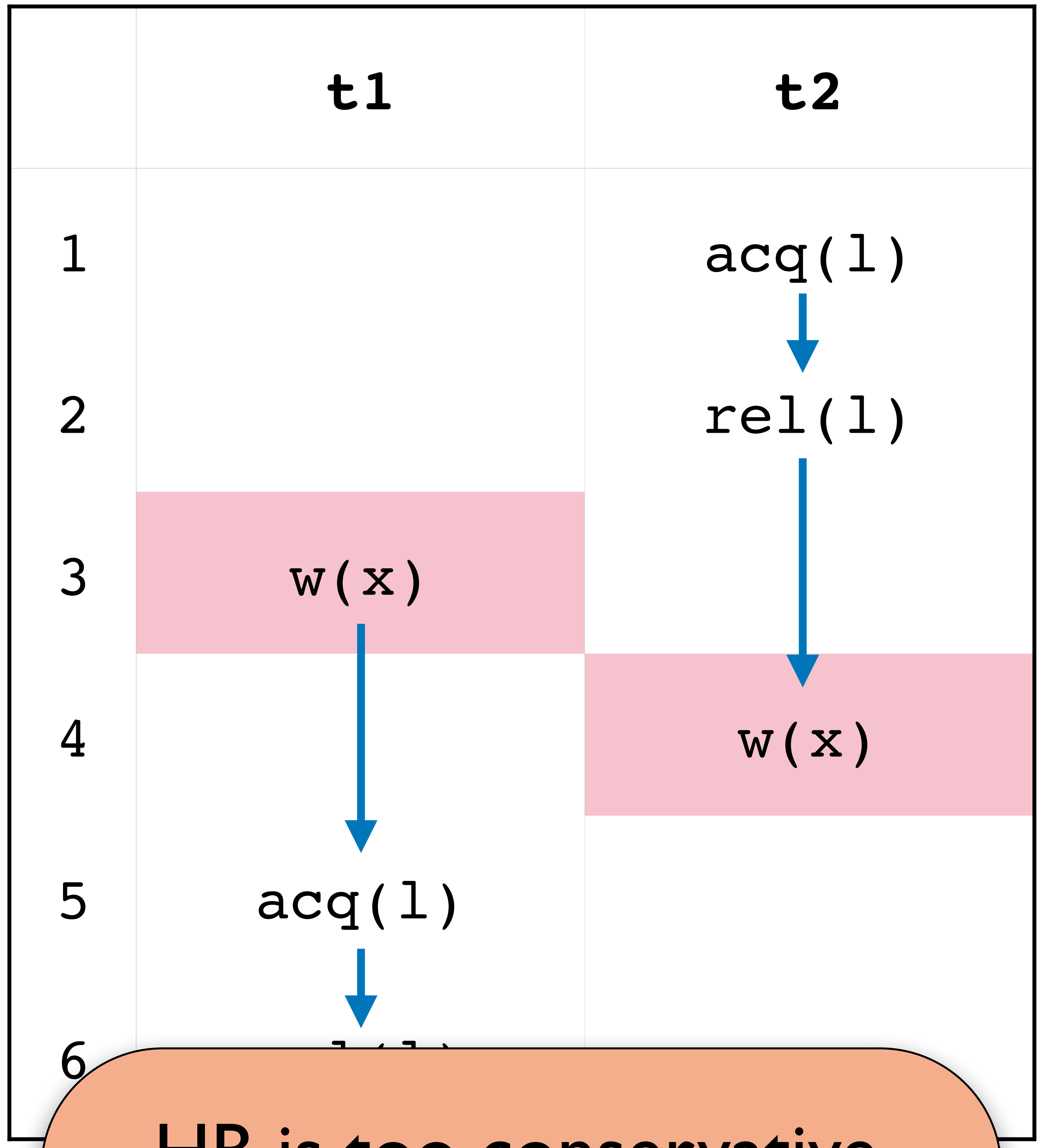


	t1	t2
1	w(x)	
2	acq(l)	
3	rel(l)	
4		acq(l)
5		rel(l)
6		w(x)

No race reported by HB

Happens-Before

- $\leq_{HB}$ orders events of an execution σ as follows
  1. Intra-thread ordering
  2. Critical sections on the same lock are ordered as in σ
     - release of earlier to acquire of later
- Race if *conflicting* events are not ordered
- Sound - no false alarms
- Race detection algorithm
  - Linear time
  - One pass streaming (does not store the trace)



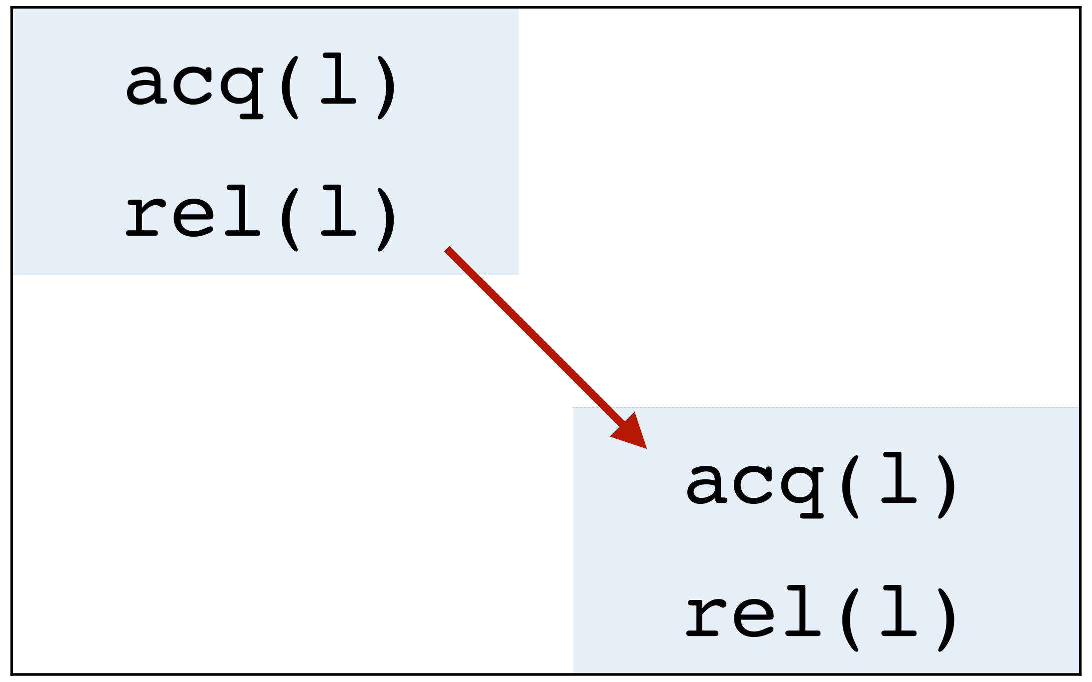
	t1	**t2**
1		`acq(l)`
2		`rel(l)`
3	`w(x)`	
4		`w(x)`
5	`acq(l)`	
6		

HB is too conservative.
Misses simple data races

Weak Causal Precedence[†]

[†] Dynamic Race Prediction in Linear Time, **PLDI 2017**

Tackling the Conservativeness of HB



$\leq_{HB}$ orders all critical sections on the same lock

- Space of reorderings = all linearizations of $\leq_{HB}$
- $\leq_{HB}$ orders too many events

- Can we relax some HB-orderings?
  - Naively ⇒ infeasible reorderings
  - Careful analysis ⇒ expensive

Can we balance soundness, and scalability and still get better prediction power than HB?

Weak Causal Precedence[†]

WCP identifies when to order critical sections on common lock

- $<_{WCP}$ orders events of an execution σ as follows

If critical sections contain events conflicting events, then they can't be reordered

[†] Dynamic Race Prediction in Linear Time, **PLDI 2017**

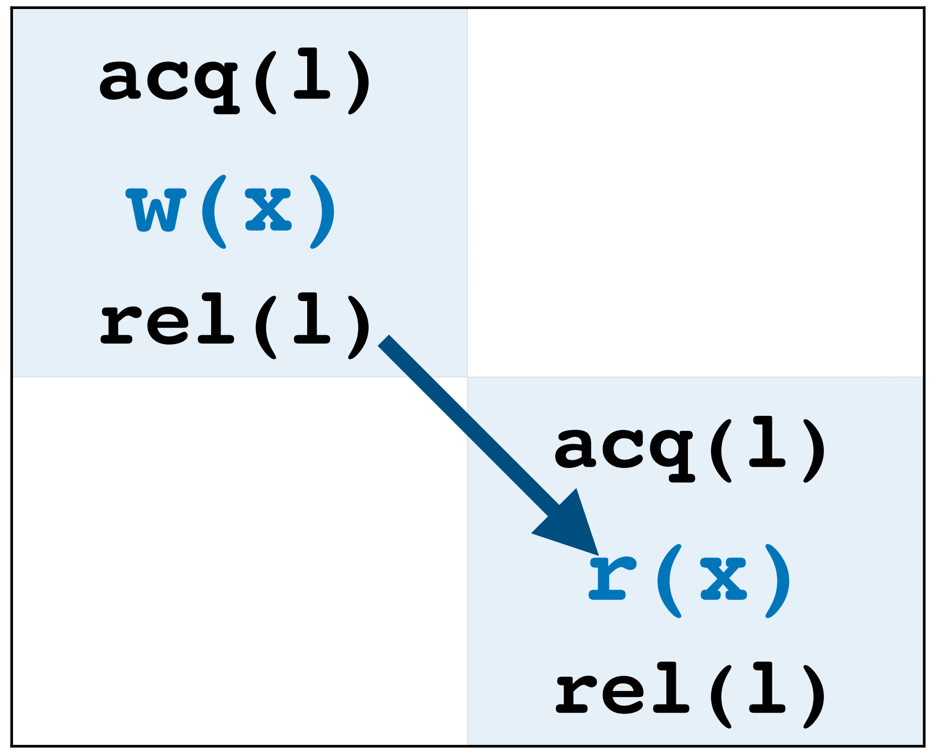
Weak Causal Precedence[†]

WCP identifies when to order critical sections on common lock

- $<_{WCP}$ orders events of an execution σ as follows

1. Critical sections C_1, C_2 on same lock are ordered when they contain conflicting events $e_1 \in C_1$, $e_2 \in C_2$:

$$\texttt{rel}(C_1) <_{WCP} e_2$$



```
acq(l)
w(x)
rel(l)
          acq(l)
          r(x)
          rel(l)
```

[†] Dynamic Race Prediction in Linear Time, **PLDI 2017**

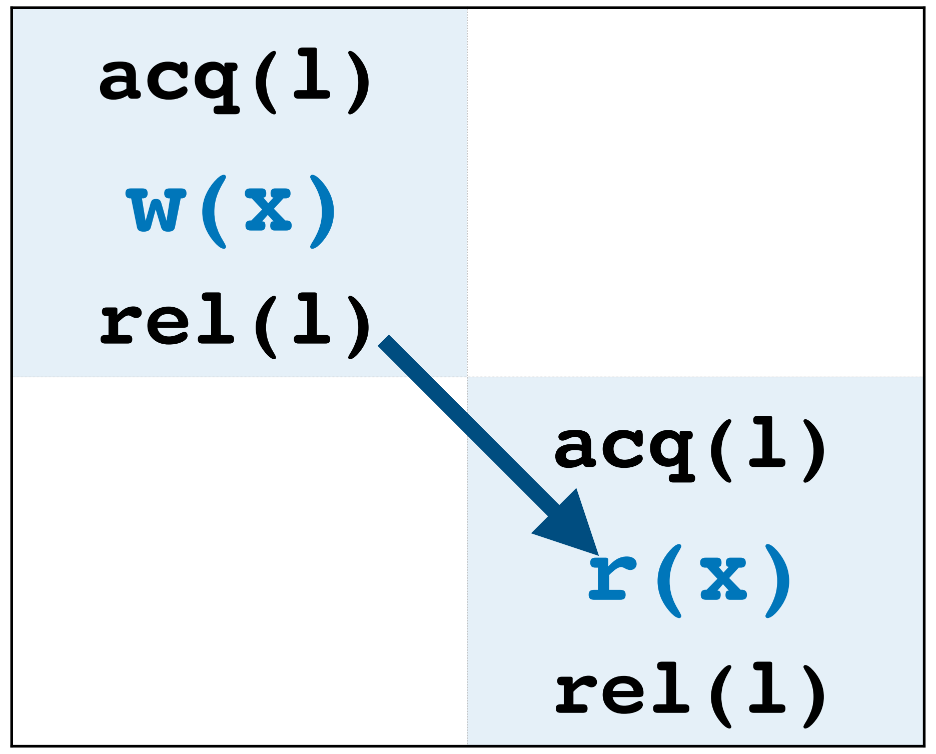
Weak Causal Precedence†

WCP identifies when to order critical sections on common lock

- $<_{\mathrm{WCP}}$ orders events of an execution σ as follows

1. Critical sections C_1, C_2 on same lock are ordered when they contain conflicting events $e_1 \in C_1, e_2 \in C_2$:

$$\texttt{rel}(C_1) <_{\mathrm{WCP}} e_2$$



```
acq(l)
w(x)
rel(l)
          acq(l)
          r(x)
          rel(l)
```

If critical sections contain events (inductively) ordered by WCP, then they can't be reordered.

---

†Dynamic Race Prediction in Linear Time, **PLDI 2017**

Weak Causal Precedence†

WCP identifies when to order critical sections on common lock

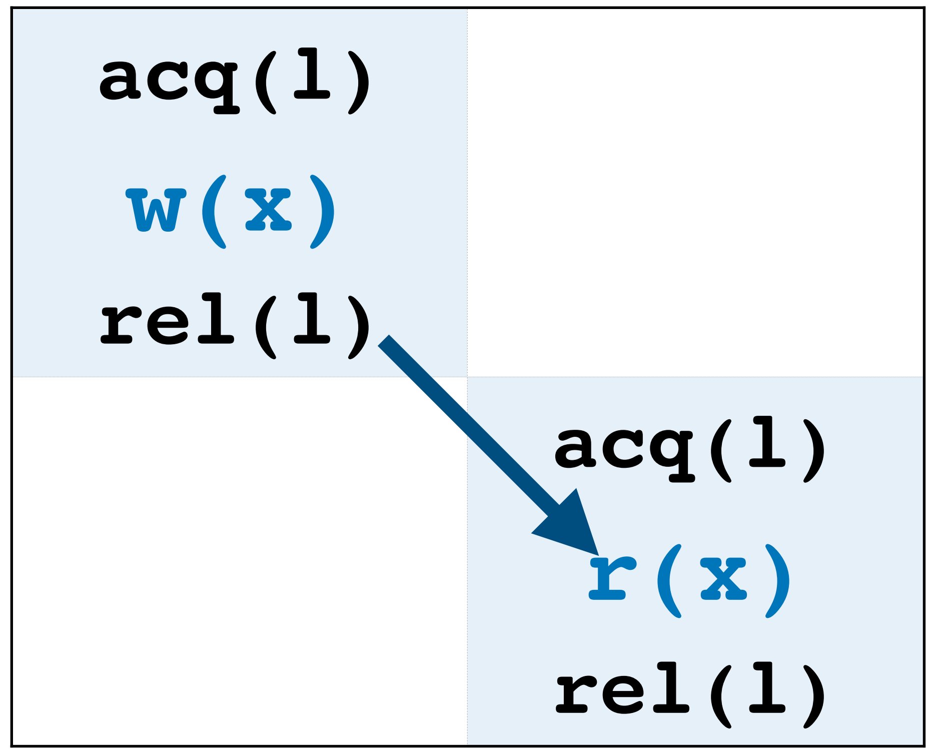
- $<_{WCP}$ orders events of an execution σ as follows

1. Critical sections C_1, C_2 on same lock are ordered when they contain conflicting events $e_1 \in C_1, e_2 \in C_2$:

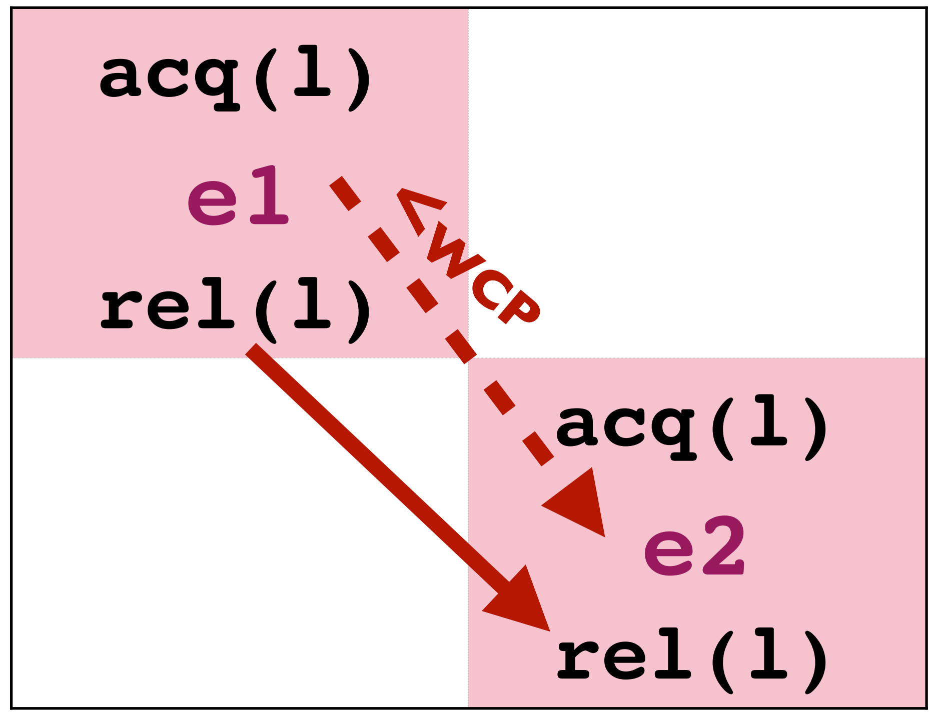
   $\texttt{rel}(C_1) <_{WCP} e_2$

2. Critical sections C_1, C_2 on same lock are ordered when they contain events $e_1 \in C_1, e_2 \in C_2$ ordered by WCP :

   $\texttt{rel}(C_1) <_{WCP} \texttt{rel}(C_2)$.



acq(l)
w(x)
rel(l)
acq(l)
r(x)
rel(l)



acq(l)
e1
rel(l)
$<_{WCP}$
acq(l)
e2
rel(l)

†Dynamic Race Prediction in Linear Time, **PLDI 2017**

Weak Causal Precedence†

WCP identifies when to order critical sections on common lock

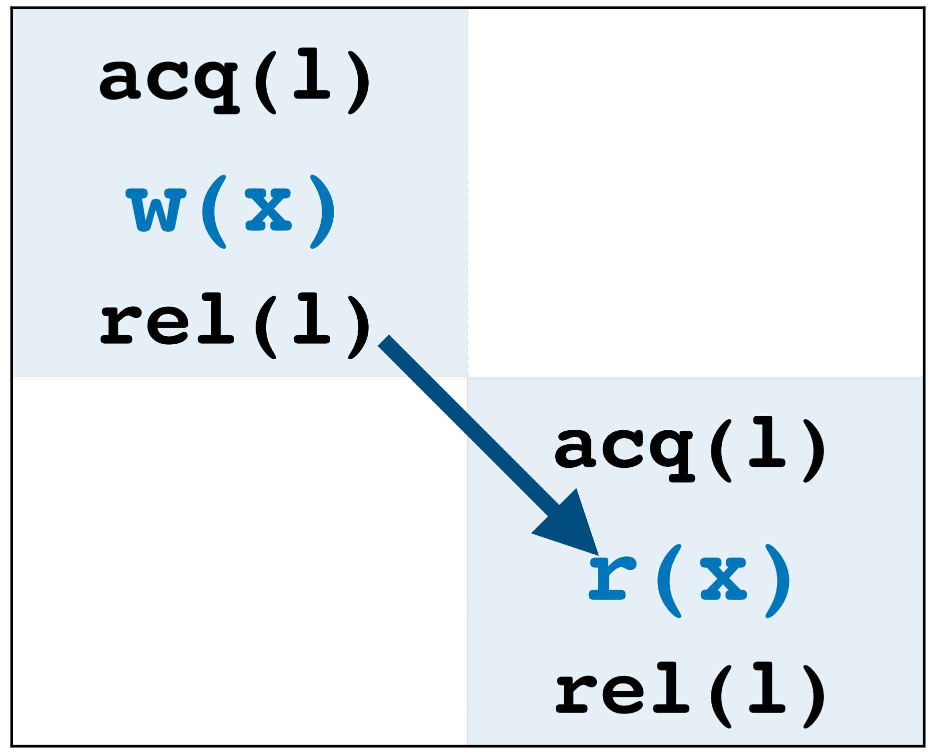
- $<_{WCP}$ orders events of an execution σ as follows

1. Critical sections C_1, C_2 on same lock are ordered when they contain conflicting events $e_1 \in C_1, e_2 \in C_2$:

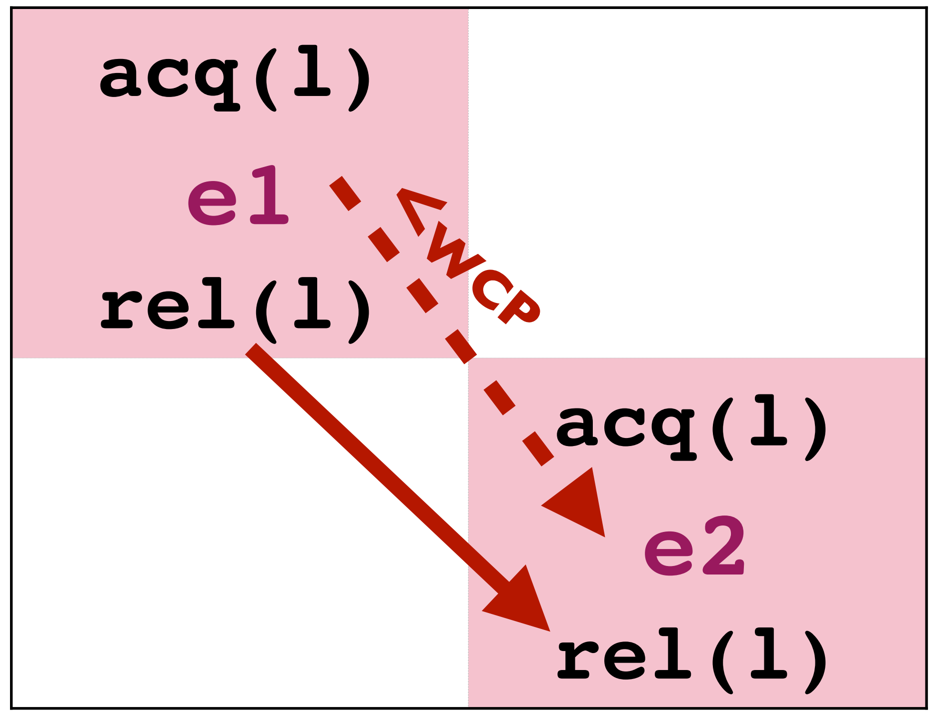
   $\texttt{rel}(C_1) <_{WCP} e_2$

2. Critical sections C_1, C_2 on same lock are ordered when they contain events $e_1 \in C_1, e_2 \in C_2$ ordered by WCP :

   $\texttt{rel}(C_1) <_{WCP} \texttt{rel}(C_2)$.



```
acq(l)
w(x)
rel(l)
        acq(l)
        r(x)
        rel(l)
```



```
acq(l)
e1
rel(l)
        acq(l)
        e2
        rel(l)
```

Ensure Soundness

---

†Dynamic Race Prediction in Linear Time, **PLDI 2017**

Weak Causal Precedence†

WCP identifies when to order critical sections on common lock

- $<_{WCP}$ orders events of an execution σ as follows

1. Critical sections C_1, C_2 on same lock are ordered when they contain conflicting events $e_1 \in C_1, e_2 \in C_2$:

   $$\texttt{rel}(C_1) <_{WCP} e_2$$

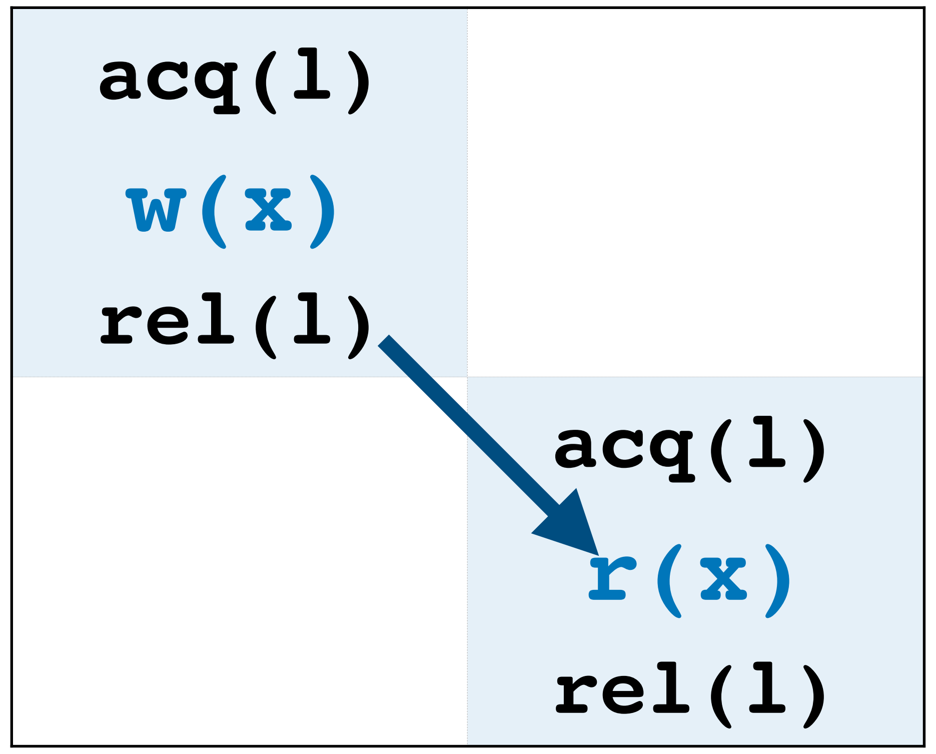
2. Critical sections C_1, C_2 on same lock are ordered when they contain events $e_1 \in C_1, e_2 \in C_2$ ordered by WCP :

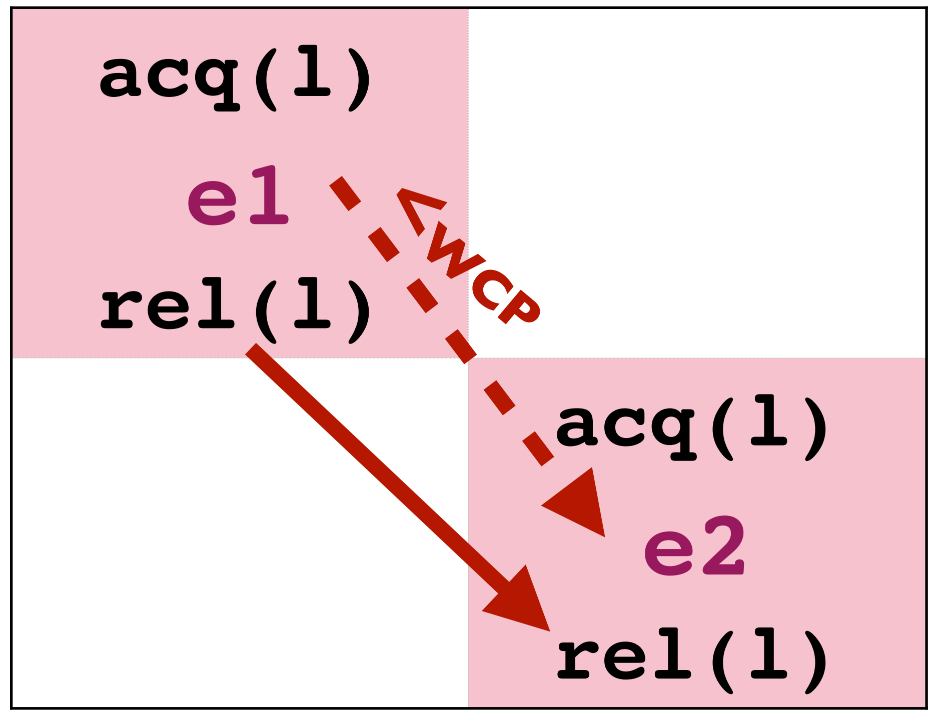
   $$\texttt{rel}(C_1) <_{WCP} \texttt{rel}(C_2).$$

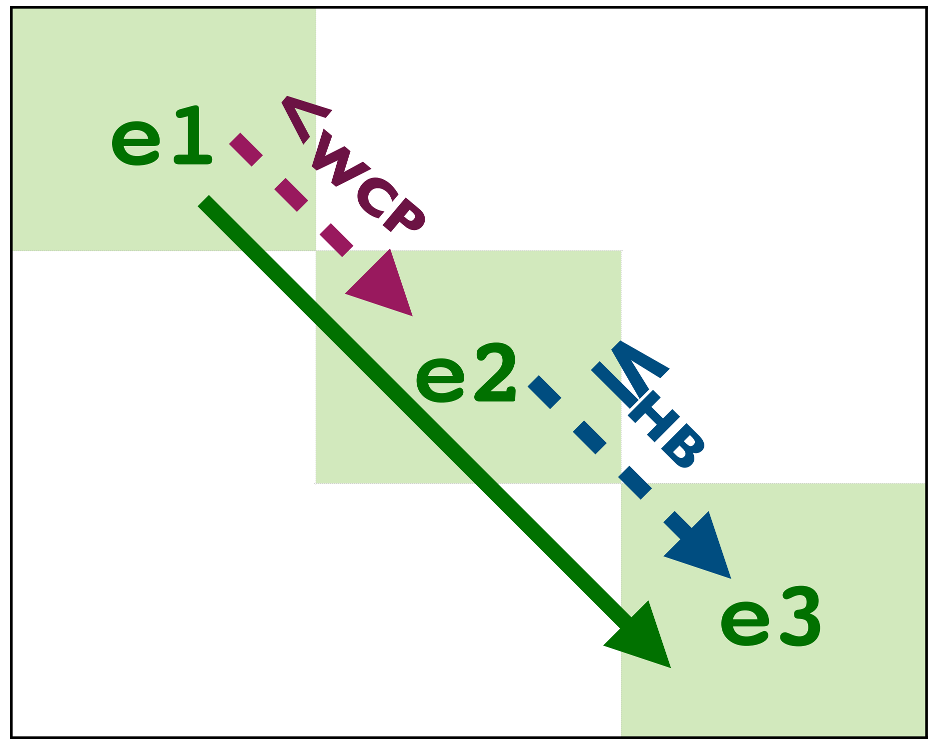
3. $<_{WCP}$ composes with $\leq_{HB}$

   $$<_{WCP} \circ \leq_{HB} \subsetneq <_{WCP}$$

   $$\leq_{HB} \circ <_{WCP} \subsetneq <_{WCP}$$







---

†Dynamic Race Prediction in Linear Time, **PLDI 2017**

WCP Soundness

Theorem (Weak Soundness for WCP).

Let σ be a trace and let (e_1, e_2) be a pair of conflicting events in σ, unordered by $<_{WCP}$.

Then, there is a correct reordering of σ that exhibits a **data race** or a **deadlock**.

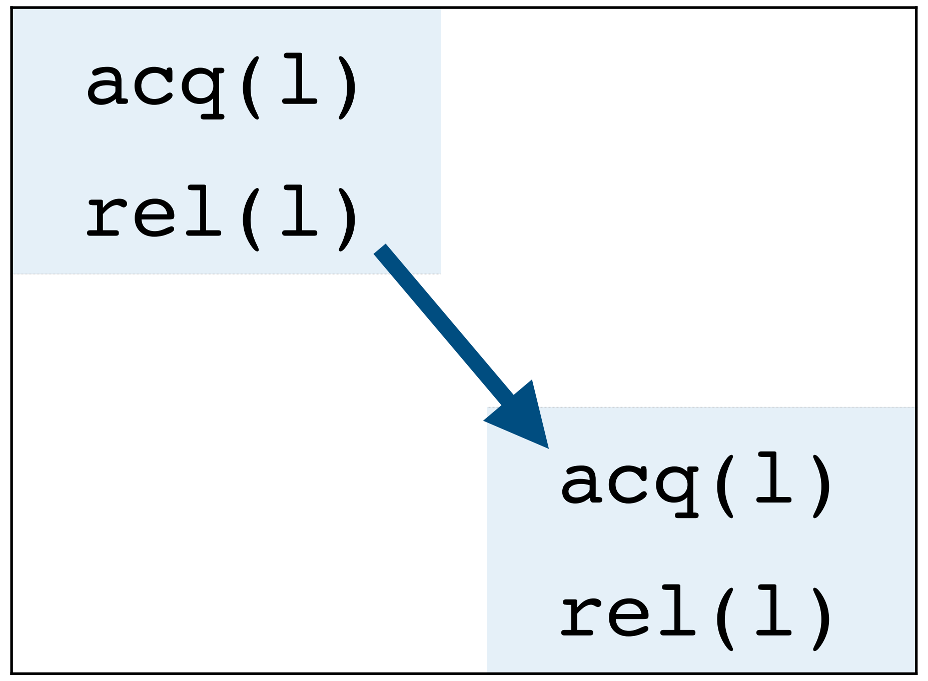
HB v/s WCP

① **All races reported by HB are also reported by WCP**

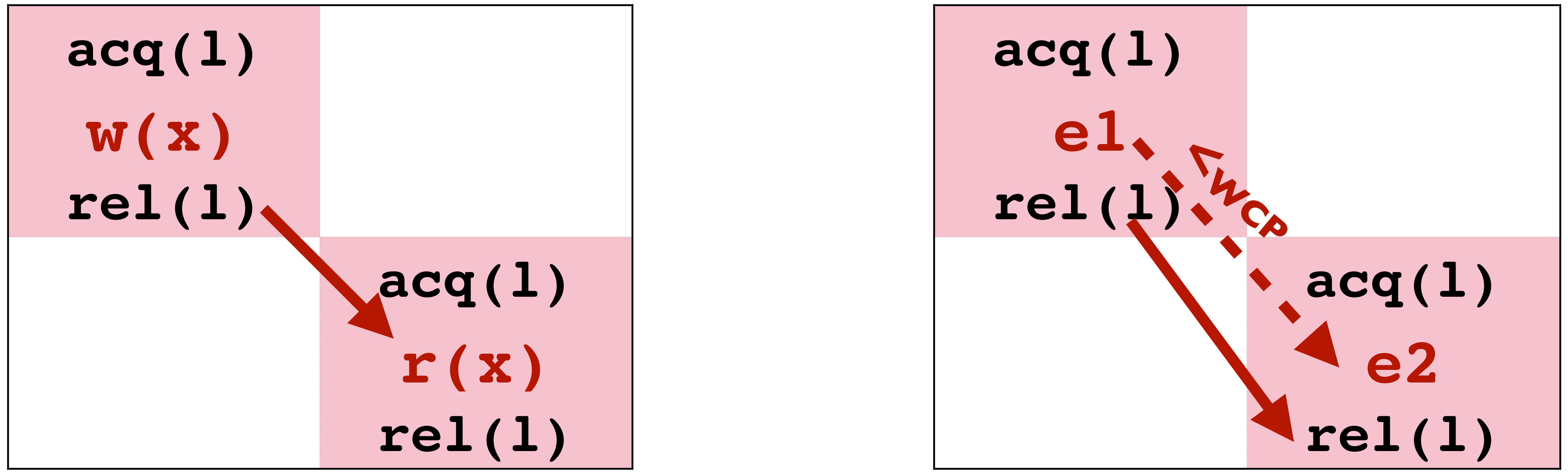
Every $\leq_{WCP}$ ordering is also an $\leq_{HB}$ ordering

② **WCP detects more races than HB**

$\leq_{WCP}$ places fewer orderings than $\leq_{HB}$



HB orders **all** critical sections on the same lock



WCP **selectively** orders critical sections on the same lock

Race Detection Algorithm using WCP

Algorithm

Implementation

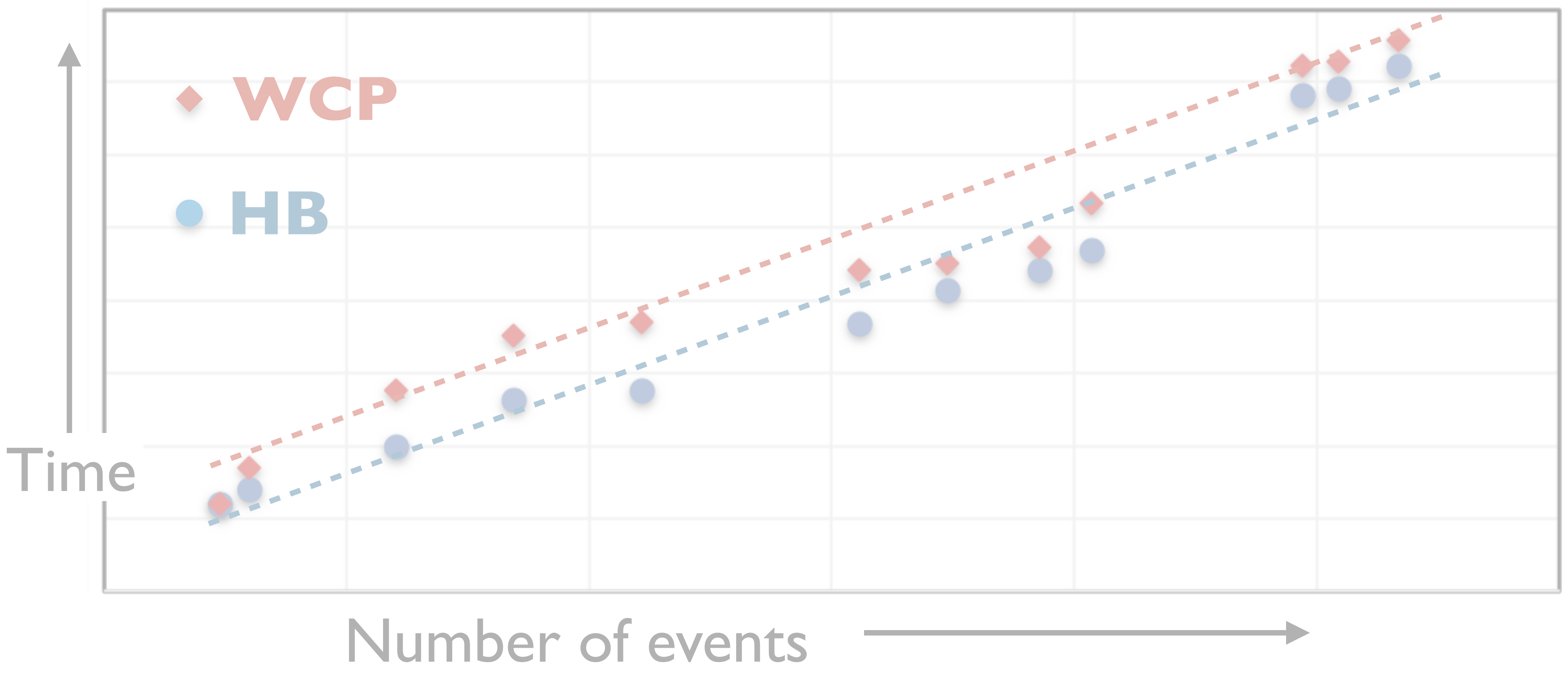
- **Linear time, one pass streaming**
  - Does not store the entire trace
  - Processes each event as it occurs
  - Constant time processing for each event
  - Detects races (conflicting events unordered by WCP) as they occur
  - Vector-clock algorithm

RAPID

github.com/umangm/rapid

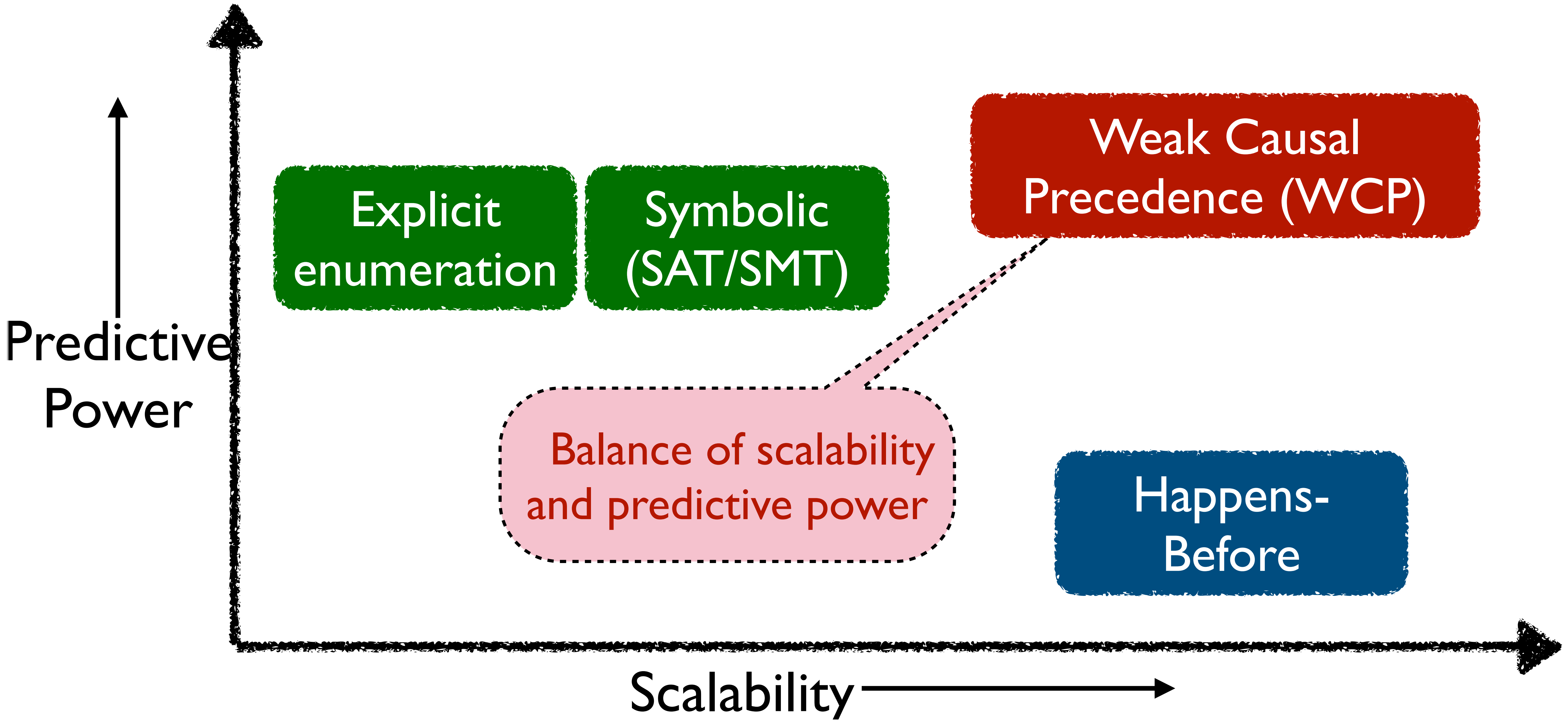
WCP Evaluation

- 18 benchmarks (Dacapo, Apache projects, IBM Contest suite, Java Grande Forum)
- Trace sizes - 50 to 216M



Techniques	Races	Avg. Time
HB	**182**	**144 s**
WCP	**190**	**198 s**
SMT*	**51**	**2258 s**

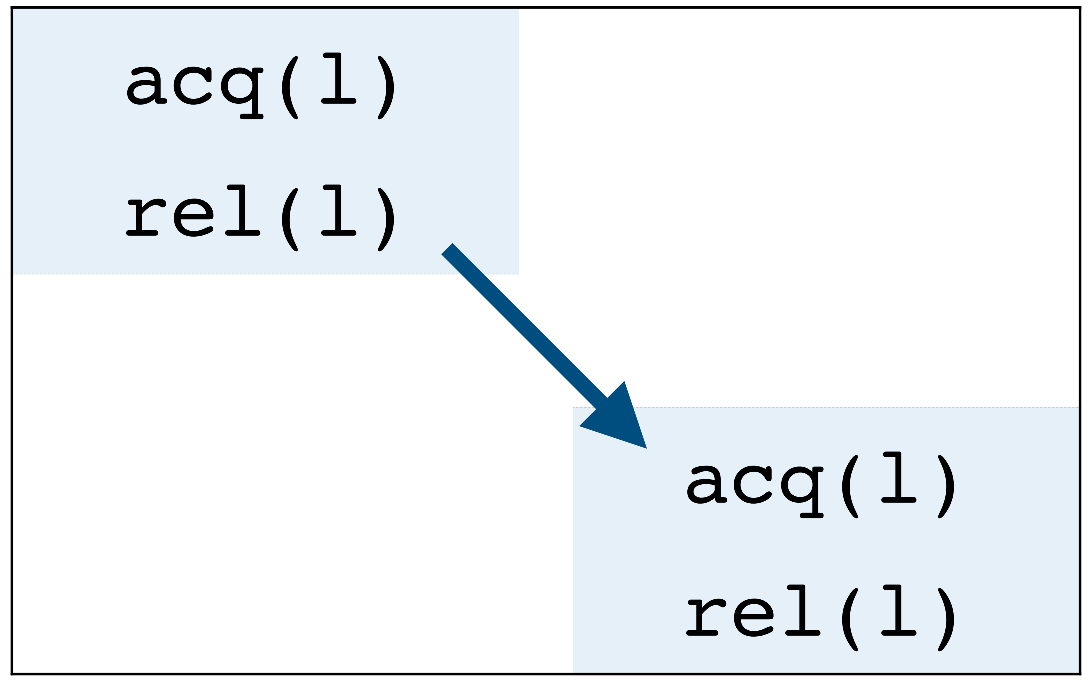
* RVPredict (Commercial race detector)



Synchronization Preserving Races[†]

[†] Optimal Prediction of Synchronization-Preserving Races, **POPL 2021**

Tackling the Conservativeness of HB (yet again)



HB orders all critical sections on the same lock

HB-principle: Consider all reorderings in which the order of critical sections is the same as the original trace

Pros

- Scalability
- Soundness

Cons

- Misses simple races

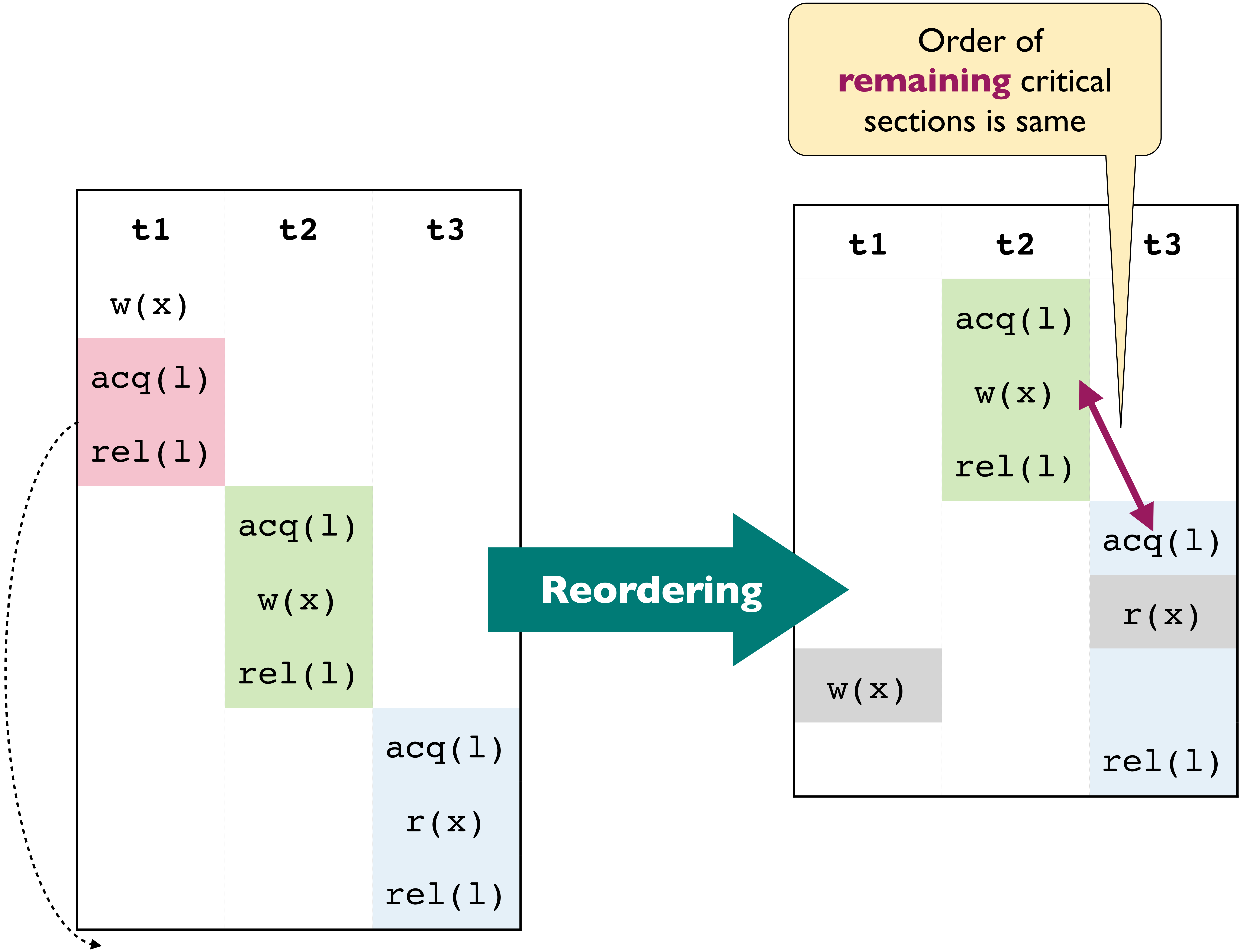
Can we -

- Reason about correct reorderings beyond the purview of HB
- While still sticking to the HB-principle

Synchronization Preserving Races

A correct reordering σ^* of σ is synchronization preserving if

- for every two critical sections C_1 and C_2 on the same lock,
- if both C_1 and C_2 occur in σ^*, then they must occur in the same order as in σ.



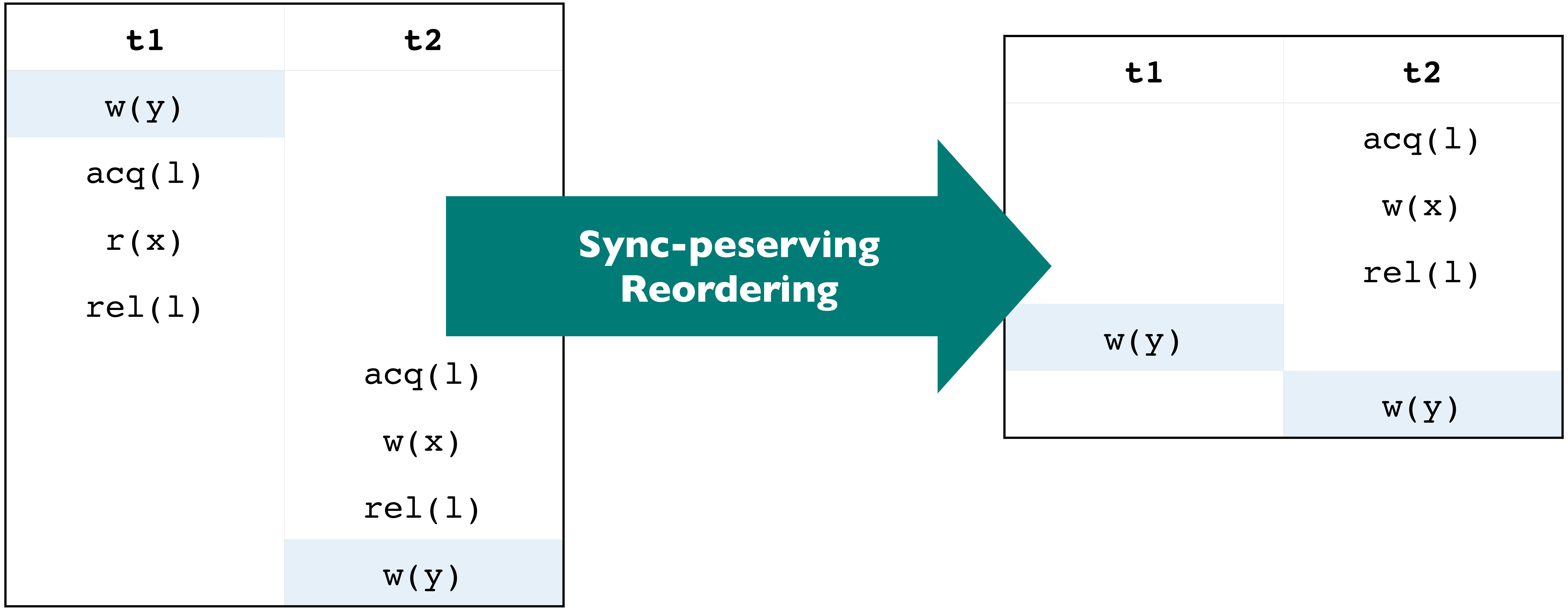
t1	t2	t3
w(x)		
acq(l)		
rel(l)		
	acq(l)	
	w(x)	
	rel(l)	
		acq(l)
		r(x)
		rel(l)

t1	t2	t3
	acq(l)	
	w(x)	
	rel(l)	
		acq(l)
		r(x)
w(x)		
		rel(l)

Order of **remaining** critical sections is same

Synchronization Preserving Races

A race (e1, e2) is called a sync-preserving race if it is witnessed by a sync-preserving correct reordering

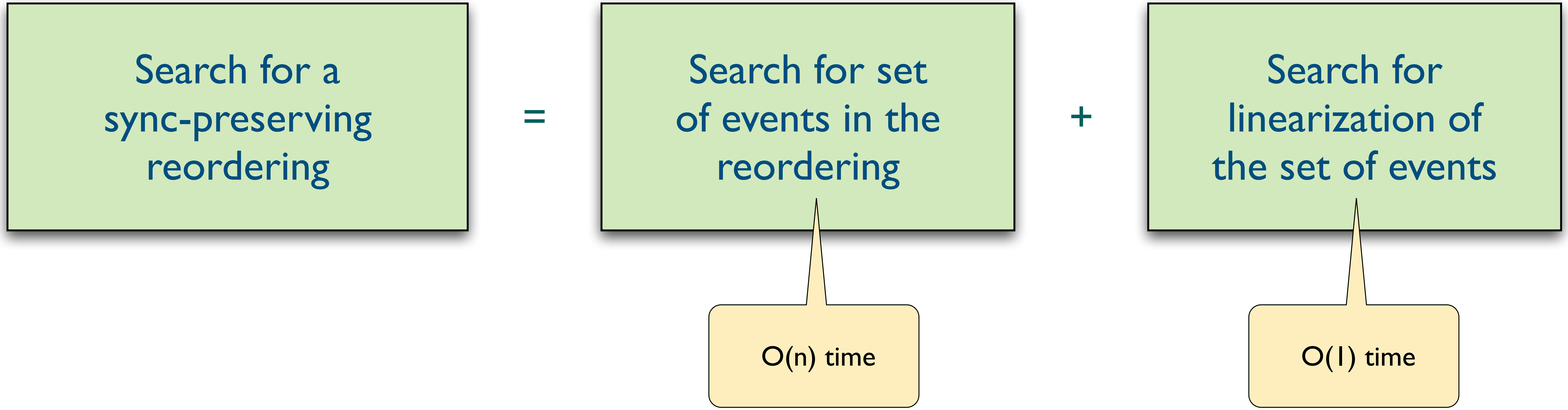


t1	t2
w(y)	
acq(l)	
r(x)	
rel(l)	
	acq(l)
	w(x)
	rel(l)
	w(y)

Sync-peserving Reordering

t1	t2
	acq(l)
	w(x)
	rel(l)
w(y)	
	w(y)

Detecting Sync-Preserving Races



Search for a sync-preserving reordering

=

Search for set of events in the reordering

+

Search for linearization of the set of events

O(n) time

O(1) time

__Theorem.__

The problem of checking if a trace σ has a sync-preserving race can be solved in O(n) time and O(n) space.

Algorithm: Key Ideas

Search for linearization
of the set of events

Lemma.

Let ρ be a sync-preserving reordering of σ that witnesses a race (e1, e2). Let $\rho^* = \sigma|_{Events(\rho)}$. Then ρ* is also a sync-preserving reordering that witnesses the race (e1, e2)

If (e1, e2) is witnessed by a sync-preserving correct reordering ρ of the observed execution σ, then it is also witnessed by a trace, all whose events are ordered as in σ.

Algorithm: Key Ideas

The SPIdeal(e1, e2) is the smallest set I such that

- pred(e1) ∈ I, pred(e1) ∈ I (Thread predecessors of e1 and e1 are in I)
- For every event e, if e ∈ I then pred(e) ∈ I
- If a read event r ∈ I then (the last write observed by r) lastWrite(r) ∈ I
- For two acquire events acq1 < acq2 of the same lock ℓ,

  if acq1 ∈ I, acq2∈ I, then match(acq2)∈ I

Search for set of events in the reordering

Lemma.

If (e1, e2) is a sync-preserving race, then it is witnessed by a reordering ρ such that Events(ρ) = SPIdeal(e1, e2)

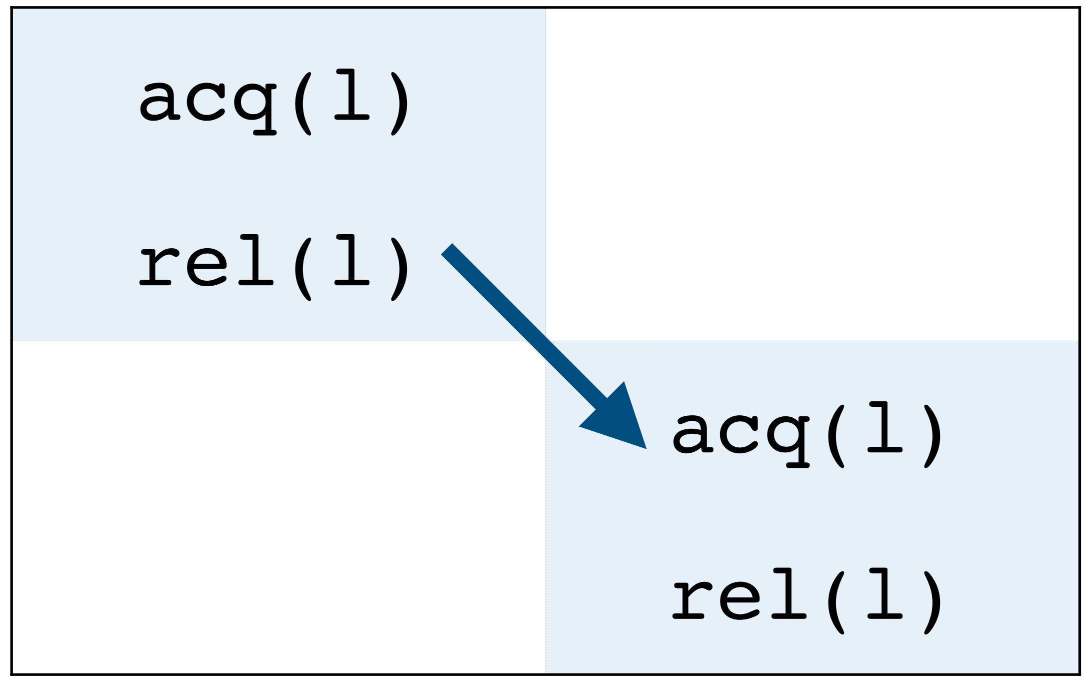
Algorithm and Complexity

- Generate the set $\mathrm{SPIdeal}(e1, e2)$
- Check if $e1 \notin \mathrm{SPIdeal}(e1, e2)$ and $e2 \notin \mathrm{SPIdeal}(e1, e2)$

Theorem.

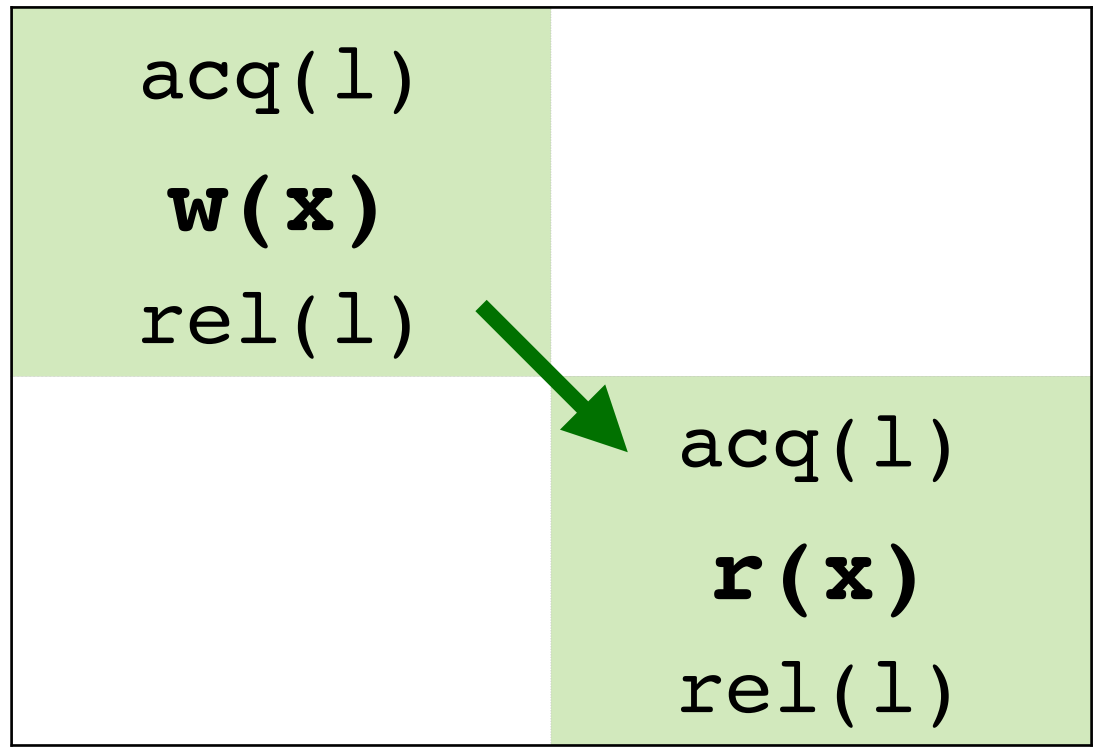
The problem of checking if a trace σ has a sync-preserving race can be solved in O(n) time and O(n) space.

HB v/s WCP v/s Sync-Preserving Races



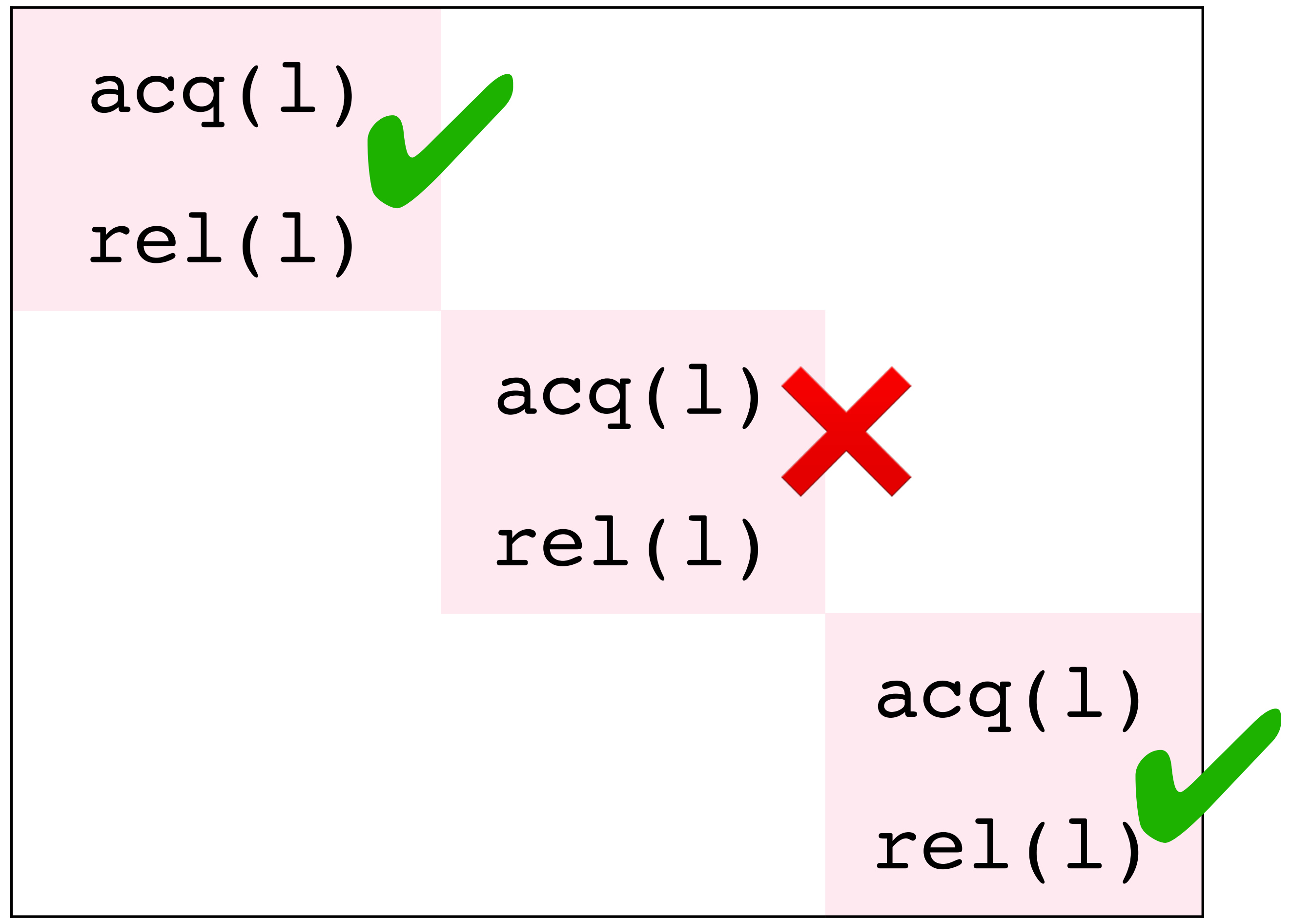
`acq(l)`	
`rel(l)`	
	`acq(l)`
	`rel(l)`

HB orders all critical sections on the same lock



`acq(l)`	
`w(x)`	
`rel(l)`	
	`acq(l)`
	`r(x)`
	`rel(l)`

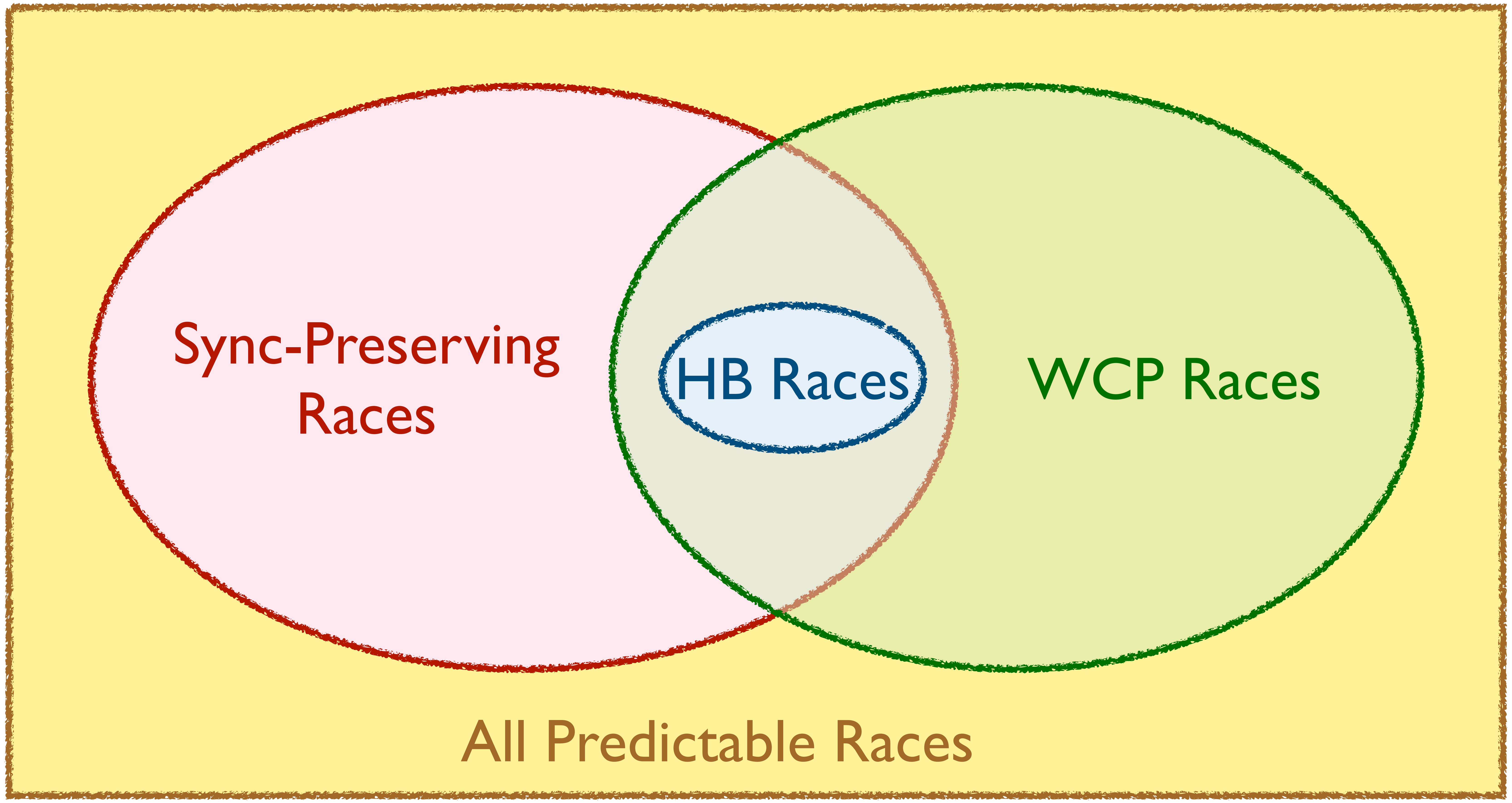
WCP selectively orders conflicting critical sections



`acq(l)` ✓		
`rel(l)`		
	`acq(l)` ✗	
	`rel(l)`	
		`acq(l)`
		`rel(l)` ✓

Sync-preserving selects which critical sections to schedule

HB v/s WCP v/s Sync-Preserving Races



HB v/s WCP v/s Sync-Preserving Races

t1	t2
w(x)	
acq(l)	
rel(l)	
	acq(l)
	rel(l)
	w(x)

HB	✗
Sync-Preserving	✓
WCP	✓

t1	t2
w(y)	
acq(l)	
r(x)	
rel(l)	
	acq(l)
	w(x)
	rel(l)
	w(y)

HB	✗
Sync-Preserving	✓
WCP	✗

t1	t2
acq(l)	
r(x)	
rel(l)	
	acq(l)
	rel(l)
	w(x)

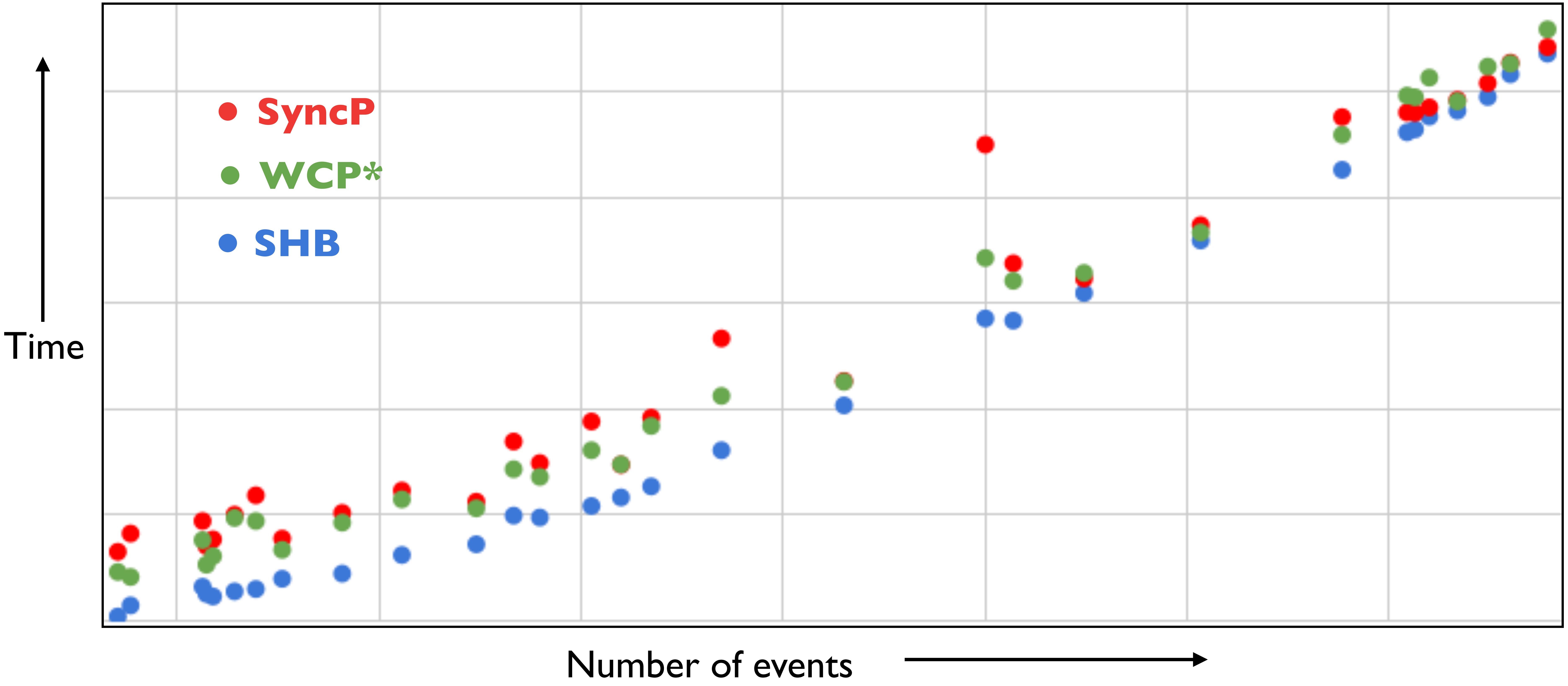
HB	✗
Sync-Preserving	✗
WCP	✓

t1	t2
acq(m)	
w(y)	
acq(l)	
r(x)	
rel(l)	
rel(m)	
	acq(m)
	rel(m)
	acq(l)
	w(x)
	rel(l)
	w(y)

HB	✗
Sync-Preserving	✗
WCP	✗

Evaluation

- 30 benchmarks: Dacapo, Apache projects, IBM Contest suite, Java Grande Forum, SIR
- Trace sizes - 50 to 600M

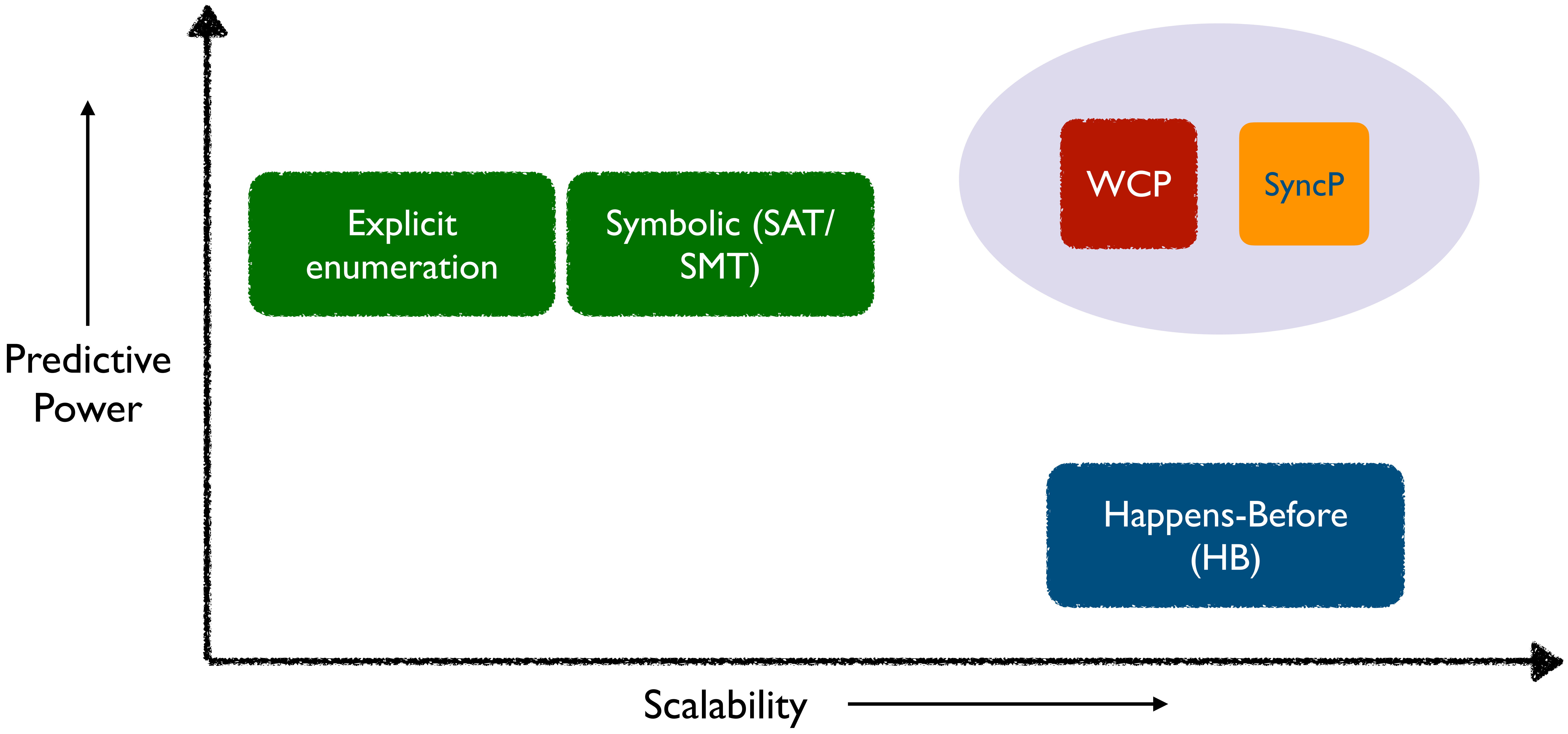


Evaluation

- 30 benchmarks: Dacapo, Apache projects, IBM Contest suite, Java Grande Forum, SIR
- Trace sizes - 50 to 600M

Techniques	Races	Racy Mem. Loc.	Max Distance	Avg Time
SHB	**157**	**1255**	**10M**	**235s**
WCP*	**134**	**1240**	**8M**	**403s**
SyncP	**178**	**1276**	**224M**	**321s**

Algorithms for Data Race Prediction



Predictive Power

Explicit enumeration

Symbolic (SAT/ SMT)

WCP

SyncP

Happens-Before (HB)

Scalability

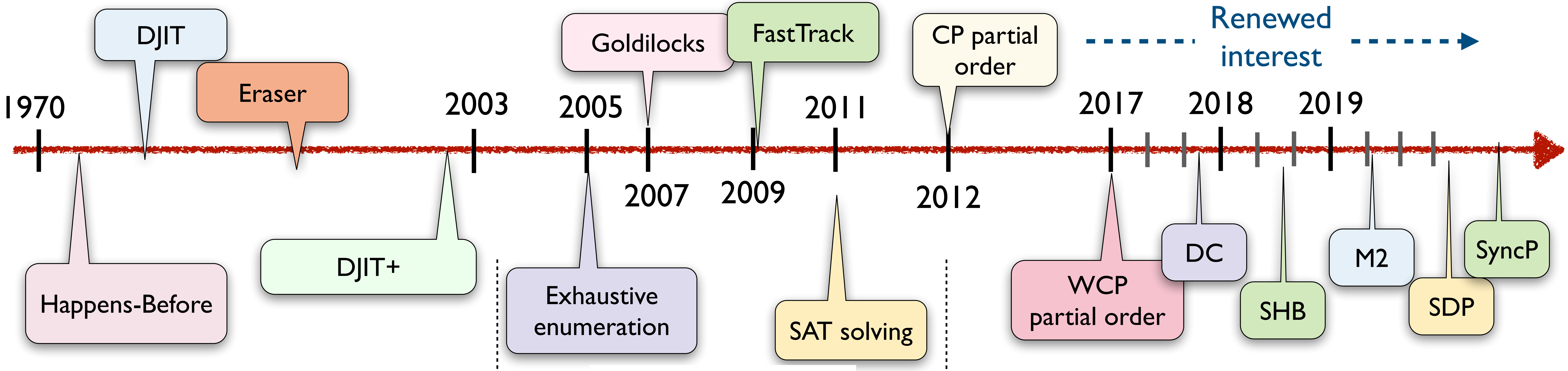
How Hard is Data Race Prediction?[†]

[†] The Complexity of Dynamic Data Race Prediction, **LICS 2020**

Some History

Data Race Prediction

Given an execution σ, is there a **correct reordering** of σ with a data race?



← Initial developments based on HB →

Complexity	PTIME
Completeness	👎

← Explicit or Symbolic Search →

Complexity	Exponential
Completeness	👍

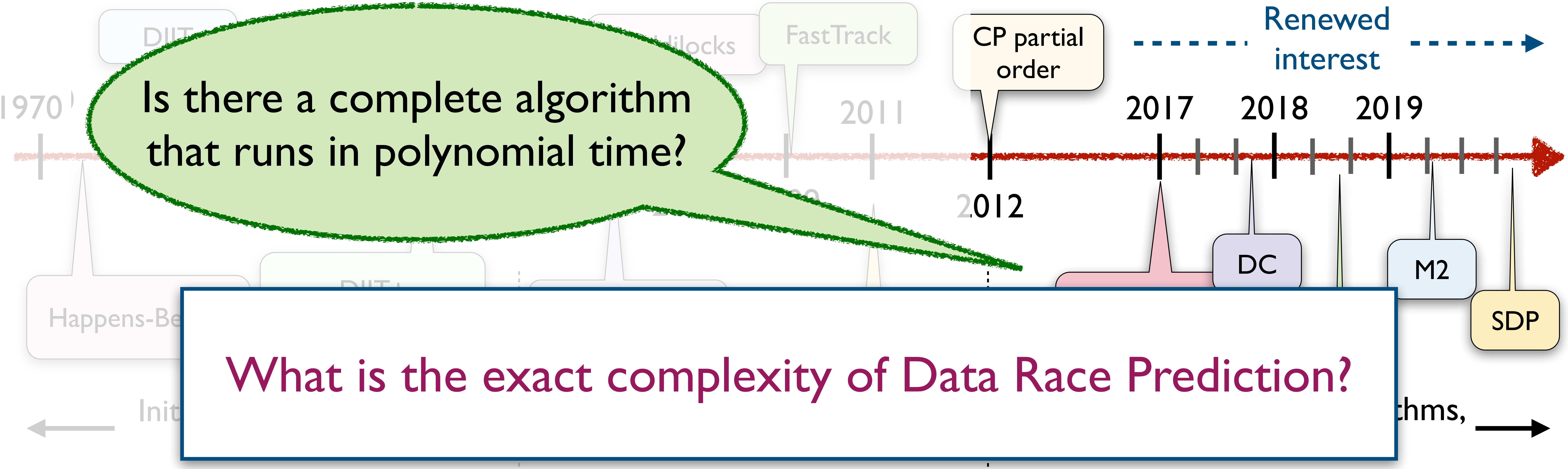
← Polynomial time algorithms, better than HB →

Complexity	PTIME
Completeness	👉

Some History

Data Race Prediction

Given an execution σ, is there a **correct reordering** of σ with a data race?



Is there a complete algorithm that runs in polynomial time?

What is the exact complexity of Data Race Prediction?

Complexity	PTIME
Completeness	👎

Complexity	Exponential
Completeness	👍

Complexity	PTIME
Completeness	

How hard is Data Race Prediction?

Data Race Prediction

- **Input:** Trace **σ** and conflicting events e_1 and e_2
  [n events, k threads, d memory locations and locks]
- **Output:** YES iff there is a correct reordering of **σ** that exhibits data race (e_1, e_2).

(Easy) Upper Bounds

1. **NP**
   - Guess an alternate reordering and check if it is a correct reordering
2. O(k^n) - Enumeration based techniques:
   - At every step, choose thread to execute
3. O(SAT(poly(n)) - SAT solving based techniques

Lower Bound

- Is it **NP**-hard? Is enumeration unavoidable?
- Is it polynomial time?

How hard is Data Race Prediction?

Data Race Prediction

- **Input:** Trace σ and events e_1 and e_2
  [N events, T threads, V memory locations and L locks]
- **Output:** YES iff there is a correct reordering of σ that exhibits data race (e_1, e_2).

Extensive study of complexity theoretic questions in data race prediction[†]

General Case

1. Poly-time Upper bound (when k is constant)
   Algorithm for race prediction $O(TN^{2(T-1)})$

exponential in T (not N)

Enumeration based approaches run in $O(T^N)$ time

2. Lower bound : **W[1]** hard in parameter T
   - **NP**-hard in general
   - Not likely to be **FPT** in T

Special Cases

3. Restricting the space of input traces
   - $O(N^2)$ time algorithm
   - Matching (conditional) lower bound

4. Restricting the space of data races to be reported
   - **Linear** time algorithm (parametric)

[†] The Complexity of Dynamic Data Race Prediction, **LICS 2020**

Fine-Grained Hardness in Data Race Prediction[†]

[†] Dynamic Data Race Prediction through the Fine-Grained Lens, **CONCUR 2021**

Linear Time Checkable Notions

- Algorithm that runs in time proportional to N, on input traces containing N events

Happens-Before	Lockset Principle	Synchronization Preserving
$O(N*T)$	$O(N*L)$	$O(N*V*T^3)$

- Multiplicative dependence on other parameters - #threads (T), #locks (L), #variables (V)
- Linear only when parameters are constant!

Is it possible to design ***purely linear time*** algorithms?

Contributions

Data Race Detection Given trace σ [N events, T threads, L locks, V variables], check if σ has a data race

Study of **fine-grained complexity** of detecting races based on several notions

Lockset Principle

$O(N*L)$

5. SETH-based $O(N^2)$ *lower bound* for lock-cover races
6. Improved *upper bound* for lock-set races: $O(N*\min(L,V))$
7. *Conditional impossibility* of SETH-based super-linear lower bound for lock-set races
8. Hitting Set based $O(N^2)$ *lower bound* for lock-set races

Happens-Before

$O(N*T)$

1. Improved *upper bound*: $O(N*\min(T, L))$
2. SETH-based $O(N^2)$ *lower bound* for read-write races
3. *Conditional impossibility* of SETH-based super-linear lower bound for general races
4. $O(N^{3/2})$ *lower bound* based on hardness of model-checking of FO(∀∃∃)

Synchronization Preserving

$O(N*V*T^3)$

9. SETH-based $O(N^3)$ *lower bound* for read-write races

†Dynamic Data Race Prediction through the Fine-Grained Lens, **CONCUR 2021**

Avenues for Future Work

- *Best* race detector that runs in linear time?
- Other concurrency bugs - deadlocks, atomicity violations
- Complementing other techniques
  - DPOR-style model checking
  - *Fuzzing*
  - controlled concurrency testing
- Other concurrency paradigms - message passing, distributed systems, weak memory

Thank You !

Looking for students and postdocs!

✉ umathur@comp.nus.edu.sg