Analysis Techniques to Detect Concurrency Errors

Cormac Flanagan
UC Santa Cruz

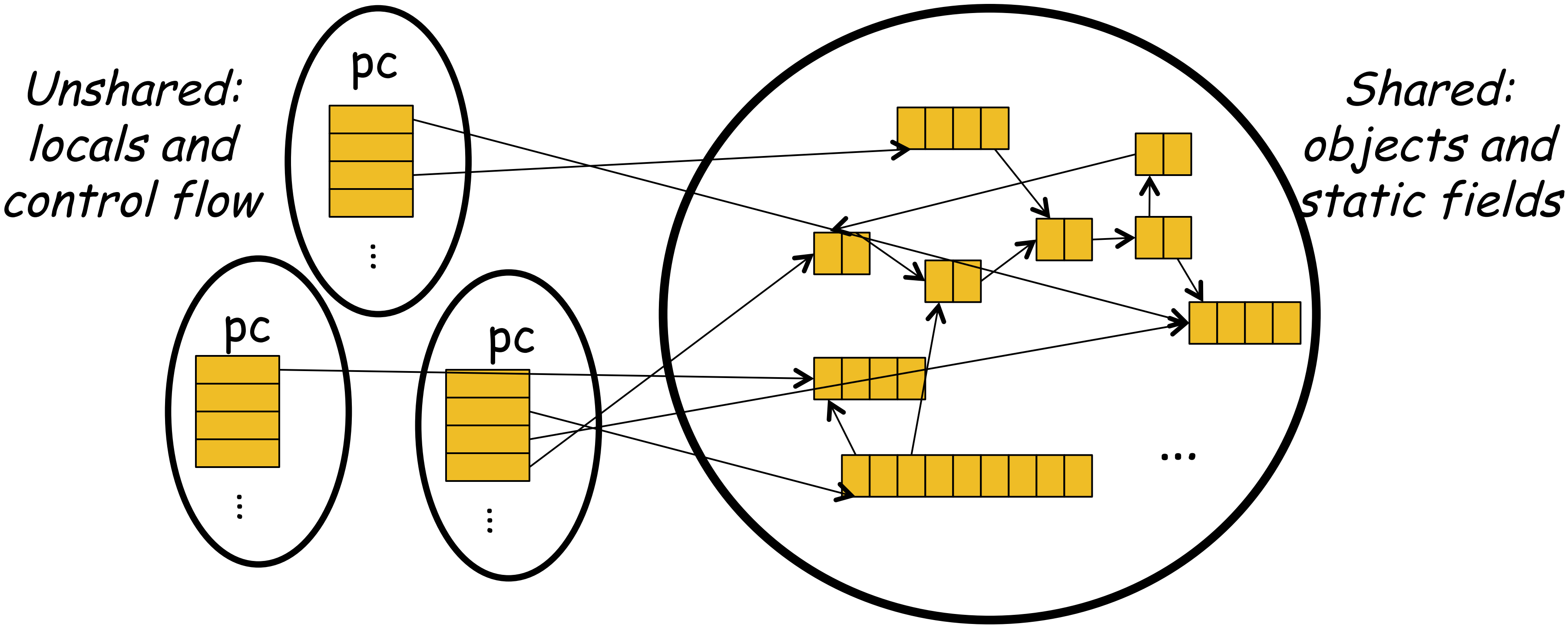
Stephen Freund
Williams College

Lecture Goals

- Enforcing concurrency properties
  - facilitates reasoning about correctness
  - race freedom, atomicity, determinism, cooperability
- Static and dynamic analyses
  - design space
  - implementation techniques
  - limitations
- Open research questions

Concurrent Programming Models

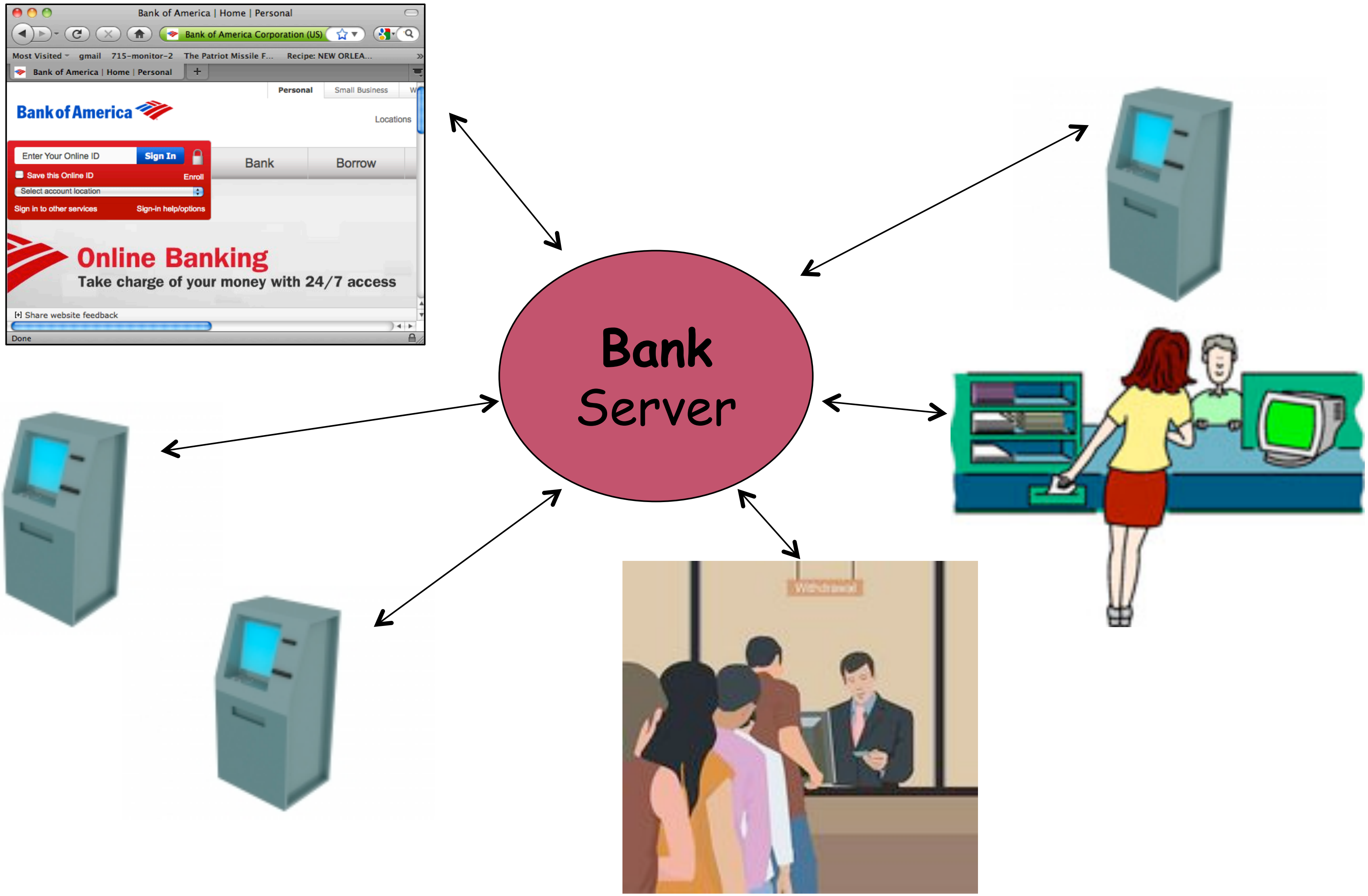
- Shared memory and explicit threads / sync



- Others
  - message passing, transactions, ...

Deterministic Parallelism

Non-Deterministic Concurrency



Open Research Problems

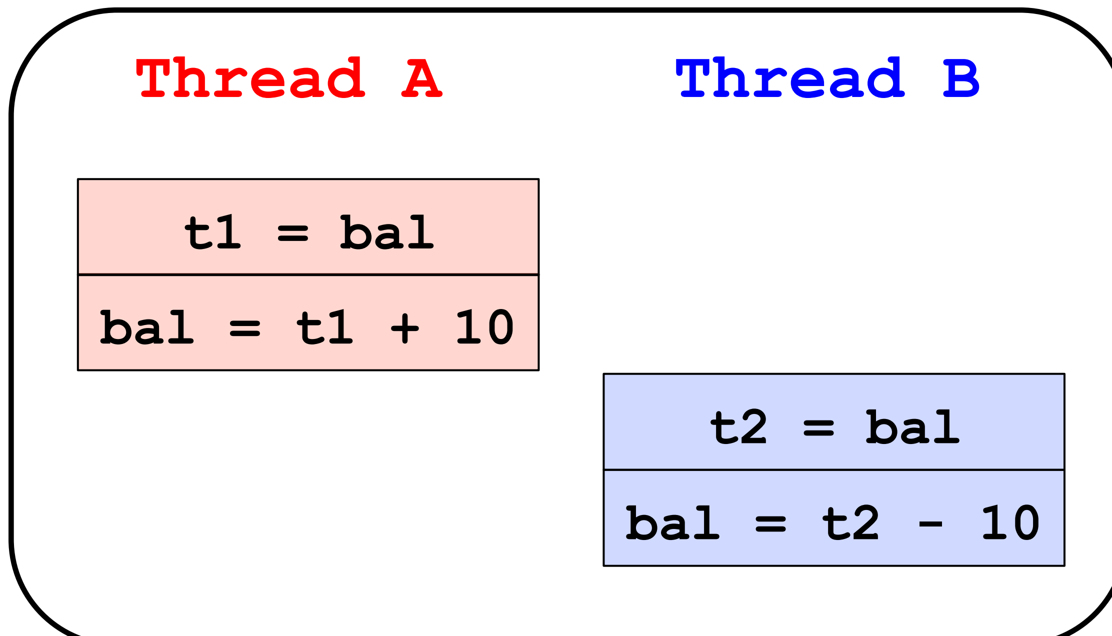
- Making concurrency/parallelism readily accessible to all programmers
- Developing programming models beyond shared memory
- How to write efficient multithreaded code
- How to write *correct* multithreaded code

Thread Interference: Data Races

- Concurrent conflicting accesses
  - Two threads read/write, write/read, or write/write the same location without intervening synchronization

Thread A
```
  ...
  t1 = bal;
  bal = t1 + 10;
  ...
```

Thread B
```
  ...
  t2 = bal;
  bal = t2 - 10;
  ...
```

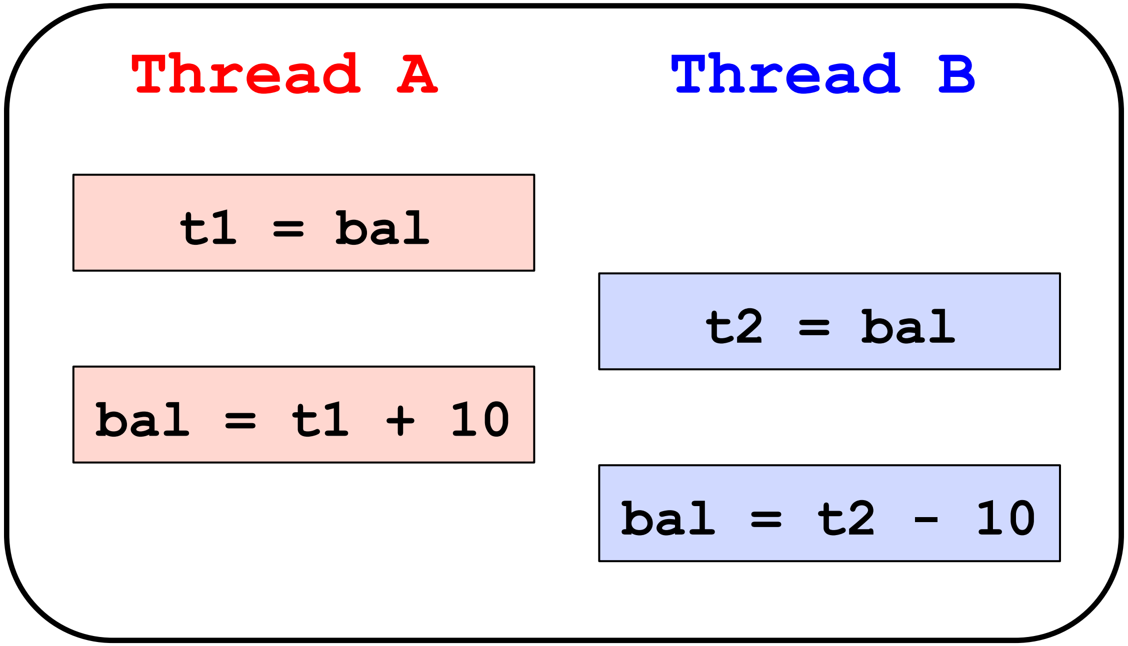


Thread Interference: Data Races

- Concurrent conflicting accesses
  - Two threads read/write, write/read, or write/write the same location without intervening synchronization

```
Thread A
  ...
  t1 = bal;
  bal = t1 + 10;
  ...
```

```
Thread B
  ...
  t2 = bal;
  bal = t2 - 10;
  ...
```



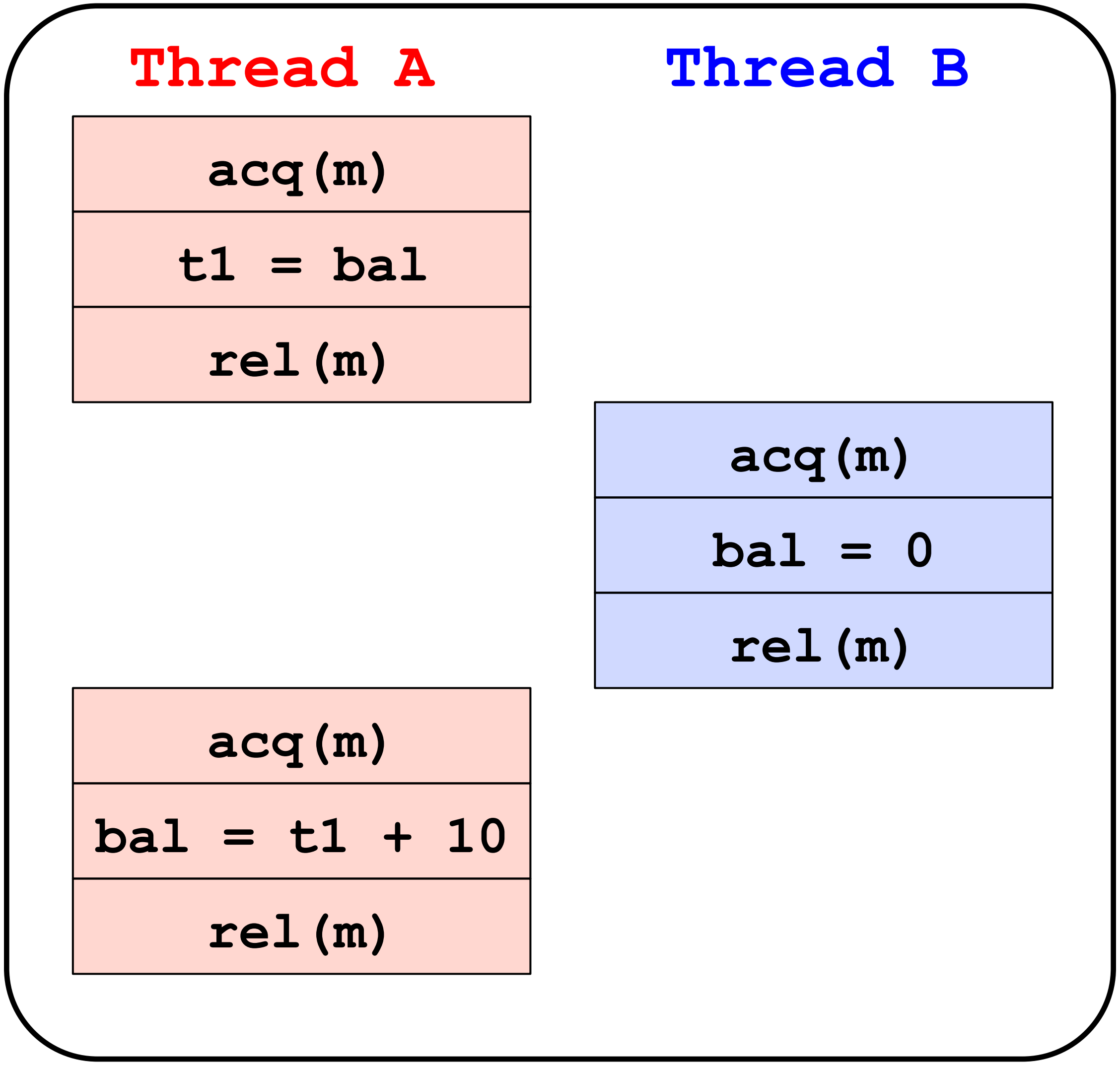
Thread Interference: Atomicity Violations

Thread A

```
...
acq(m);
t1 = bal;
rel(m);

acq(m);
bal = t1 + 10;
rel(m);
```

Thread B

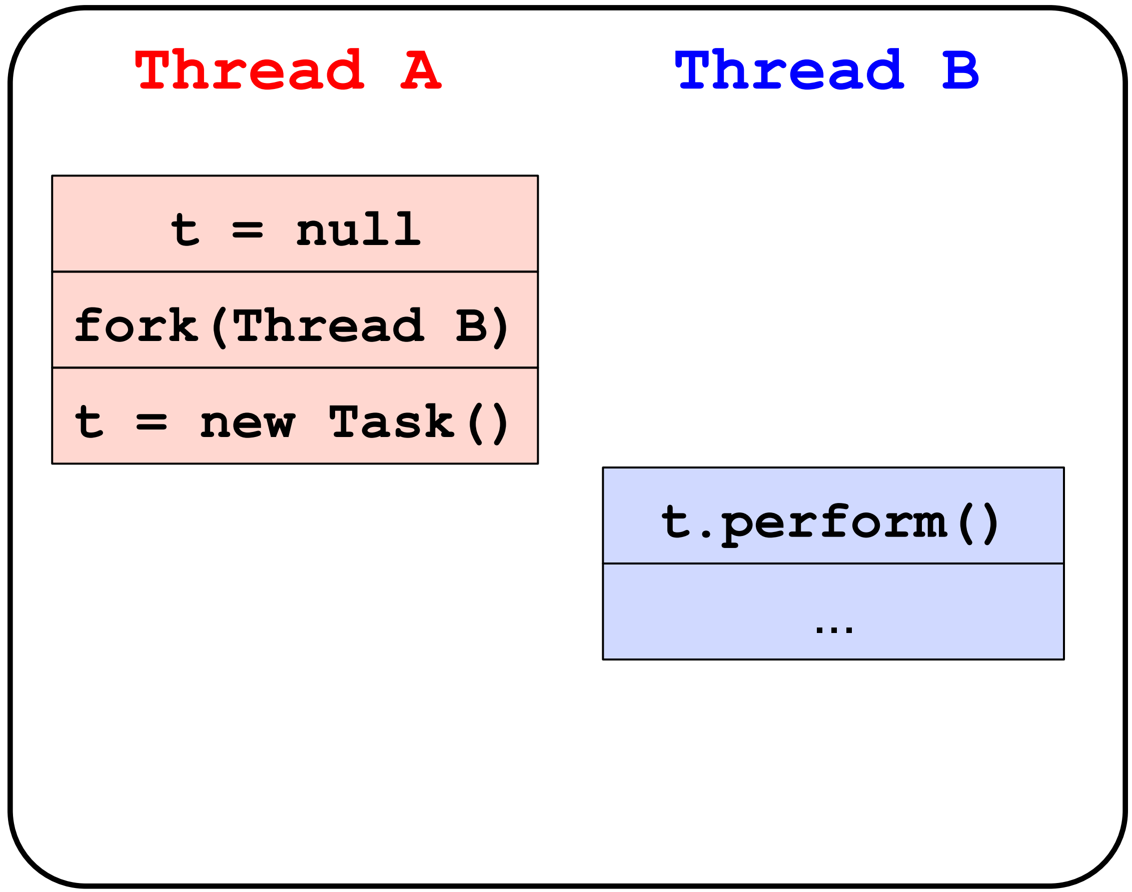
```
...
acq(m);
bal = 0
rel(m);
```



Thread Interference: Ordering Violations

```
Thread A
   ...
   t = null;
   fork(Thread B)
   t = new Task()
```

```
Thread B
   t.perform();
   ...
```



Thread Interference: Unintended Sharing

```
void work() {
  static int local = 0;
  local++;
  ...
}
```

Thread A

```
work();
```

Thread B

```
work();
```

Thread A	Thread B
t1 = local	
	t2 = local
local = t1+1	
	local = t2+1

Thread Interference: Deadlock

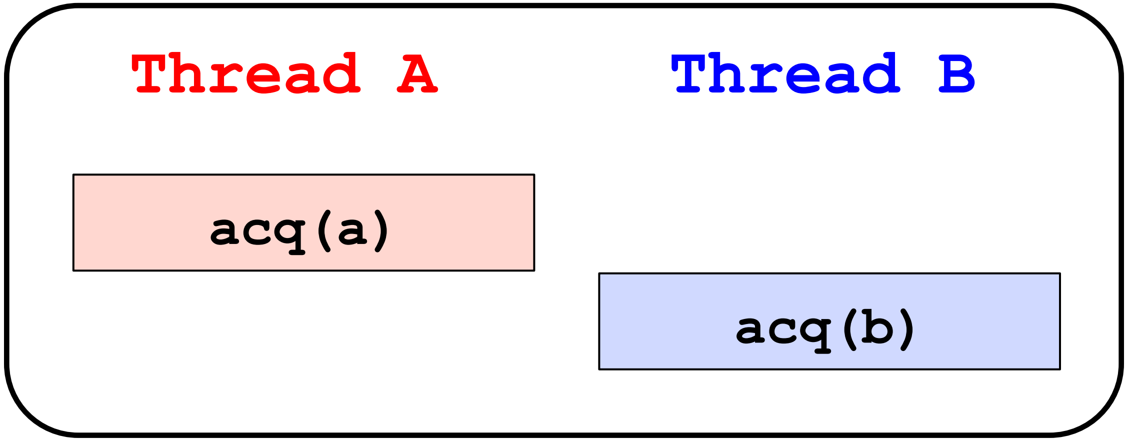
```
class Account {
  int bal;
  synchronized void deposit(int  n) { bal = bal + n; }

  synchronized void transfer(Account other, int n) {
    other.deposit(n);
    this.deposit(-n);
  }
}
```

Thread A

```
a.transfer(b,10);
```

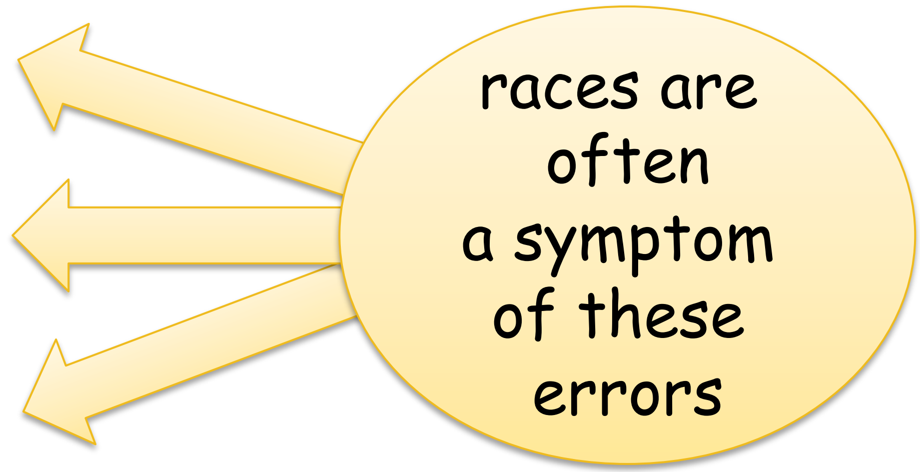
Thread B

```
b.transfer(a,10);
```



Data Race Detection

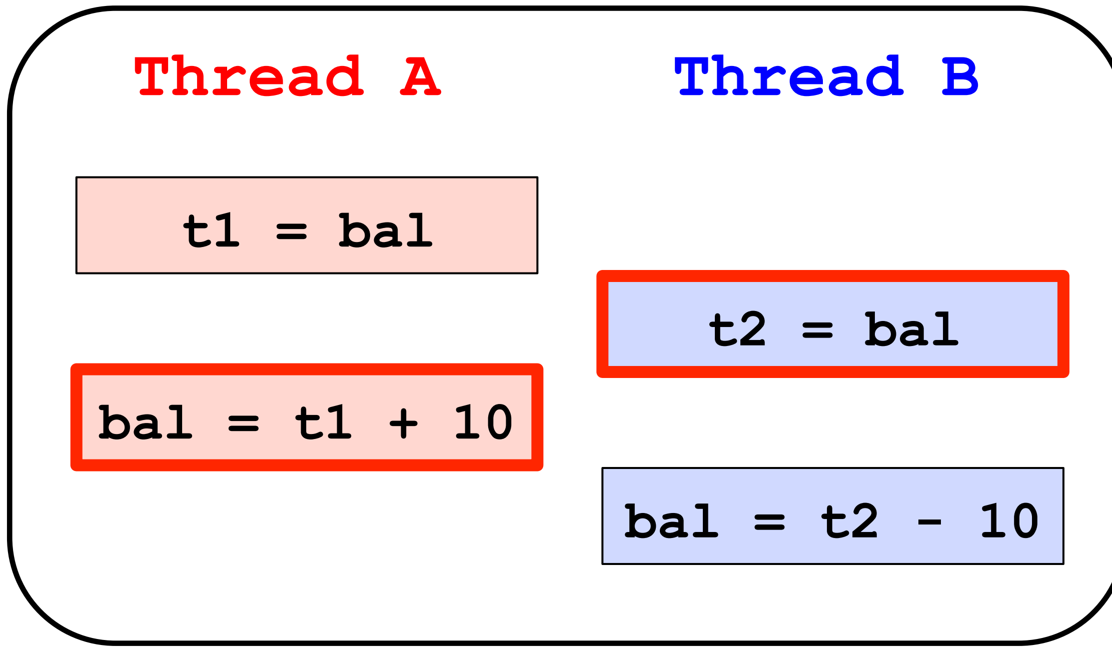
- Atomicity violations
- Ordering violations
- Unintended sharing
- Deadlocks and livelocks



Thread Interference: Atomicity Violation

```
Thread A
  ...
  t1 = bal;
  bal = t1 + 10;
  ...
```

```
Thread B
  ...
  t2 = bal;
  bal = t2 - 10;
  ...
```



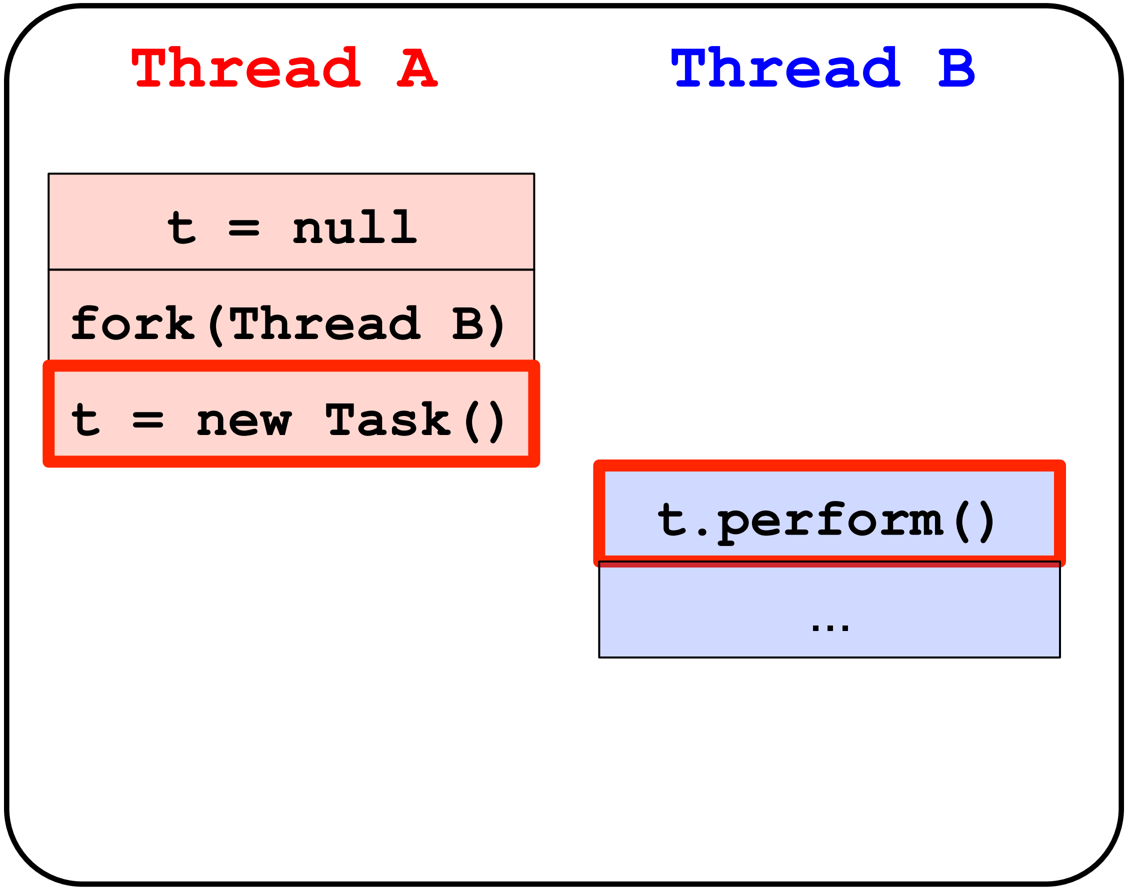
Thread Interference: Ordering Violations

Thread A

```
...
t = null;
fork(Thread B)
t = new Task()
```

Thread B

```
t.perform();
...
```



Thread Interference: Unintended Sharing

```
void work() {
   static int local = 0;
   local++;
   ...
}
```

Thread A

```
work();
```

Thread B

```
work();
```

Thread A	**Thread B**
t1 = local	
	t2 = local
local = t1+1	
	local = t2+1

Are All Race Conditions Errors?

- Implementing flag synchronization

```
boolean done = false;
```

Thread A

```
x = 1;
done = true;
```

Thread B

```
if (done) t = x;
```

- Implementing fast reads

```
int bal = 0;
```

Thread A

```
synchronized (m) {
  bal = bal + n;
}
```

Thread B

```
t = bal;
```

Are All Race Conditio

- Treated as "synchronization"
- Documents potential sharing
- Improves program semantics
- In C++: std::atomic<> types

- Implementing flag synchronization

```
volatile boolean done = false;
```

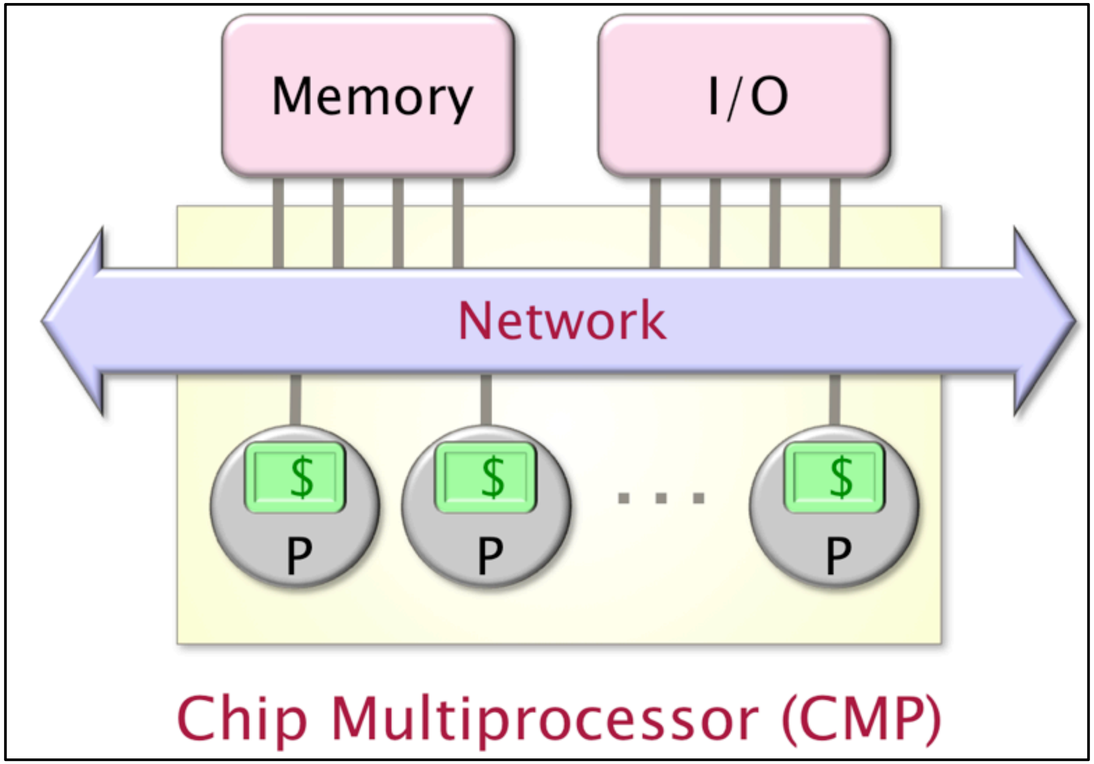
Thread A	Thread B
`x = 1;`	`if (done) t = x;`
`done = true;`	

- Implementing fast reads

```
volatile int bal = 0;
```

Thread A	Thread B
`synchronized (m) {`	`t = bal;`
`  bal = bal + n;`	
`}`	

Data Races and Memory Models



- Each processor/core has a cache
- When do writes to *x* become visible to other processors (threads)?

Memory Models

- Sequential Consistency
  - Operations by threads are interleaved in some global sequential order.
  - A read yields the value most recently written to that location according to this order.
  - Simple, intuitive

Java Example

```
int x;
int y;
Initially x == y == 0;
```

Thread A

```
x = 10;
y = 20;
```

Thread B

```
r1 = y;
r2 = x;
print r1 + r2;
```

What's Printed? 30? 20? 10? 0?

Memory Models

- Sequential Consistency
  - Operations by threads are interleaved in some global sequential order.
  - A read yields the value most recently written to that location according to this order.

- Relaxed Models (JMM, x86-TSO, etc.)
  - writes may be buffered in caches
  - more than one value written to *x* may be visible
  - necessary for hardware performance
  - (also enables compiler optimizations)

Example

```
int x = 0;
boolean done = false;
```

Thread A

```
x = 10;
done = true;
```

Thread B

```
while (!done) { }
print x;
```

```
int x = 0;
volatile boolean done = false;
```

Thread A

```
x = 10;
done = true;
```

Thread B

```
while (!done) { }
print x;
```

Why Look For Races?

- Programmers make errors leading to data races:
  - Missing locking
  - Missing "volatile" annotations
  - ...

- Must know about races to reason about any more sophisticated concurrency property

- Memory Model Guarantee:
  - Data-Race Freedom → Seq. Consistent Behavior

Data Race Detection

- Automated Tools to Find Data Races
  - Active area of research for > 20 years
  - More than 100 academic papers on the subject

- Key dimensions of the design space are not unique to data-race detection
  - type-checking
  - array-bounds
  - pointer errors
  - etc.

Static Data Race Detection

- Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Annotations to document concurrency invariants
- Example Tools:
  - RCC/Java — type-based
  - CHESS — state exploration
  - ESC/Java — "functional verification" (theorem proving-based)

Static Data Race Detection

- Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Annotations to document concurrency invariants
- Disadvantages of static:
  - Undecidable...
  - Tools produce "false positives" or "false negatives"
  - May be slow, require programmer annotations
  - May be hard to interpret results

Dynamic Data Race Detection

- Advantages
  - Can avoid "false positives"
  - No need for language extensions or sophisticated static analysis

- Disadvantages
  - Run-time overhead (5-20x for best tools)
  - Memory overhead for analysis state
  - Reasons only about observed executions
    - sensitive to test coverage
    - (some generalization possible...)

Dynamic Analysis Design Space

- Soundness
  - every actual data race is reported
- Completeness
  - all reported warnings are actually races
- Coverage
  - generalize to additional traces?
- Overhead
  - run-time slowdown
  - memory footprint
- Programmer overhead

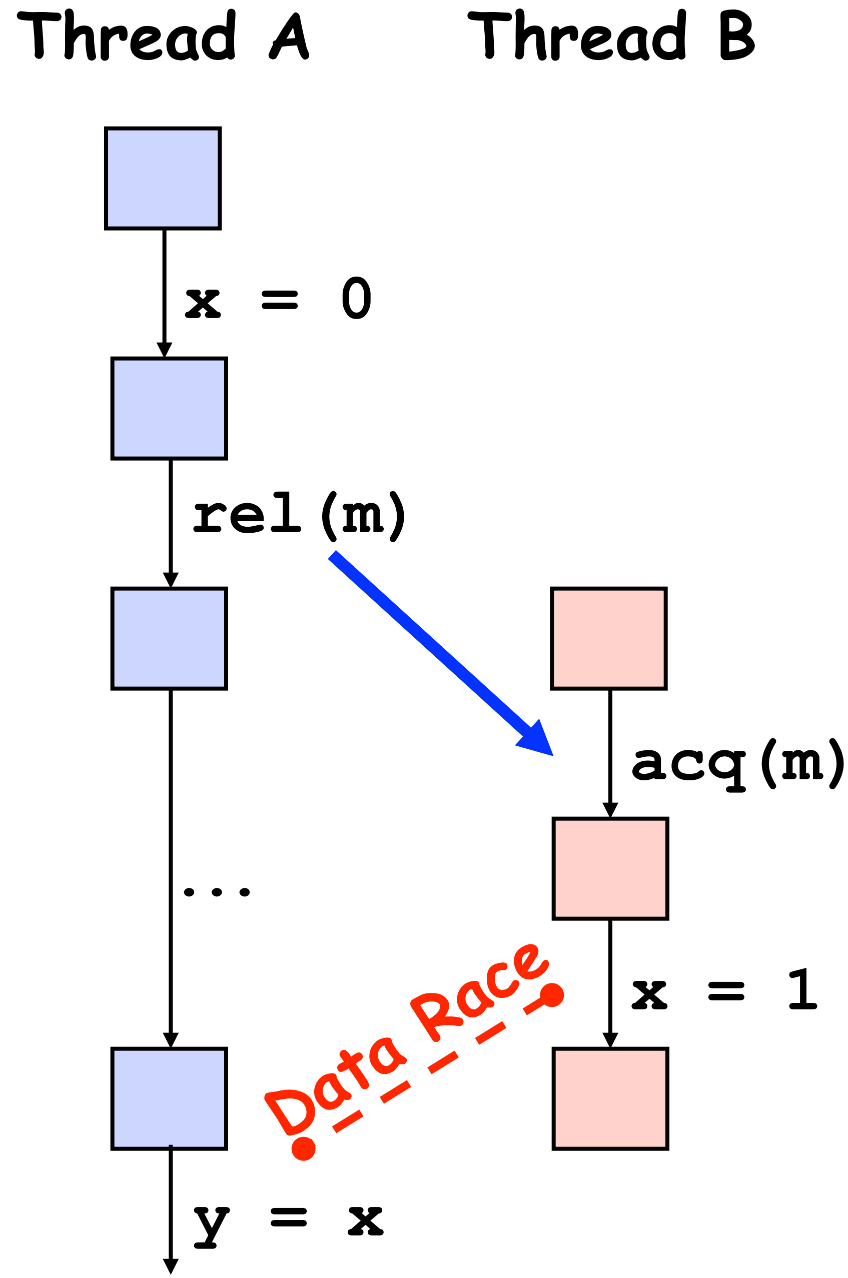
Overview of Analysis Techniques

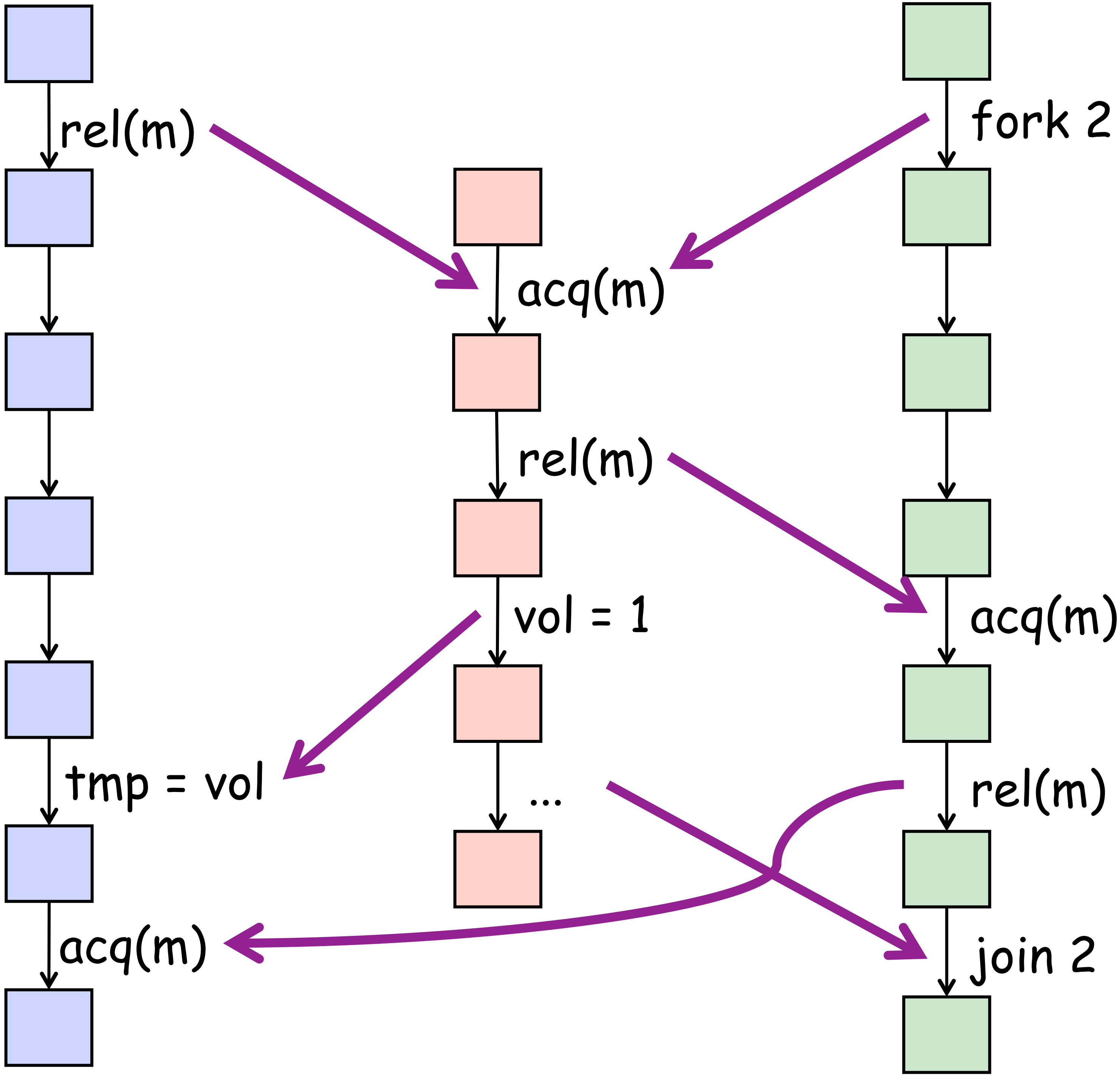
- Lamport's Happens-Before Relation [Lamport 78]
  - enables precise definition of data race

- Four points in design space
  1. LockSet
  2. Vector Clocks
  3. Hybrid LockSet/VC
  4. FastTrack

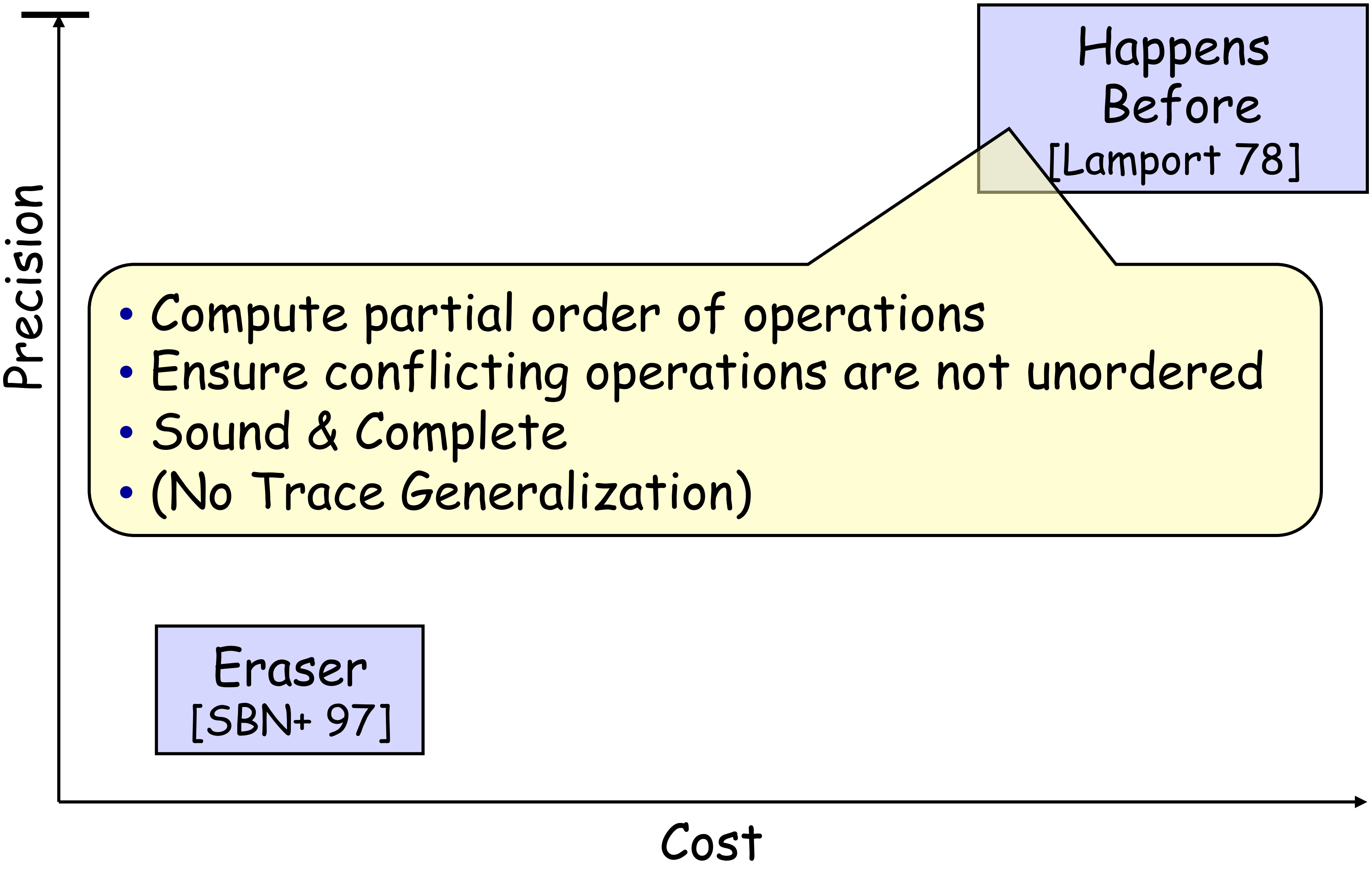
Happens-Before

- Event Ordering:
  - program order
  - synchronization order
  - transitivity
- Types of Data Races:
  - Write-Write
  - Write-Read
    - (write then read)
  - Read-Write
    - (read then write)

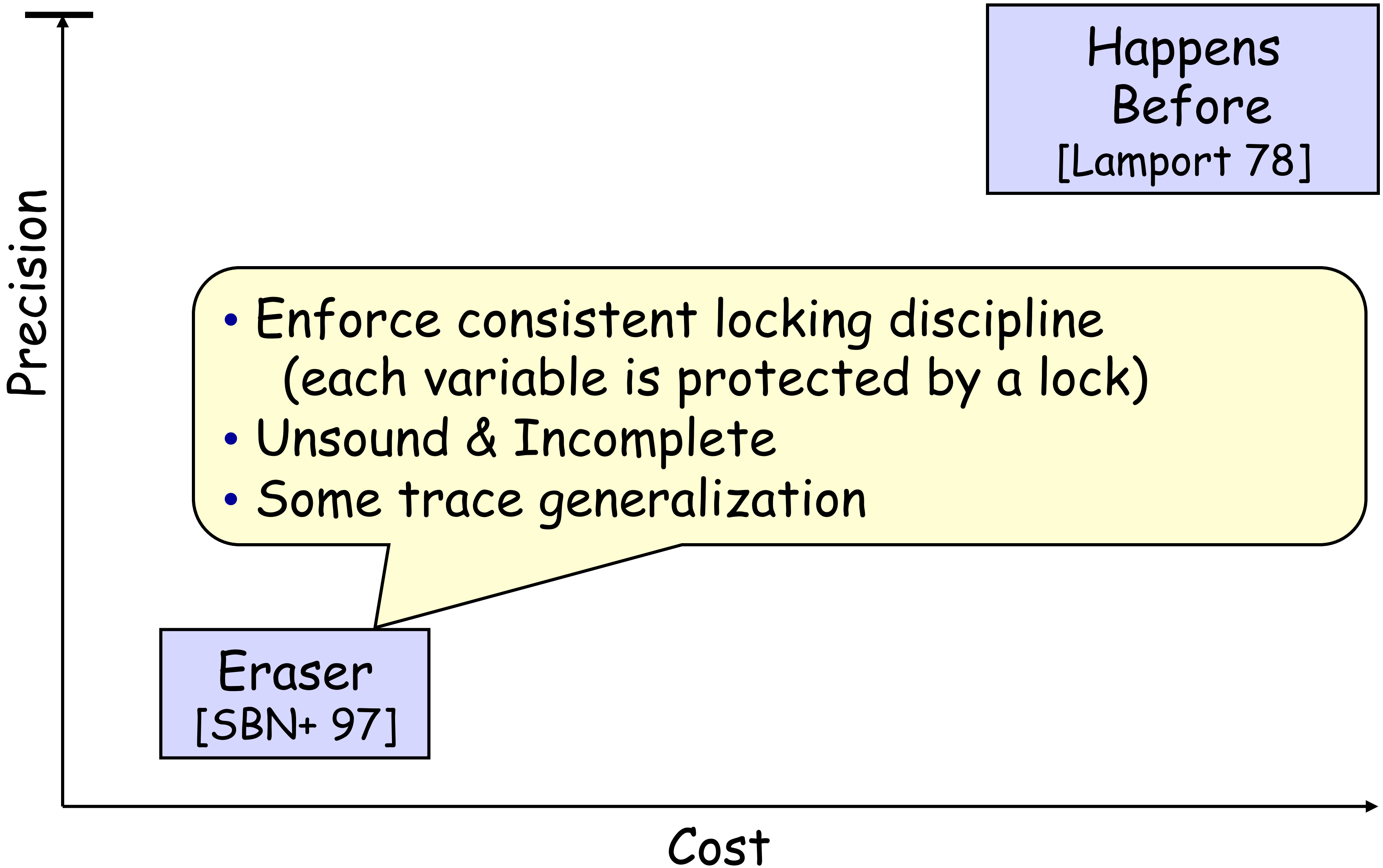



rel(m)
fork 2
acq(m)
rel(m)
acq(m)
vol = 1
tmp = vol
...
rel(m)
acq(m)
join 2

Dynamic Data Race Detection



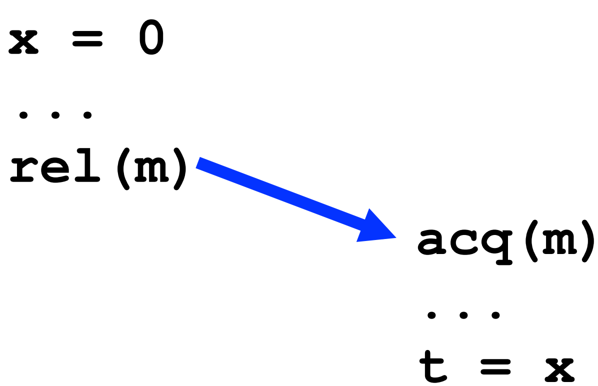
Dynamic Data Race Detection



Approximating Happens-Before

- Track *lockset* for each memory location
  - LockSet(x): set of locks held on all accesses to location x

- If m ∈ LockSet(x):



If LockSet(x) is empty:

Lockset Example

Thread A

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

Thread B

```
synchronized(y) {
  o.f = 2;
}
```

- First access to `o.f`:

  LockSet(`o.f`) := Held(curThread)

  = { x, y }

Lockset Example

Thread A

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

Thread B

```
synchronized(y) {
  o.f = 2;
}
```

- Subsequent access to `o.f`:

  LockSet(`o.f`) := LockSet(`o.f`) $\cap$ Held(curThread)

  $= \{ x, y \} \cap \{ x \} = \{ x \}$

Lockset Example

Thread A

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

→

Thread B

```
synchronized(y) {
  o.f = 2;
}
```

- Subsequent access to `o.f`:

  $\text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held}(\text{curThread})$

  $= \{ x \} \cap \{ y \} = \{ \}$

  DATA RACE!

Lockset Properties

- Relatively good performance (slowdowns < ~15x)
- Sound:

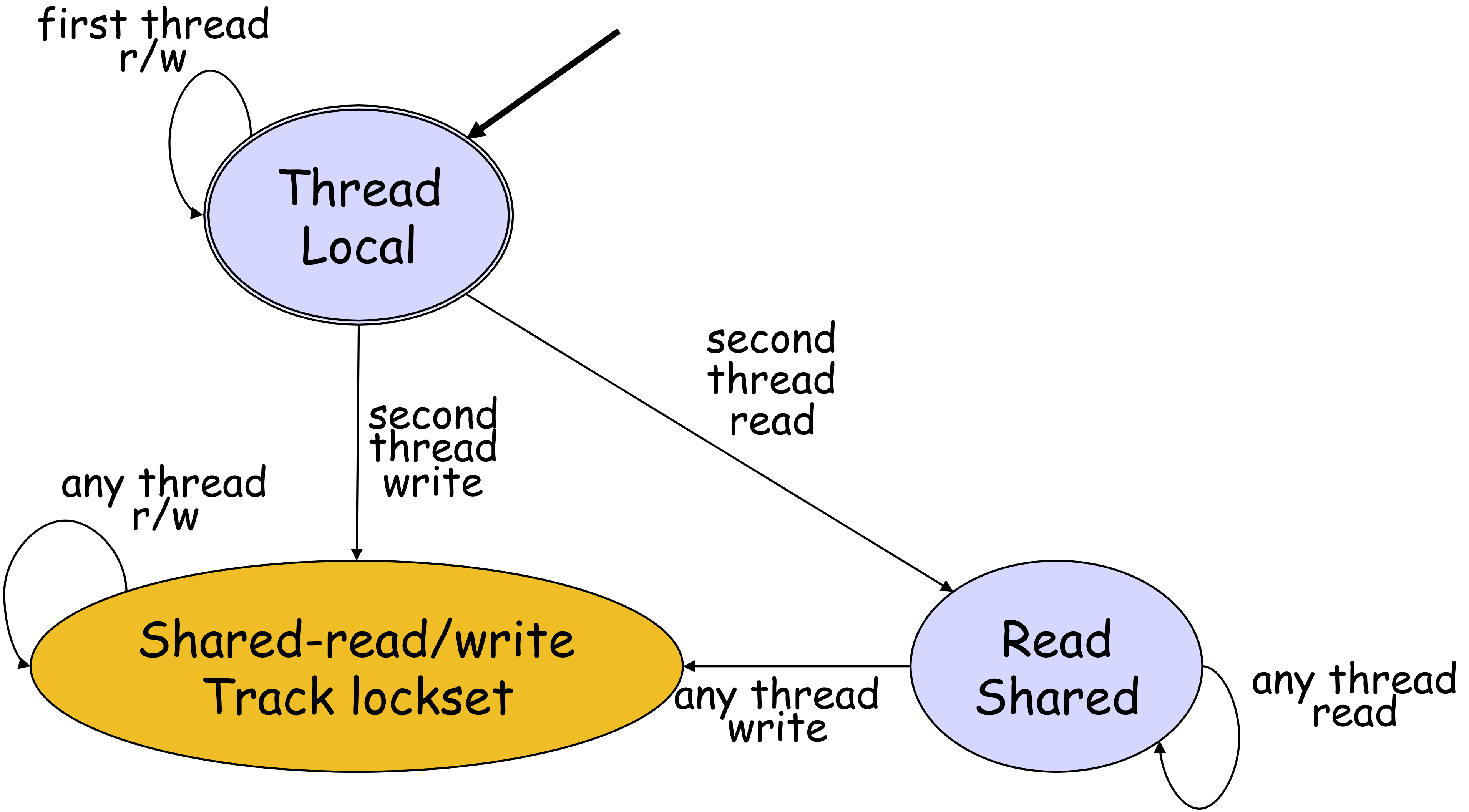
  No warnings → data-race-free execution

- Incomplete:

  Warning ↛ data race on execution

  - thread-local data, read-shared data, etc

Per-Variable State Machine



Lockset Properties

- Extensions help reduce false alarms but
  - introduce (rare) unsoundnesses
  - and still not complete...

```
boolean ready = false;
int data = 0;
```

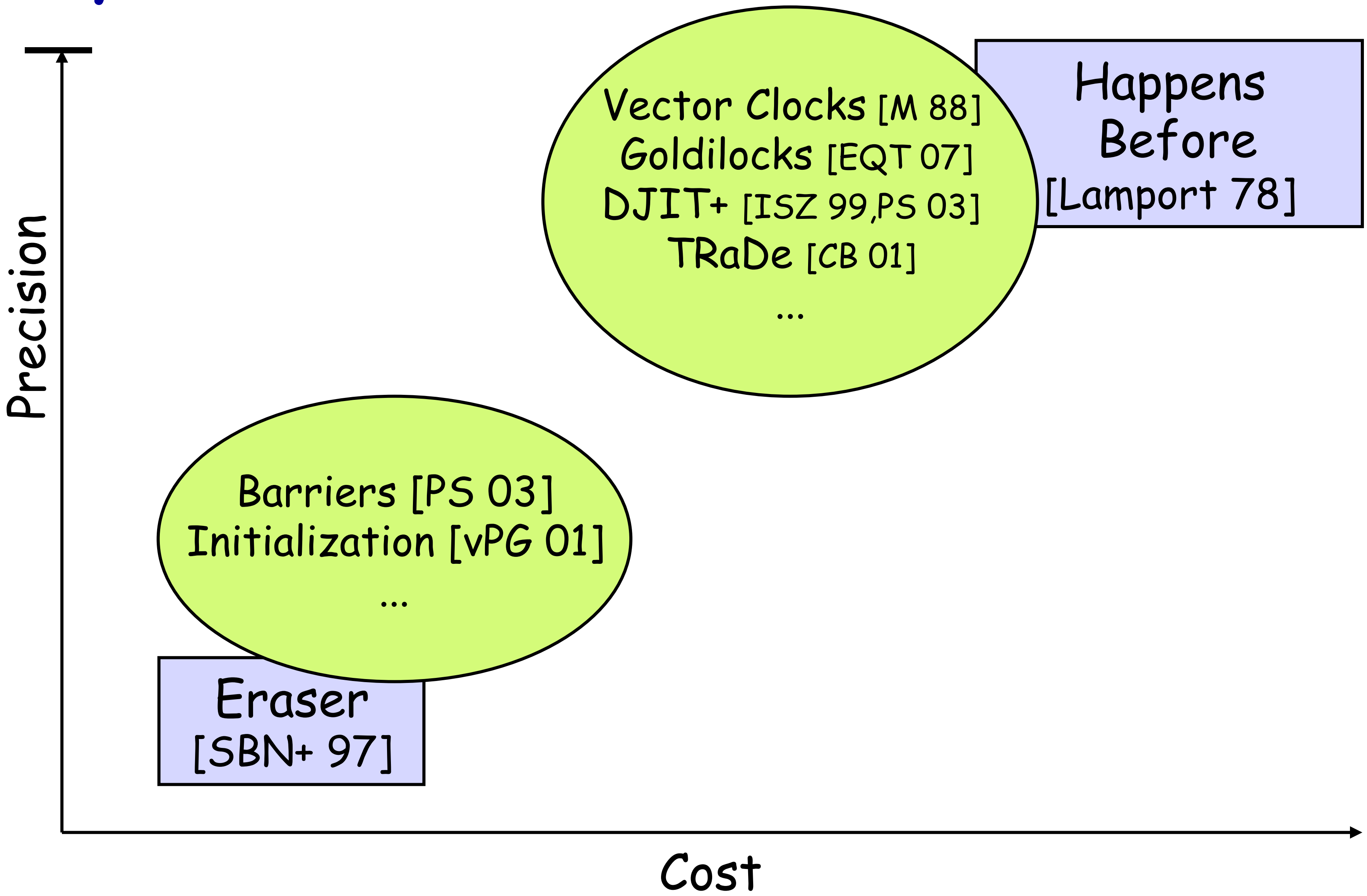
Thread A

```
data = 42;
sync(m) {
 ready = true;
}
```

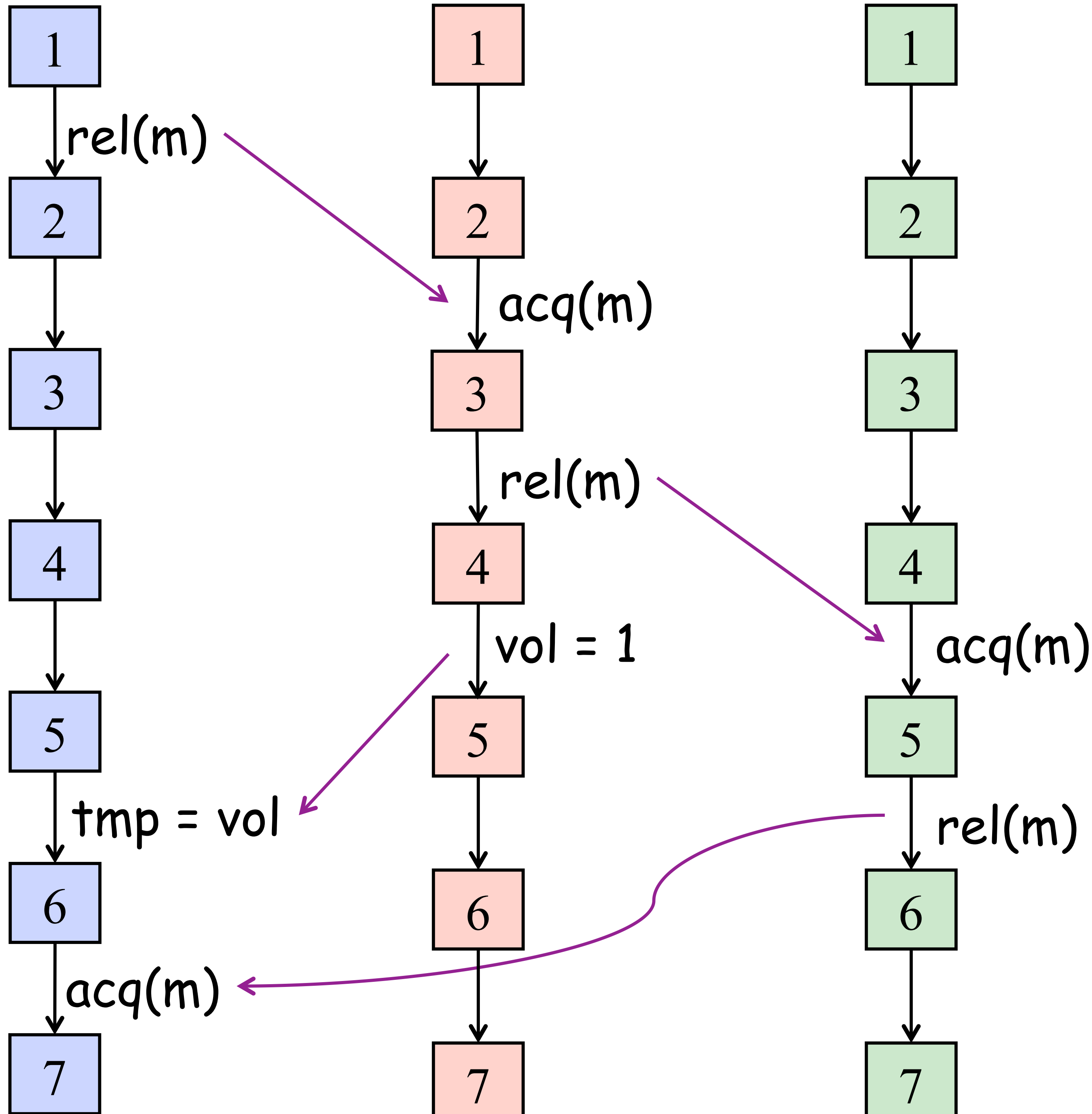
Thread B

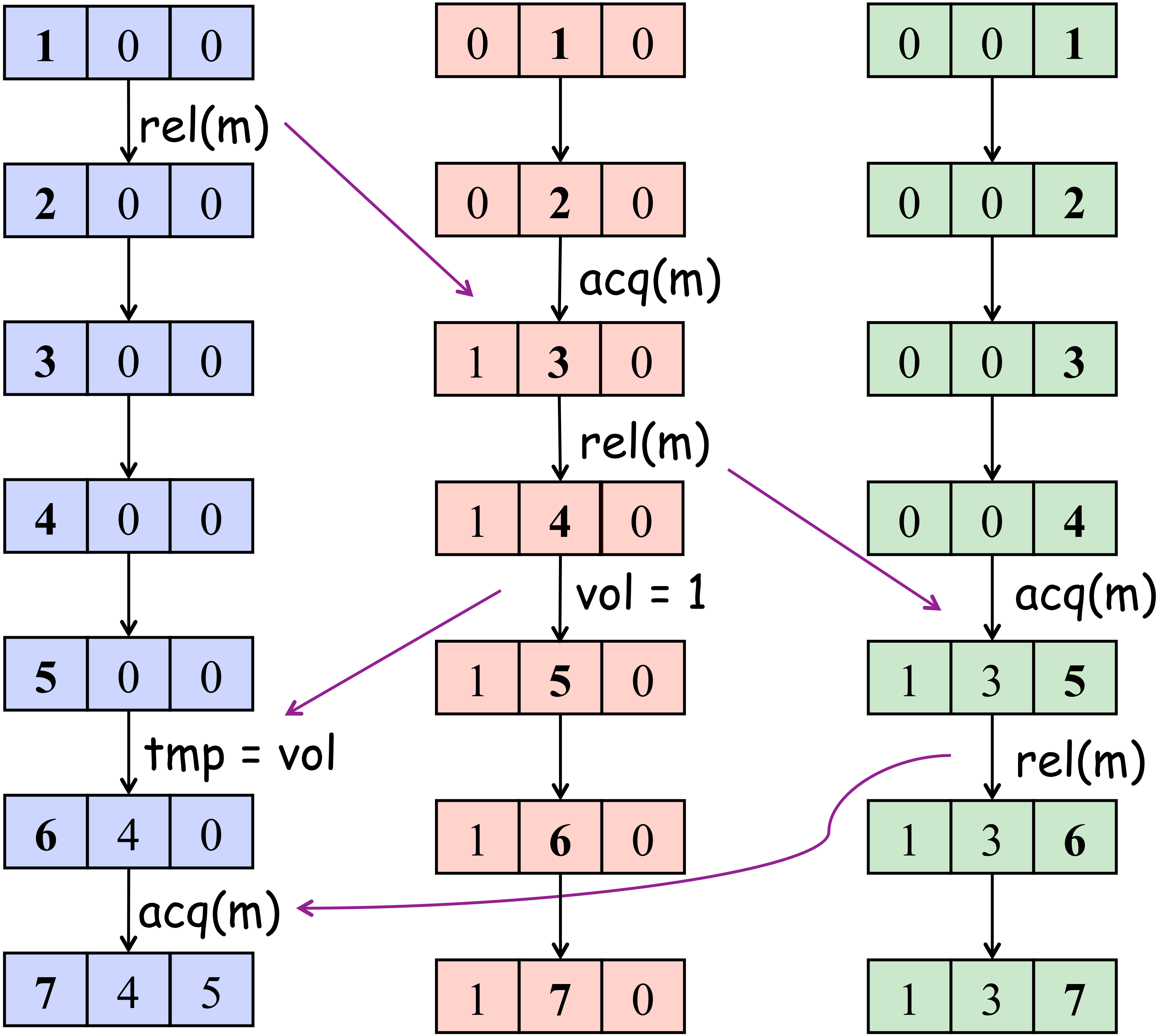
```
sync(m) {
 tmp = ready;
}
if(tmp)
  print(data)
```

Dynamic Data-Race Detection

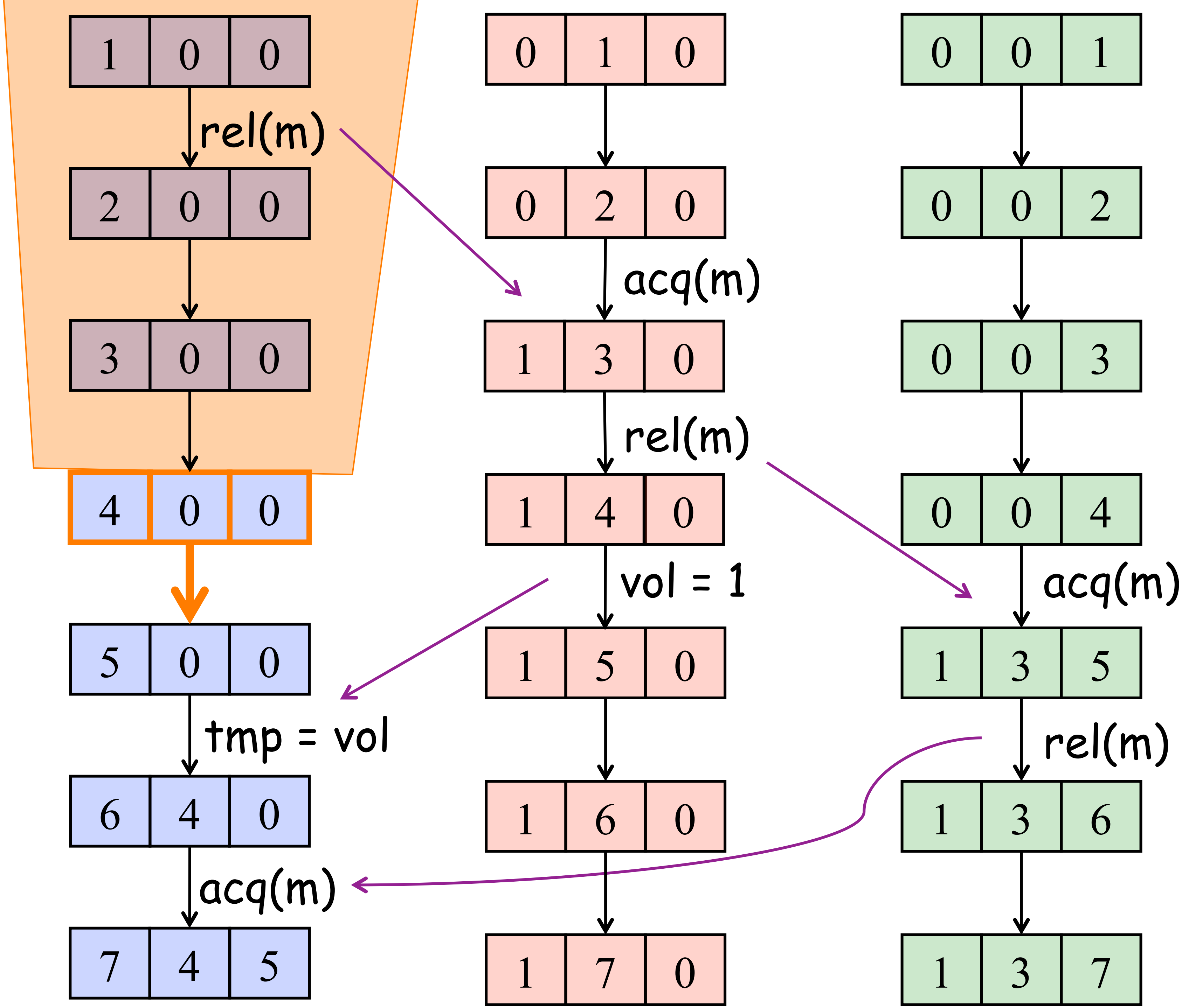


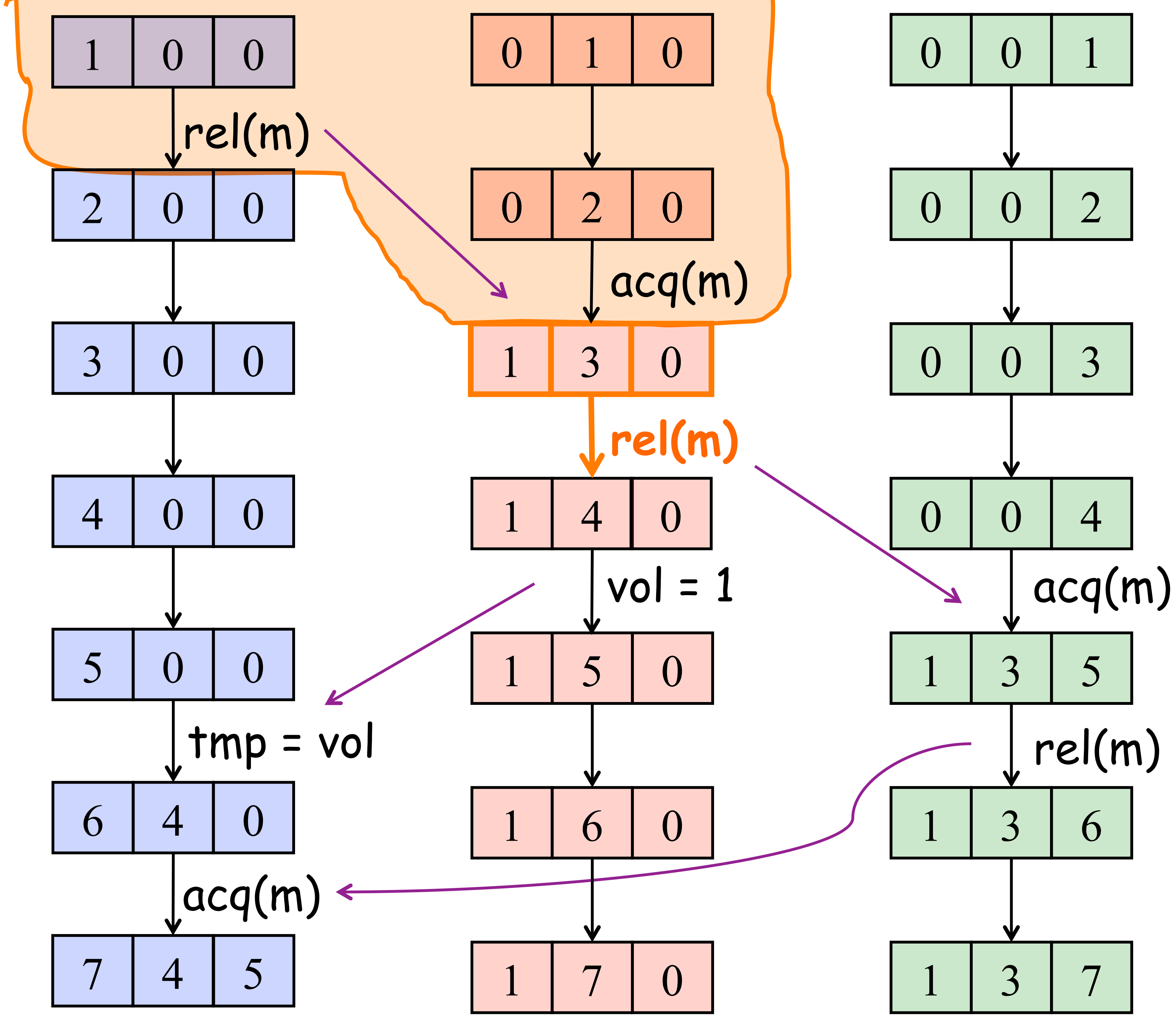
Precise Happens-Before



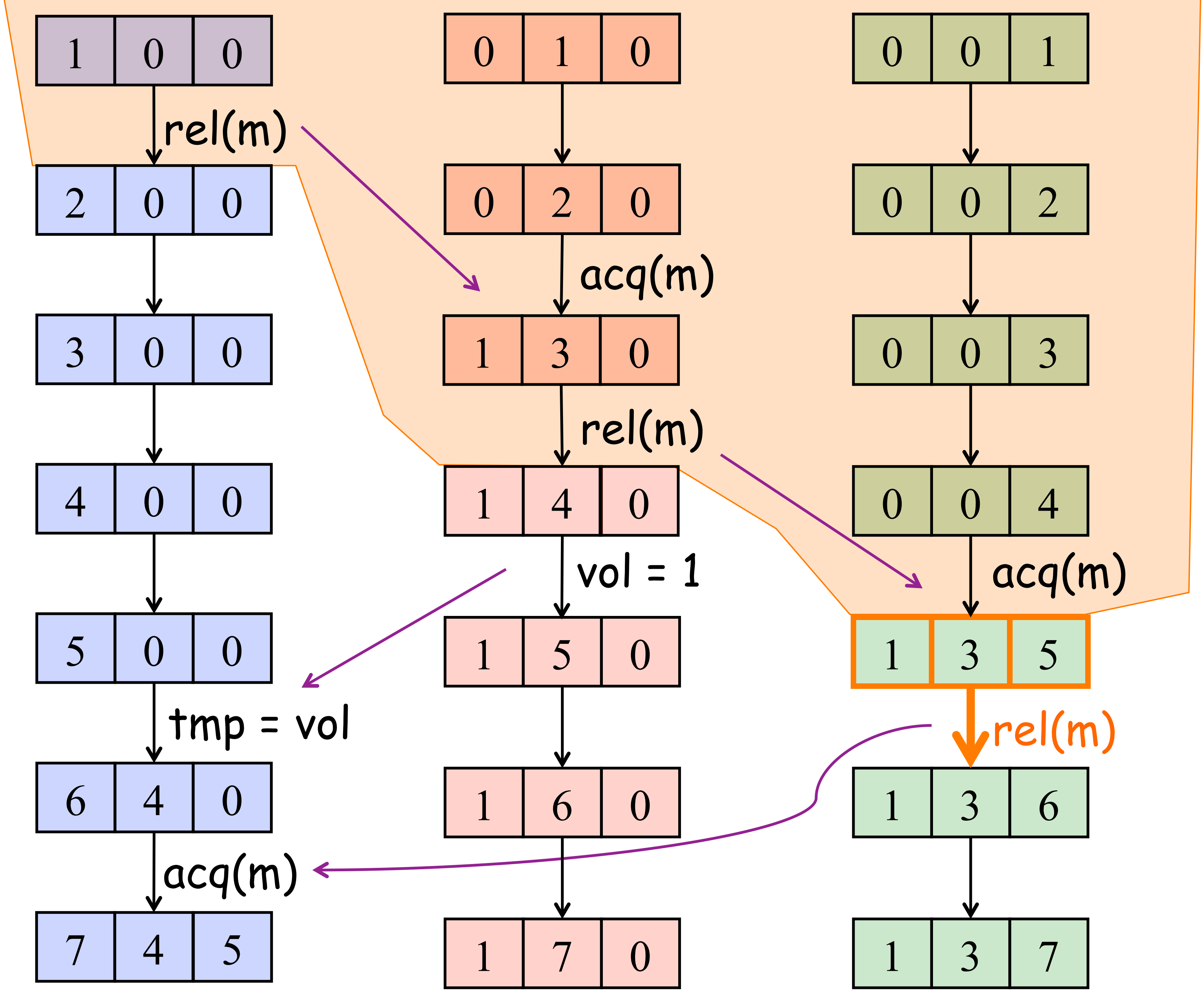


1 0 0
rel(m)
2 0 0
3 0 0
4 0 0
5 0 0
tmp = vol
6 4 0
acq(m)
7 4 5
0 1 0
0 2 0
acq(m)
1 3 0
rel(m)
1 4 0
vol = 1
1 5 0
1 6 0
1 7 0
0 0 1
0 0 2
0 0 3
0 0 4
acq(m)
1 3 5
rel(m)
1 3 6
1 3 7

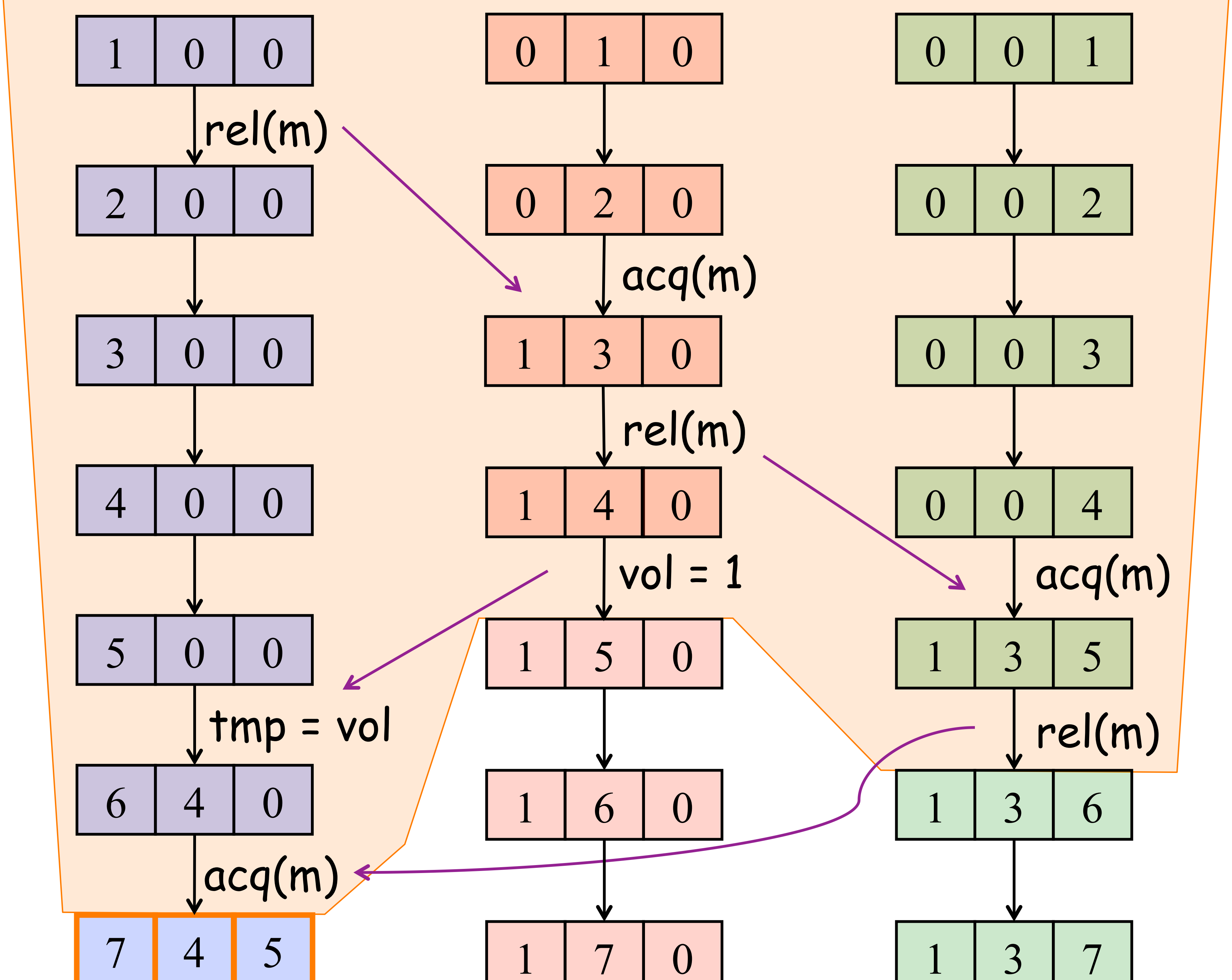

1 0 0
rel(m)
2 0 0
3 0 0
4 0 0
5 0 0
tmp = vol
6 4 0
acq(m)
7 4 5
0 1 0
0 2 0
acq(m)
1 3 0
rel(m)
1 4 0
vol = 1
1 5 0
1 6 0
1 7 0
0 0 1
0 0 2
0 0 3
0 0 4
acq(m)
1 3 5
rel(m)
1 3 6
1 3 7



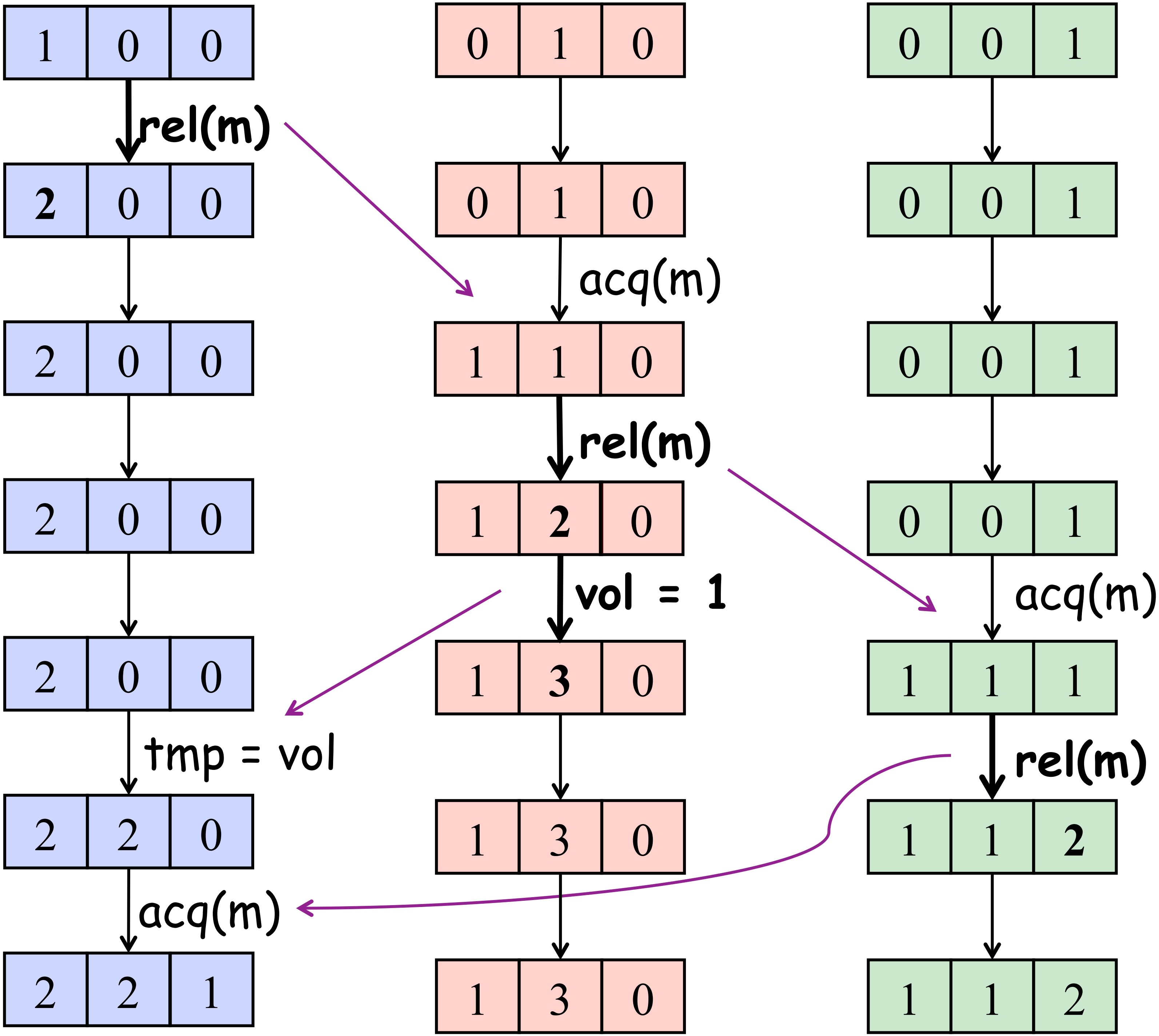
rel(m)
acq(m)
rel(m)
vol = 1
acq(m)
tmp = vol
rel(m)
acq(m)



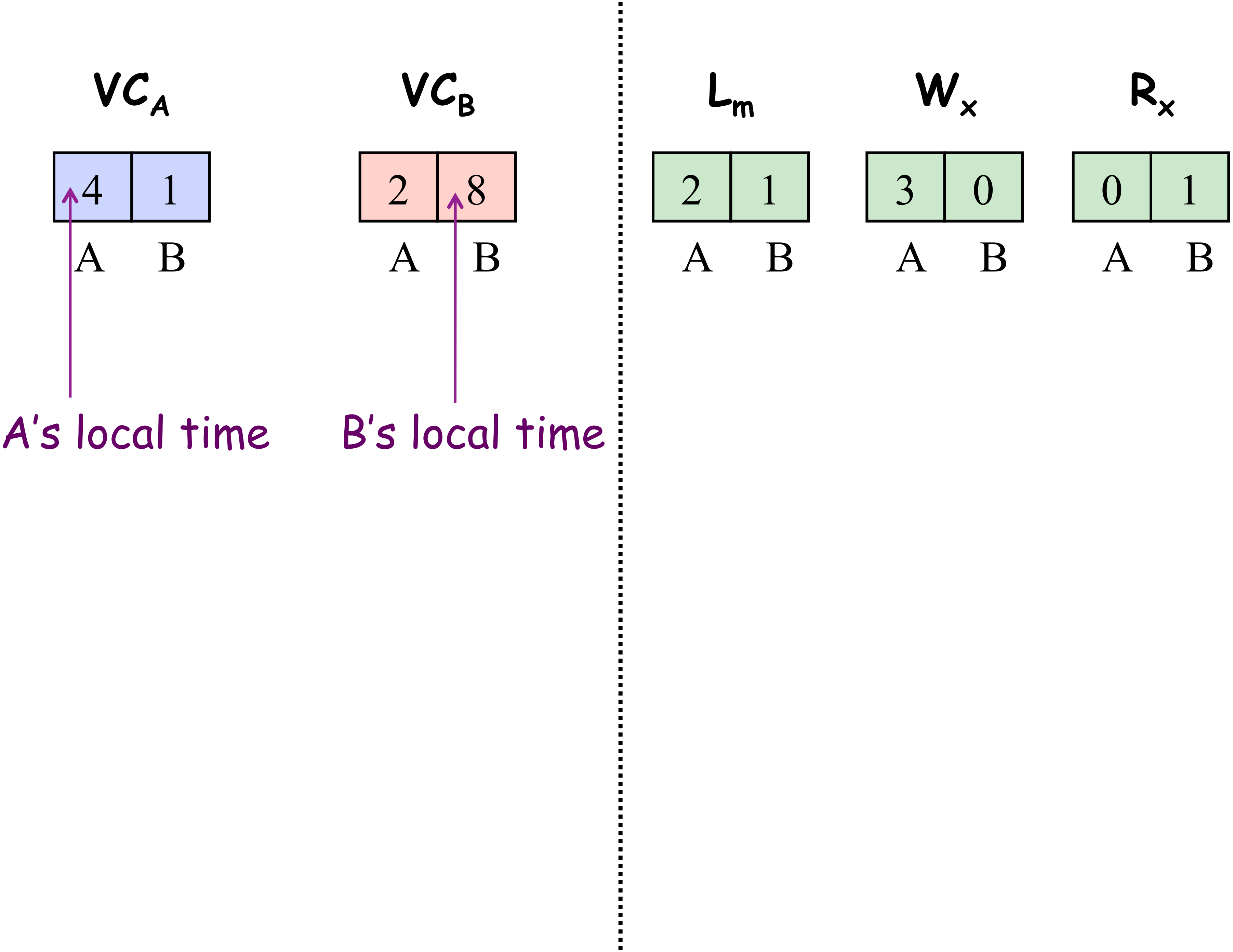
1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 2 0
0 0 2
acq(m)
3 0 0
1 3 0
0 0 3
rel(m)
4 0 0
1 4 0
0 0 4
vol = 1
acq(m)
5 0 0
1 5 0
1 3 5
tmp = vol
rel(m)
6 4 0
1 6 0
1 3 6
acq(m)
7 4 5
1 7 0
1 3 7



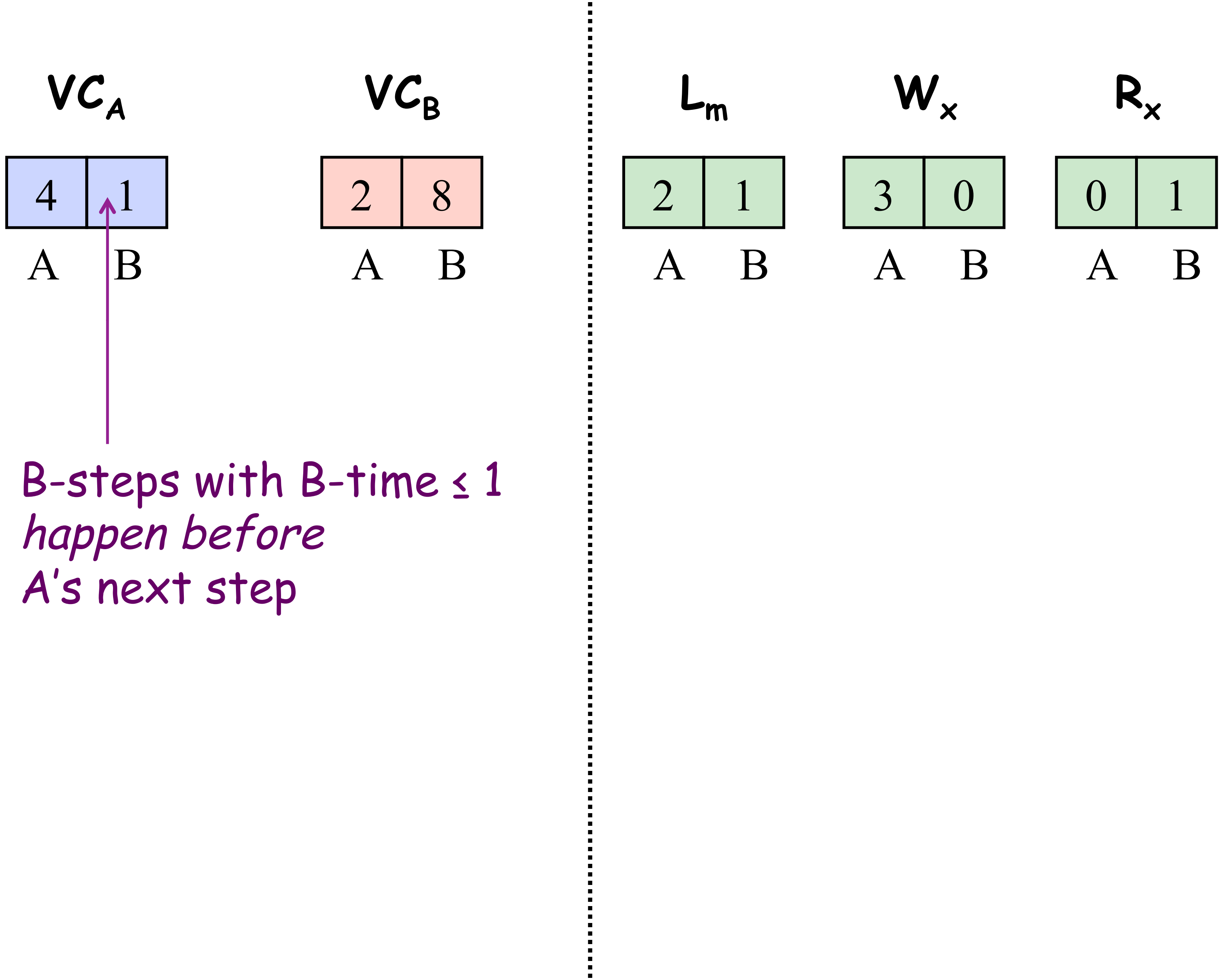
1 0 0
rel(m)
2 0 0
3 0 0
4 0 0
5 0 0
tmp = vol
6 4 0
acq(m)
7 4 5
0 1 0
0 2 0
acq(m)
1 3 0
rel(m)
1 4 0
vol = 1
1 5 0
1 6 0
1 7 0
0 0 1
0 0 2
0 0 3
0 0 4
acq(m)
1 3 5
rel(m)
1 3 6
1 3 7



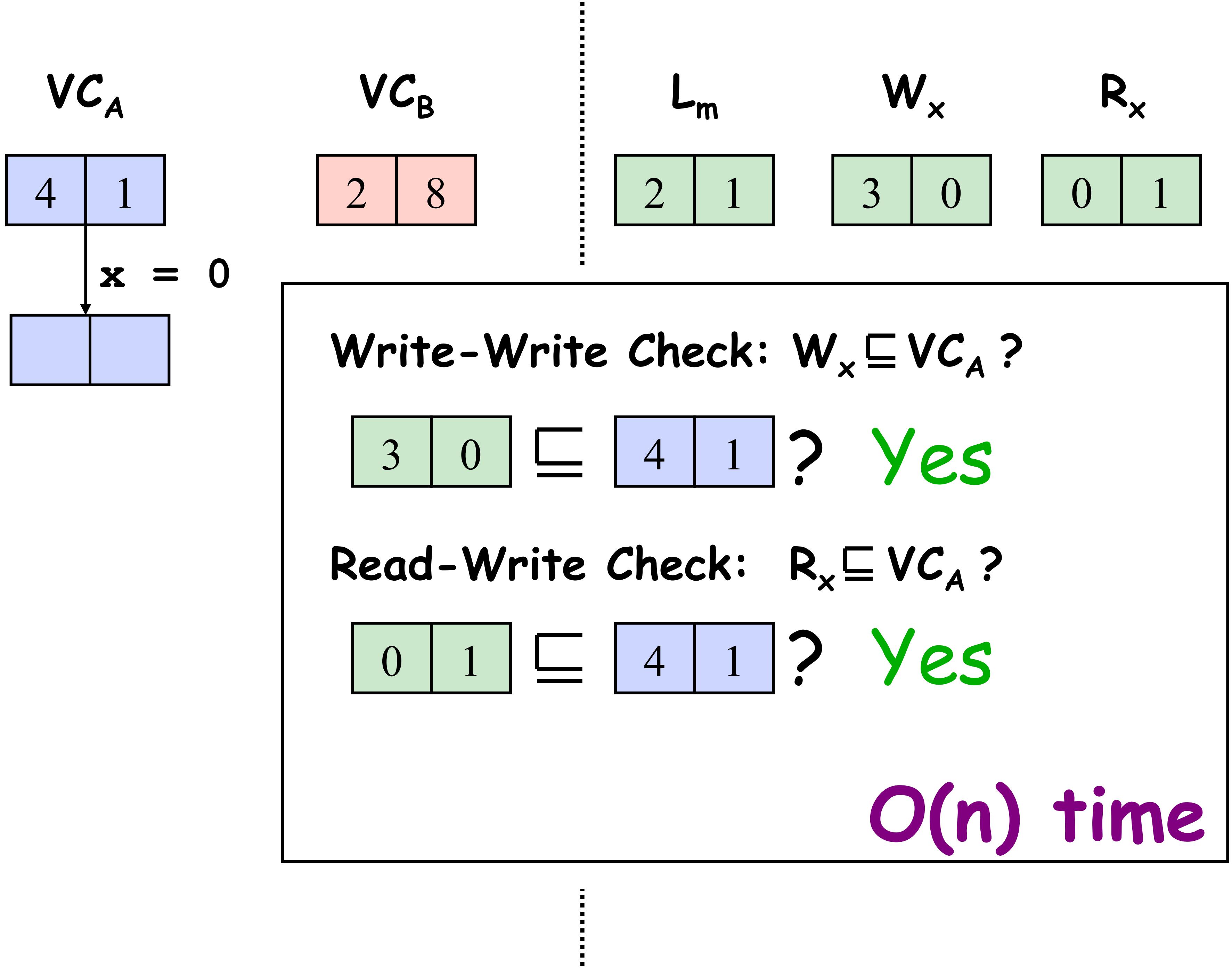
1 0 0
rel(m)
2 0 0
2 0 0
2 0 0
2 0 0
tmp = vol
2 2 0
acq(m)
2 2 1
0 1 0
0 1 0
acq(m)
1 1 0
rel(m)
1 2 0
vol = 1
1 3 0
1 3 0
1 3 0
0 0 1
0 0 1
0 0 1
0 0 1
acq(m)
1 1 1
rel(m)
1 1 2
1 1 2



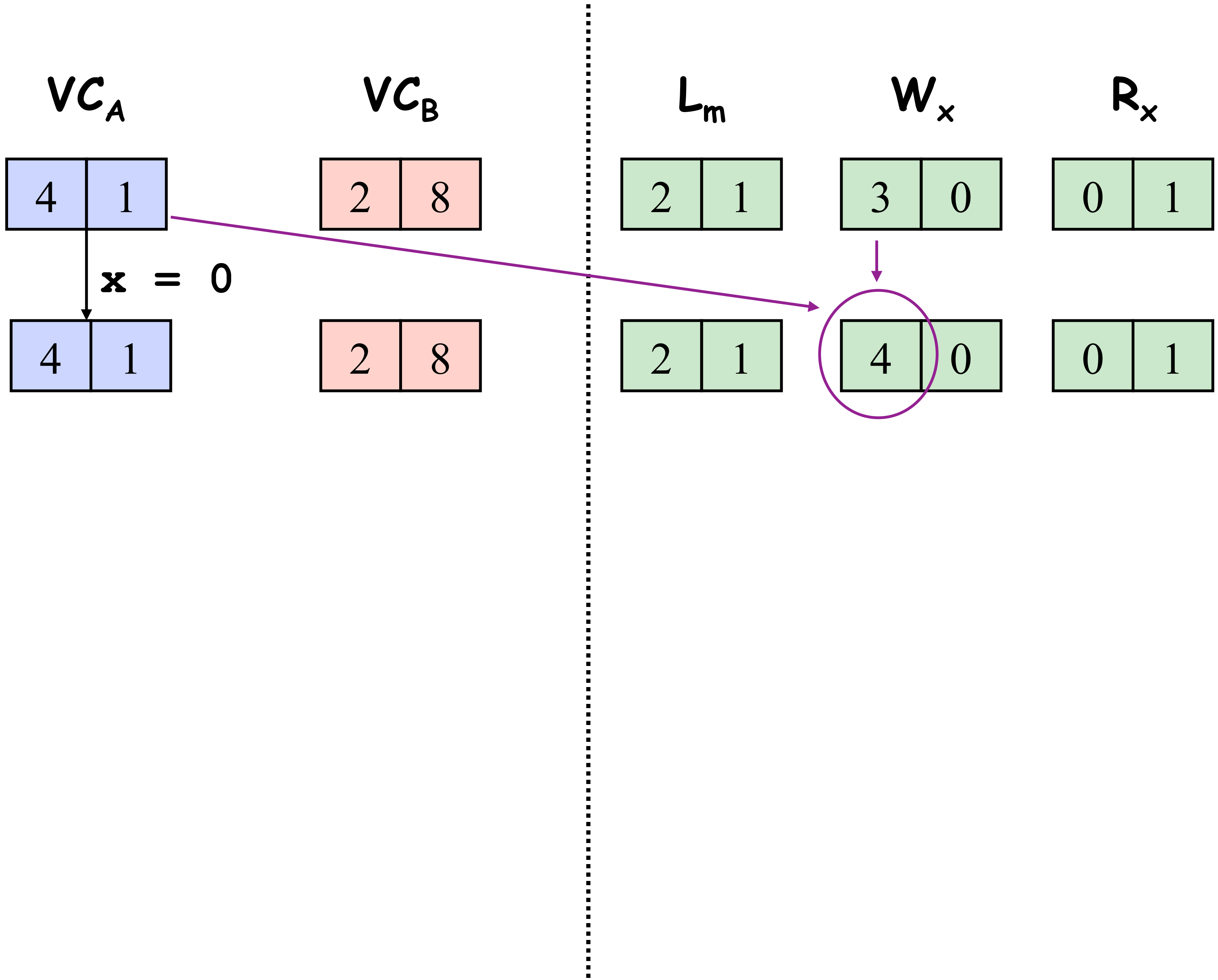
VC_A
4 1
A B
VC_B
2 8
A B
L_m
2 1
A B
W_x
3 0
A B
R_x
0 1
A B
A's local time
B's local time



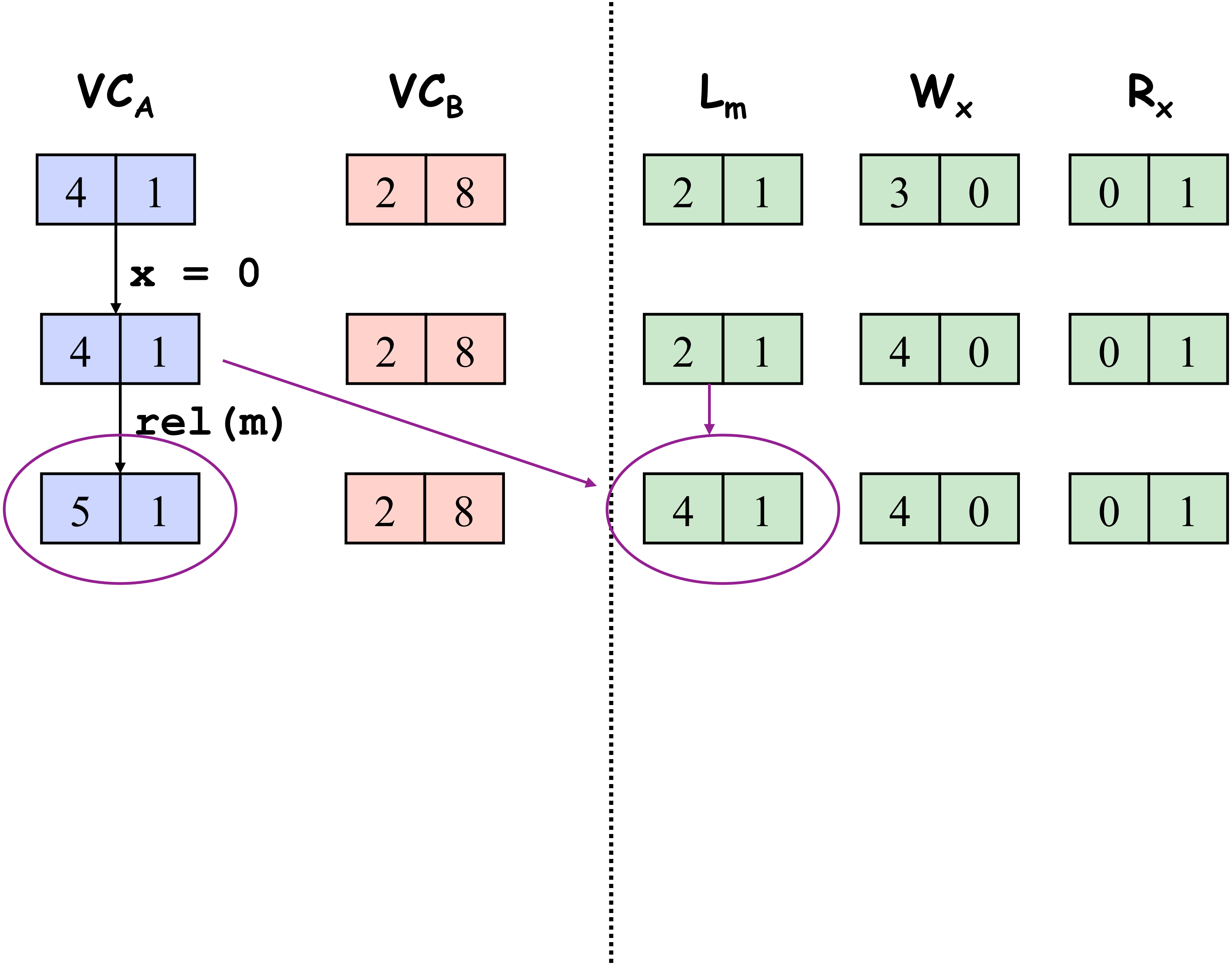
VC_A
4 1
A B
VC_B
2 8
A B
L_m
2 1
A B
W_x
3 0
A B
R_x
0 1
A B
B-steps with B-time ≤ 1
happen before
A's next step



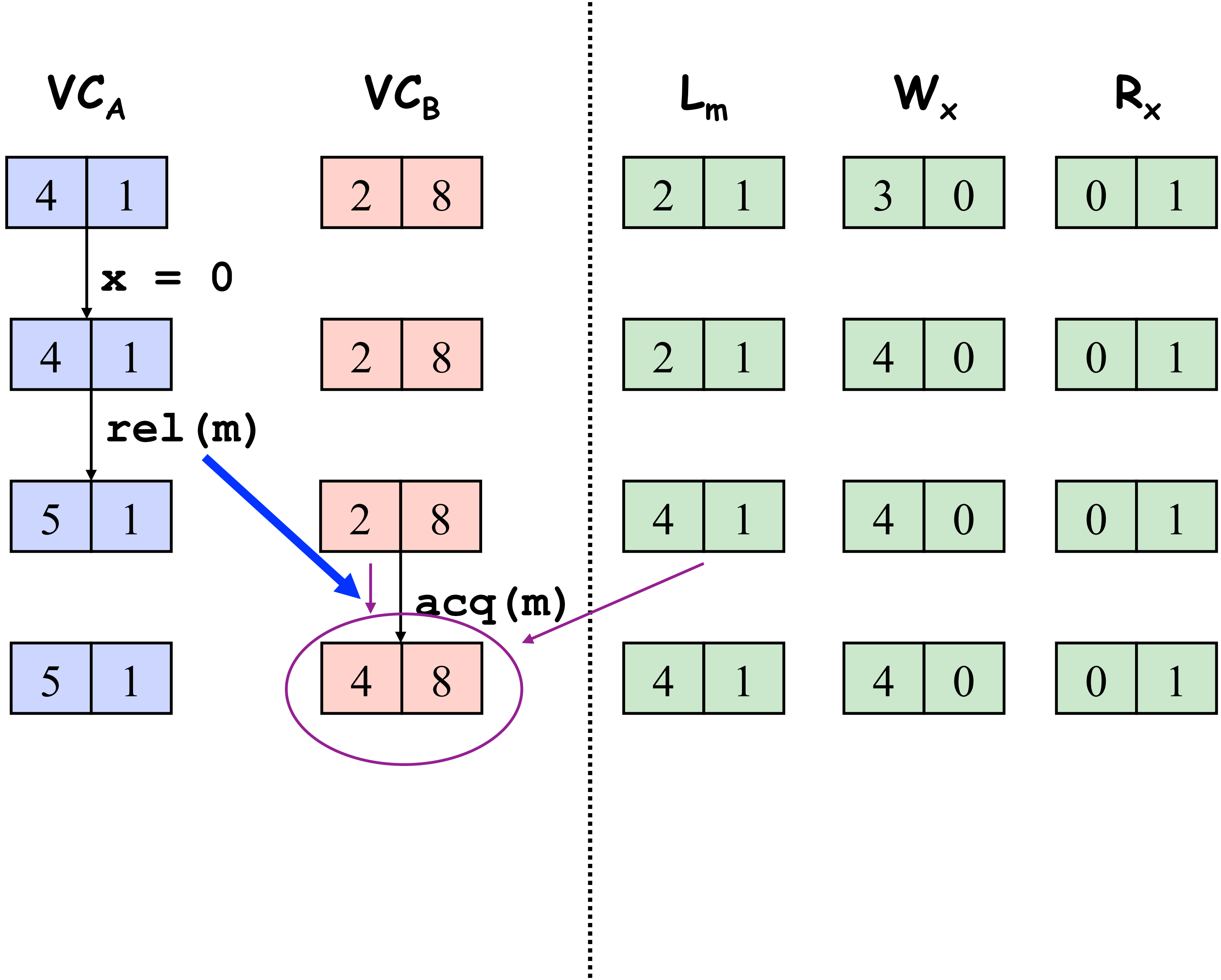
VC_A
4 1
x = 0
VC_B
2 8
L_m
2 1
W_x
3 0
R_x
0 1
Write-Write Check: W_x ⊑ VC_A ?
3 0 ⊑ 4 1 ? Yes
Read-Write Check: R_x ⊑ VC_A ?
0 1 ⊑ 4 1 ? Yes
O(n) time



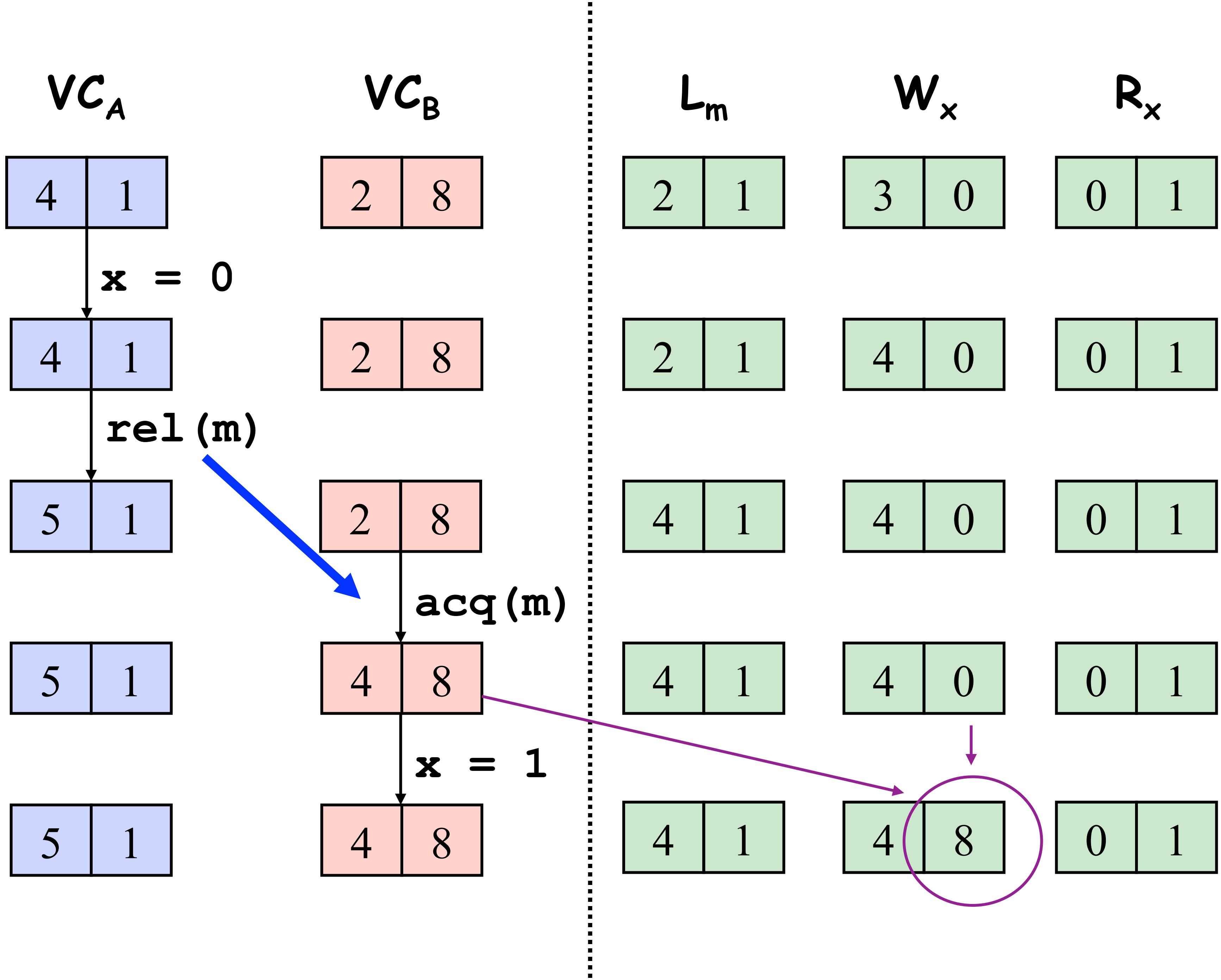
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1



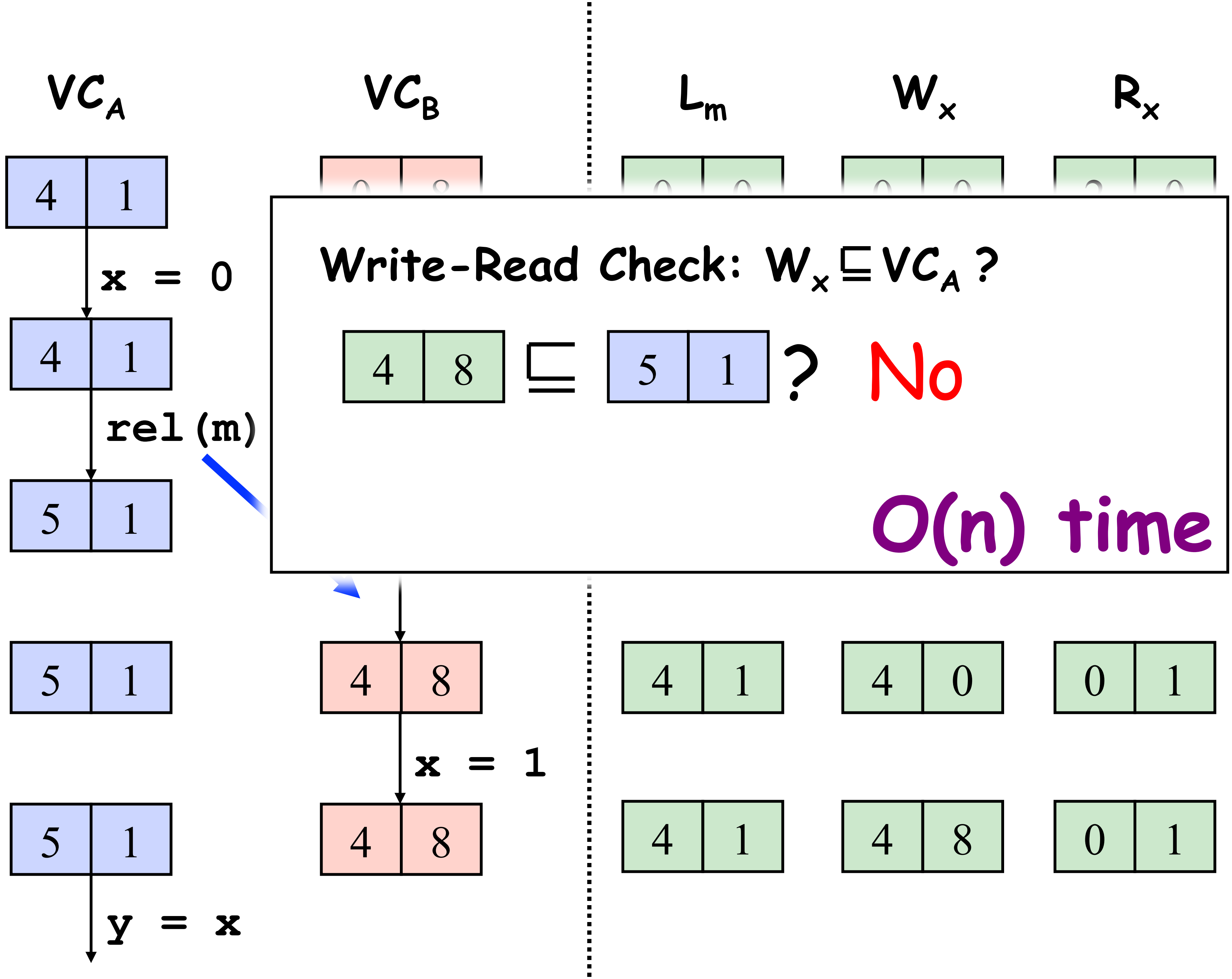
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1



VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1
acq(m)
5 1
4 8
4 1
4 0
0 1



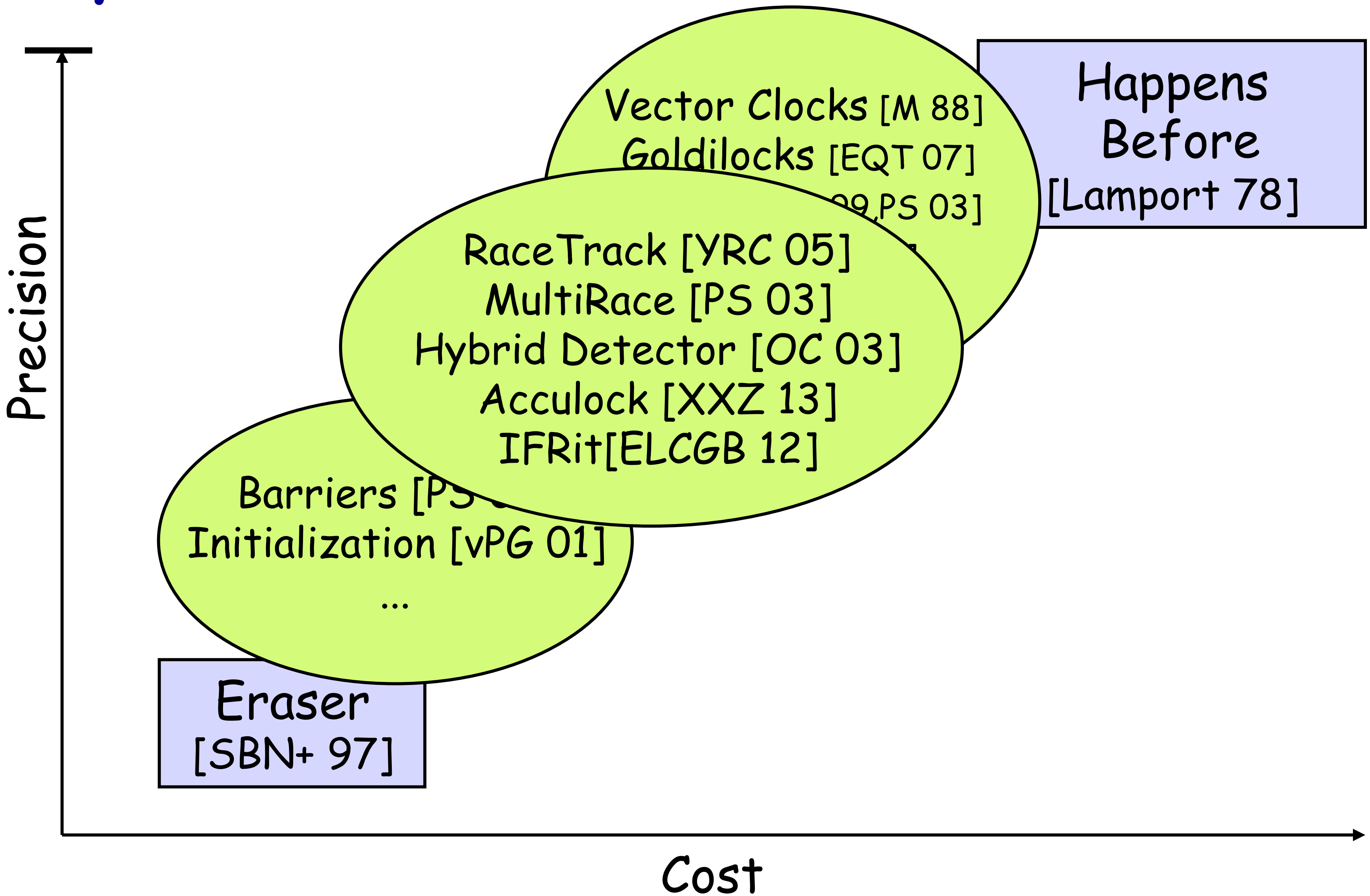
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1
acq(m)
5 1
4 8
4 1
4 0
0 1
x = 1
5 1
4 8
4 1
4 8
0 1


VC_A
VC_B
L_m
W_x
R_x
x = 0
rel(m)
Write-Read Check: W_x ⊑ VC_A ?
4 8 ⊑ 5 1 ? No
O(n) time
x = 1
y = x

VectorClocks for Data-Race Detection

- Sound
  - No warnings → data-race-free execution
- Complete
  - Warning → data-race exists
- Slow performance
  - (slowdowns > 50x)

Dynamic Data-Race Detection



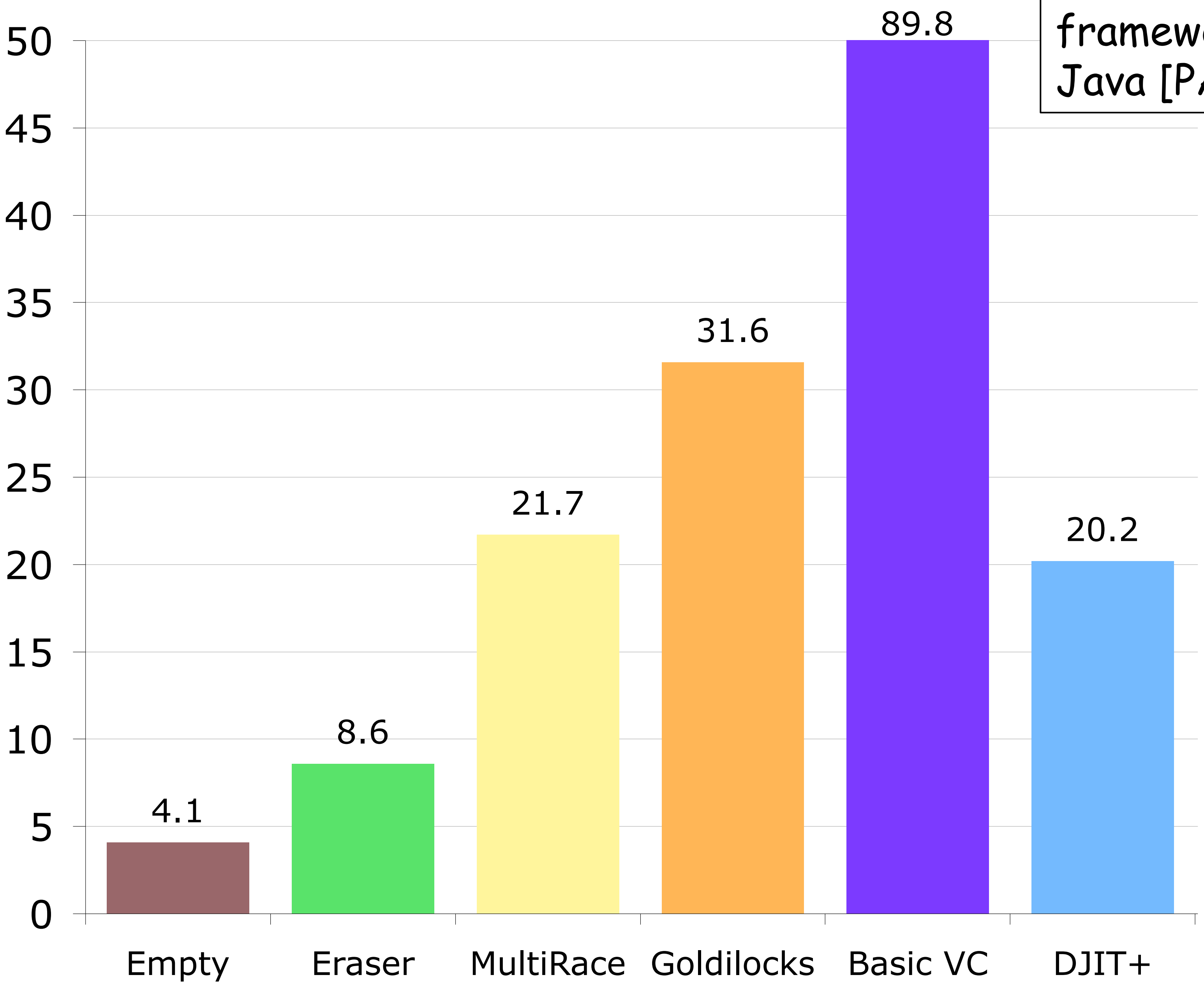
Combined Approaches

- MultiRace [PS 03,07]
  - Use LockSet for x
  - Switch to VC if LockSet becomes empty
  - (adaptive granularity as well)

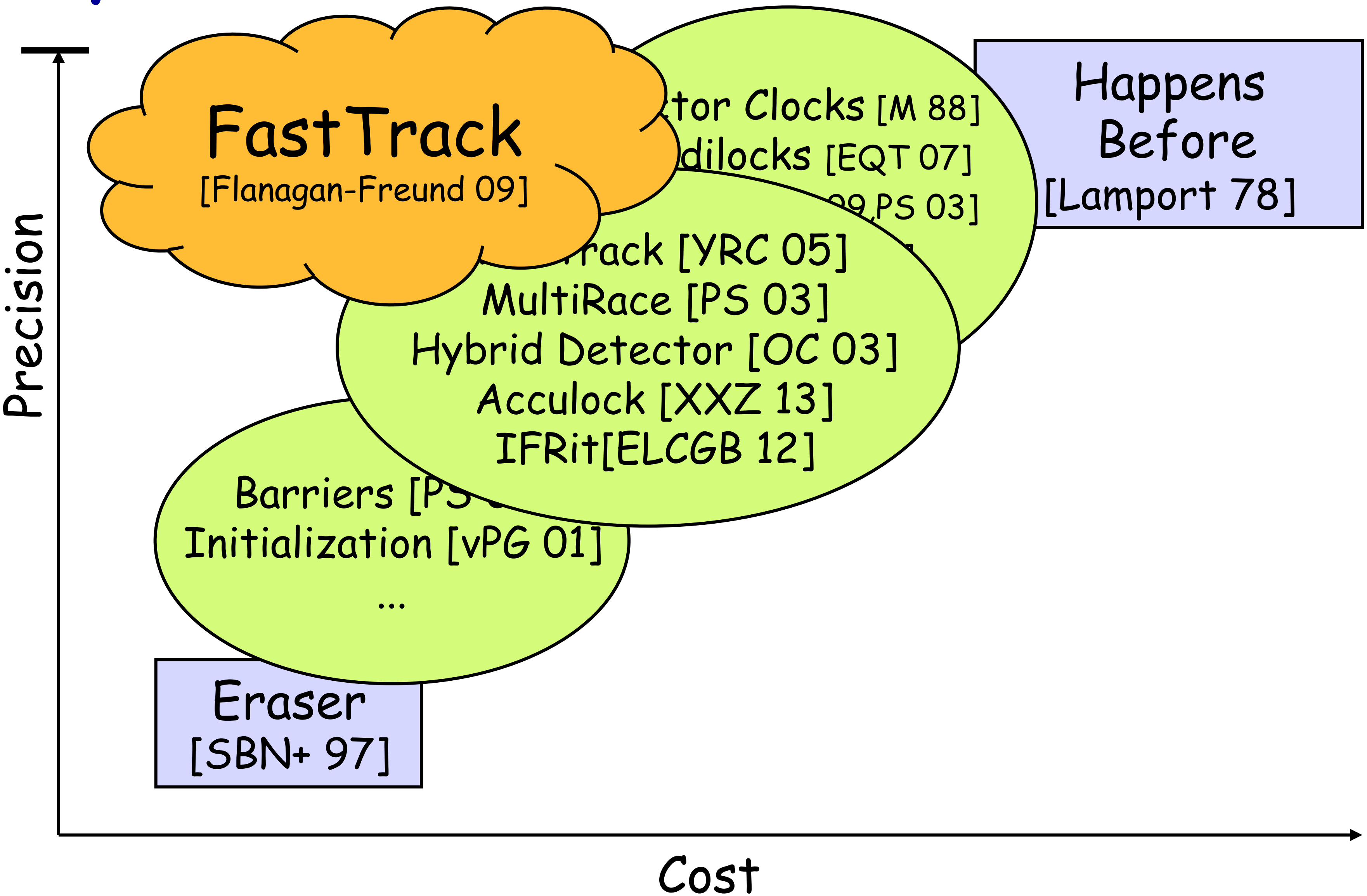
- RaceTrack [YRC 05]
  - Use Locket for x with extensions to Eraser state machine.
  - Use VCs to reason about fork/join and wait/notify

Slowdown (x Base Time)

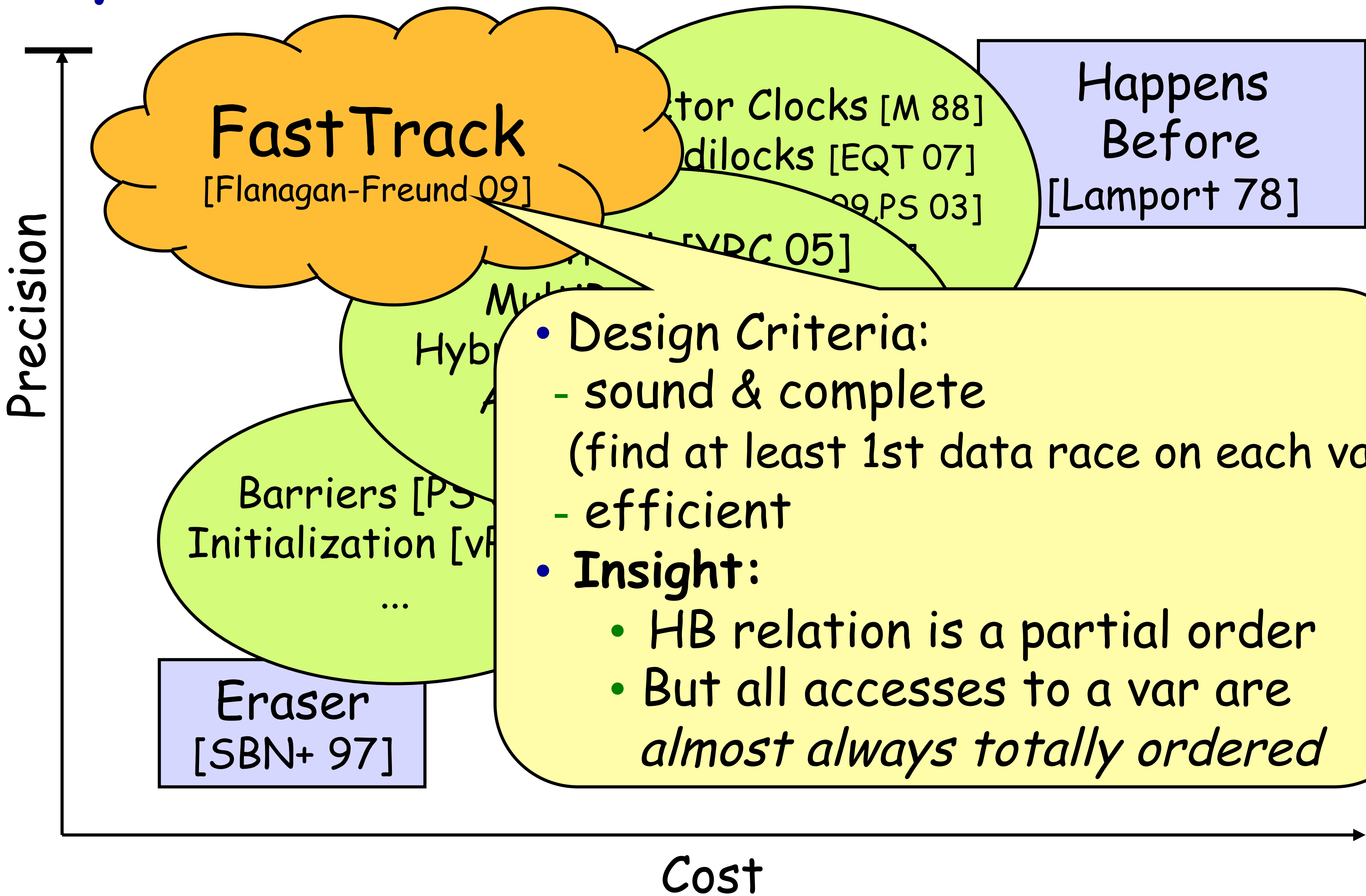
Tools implemented in RoadRunner framework for Java [PASTE 10]



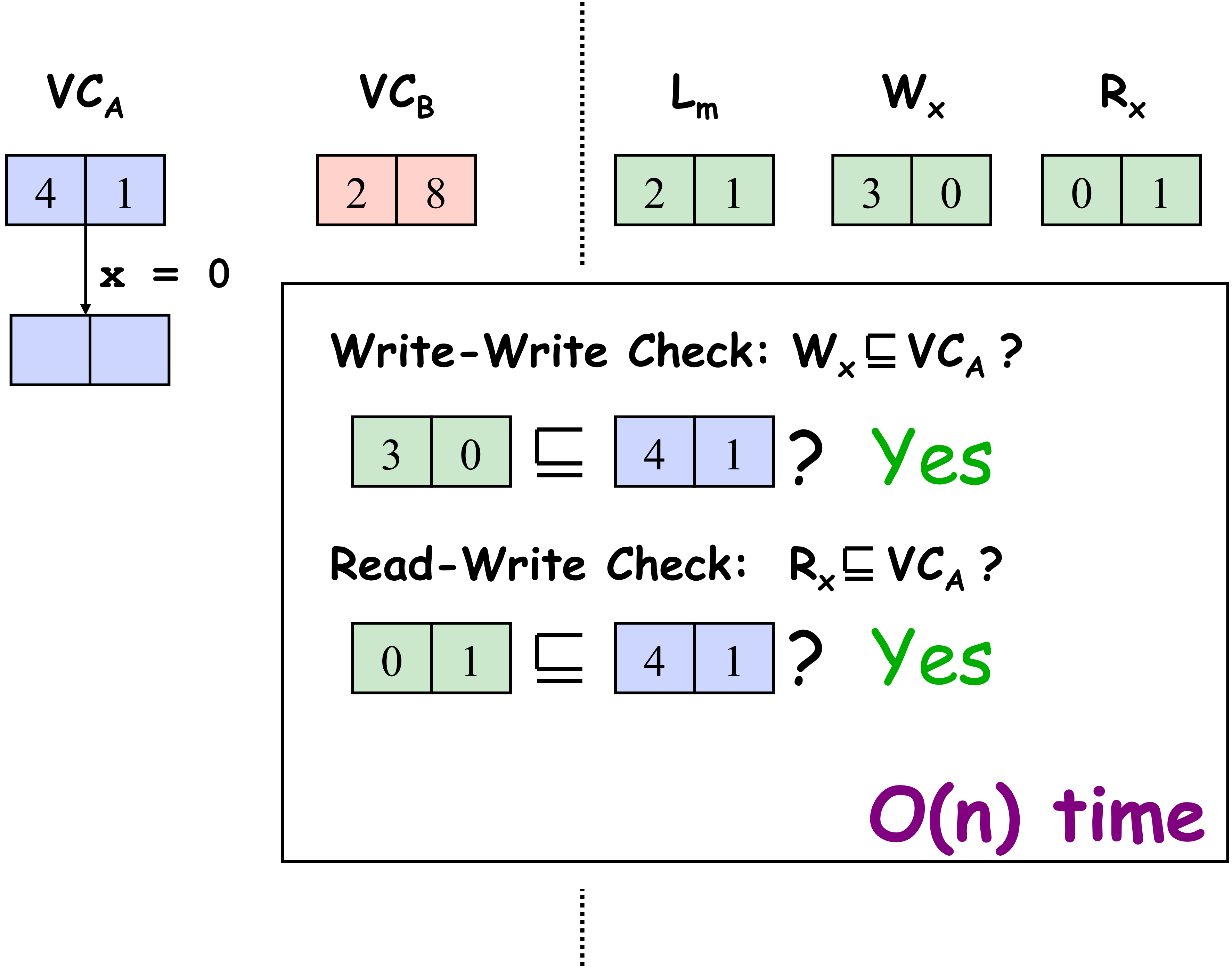
Dynamic Data-Race Detection



Dynamic Data-Race Detection

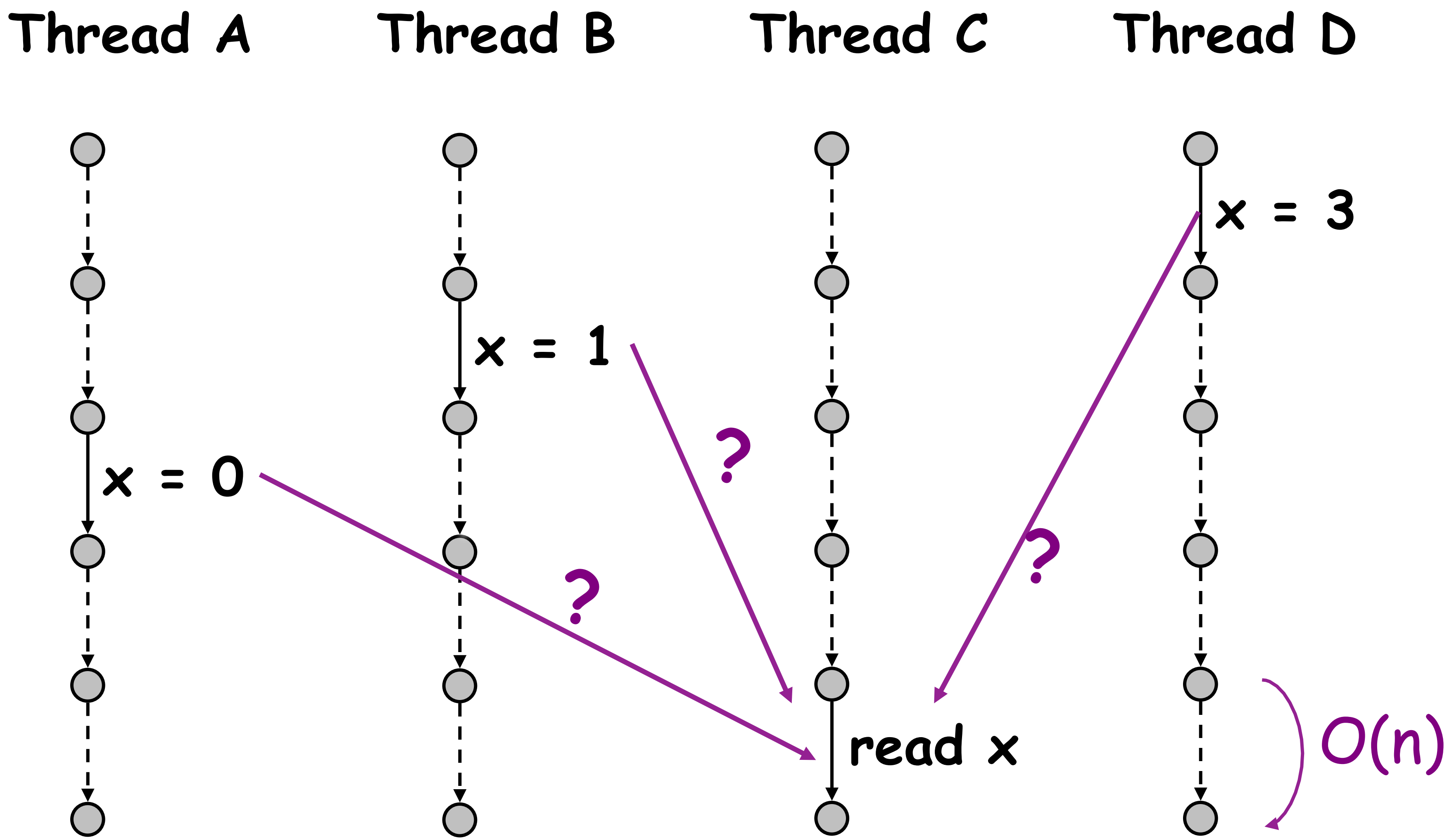


- Design Criteria:
  - sound & complete
    (find at least 1st data race on each var)
  - efficient
- **Insight:**
  - HB relation is a partial order
  - But all accesses to a var are *almost always totally ordered*

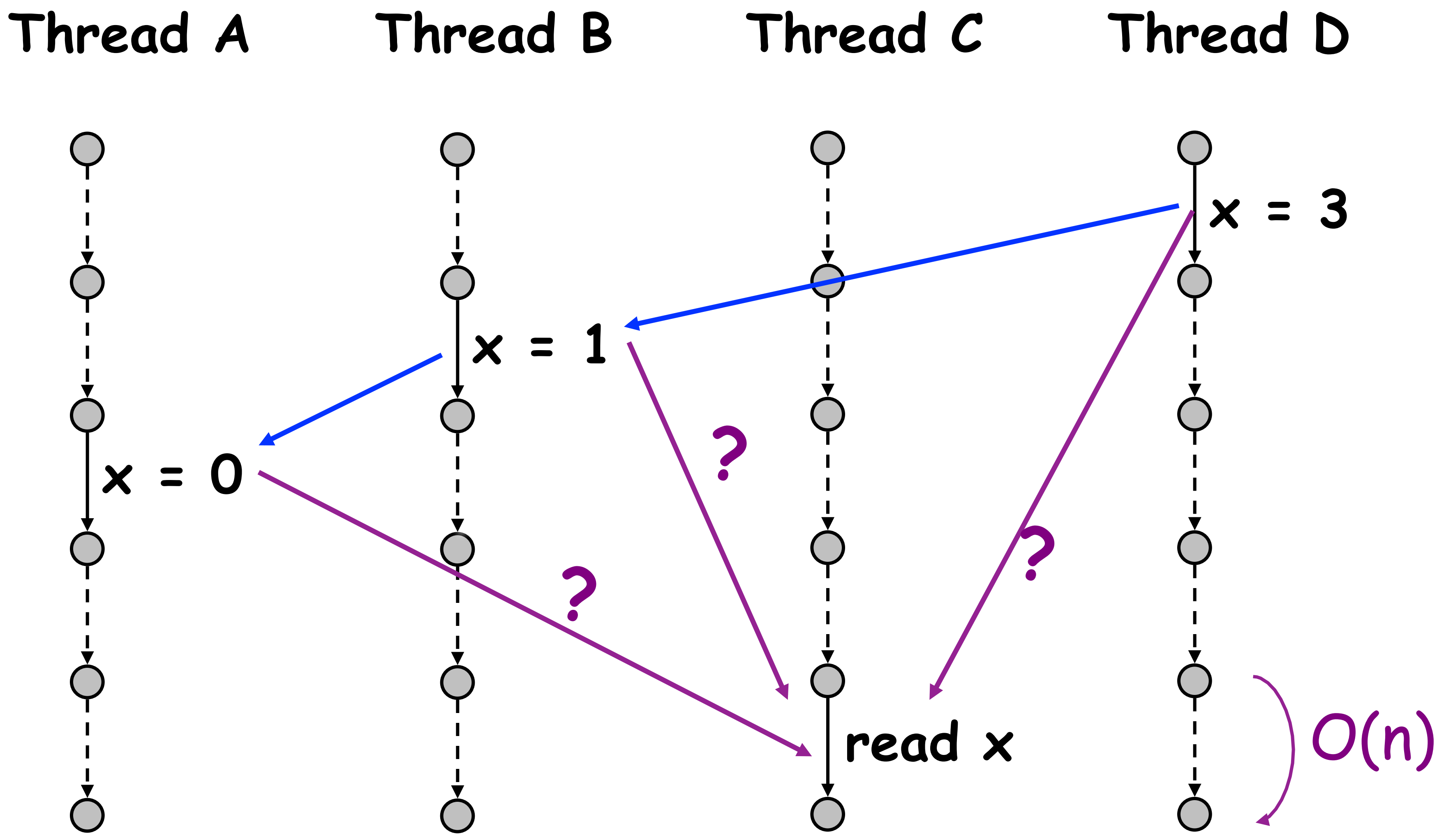


VC_A
4 1
x = 0
VC_B
2 8
L_m
2 1
W_x
3 0
R_x
0 1
Write-Write Check: $W_x \sqsubseteq VC_A$?
3 0 $\sqsubseteq$ 4 1 ? Yes
Read-Write Check: $R_x \sqsubseteq VC_A$?
0 1 $\sqsubseteq$ 4 1 ? Yes
O(n) time

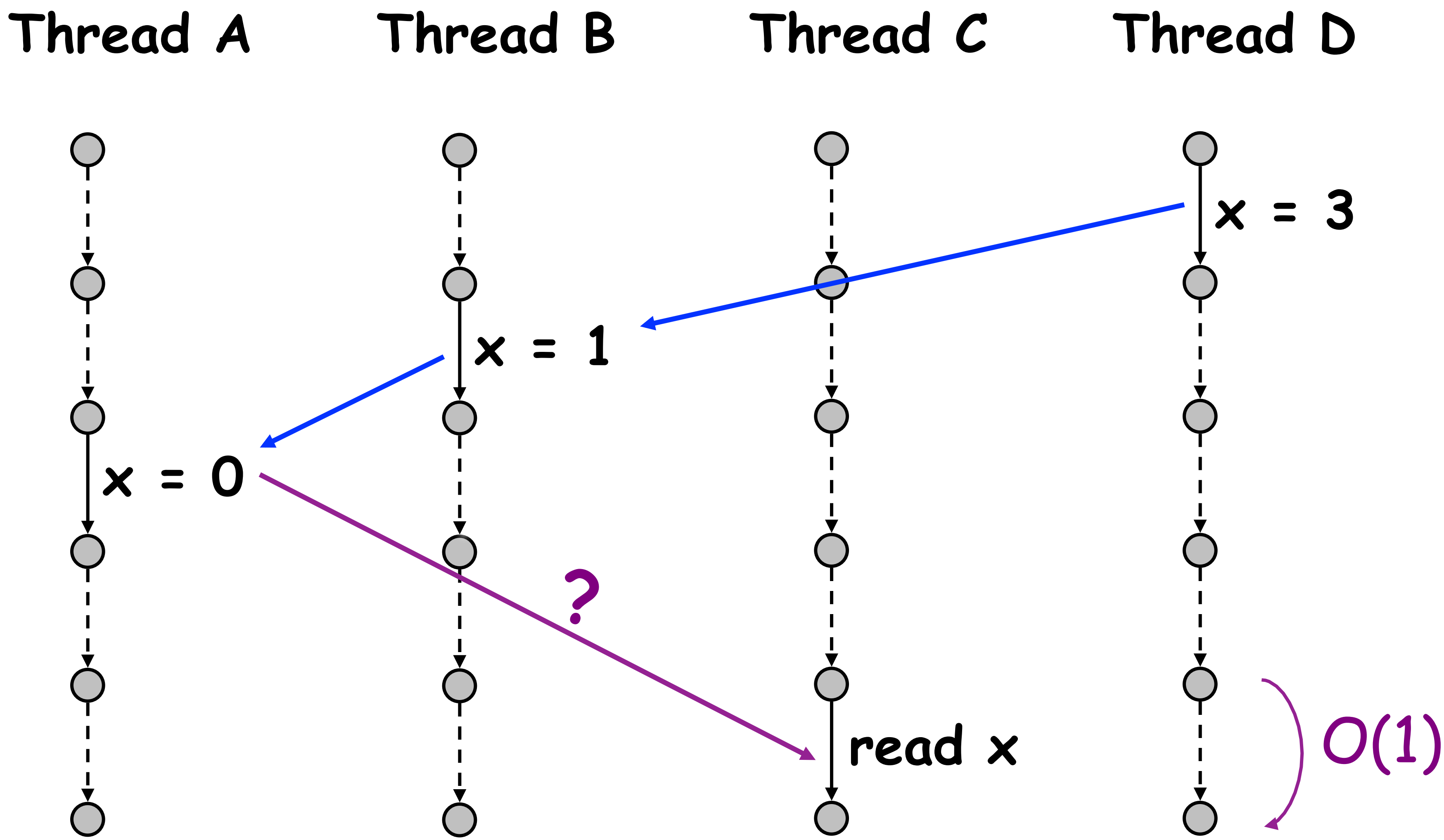
Write-Write and Write-Read Data Races

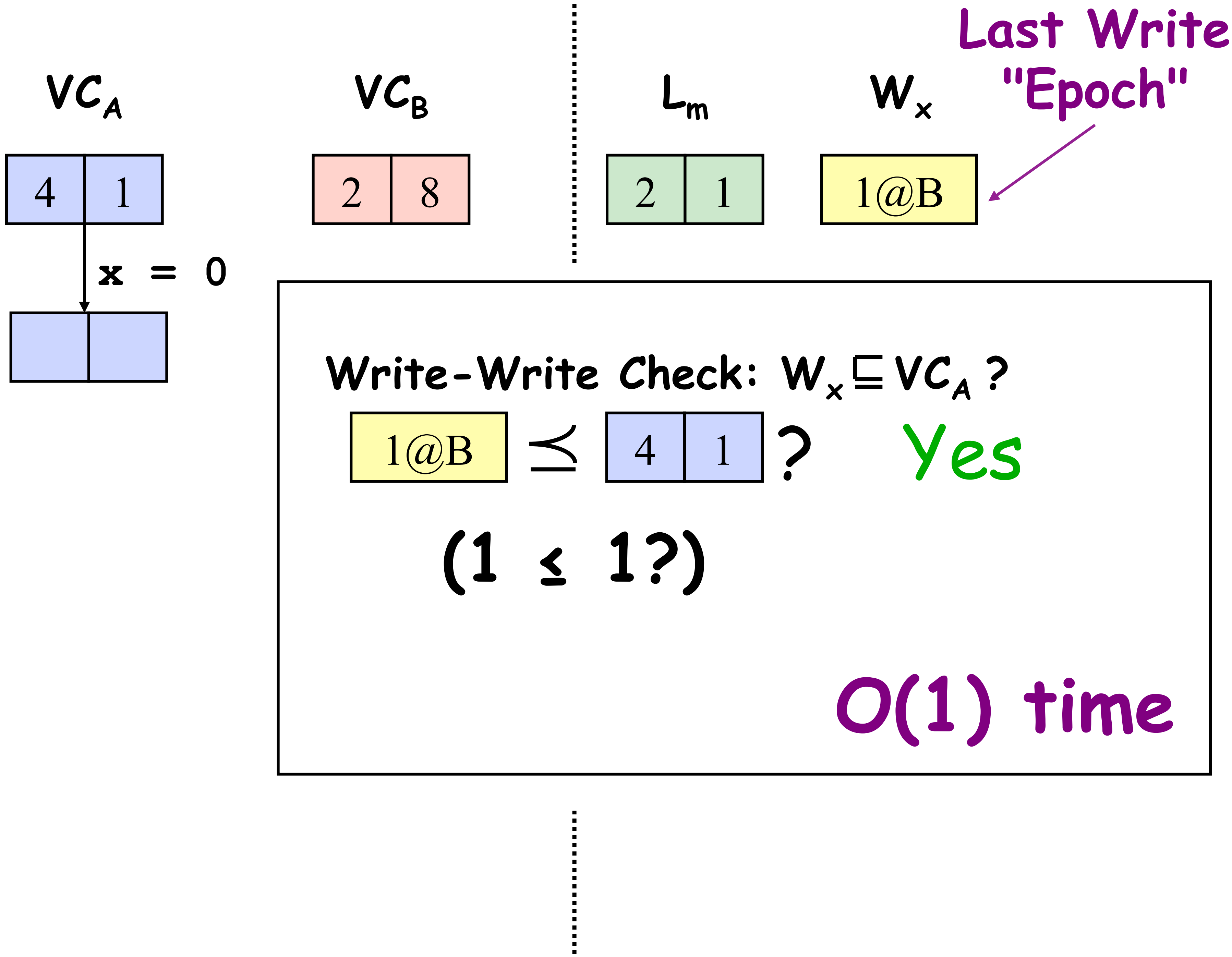


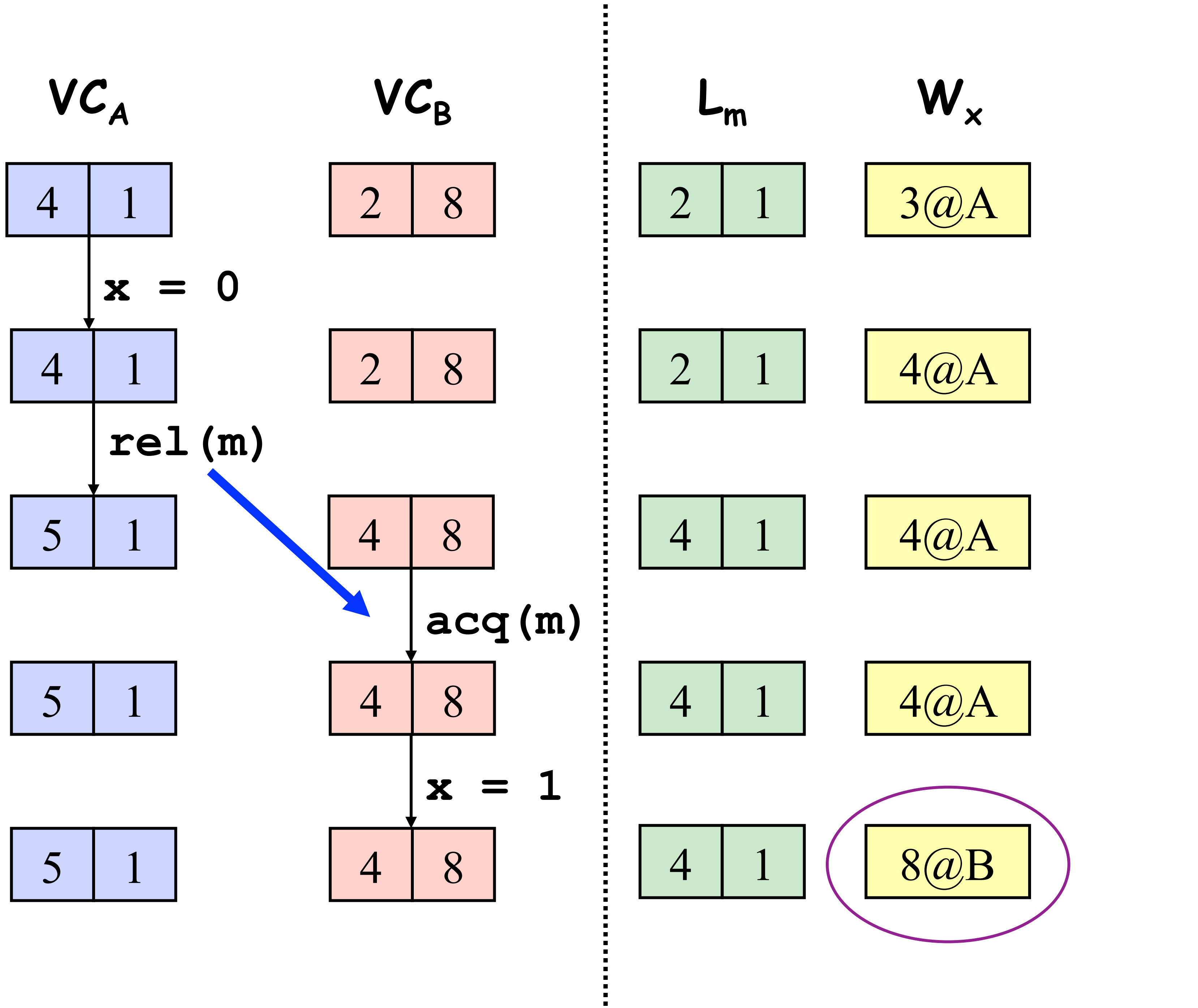
No Data Races Yet: Writes Totally Ordered

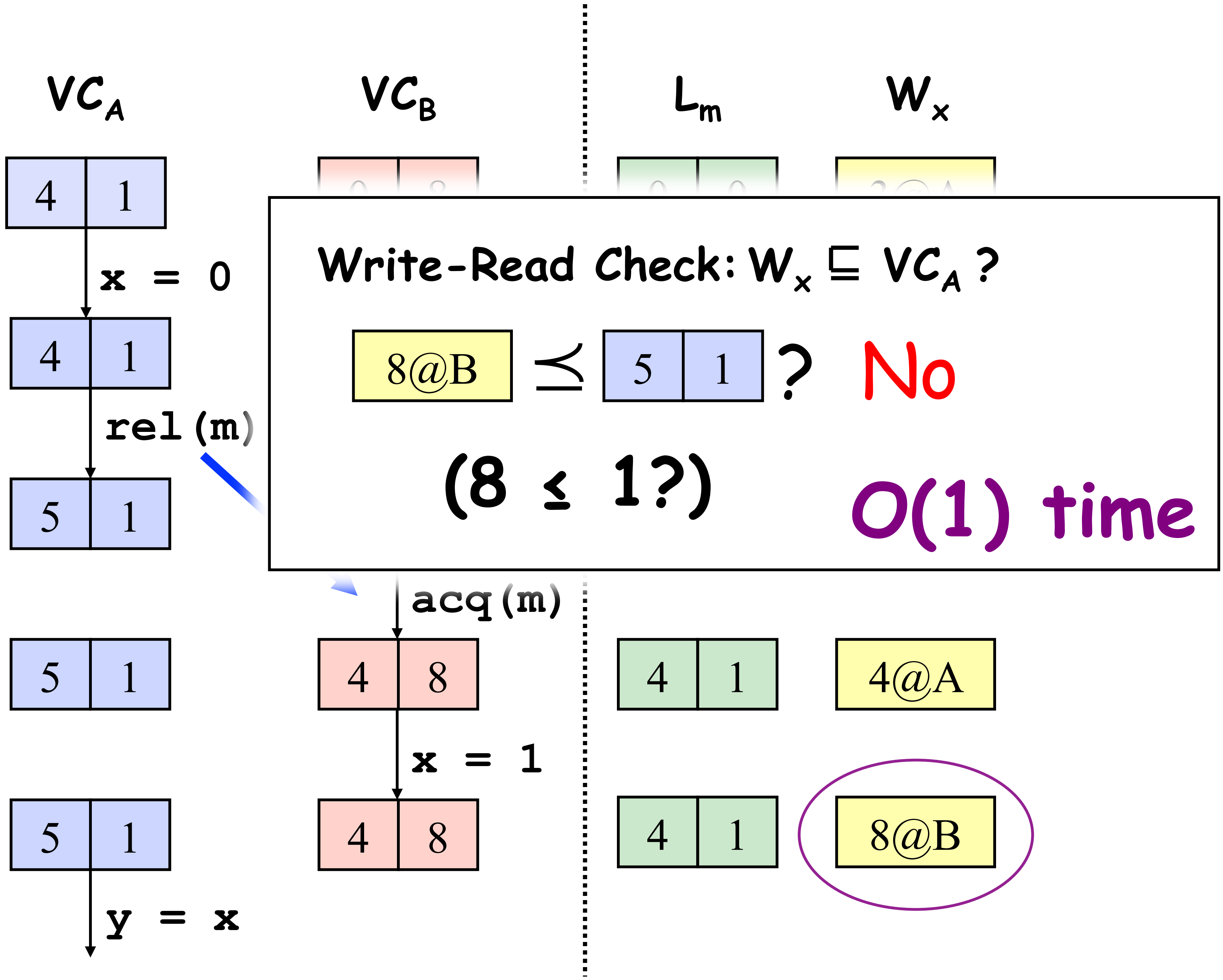


No Data Races Yet: Writes Totally Ordered



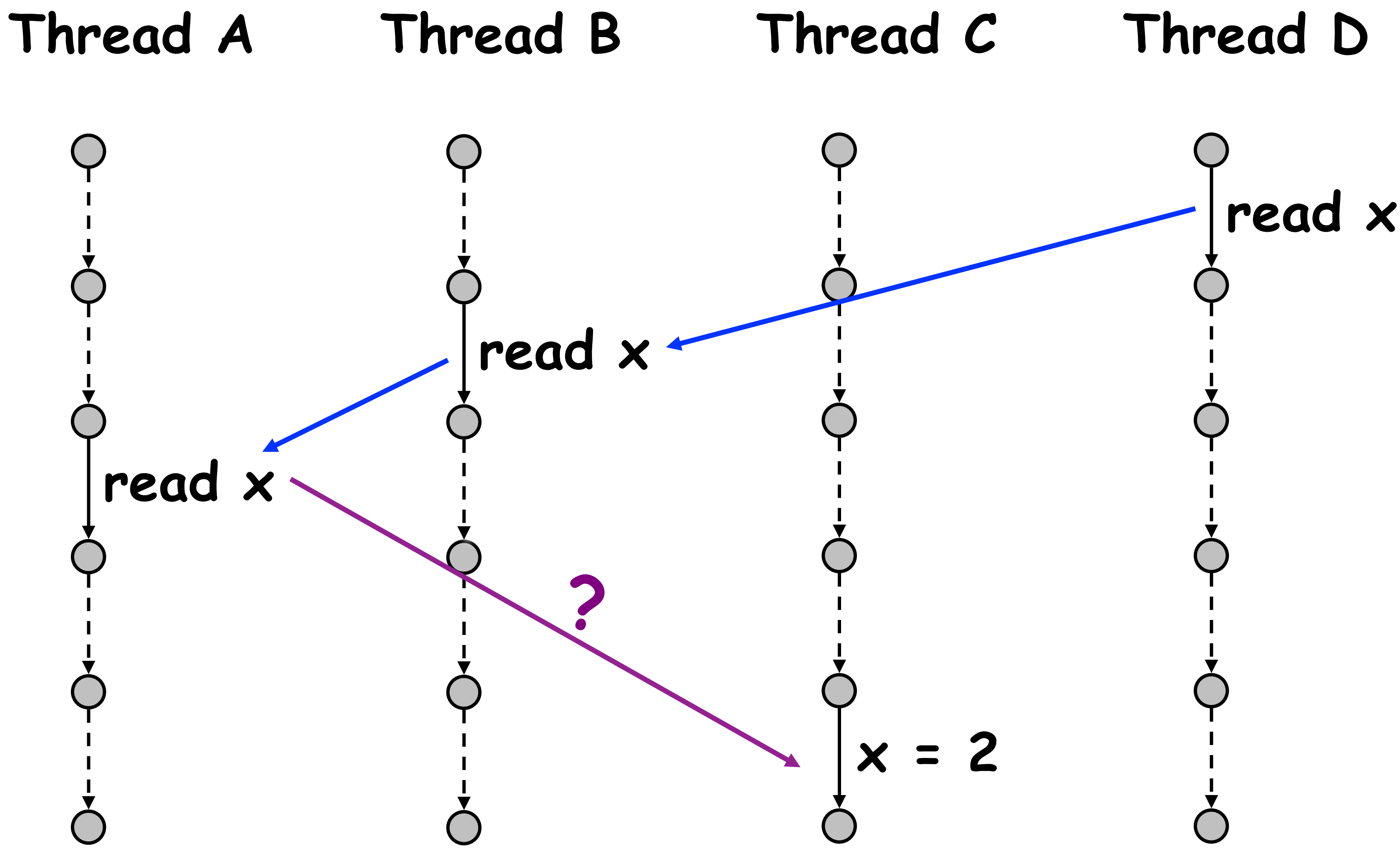

Last Write "Epoch"
VC_A
VC_B
L_m
W_x
4 1
2 8
2 1
1@B
x = 0
Write-Write Check: W_x ⊑ VC_A ?
1@B ⪯ 4 1 ?
Yes
(1 ≤ 1?)
O(1) time


VC_A
VC_B
L_m
W_x
4 1
2 8
2 1
3@A
x = 0
4 1
2 8
2 1
4@A
rel(m)
5 1
4 8
4 1
4@A
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B



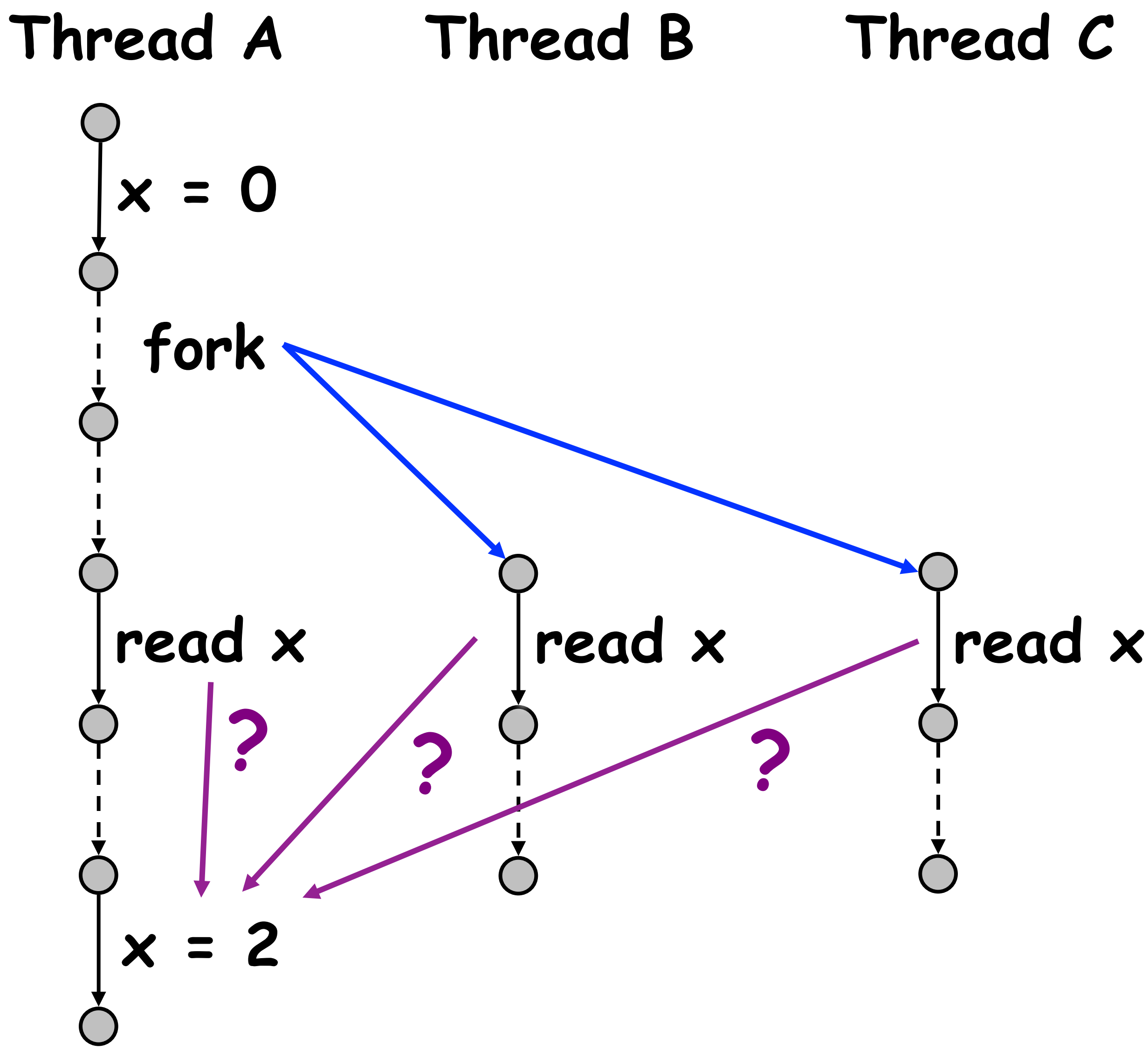
VC_A
VC_B
L_m
W_x
4 1
x = 0
4 1
rel(m)
5 1
Write-Read Check: W_x ⊑ VC_A ?
8@B ≼ 5 1 ? No
(8 ≤ 1?)
O(1) time
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B
y = x

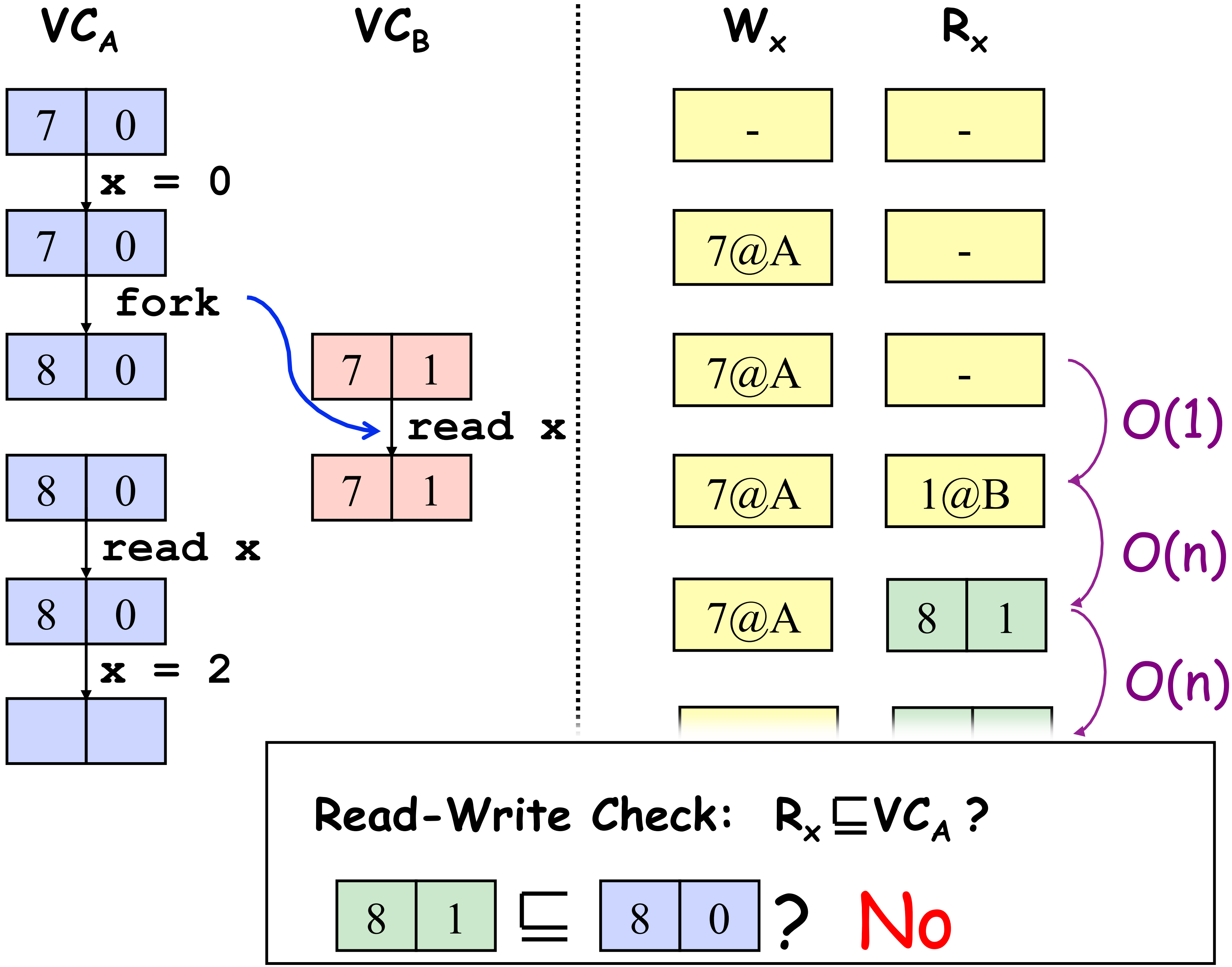
Read-Write Data Races -- Ordered Reads



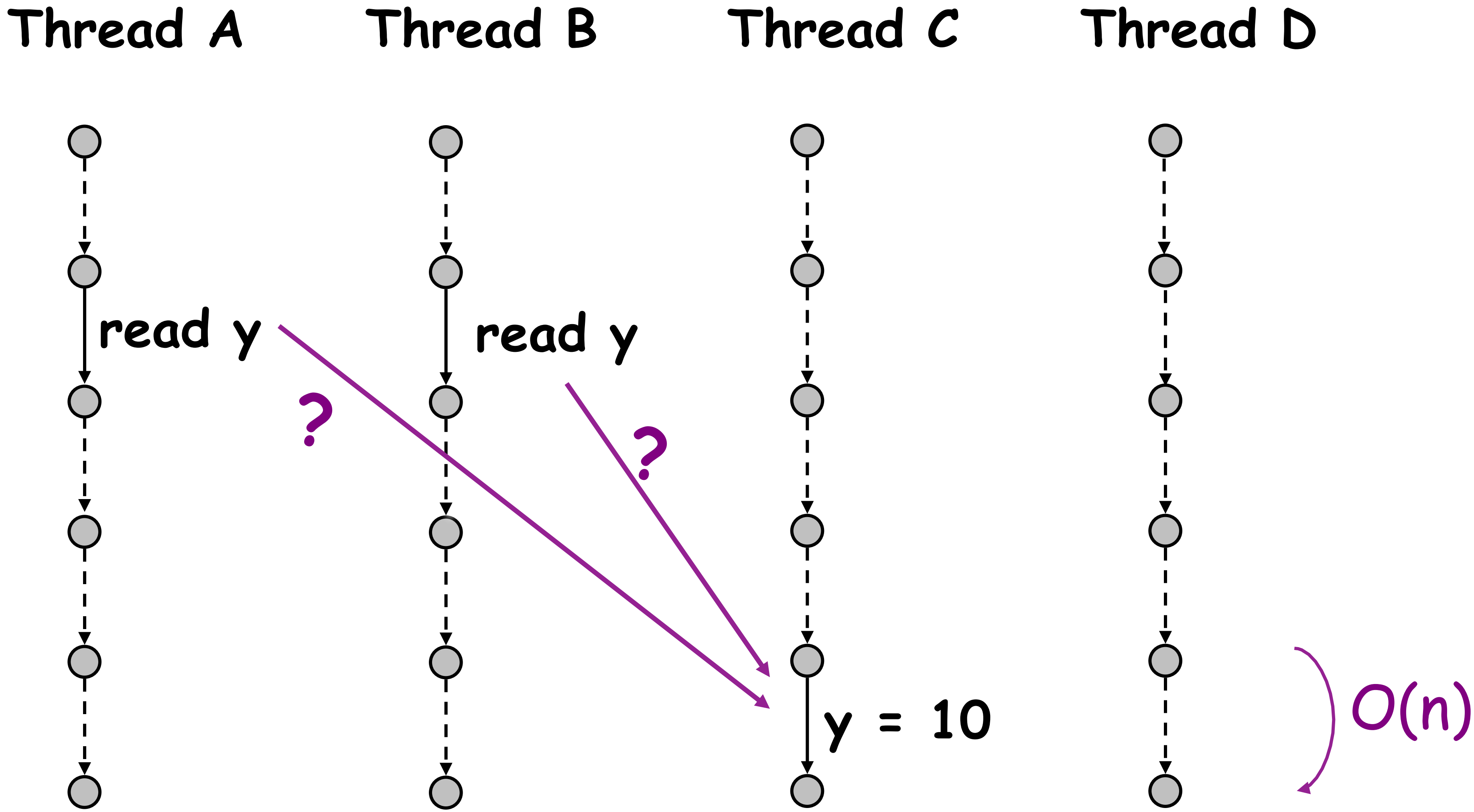
Most common case: thread-local, lock-protected, ...

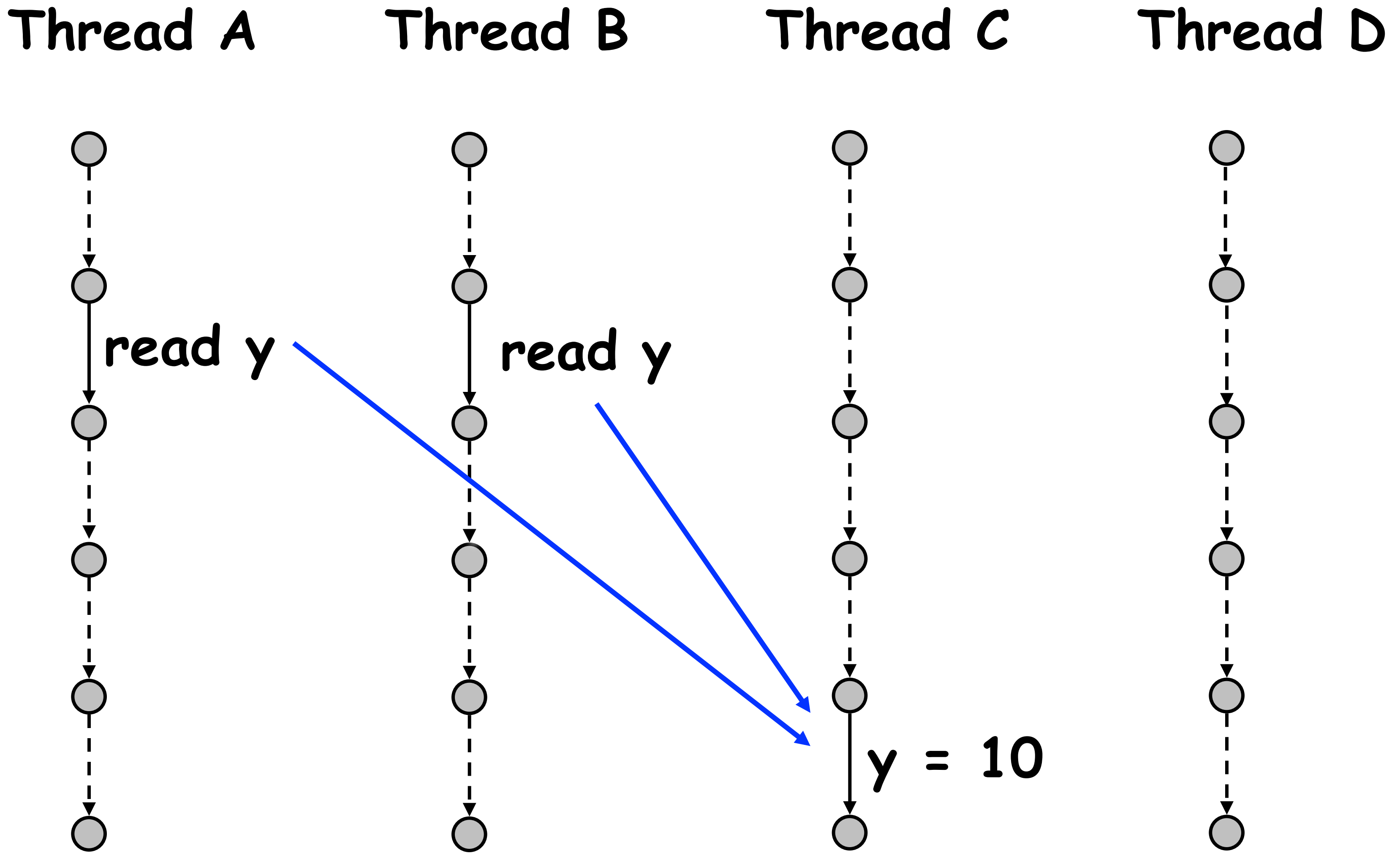
Read-Write Data Races -- Unordered Reads

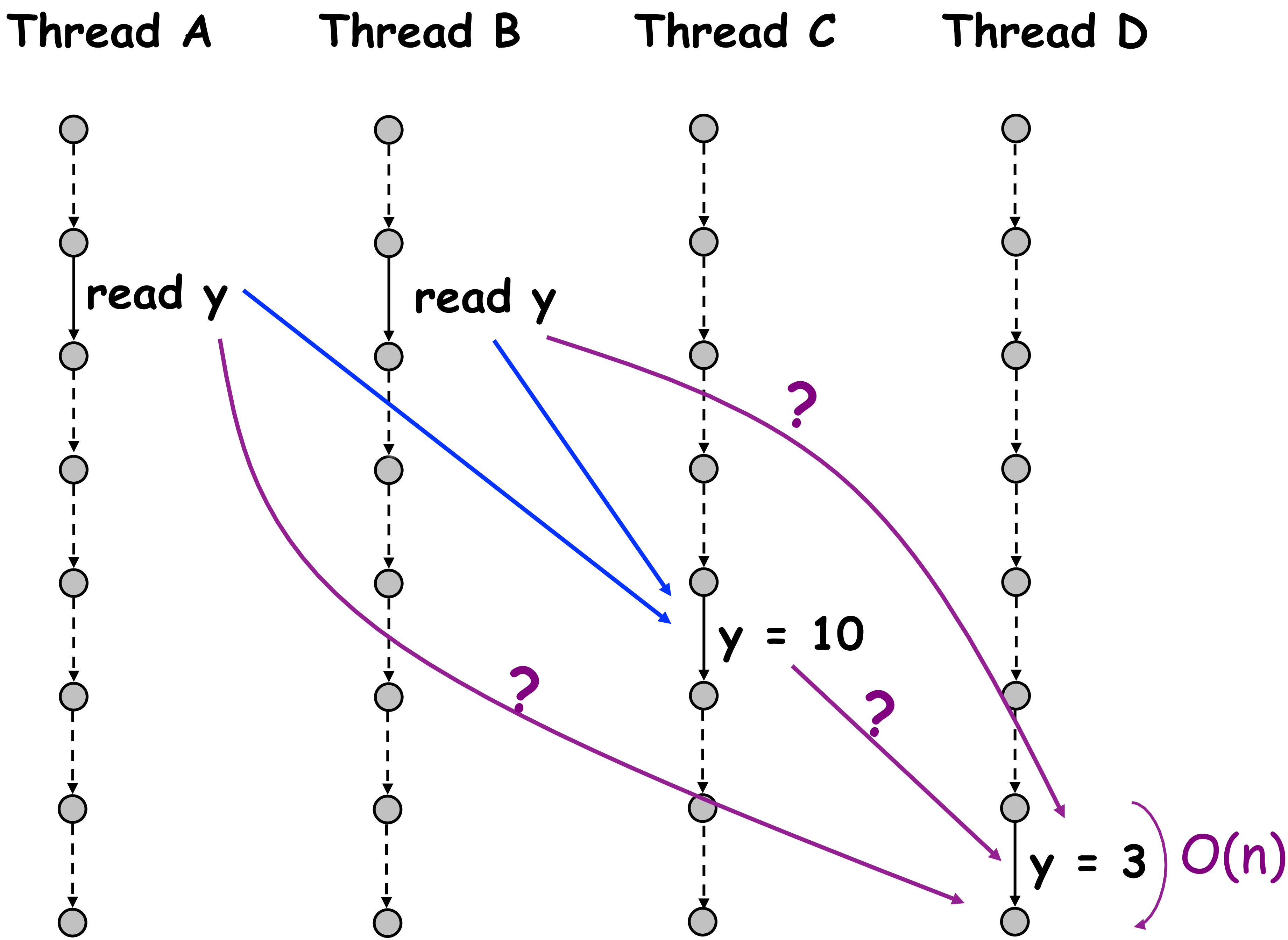




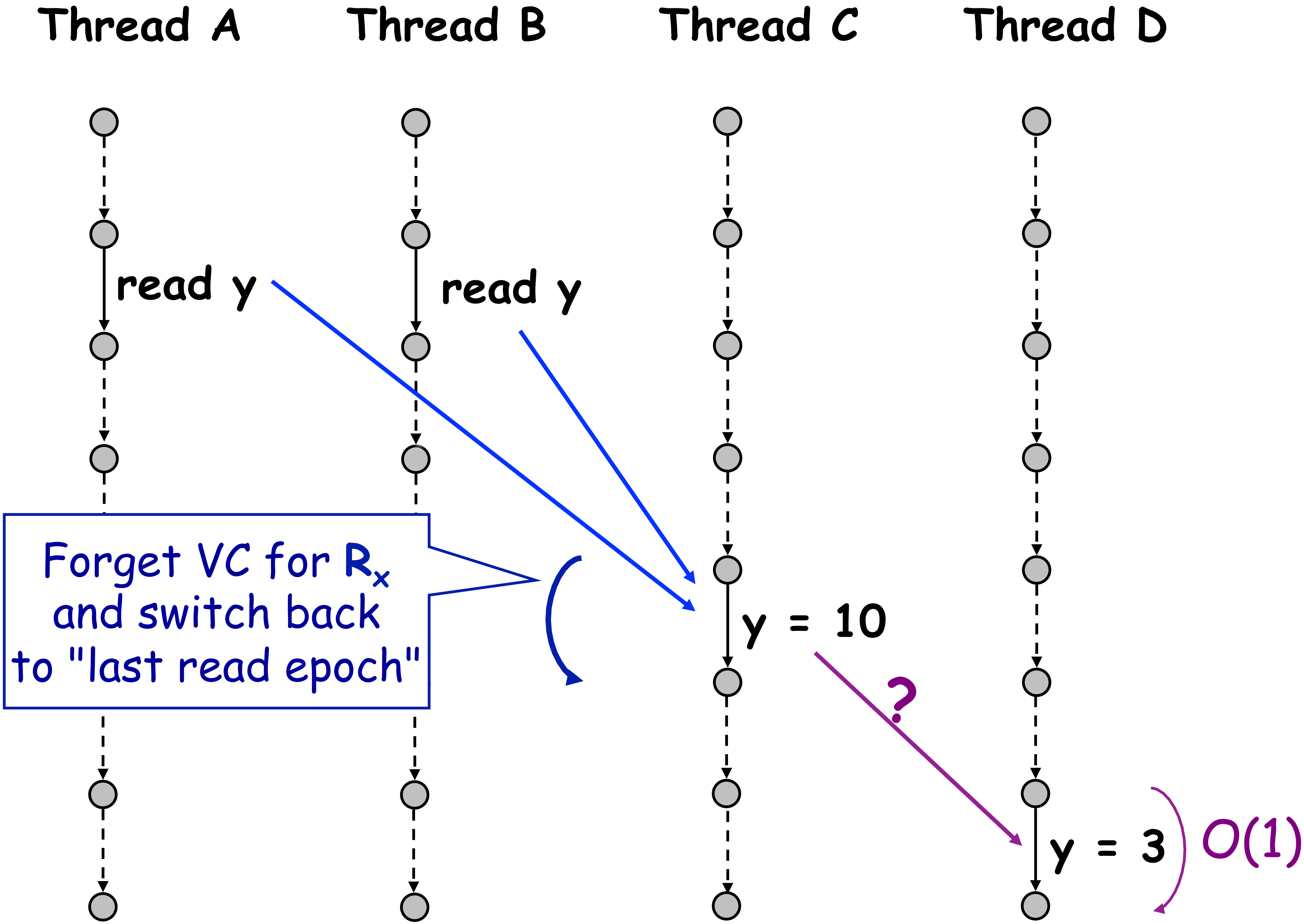
VC_A
VC_B
W_x
R_x
7 0
x = 0
7 0
fork
8 0
7 1
read x
7 1
8 0
read x
8 0
x = 2
-
-
7@A
-
7@A
-
O(1)
7@A
1@B
O(n)
7@A
8 1
O(n)
Read-Write Check: R_x ⊑ VC_A ?
8 1 ⊑ 8 0 ? No


Thread A
Thread B
Thread C
Thread D
read y
read y
?
?
y = 10
O(n)

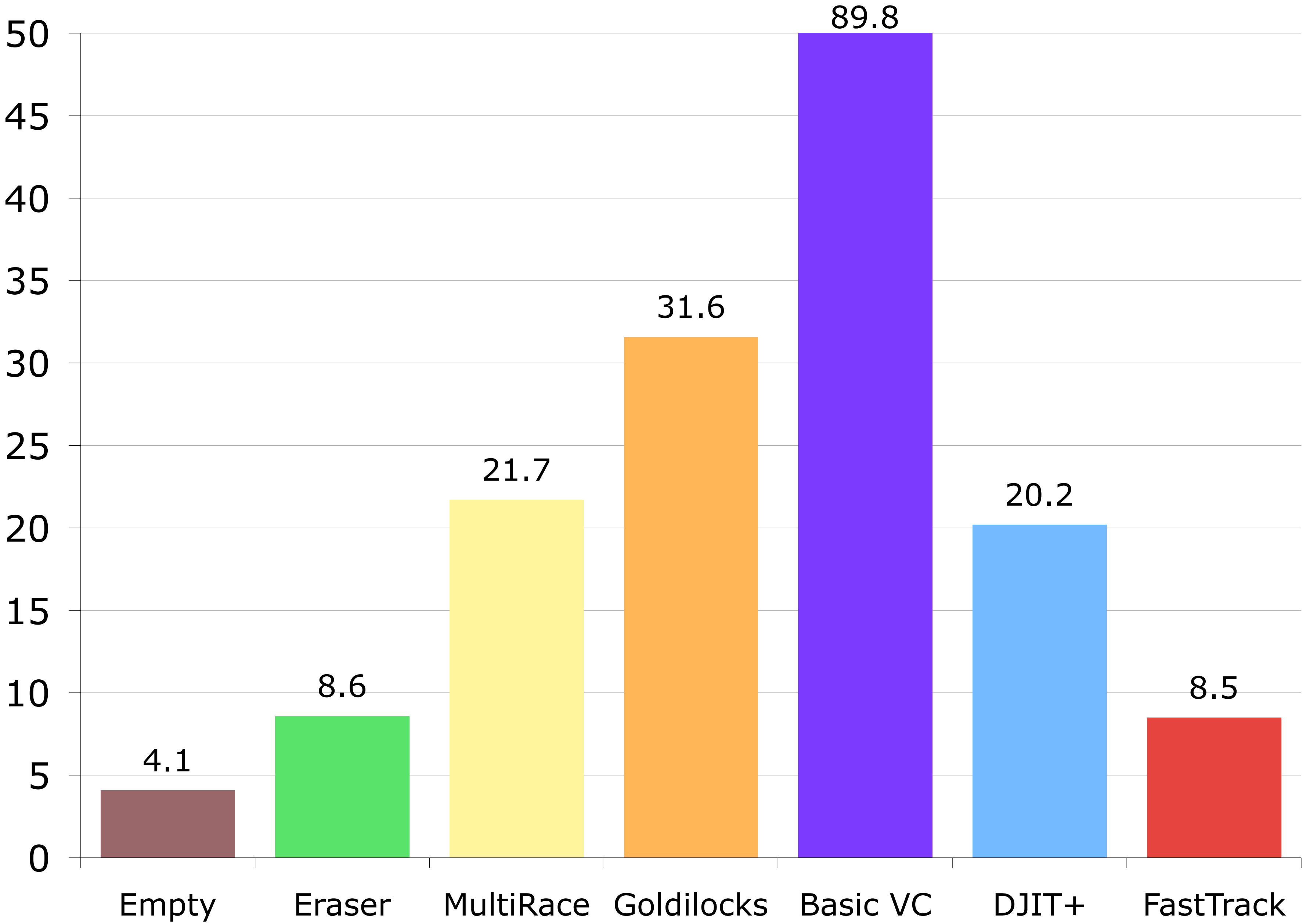

Thread A
Thread B
Thread C
Thread D
read y
read y
y = 10



Thread A
Thread B
Thread C
Thread D
read y
read y
?
?
?
y = 10
y = 3
O(n)


Thread A
Thread B
Thread C
Thread D
read y
read y
Forget VC for R_x and switch back to "last read epoch"
y = 10
?
y = 3
O(1)

Slowdown (x Base Time)



Memory Usage

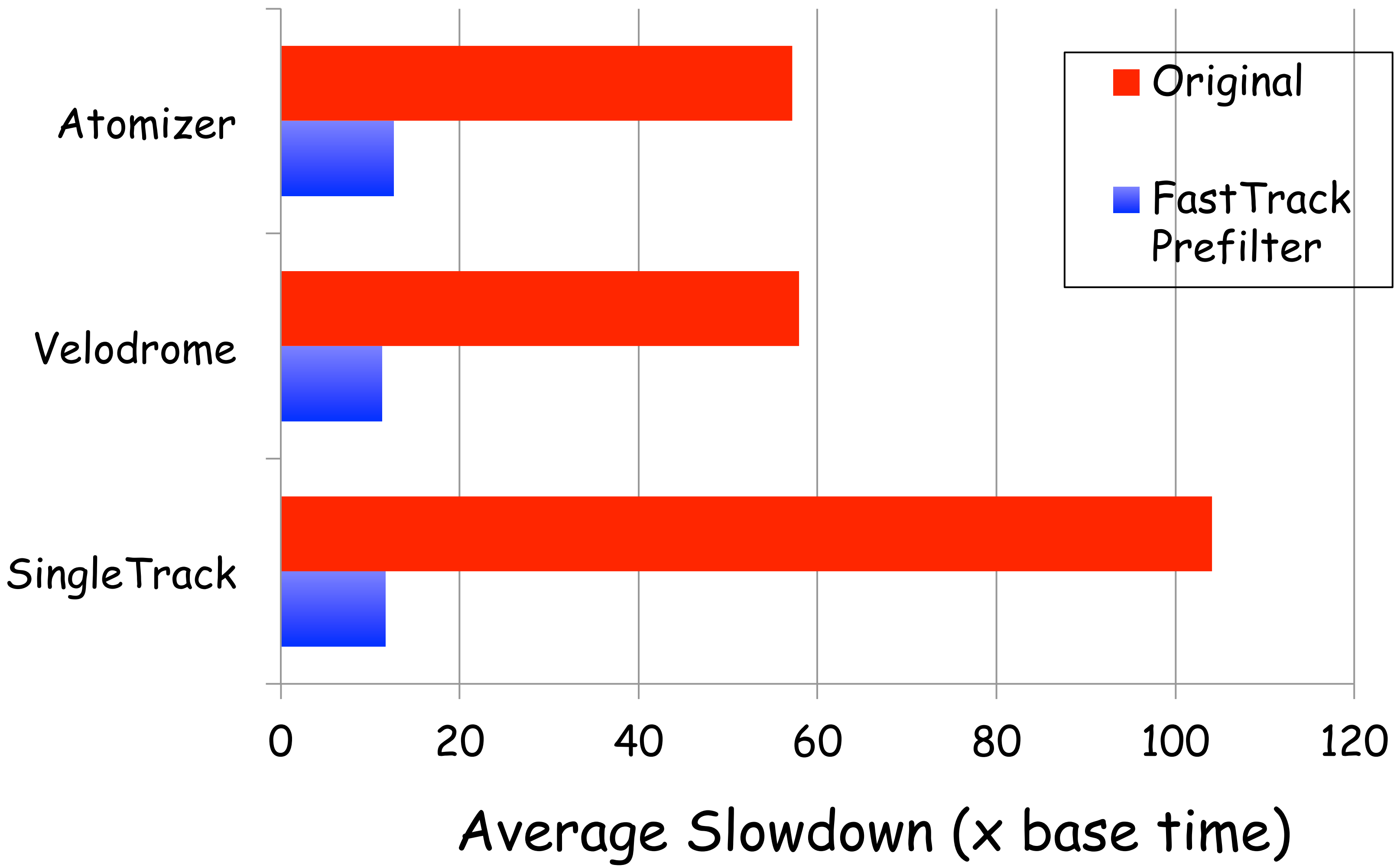
- FastTrack allocated ~200x fewer VCs

Checker	Memory Overhead
Basic VC, DJIT+	7.9x
FastTrack	2.8x
Empty	2.0x

(Note: VCs for dead objects are garbage collected)

- Improvements
  - accordion clocks [CB 01]
  - analysis granularity [PS 03, YRC 05]

Precise Data Race Classification for Other Checkers



and ~40% reduction in false alarms in Atomizer…

Eclipse 3.4

- Scale
  - > 6,000 classes
  - 24 threads
  - custom sync. idioms
- Precision (tested 5 common tasks)
  - Eraser: ~1000 warnings
  - FastTrack: ~30 warnings
- Performance on compute-bound tasks
  - > 2x speed of other precise checkers
  - same as Eraser

Verifying Race Freedom with Types

```
class Ref {
  int i;
  void add(Ref r) {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   sync(x,y) { x.add(y); }
   sync(x,y) { x.add(y); }
}
assert x.i == 6;
```

Property: Each shared variable must be protected by a lock.

Verifying Race Freedom with Types

```
class Ref {
  int i guarded_by this;
  void add(Ref r) requires this, r {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   sync(x,y) { x.add(y); }
   sync(x)   { x.add(y); }
}
assert x.i == 6;
```

Property: Each shared variable must be protected by a lock.

← Error: lock y not held (points to `sync(x) { x.add(y); }`)

Client-Side Locking

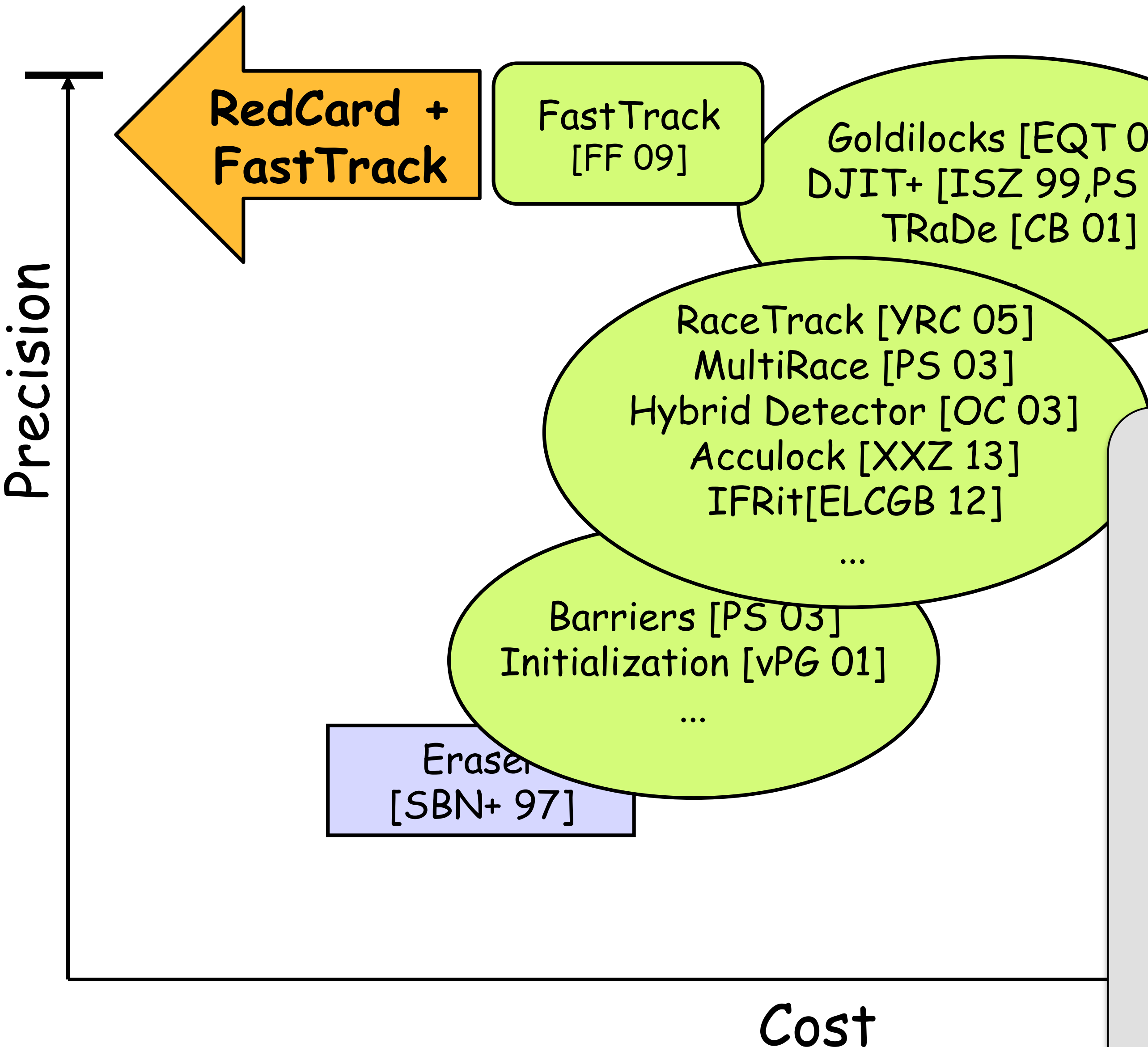
```
class Ref<ghost g> {
  int i guarded_by g;
  void add(Ref<g> r) requires g {
    i = i + r.i;
  }
}

Object m = new Object();
Ref<m> x = new Ref<m>(0);
Ref<m> y = new Ref<m>(3);
parallel {
   sync(m) { x.add(y); }
   sync(m) { x.add(y); }
}
assert x.i == 6;
```

Static Race Detection In Practice

- Rcc/Java [Flanagan-Freund 00-06]
- Other Systems
  - Ownership types [Boyapati et al 01]
  - RacerX [Engler-Ashcraft 02]
  - Chord [Naik et al 06]
  - Object Use Graphs [vonPraun-Gross 03]

- Limitations
  - scalability
  - unsound or incomplete

Static Analysis to Optimize Dynamic Checks



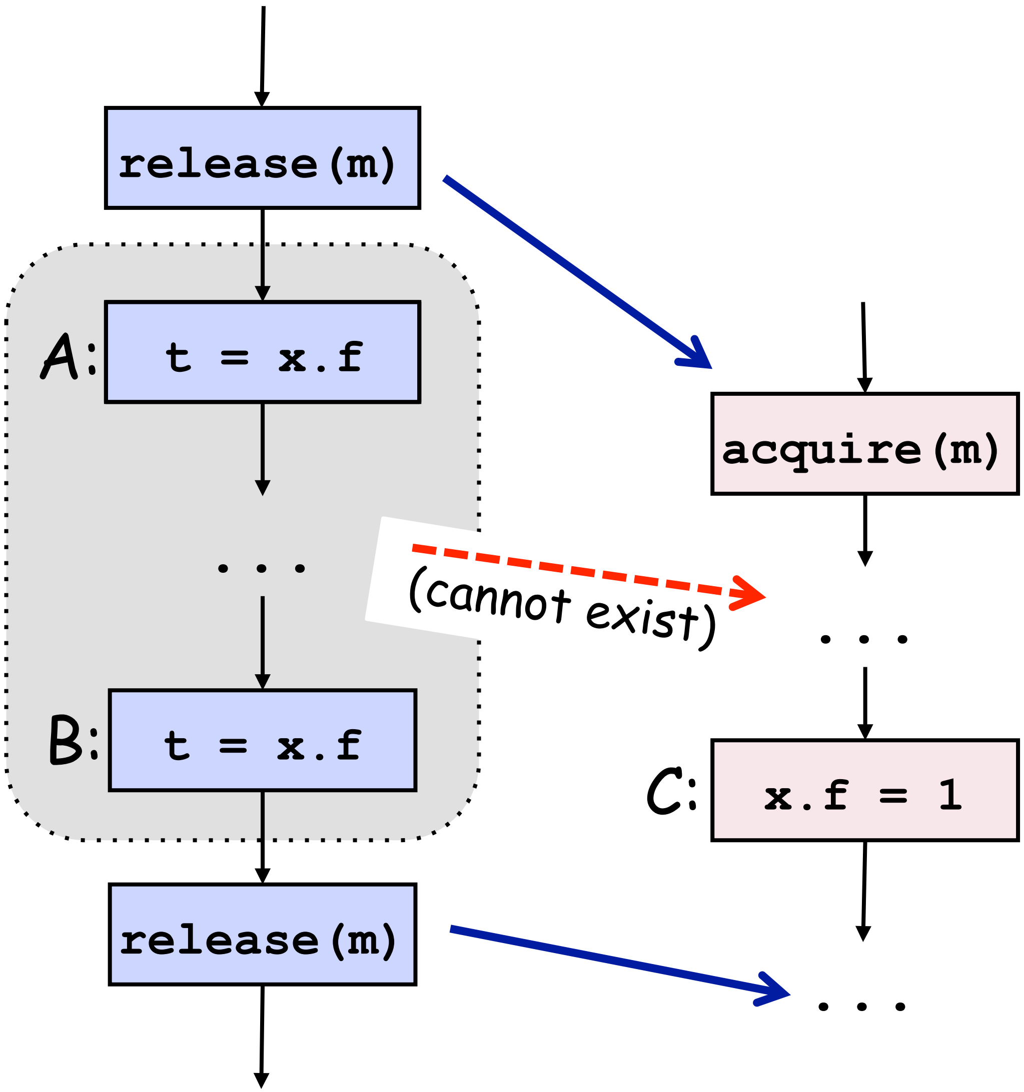
In source code:

```
...
t = x.f^Check;
...
u = y.g^NoCheck;
...
```

Options for Skip checks

1. on race-free access.
2. *that are redundant.*

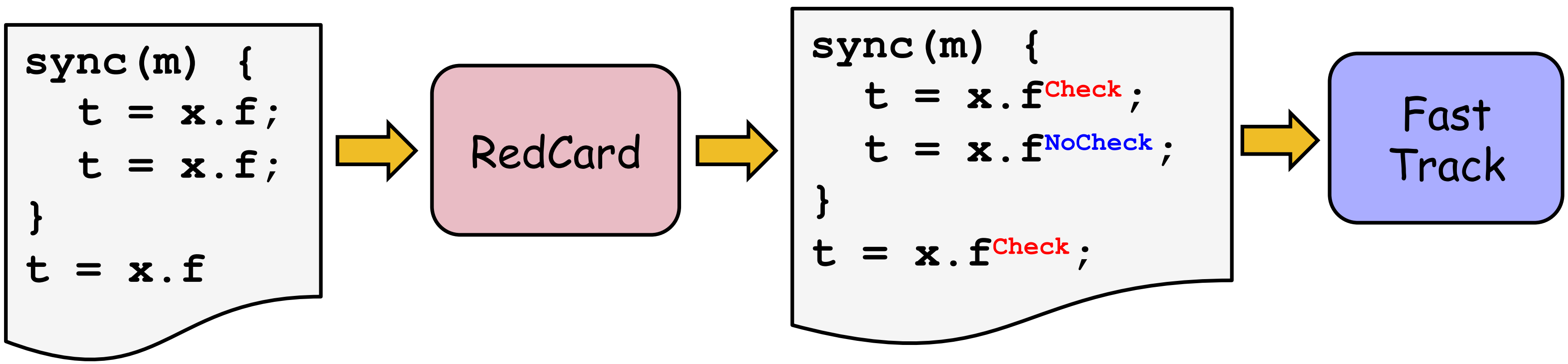
Release-Free Spans



- Sequence of ops by one thread
- No outgoing edges
  - eg: no releases, forks, waits, ...
- If B races with C then A races with C
- Race check on B is *redundant*

RedCard: Redundant Check Elimination [ECOOP 2013]

- Find accesses always touching memory previously accessed within current release-free span
- Remove checks on those accesses



- No change in precision
  - No missed races
  - No spurious warnings

Other Uses of Similar Notions

- Interference-Free Regions [Effinger-Dean et al 11, 12]
  - compiler optimizations, imprecise race detection
- Similar optimizations for specific race detection algorithms
  - Eraser-based [vonPraun-Gross 02, Choi et al 03]
  - X10 task parallelism [Raman et al 10]

- RedCard
  - works with any precise race detector
  - more sophisticated (but expensive) analysis
  - extensions for additional forms of redundancy

Available Paths Analysis

- For each program point, compute Context
  - Available Paths: expressions describing memory previously accessed in current span

```
{ }       ———•
                t = x.f^Check;
{ x.f }   ———•
                t = x.f^NoCheck;
{ x.f }   ———•
                rel(m);
{ }       ———•
                t = x.f^Check;
{ x.f }   ———•
```

x.f is an available path

(for simplicity, assume no distinction between reads and writes)

Must Aliases

- Include must-alias constraints in C

> **x.h** is an available path and **y = x**

```
{}                                 ——•
                                       x = z.g^Check;
{z.g,        x = z.g          }    ——•
                                       y = z.g^NoCheck;
{z.g,        x = z.g, y = z.g}     ——•
                                       t1 = x.h^Check;
{z.g, x.h,   x = z.g, y = z.g}     ——•
                                       t2 = y.h^NoCheck;
```

- Implement via any sound decision proc. (Z3)
- Similar to type state tracking [Fink et al 08]

Redundant Array Accesses

```
for (int i = 0; i < a.length; i++) {
  a[i]^Check = ...;
}
for (int i = 0; i < a.length; i++) {
  a[i]^NoCheck = ...;
}
```

- Context extensions
  - Paths for array accesses
    - single: `a[i]`
    - range: `∀(i ∈ 0 to n).a[i]`
  - Linear inequalities

```
i = 0;
while (i < a.length) {
  a[i]^Check = 0;
  i = i + 1;
}
•
a[k]^NoCheck = 1;
```

∀(j ∈ 0 to a.length).a[j]

```
i = 0;
while (i < a.length) {
  a[i]Check = 0;
  i = i + 1;
}

   ]NoCheck = 1;
```

Loop Invariant:

∀(j ∈ 0 to i).a[j]

Inferred Via Cartesian Predicate Abstraction [BMMR 01, FQ 02]

```
i = 0;
while (i < a.length) {
    a[i]^Check = 0;
    i = i + 1;
}
a[k]^NoCheck = 1;
```

$i = 0$

$i < a.length$
$\forall (j \in 0 \text{ to } i).a[j]$

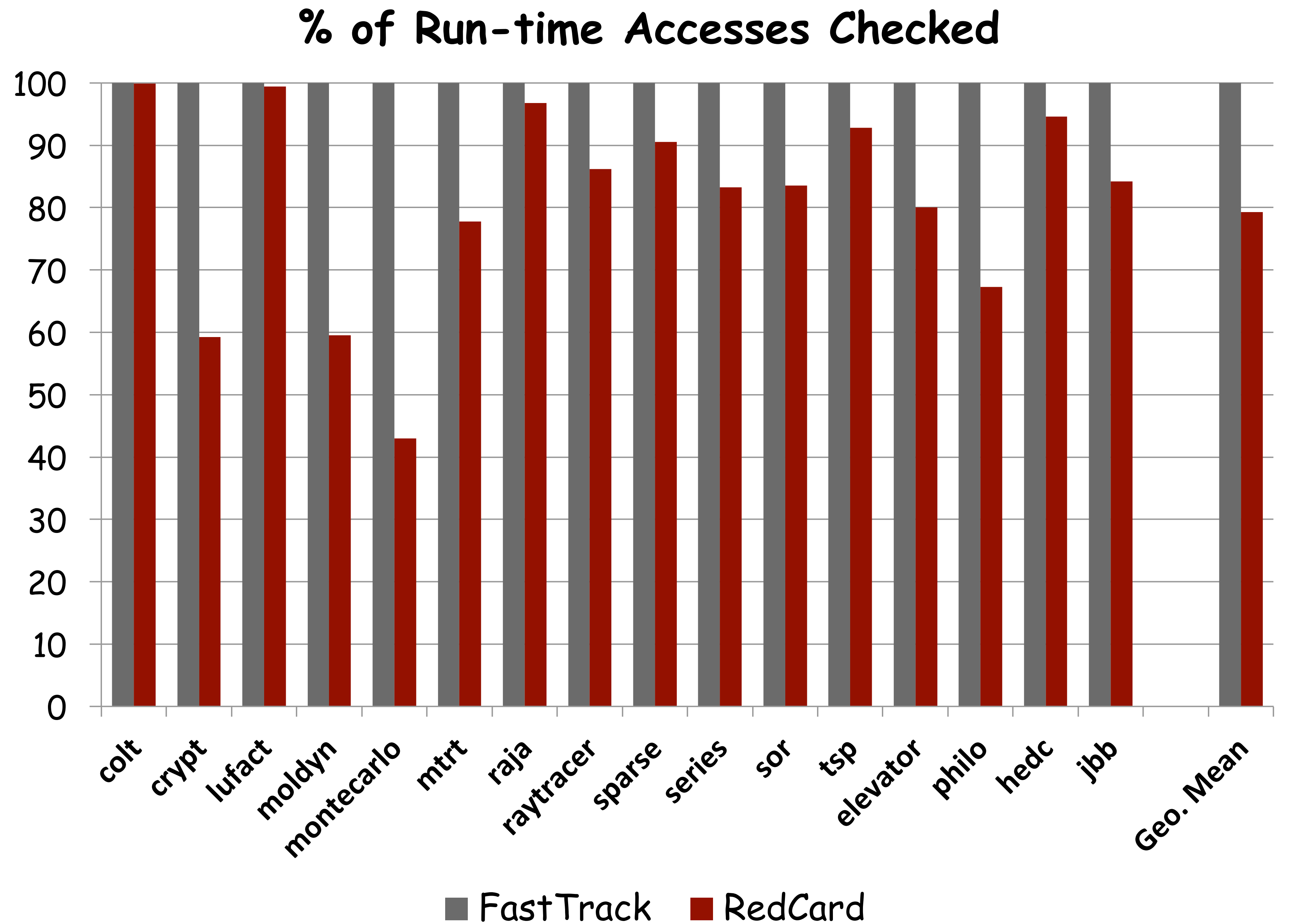
$i < a.length$
$a[i]$
$\forall (j \in 0 \text{ to } i).a[j]$

$i = i'+1$
$i' < a.length$
$a[i']$
$\forall (j \in 0 \text{ to } i').a[j]$

$\forall (j \in 0 \text{ to } a.length).a[j]$

RedCard Implementation for Java

- WALA framework for Java bytecode [IBM]
  - Dataflow analysis over SSA-based CFGs
  - Z3 [deMoura-Bjørner 08] to reason about Contexts
- Infers and outputs list of "NoCheck" accesses
- Two Modes
  - Intra-procedural
  - Inter-procedural (0-CFA, CHA)
- Analysis Time: ~18 sec per KLOC



% of Run-time Accesses Checked
100
90
80
70
60
50
40
30
20
10
0
colt
crypt
lufact
moldyn
montecarlo
mtrt
raja
raytracer
sparse
series
sor
tsp
elevator
philo
hedc
jbb
Geo. Mean
FastTrack
RedCard

Proxy Fields

- Field `y` has *proxy field* `x` if all spans accessing `p.y` also access `p.x`

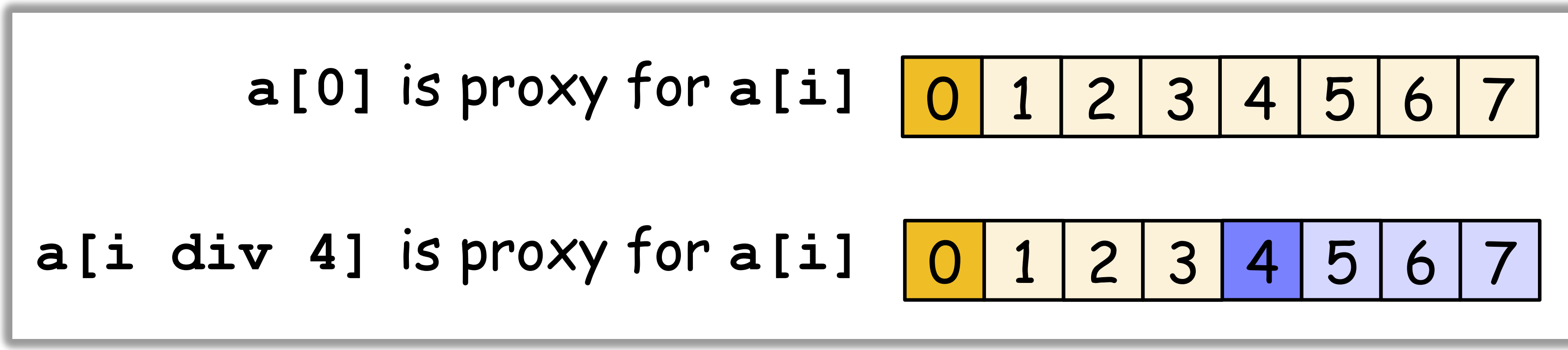
  If `p.y` has race then `p.x` has race

- Label `p.y` as "NoCheck"
- Still identify all traces with data races

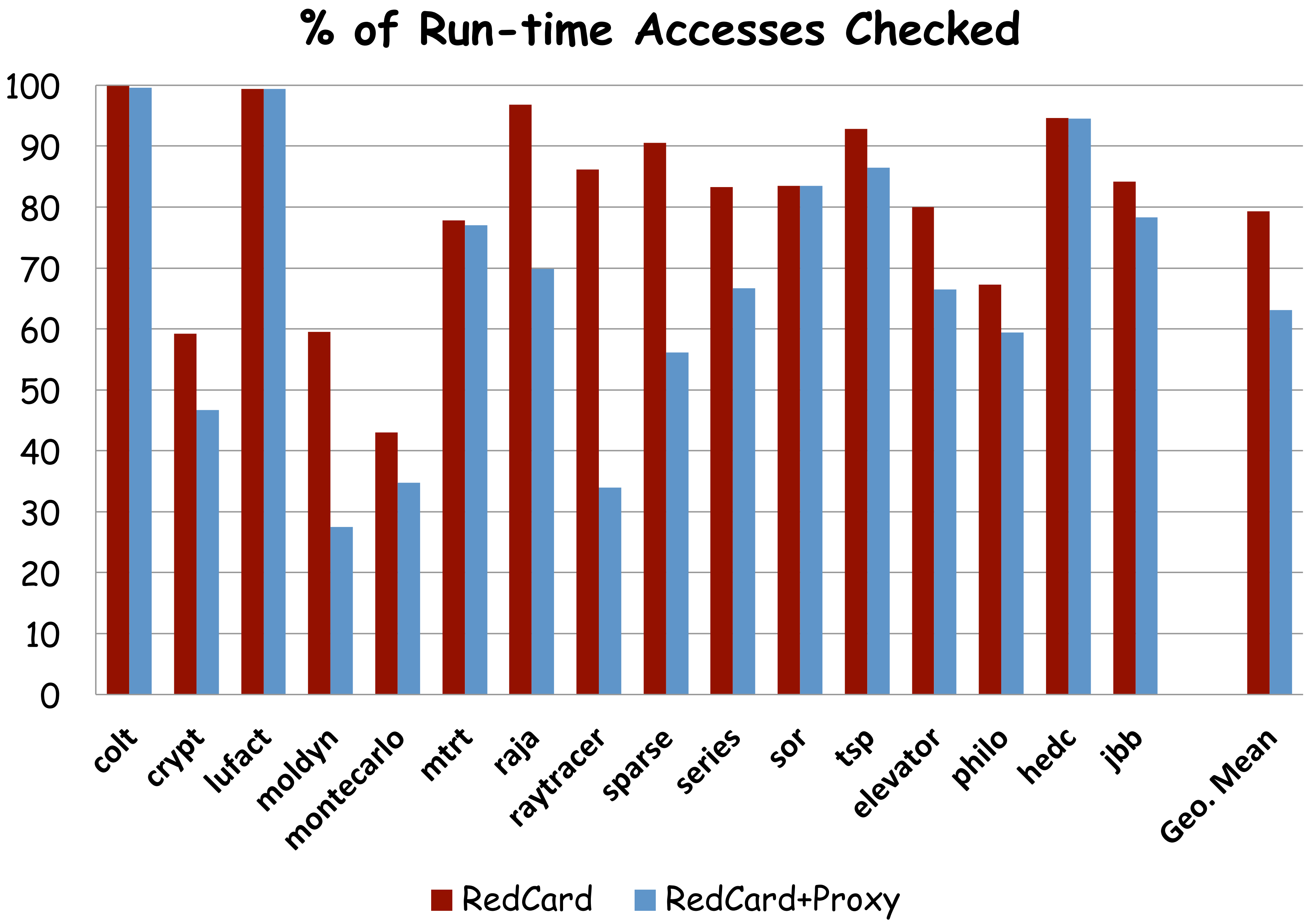
```
class Point {
  private int x,y;

  void move() {
    this.x^Check = ...;
    this.y^NoCheck = ...;
  }

  int dot(Point o) {
    return
        this.x^Check
         * o.x^Check
      + this.y^NoCheck
         * o.y^NoCheck;
  }

  int getX() {
    return this.x^Check;
  }
}
```

Array Proxies

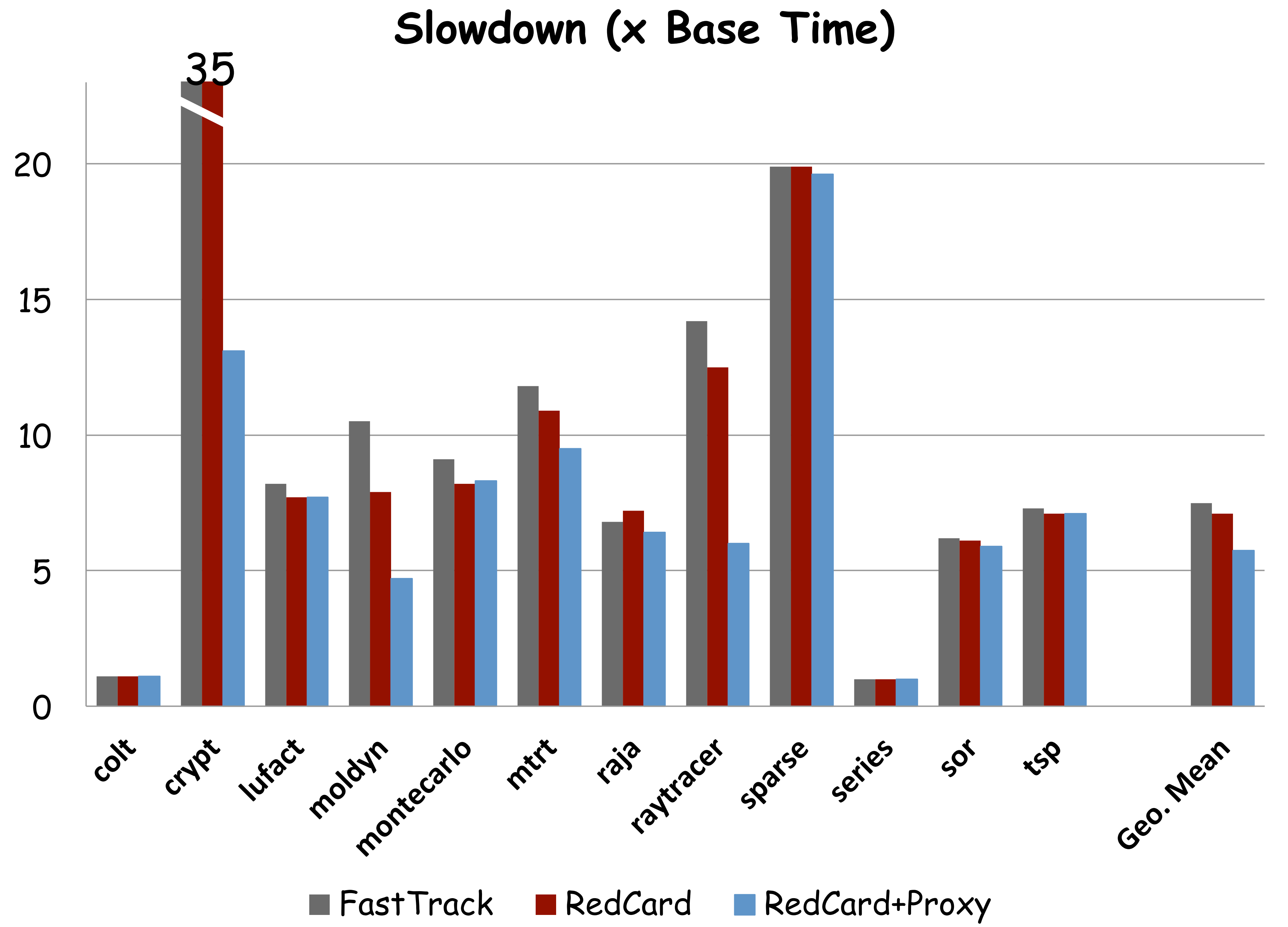
- Array element can be proxy for other elements



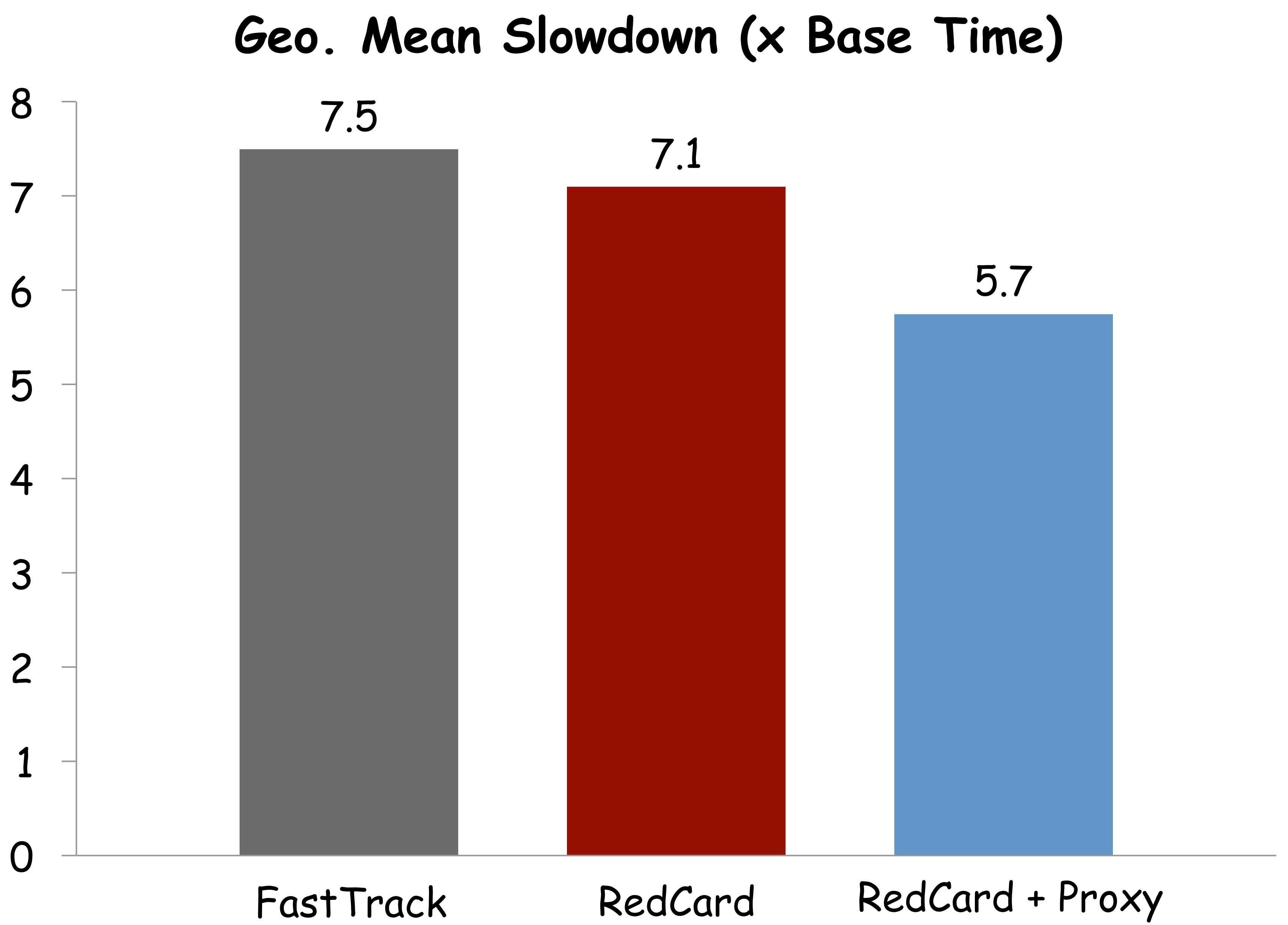
- RedCard identifies common array proxy patterns
- b[j] is "NoCheck" if b[j] has proxy other than itself
  - may-alias info about b computed by separate analysis



% of Run-time Accesses Checked
100
90
80
70
60
50
40
30
20
10
0
colt
crypt
lufact
moldyn
montecarlo
mtrt
raja
raytracer
sparse
series
sor
tsp
elevator
philo
hedc
jbb
Geo. Mean
RedCard
RedCard+Proxy



Slowdown (x Base Time)
35
20
15
10
5
0
colt
crypt
lufact
moldyn
montecarlo
mtrt
raja
raytracer
sparse
series
sor
tsp
Geo. Mean
FastTrack
RedCard
RedCard+Proxy



Geo. Mean Slowdown (x Base Time)
8
7
6
5
4
3
2
1
0
7.5
7.1
5.7
FastTrack
RedCard
RedCard + Proxy

Where To Go From Here?

- Static Race Checking Analysis
- Performance (goal is always-on precise detection...)
  - HW support
  - static-dynamic hybrid analyses
  - sampling
- Coverage
  - symbolic model checking, specialized schedulers
- Classify malignant/benign data races
  - which data races are most critical?
- How to respond to data races? warn/fail-fast/recover?
- Reproducing traces exhibiting rare data races
  - record and replay
- Generalization: reason about traces beyond the observed trace

Key References

- Hans-J. Boehm and Sarita V. Adve, "You Don't Know Jack About Shared Variables or Memory Models", CACM 2012.
- Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", CACM 1978.
- Martin Abadi, Cormac Flanagan, and Stephen N. Freund, "Types for Safe Locking: Static Race Detection for Java", TOPLAS 2006.
- Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended static checking for Java", PLDI 2002.
- S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs", TOCS 1997.

Key References

- Friedemann Mattern, "Virtual Time and Global States of Distributed Systems", Workshop on Parallel and Distributed Algorithms 1989.
- Yuan Yu, Tom Rodeheffer, and Wei Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking", SOSP 2005.
- Eli Pozniansky and Assaf Schuster, "MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs", Concurrency and Computation: Practice and Experience 2007.
- Robert O'Callahan and Jong-Deok Choi, "Hybrid Dynamic Data Race Detection", PPOPP 2003.
- Cormac Flanagan and Stephen N. Freund, "FastTrack: efficient and precise dynamic race detection", CACM 2010.
- Cormac Flanagan and Stephen N. Freund, "The RoadRunner dynamic analysis framework for concurrent programs", PASTE 2010.

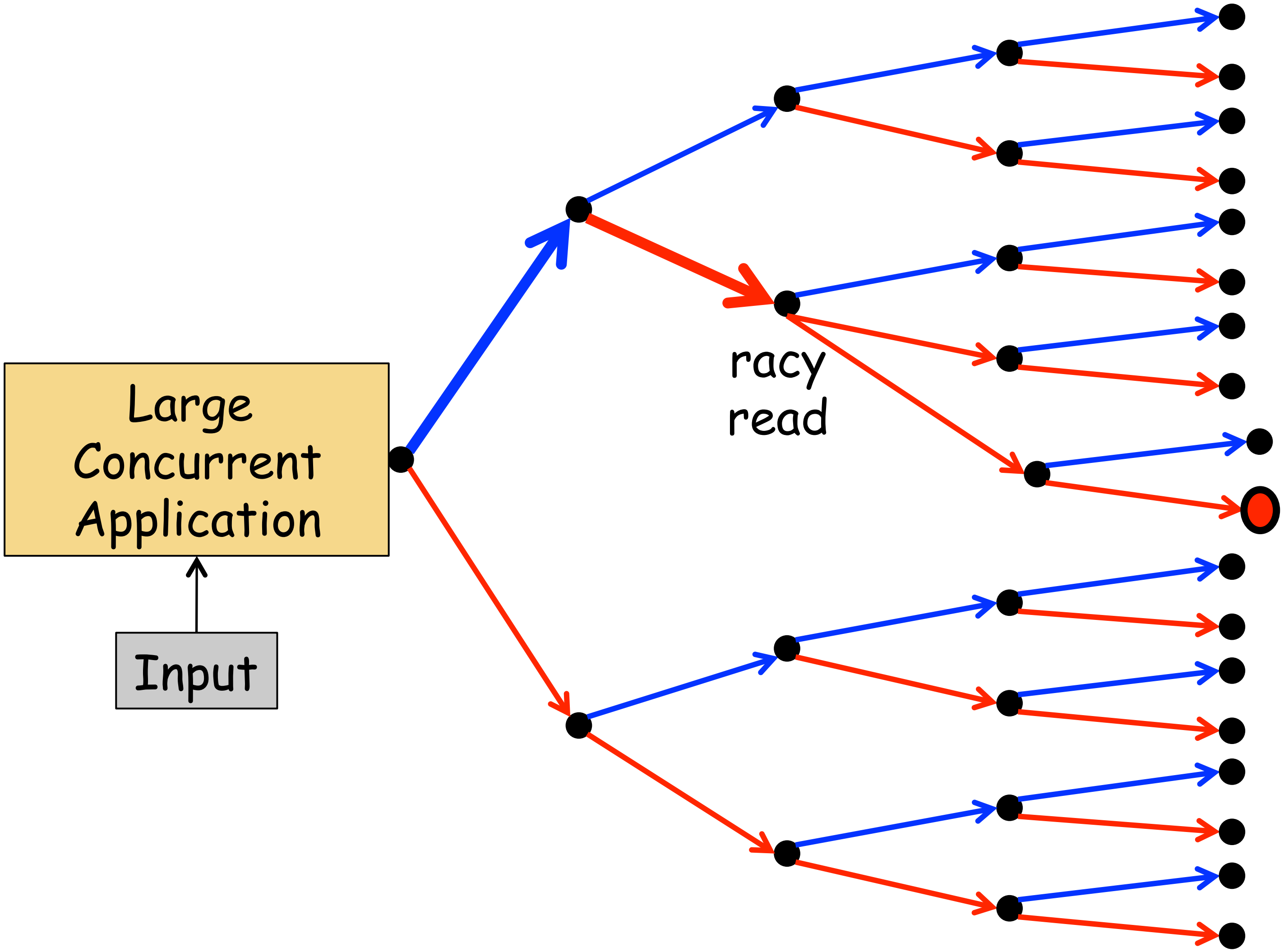
Key References

- Cormac Flanagan and Stephen N. Freund, "Adversarial memory for detecting destructive races", PLDI 2010.
- Cormac Flanagan and Stephen N. Freund, "RedCard: Redundant Check Elimination for Dynamic Race Detectors", ECOOP 2013.

Jumble: Diagnosing Bad Races

- FastTrack finds real **race conditions**
  - races correlated with defects
  - cause unintuitive behavior, especially on relaxed memory models
  - but some are intentional/benign...
- Which race conditions are **real bugs**?
  - that cause erroneous behaviors (crashes, etc)
  - and are not "benign race conditions"

Controlling Scheduling Non-Determinism



Thread A

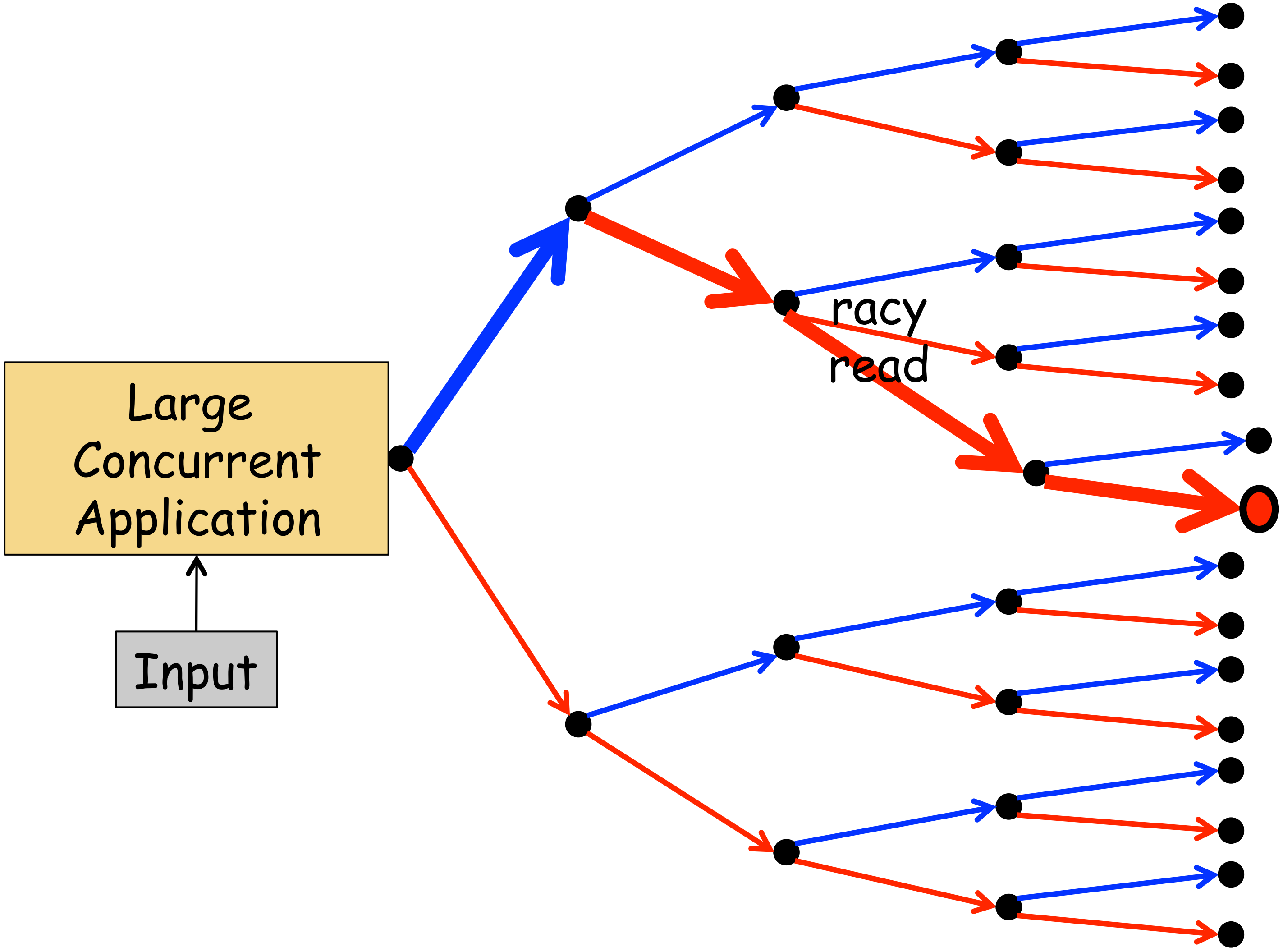
```
p = new Pt();
...
p = null;
```

Thread B

```
...
if p != null
 p.draw();
```

(eg: CalFuzzer)

Adversarial Memory [PLDI 2010]

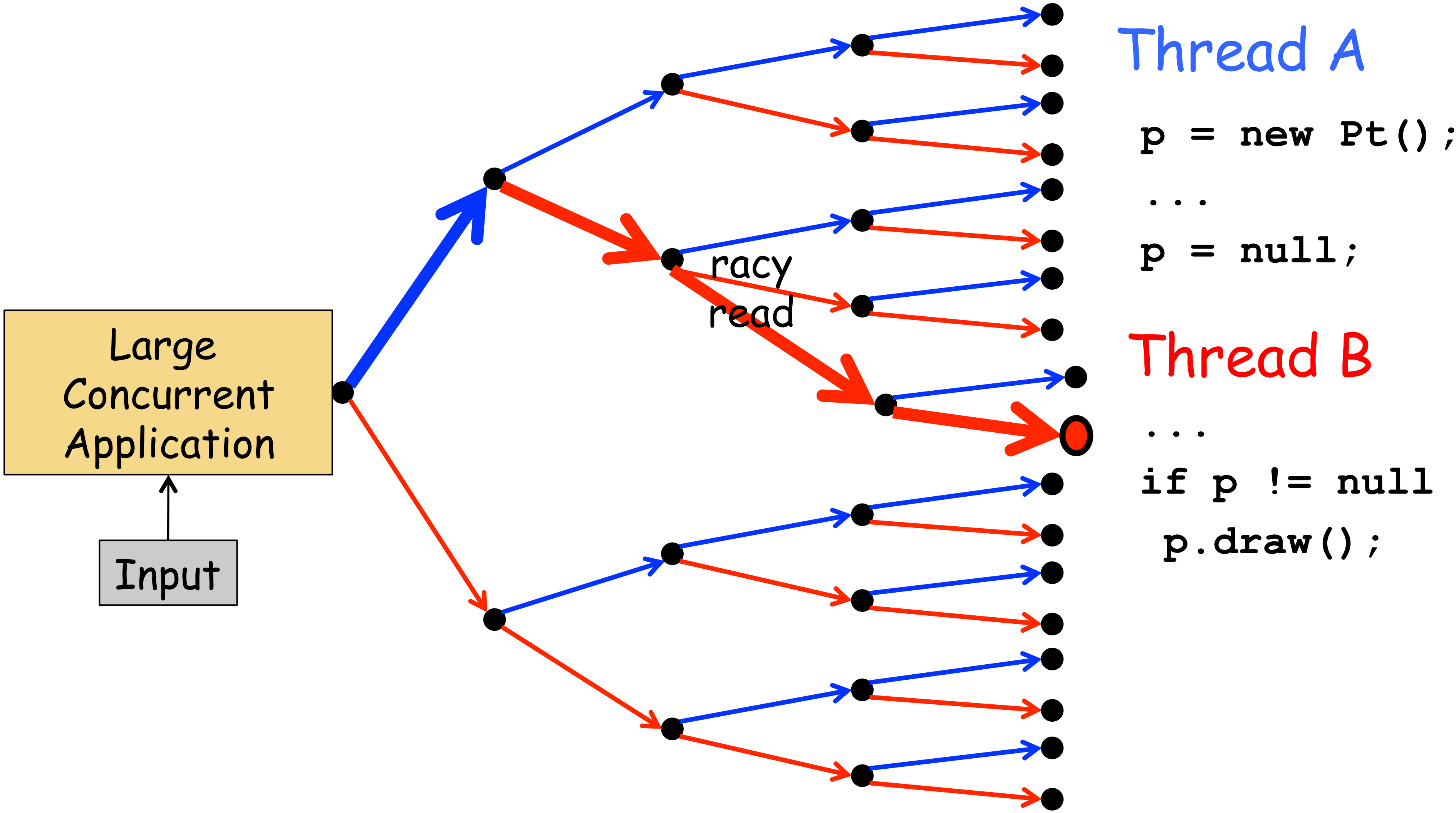


Adversarial memory exploits memory nondeterminism.

Racy read sees old value likely to crash application.

complements schedule-based approaches, quite effective.

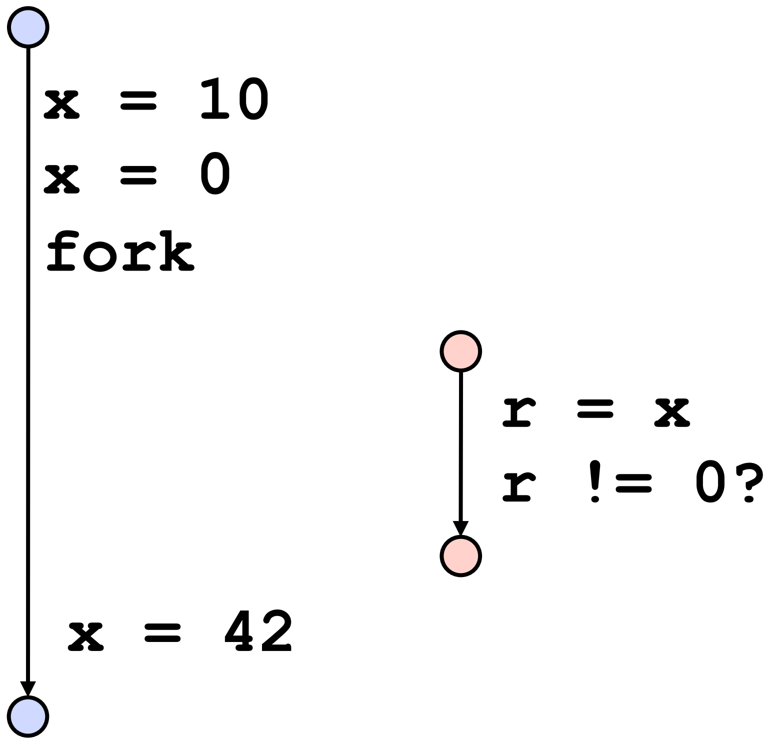
Adversarial Memory [PLDI 2010]

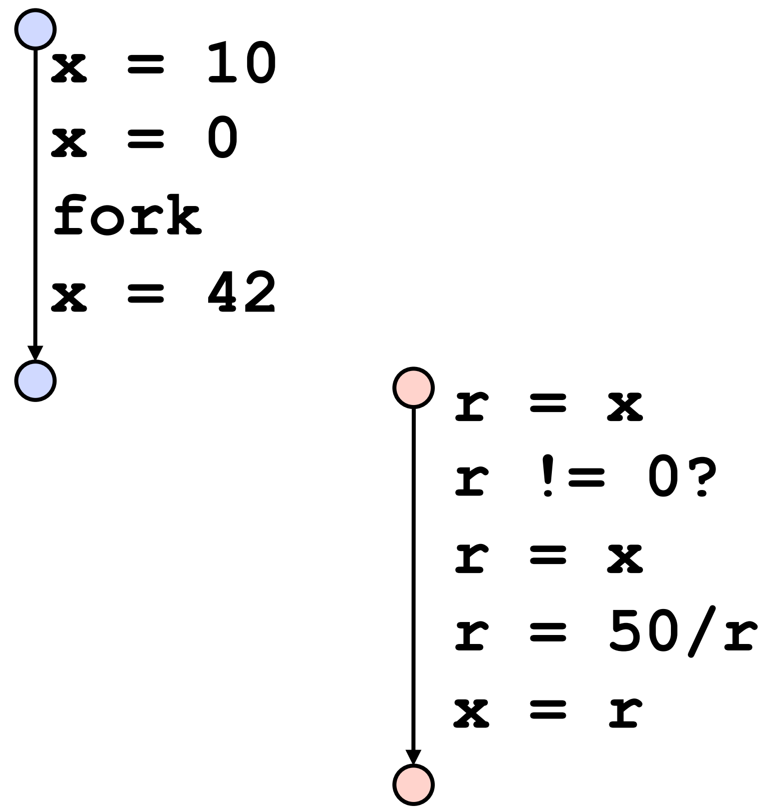


Sequentially Consistent Memory Model

```
int x = 10;
x = 0;
fork{ if (x != 0) x = 50/x; }
x = 42;
```

- Intuitive memory model
- Each read sees most recent write
- (No memory caches)





Jumble

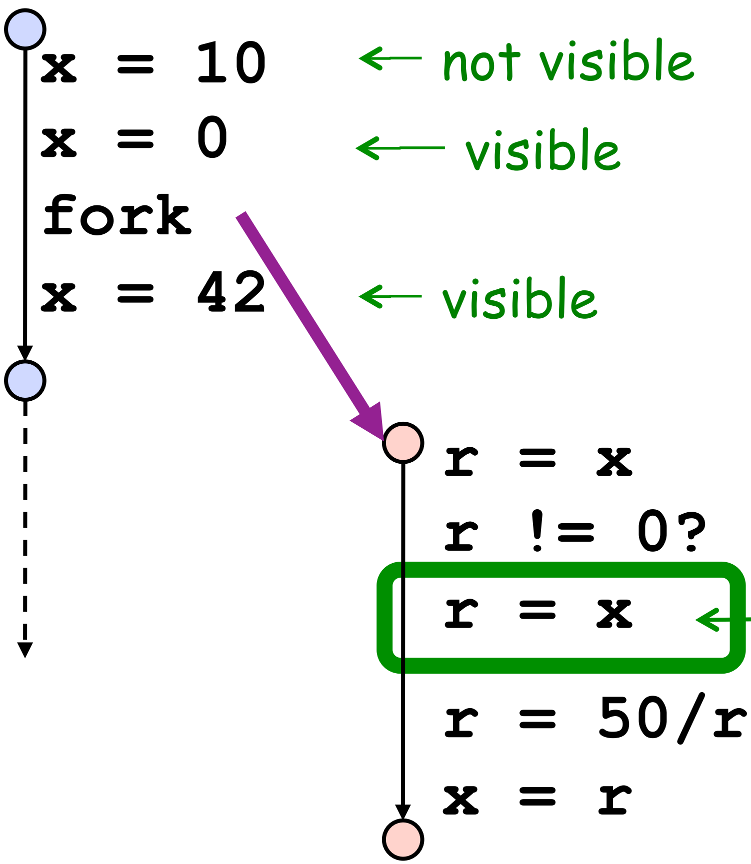
```
int x = 10;
x = 0;
fork{ if (x != 0) x = 50/x; }
x = 42;
```

Record:

- write buffer for racy vars
- happens-before relation

At each read:

- determine visible writes
- return old writes to crash app with higher probability than typical memory impl.



heuristically pick 0

division by zero

Jumble Precision: failures out of 100 runs

Benchmark: racy field	No Jumble	SC	Oldest	Oldest but diff	Random	Random but diff
montecarlo: DEBUG	0	0	0	0	0	0
mtrt: threadCount	0	0	0	0	0	0
point: p	0	0	0	0	0	0
point: x	0	0	60	52	32	30
point: y	0	0	48	53	27	30
jbb: elapsed_time	0	0	100	0	15	5
jbb: mode	0	0	100	100	95	98
raytracer:checksum1	0	0	100	100	100	100
sor: arrays	0	0	100	100	100	100
lufact: arrays	0	0	100	100	100	100
moldyn: arrays	0	0	100	100	100	100
tsp: MinTourLen	0	0	100	100	100	100

- 27 racy fields (found with FastTrack)
- ran Jumble manually once for each field
- found 4 destructive races

Student Contributors

- Jaeheon Yi, UC Santa Cruz (now at Google)
- Caitlin Sadowski, UC Santa Cruz (now at Google)
- Tom Austin, UC Santa Cruz (now at San Jose State)
- Tim Disney, UC Santa Cruz

- Ben Wood, Williams College (now at Wellesley College)
- Diogenes Nunez, Williams College (now at Tufts)
- Antal Spector-Zabusky, Williams College (now at UPenn)
- James Wilcox, Williams College (now at UW)
- Parker Finch, Williams College
- Emma Harrington, Williams College

Approximating Redundancy

- Record execution trace
- Annotate accesses in source based on dynamic occurrences in trace.

```
sync(m) {
  t = x.f^NonRedundant;
  t = x.f^Redundant;
  ...
  t = y.f^Redundant;
}
t = x.f^NonRedundant;
```

Check on this line is necessary at least once.

Check on this line is always redundant.

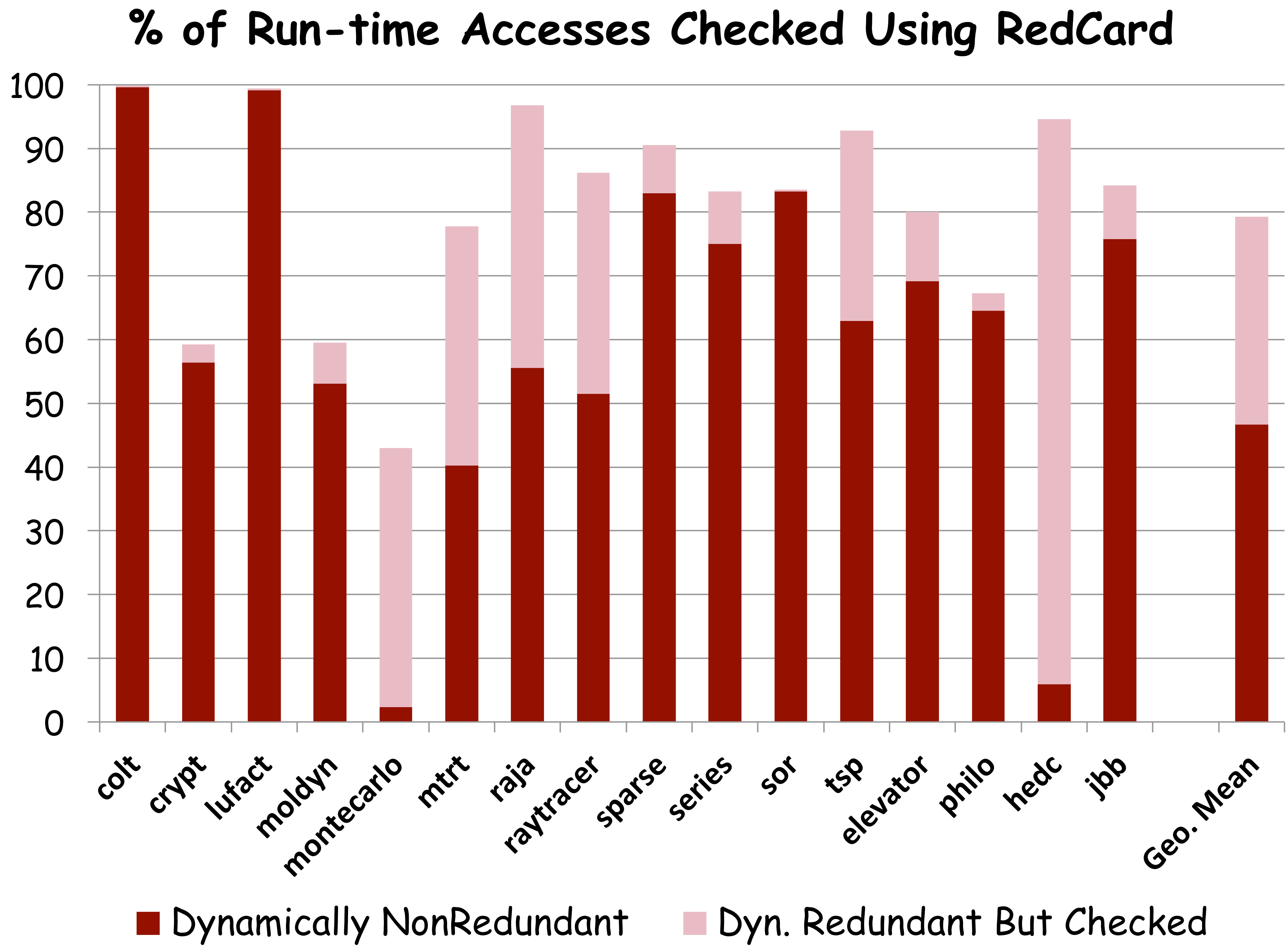
Approximating Redundancy

- Record execution trace
- Annotate accesses in source based on dynamic occurrences in trace.

```
sync(m) {
   t = x.f^NonRedundant;
   t = x.f^Redundant;
   ...
   t = y.f^Redundant;
}
t = x.f^NonRedundant;
```

```
sync(m) {
   t = x.f^Check;
   t = x.f^NoCheck;
   ...
   t = y.f^Check;
}
t = x.f^Check;
```

- Compare to RedCard annotations
  - `NoCheck` Accesses $\subseteq$ `Redundant` Accesses



% of Run-time Accesses Checked Using RedCard
100
90
80
70
60
50
40
30
20
10
0
colt
crypt
lufact
moldyn
montecarlo
mtrt
raja
raytracer
sparse
series
sor
tsp
elevator
philo
hedc
jbb
Geo. Mean
Dynamically NonRedundant
Dyn. Redundant But Checked

Where To Go From Here?

- Static Race Checking Analysis
- Performance (goal is always-on precise detection...)
  - HW support
  - static-dynamic hybrid analyses
  - sampling
- Coverage
  - symbolic model checking, specialized schedulers
- Classify malignant/benign data races
  - which data races are most critical?
- How to respond to data races? warn/fail-fast/recover?
- Reproducing traces exhibiting rare data races
  - record and replay
- Generalization
  - reason about traces beyond the observed trace

Increasing Redundancy

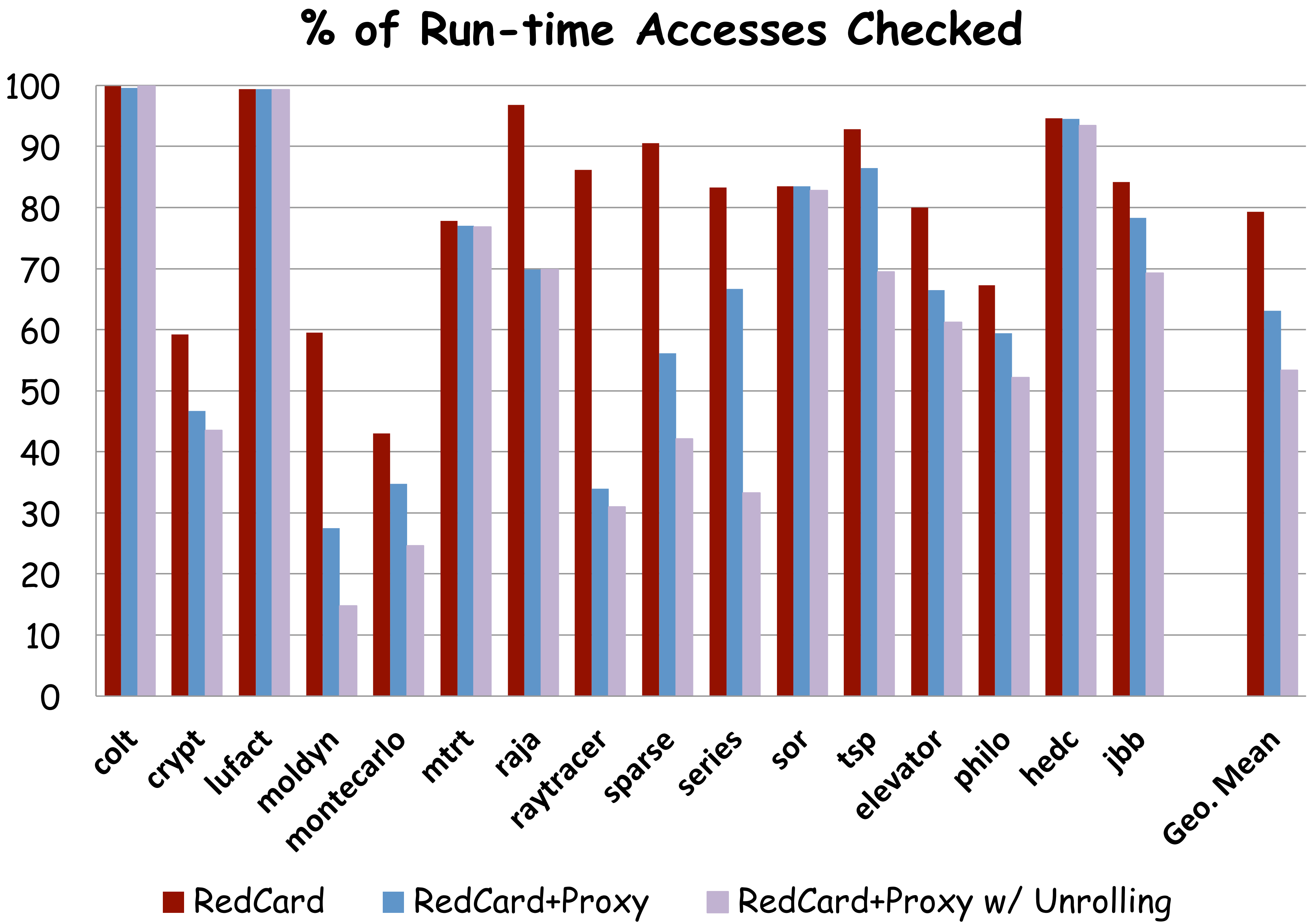
- Unroll first iteration of loops [Choi et al 03]

```
for (i = 0; i < N; i++)
  p.f^Check.m();
```

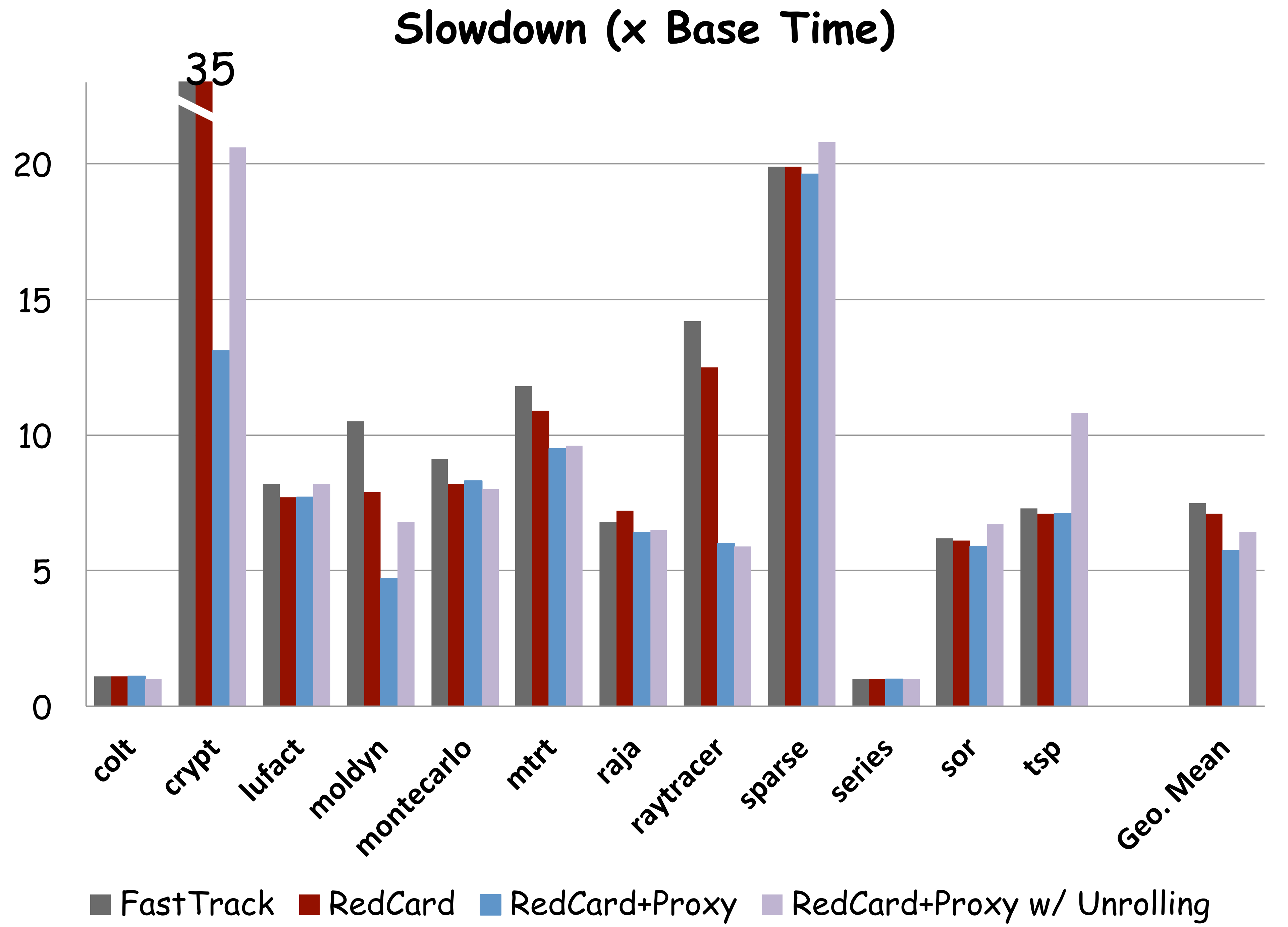
⇨

```
i = 0;
if (i < N) {
  p.f^Check.m();
  for (i = 1; i < N; i++)
    p.f^NoCheck.m();
}
```

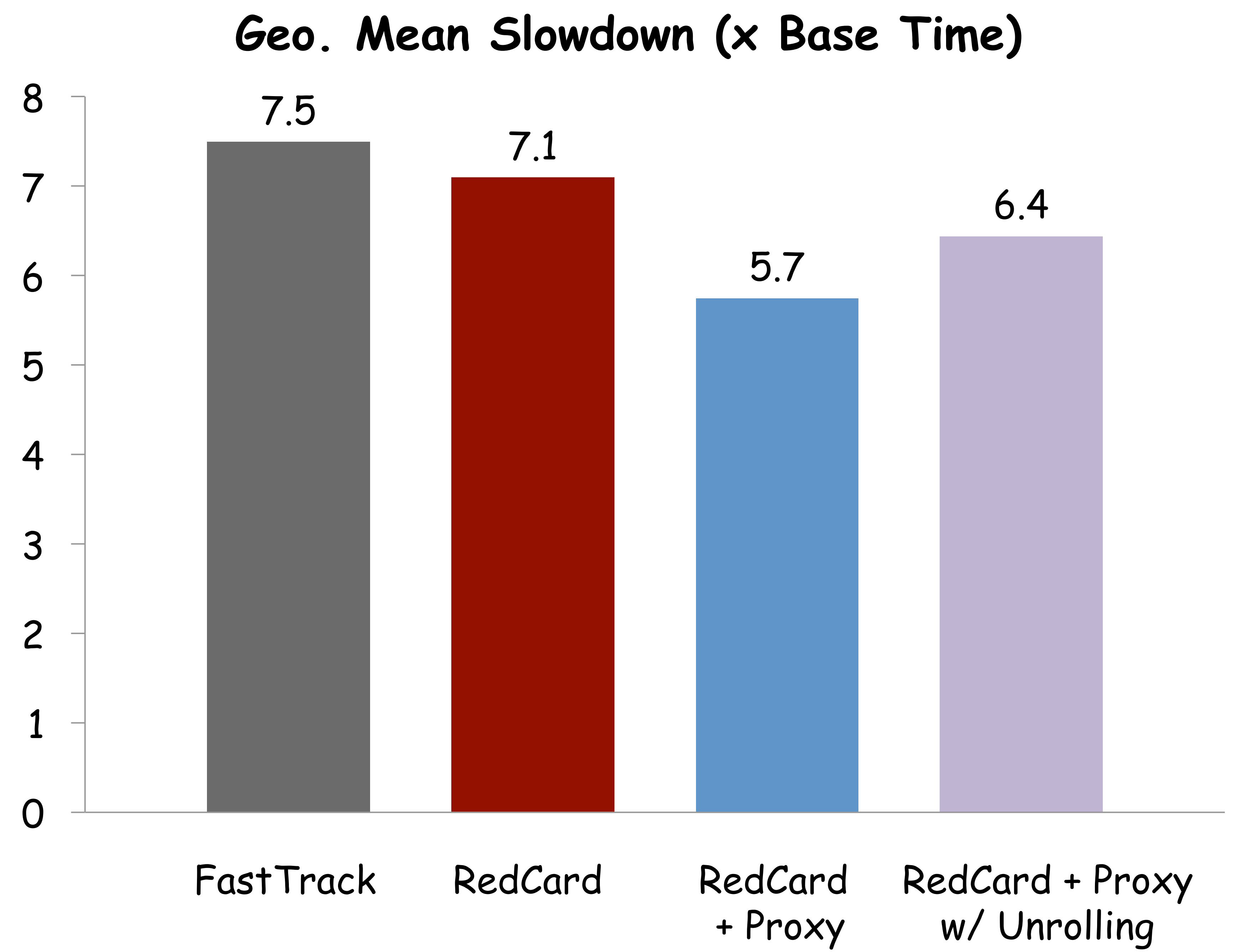
- Other transformations:
  - method specialization
  - redundant synchronization elimination
  - ...



% of Run-time Accesses Checked
100
90
80
70
60
50
40
30
20
10
0
colt
crypt
lufact
moldyn
montecarlo
mtrt
raja
raytracer
sparse
series
sor
tsp
elevator
philo
hedc
jbb
Geo. Mean
RedCard
RedCard+Proxy
RedCard+Proxy w/ Unrolling



Slowdown (x Base Time)
35
20
15
10
5
0
colt
crypt
lufact
moldyn
montecarlo
mtrt
raja
raytracer
sparse
series
sor
tsp
Geo. Mean
FastTrack
RedCard
RedCard+Proxy
RedCard+Proxy w/ Unrolling


Geo. Mean Slowdown (x Base Time)
8
7
6
5
4
3
2
1
0
7.5
7.1
5.7
6.4
FastTrack
RedCard
RedCard + Proxy
RedCard + Proxy w/ Unrolling