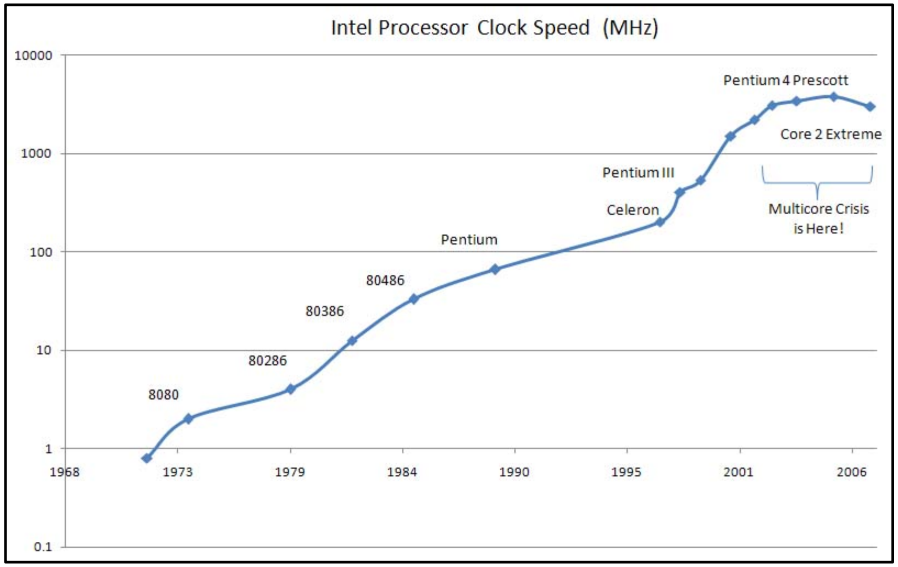
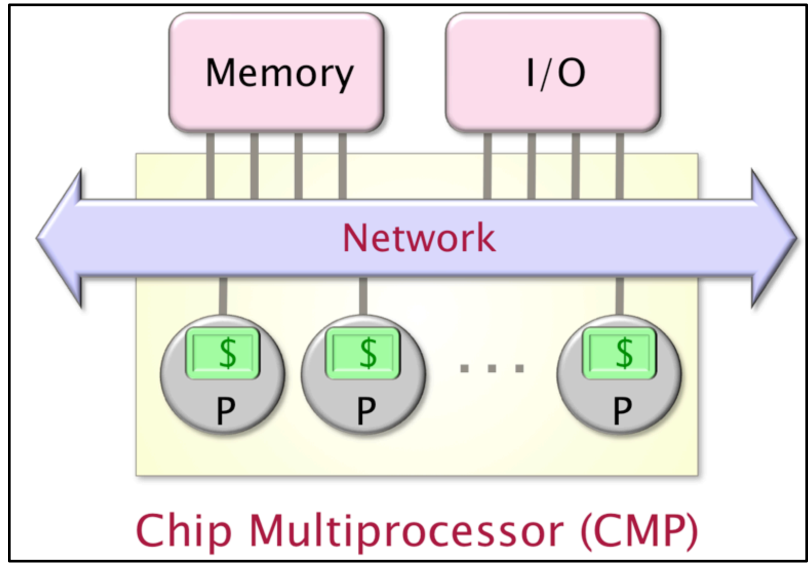
FastTrack: Efficient and Precise Dynamic Race Detection
(+ identifying destructive races)

Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College

Multithreading and Multicore

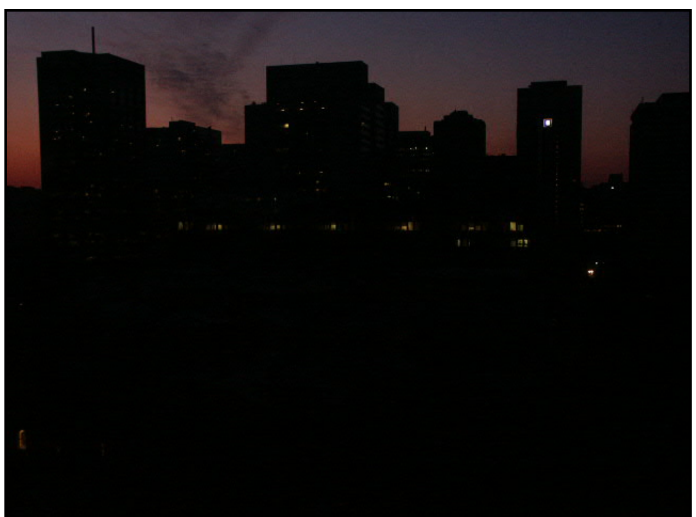




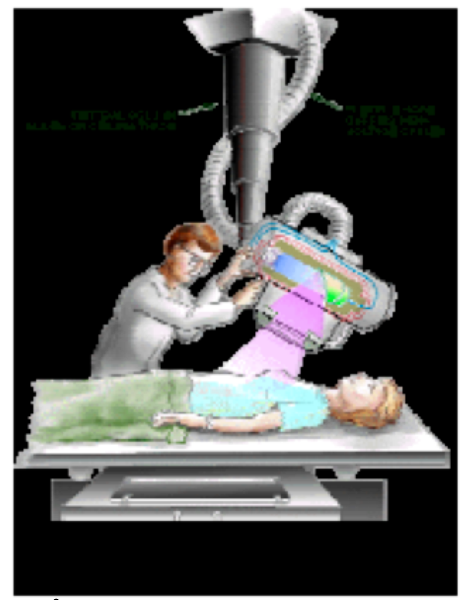
- Multithreaded programming is notoriously difficult, in part due to schedule-dependent behavior
  - race conditions, deadlocks, atomicity violations, ...
  - difficult to detect, reproduce, or eliminate

Race Conditions

- Two threads access a shared variable without synchronization, and at least one thread does a write
- Very common

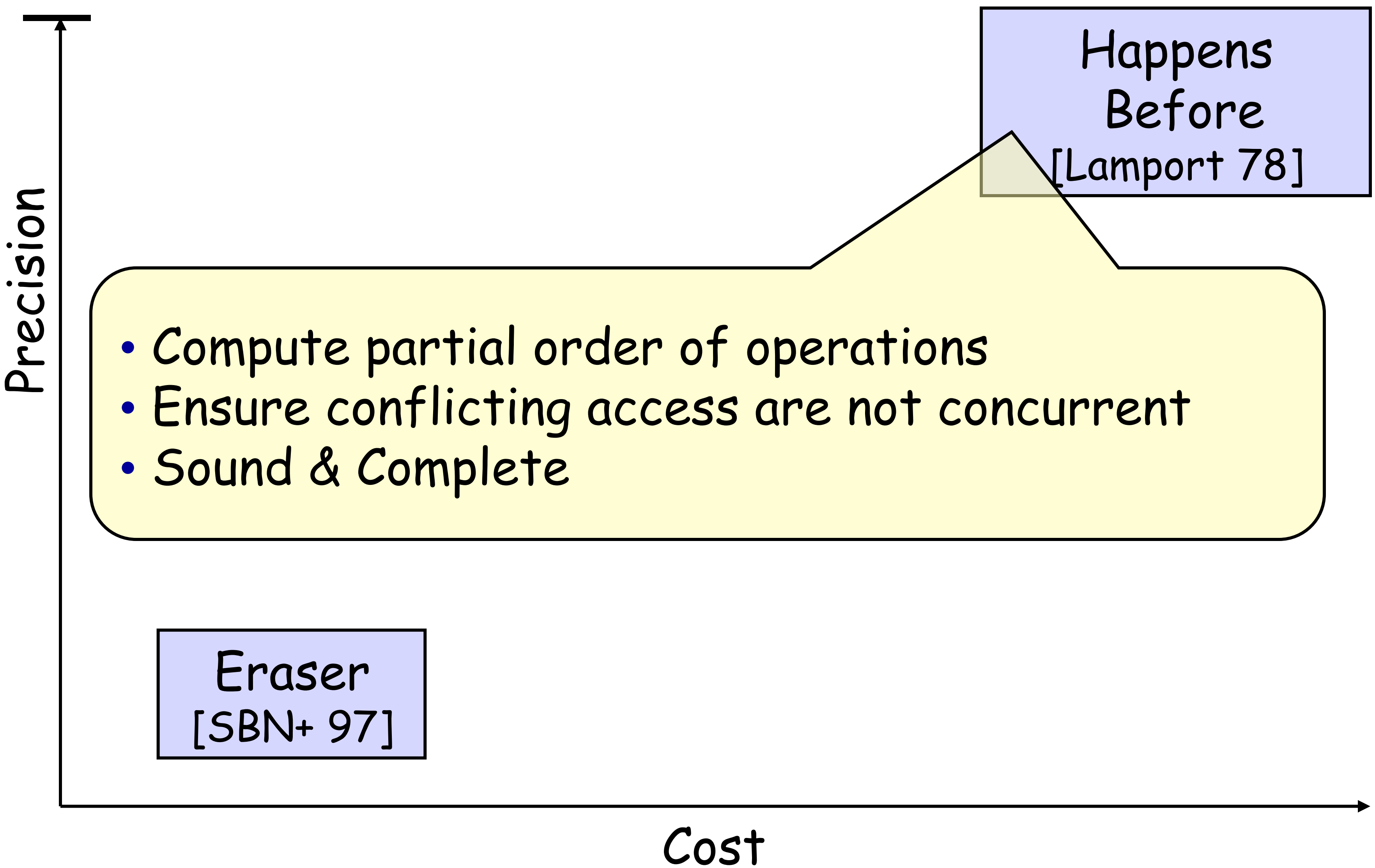


2003 Blackout ($6 Billion)

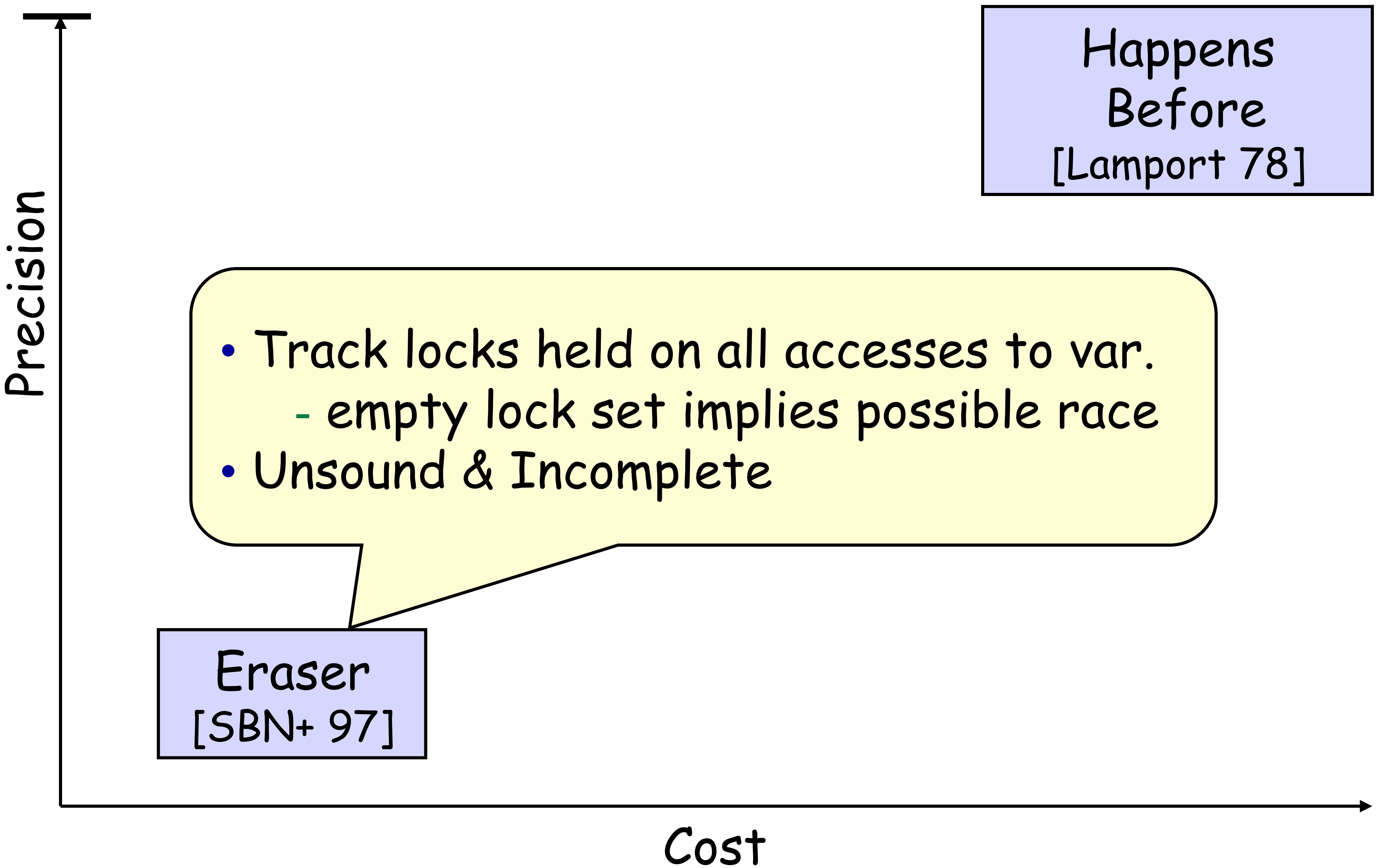


Therac-25

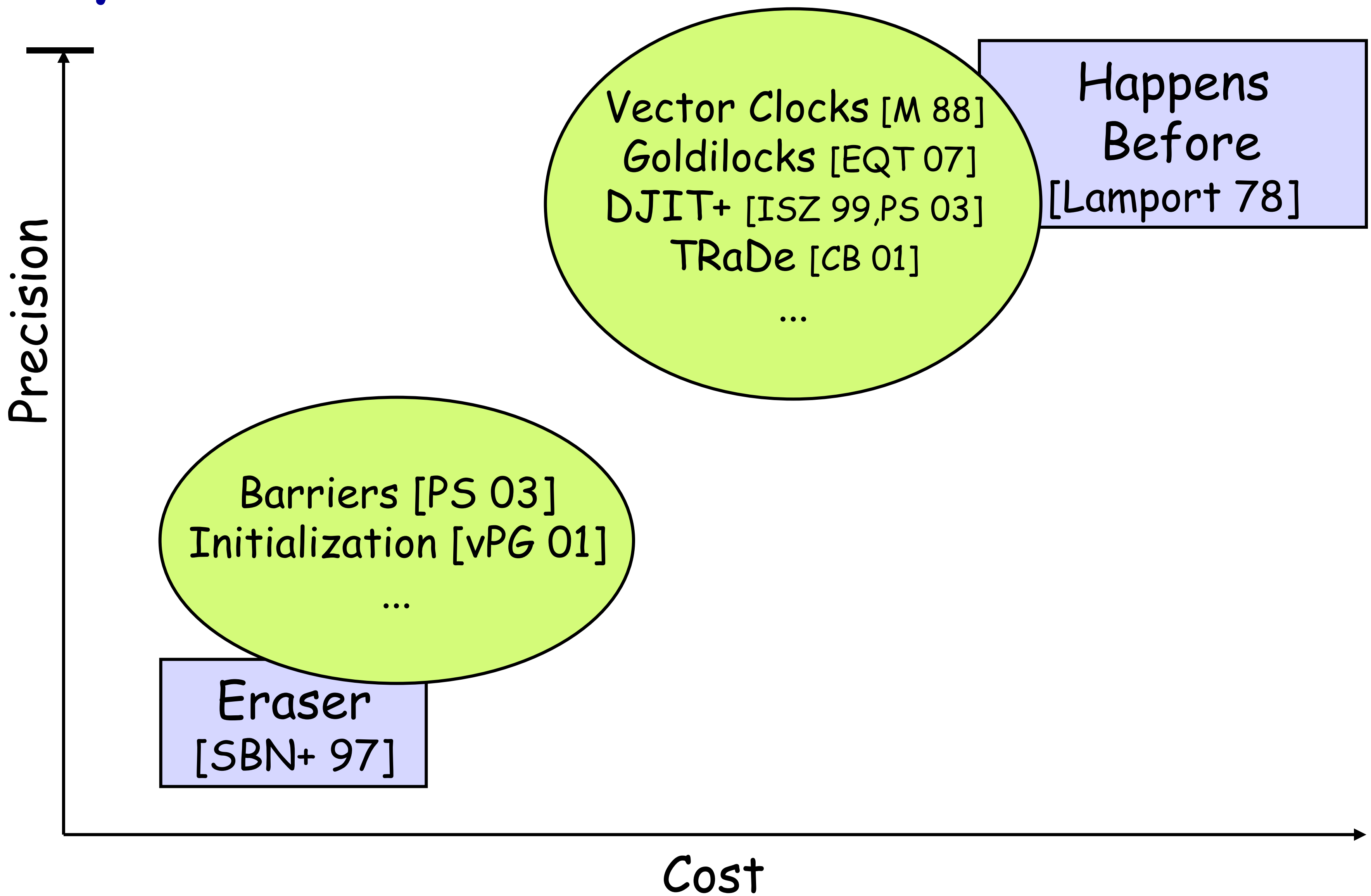
Dynamic Race Detection



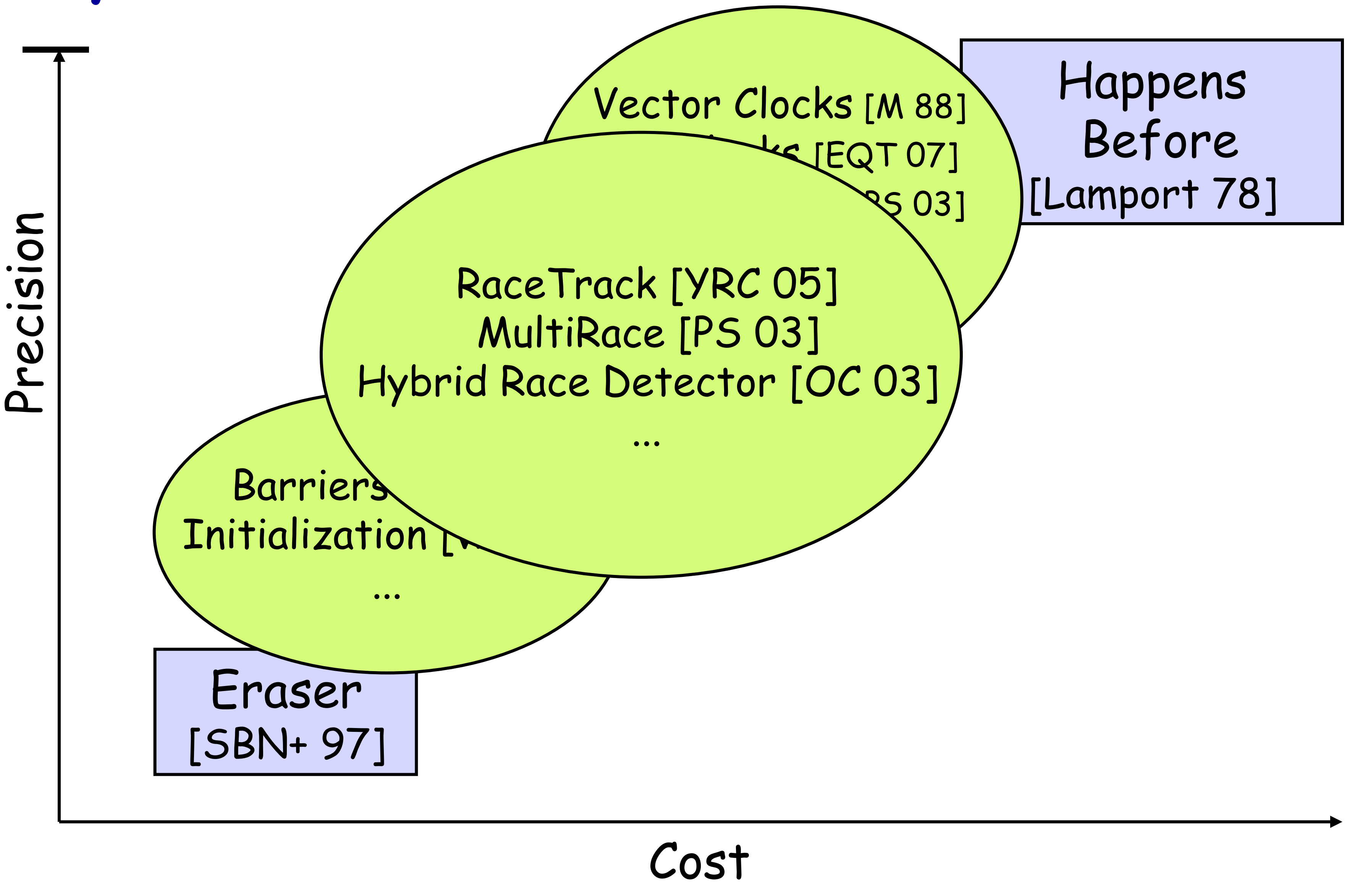
Dynamic Race Detection



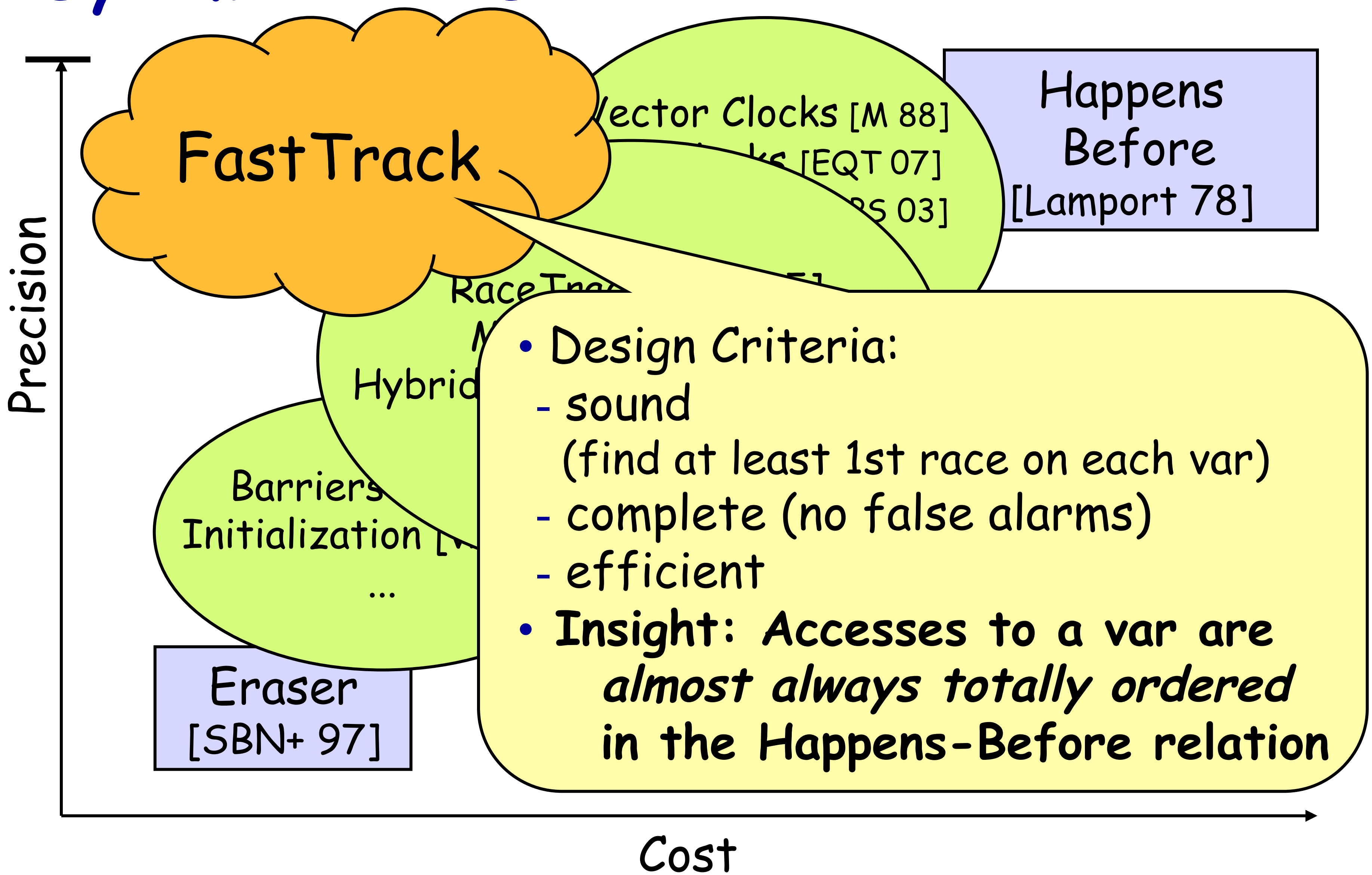
Dynamic Race Detection



Dynamic Race Detection

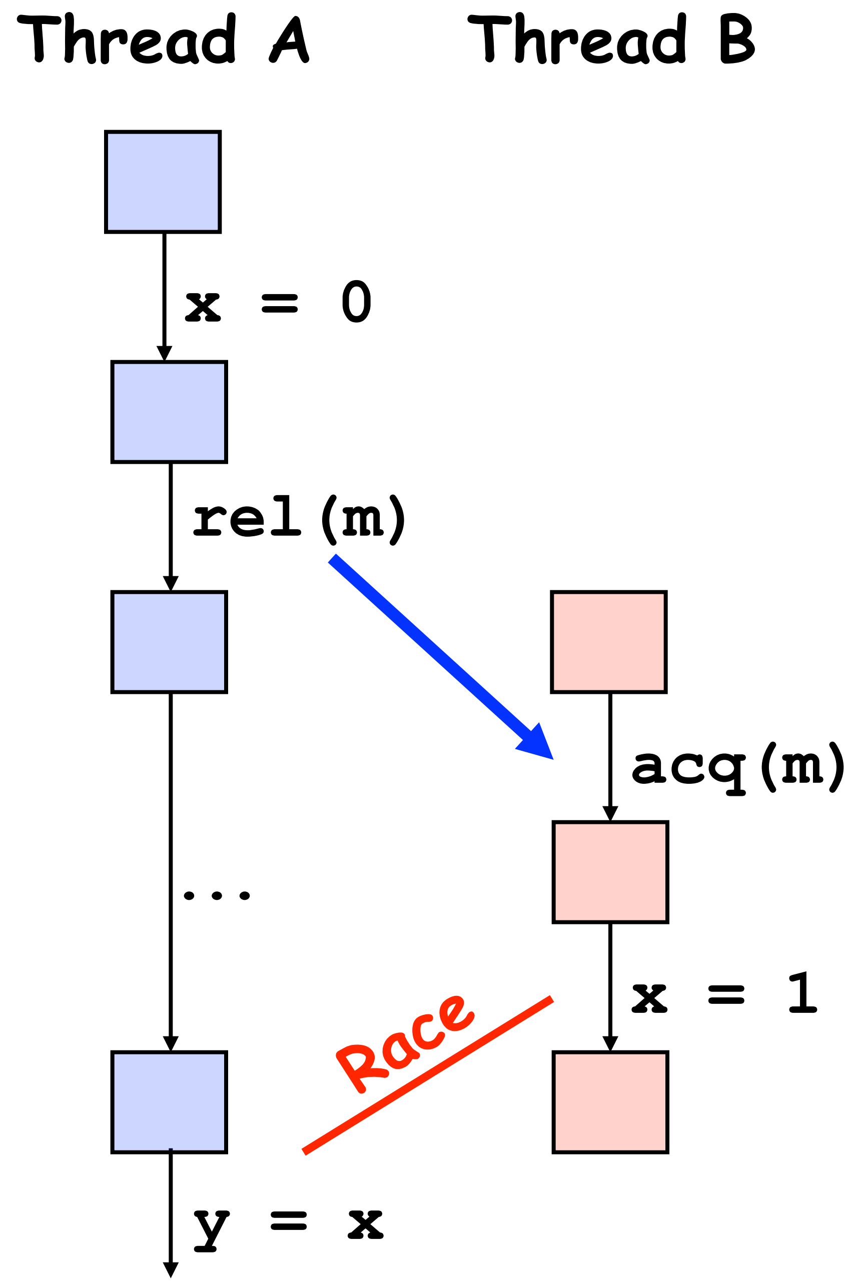


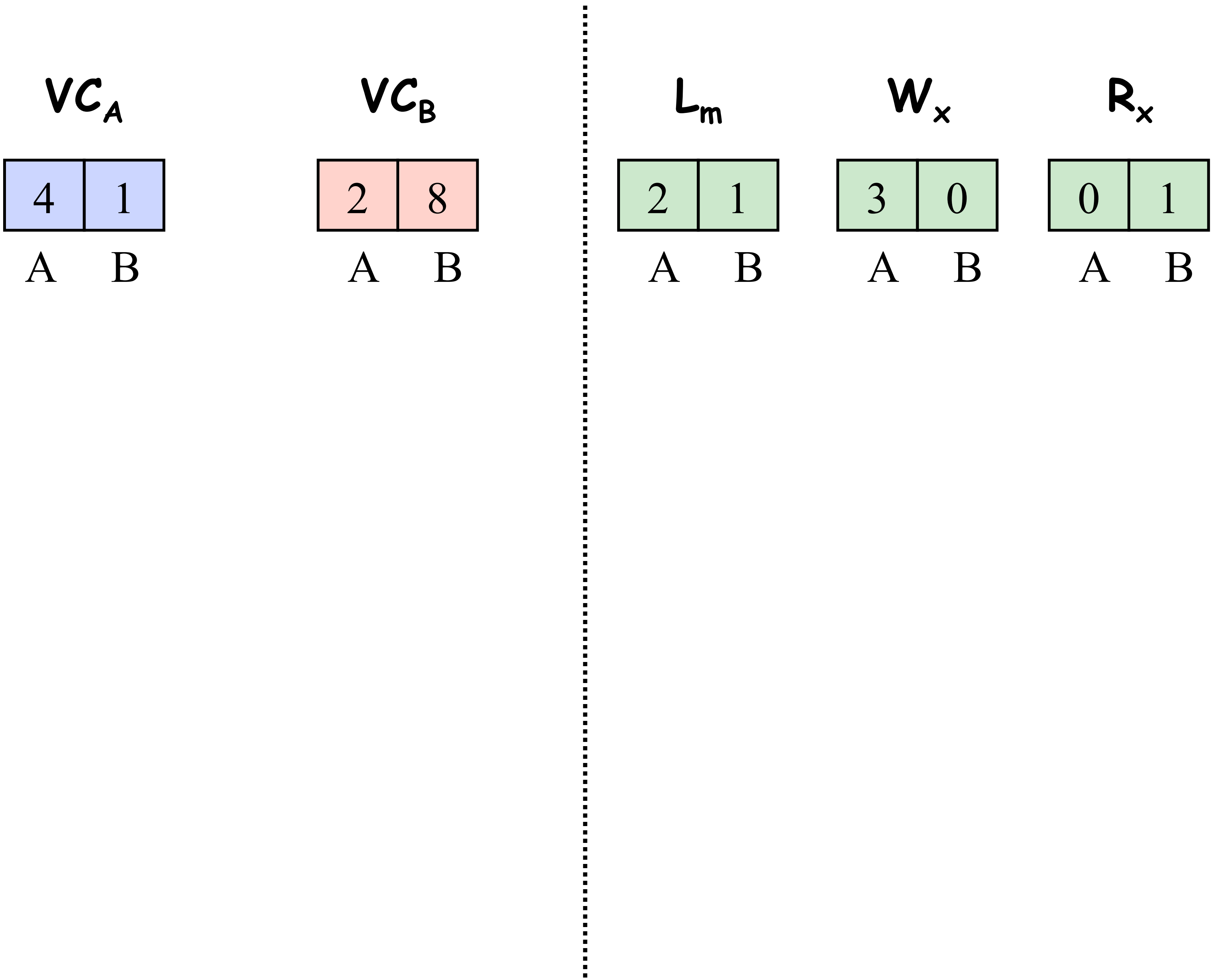
Dynamic Race Detection

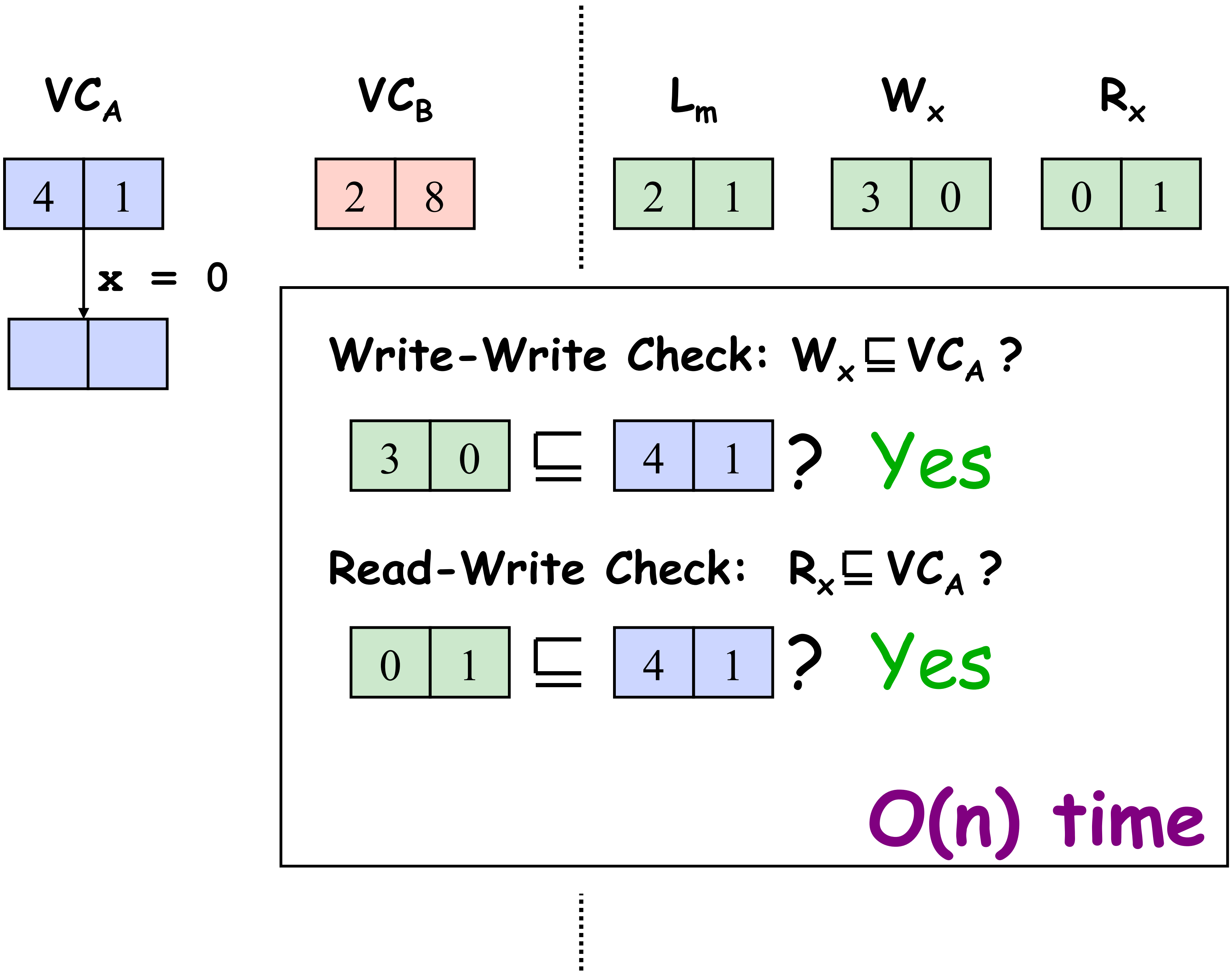


Happens-Before

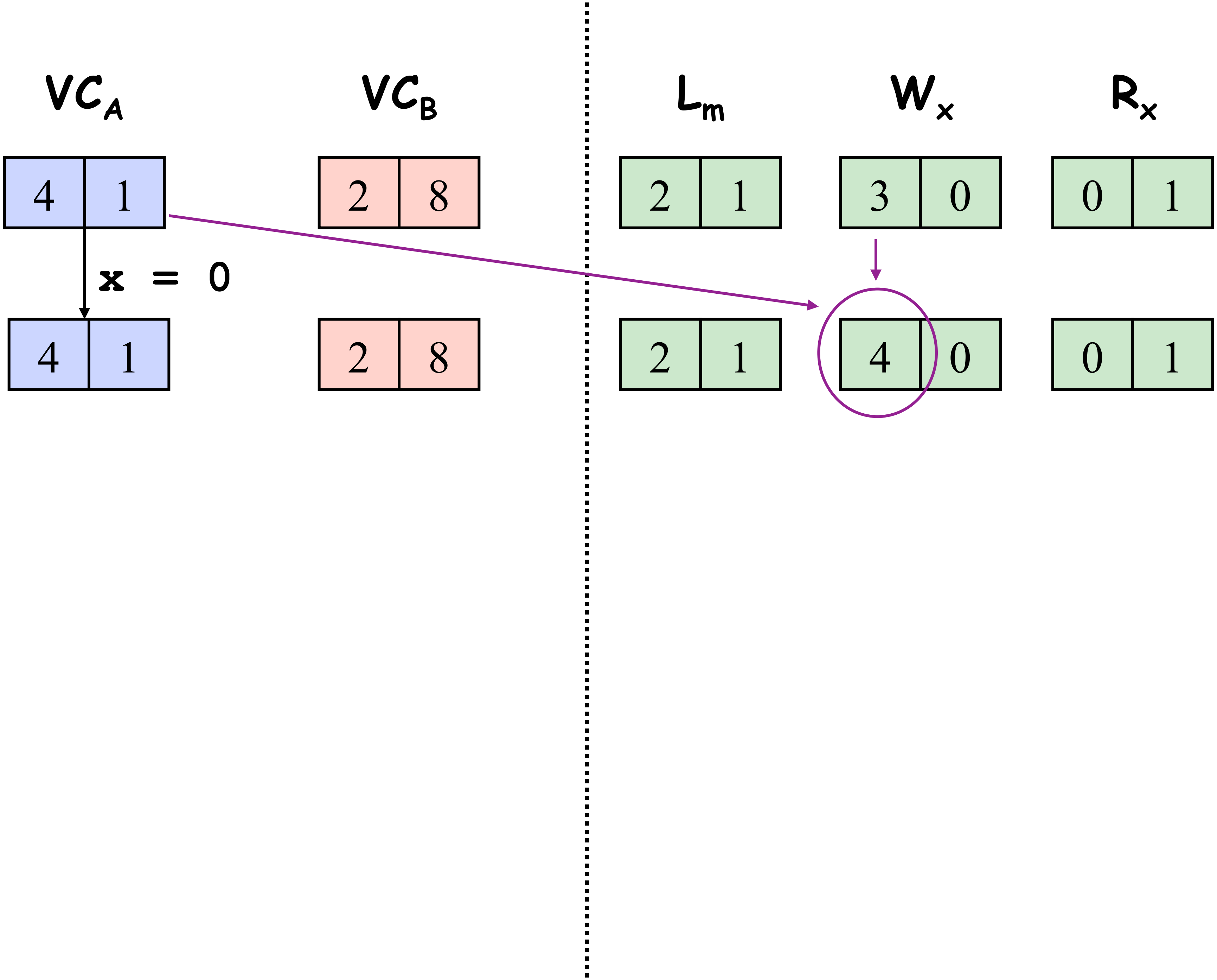
- Event Ordering:
  - program order
  - synchronization order
- Types of Races:
  - Write-Write
  - Write-Read
    - (write before read)
  - Read-Write
    - (read before write)



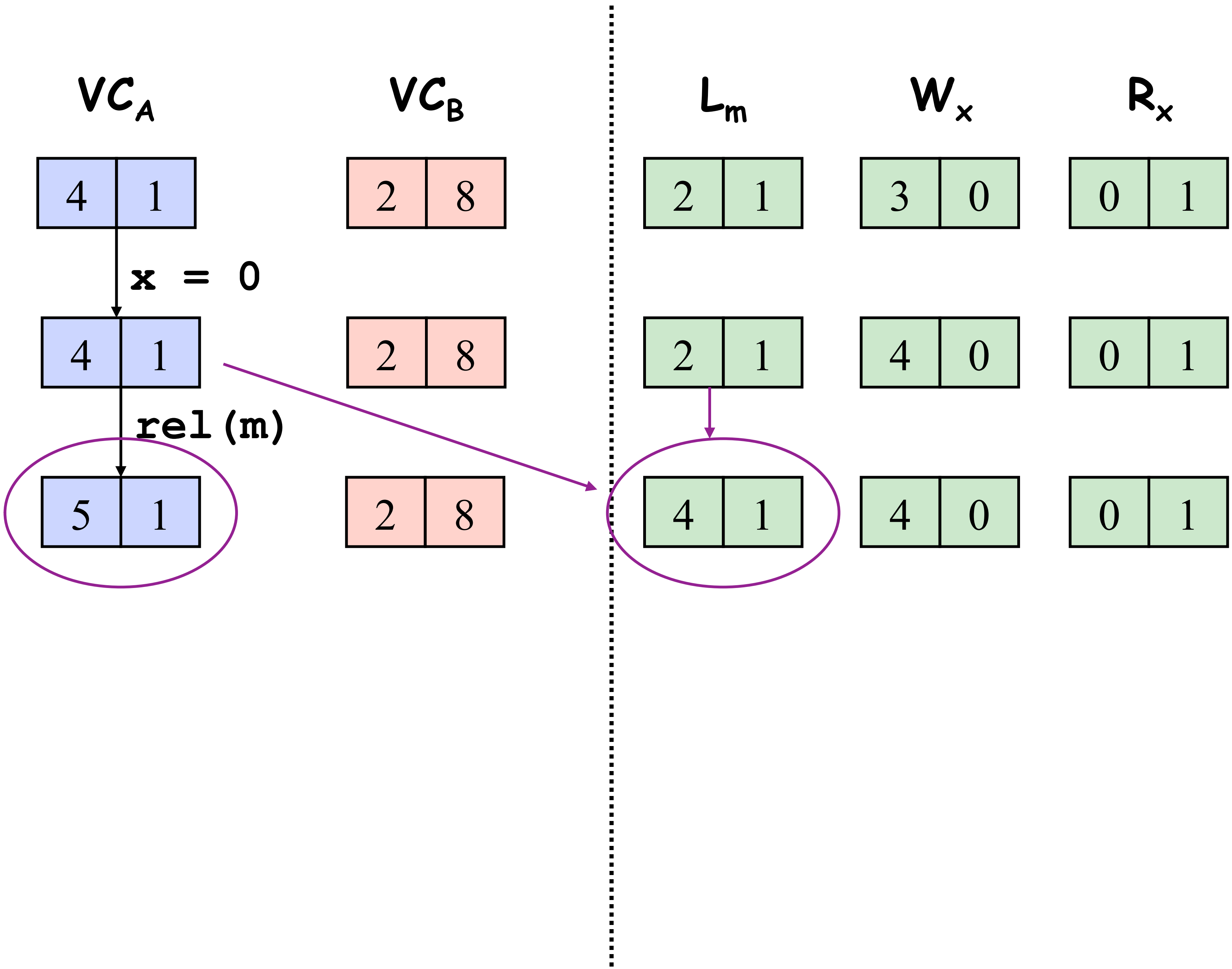

VC_A
4 1
A B
VC_B
2 8
A B
L_m
2 1
A B
W_x
3 0
A B
R_x
0 1
A B

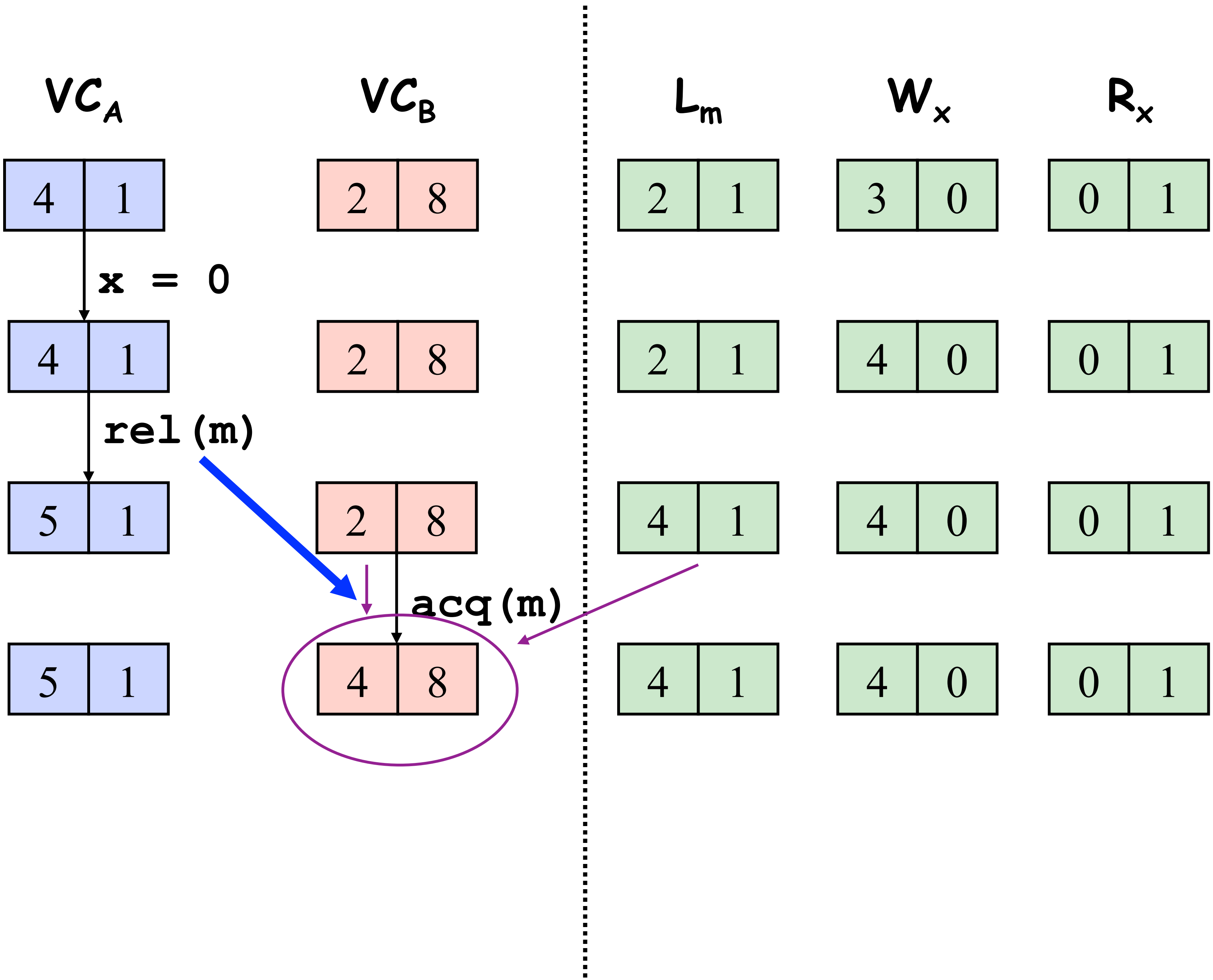


VCA
4 1
x = 0
VCB
2 8
Lm
2 1
Wx
3 0
Rx
0 1
Write-Write Check: Wx ⊑ VCA ?
3 0 ⊑ 4 1 ? Yes
Read-Write Check: Rx ⊑ VCA ?
0 1 ⊑ 4 1 ? Yes
O(n) time

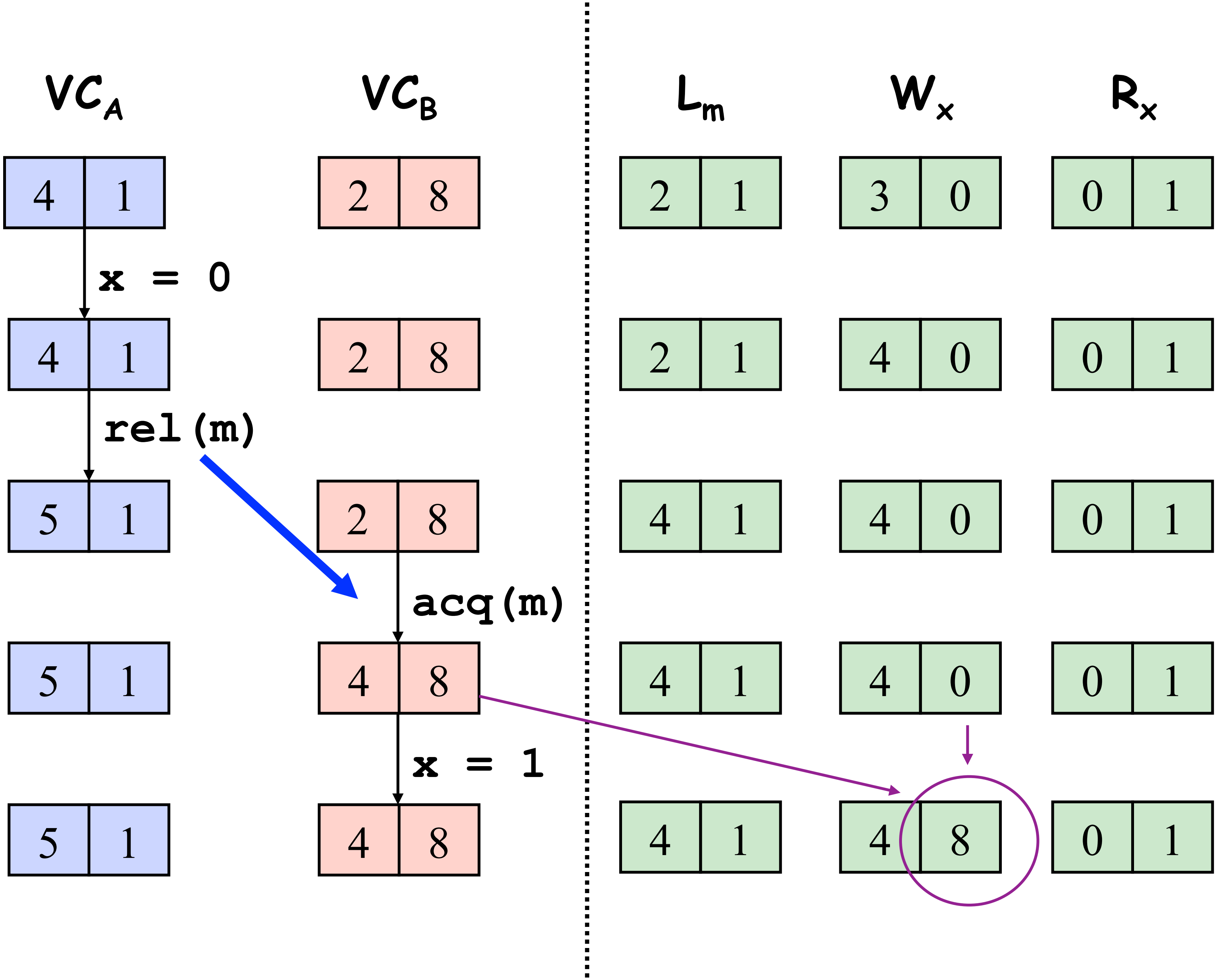


VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1

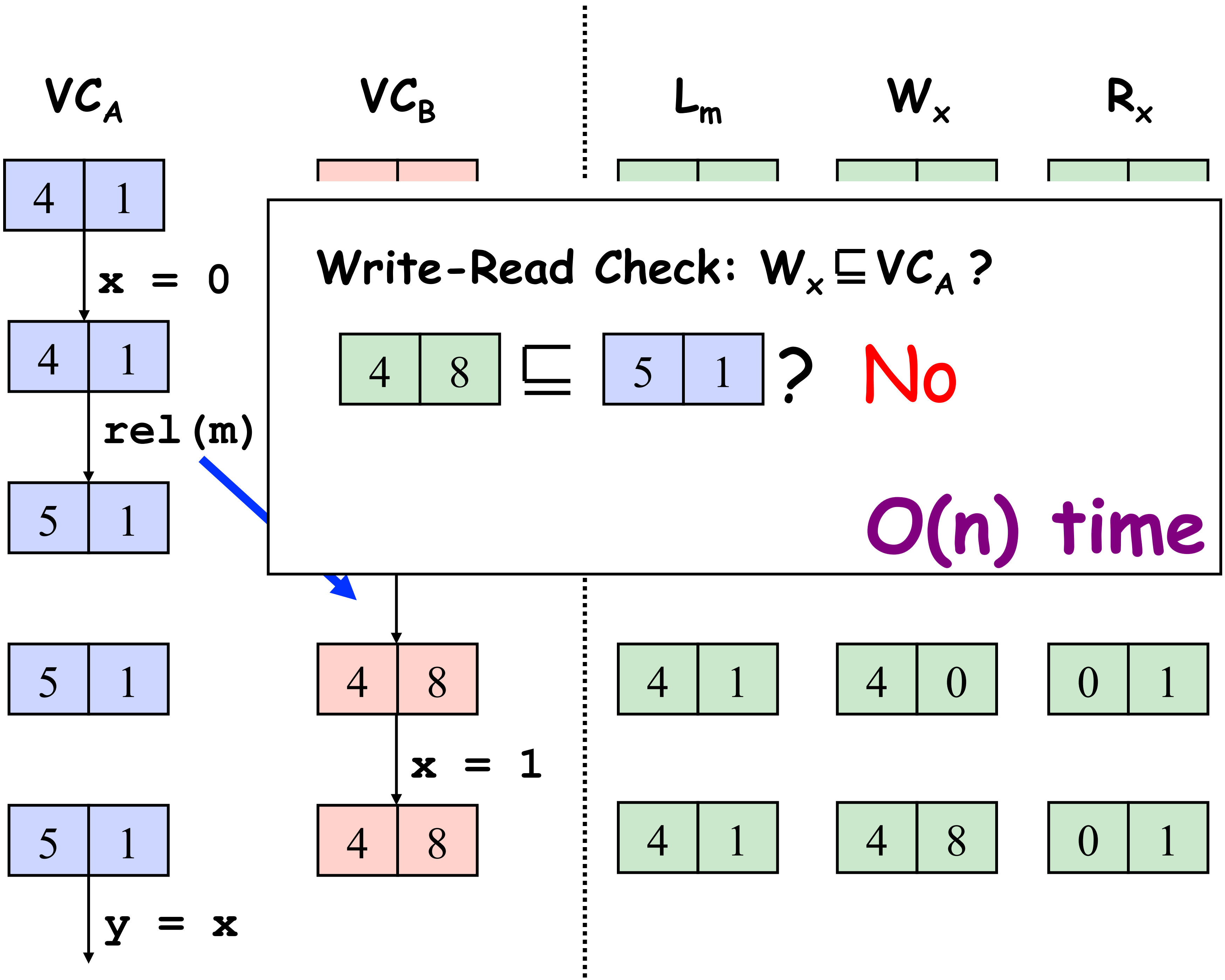

VCA
VCB
Lm
Wx
Rx
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1



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

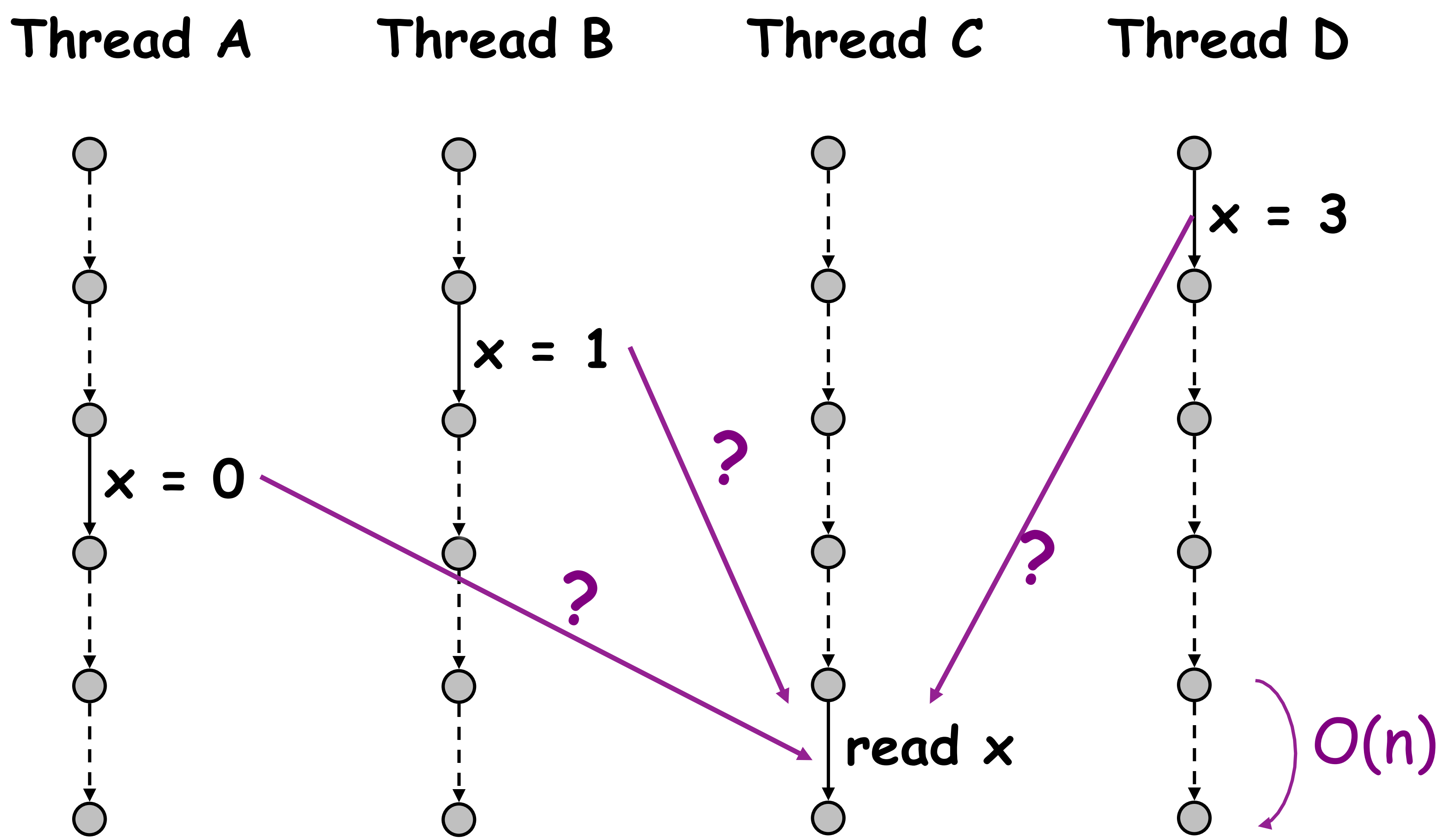


VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1
acq(m)
5 1
4 8
4 1
4 0
0 1
x = 1
5 1
4 8
4 1
4 8
0 1

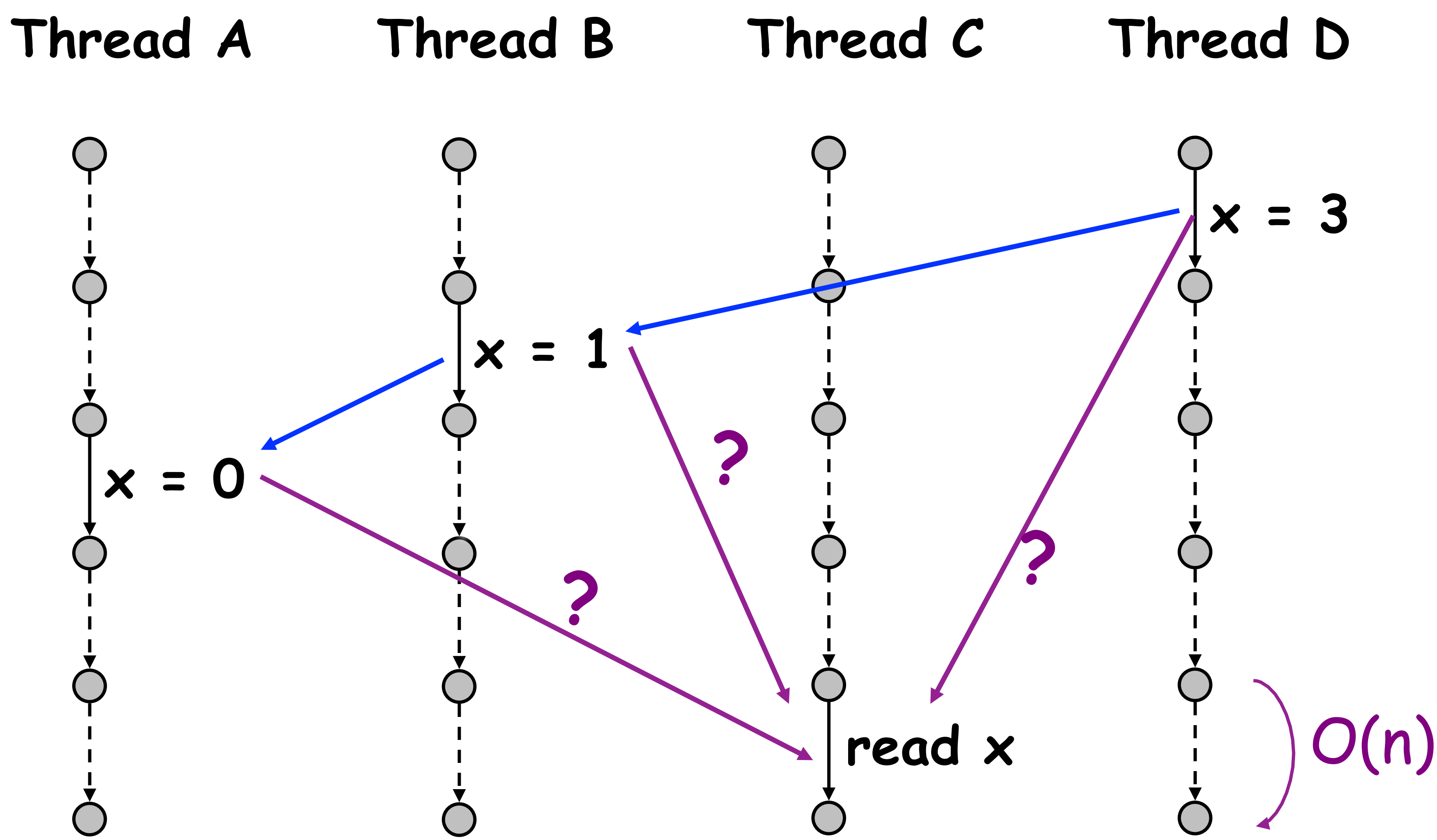


VC_A
VC_B
L_m
W_x
R_x
4 1
x = 0
4 1
rel(m)
5 1
5 1
4 8
4 1
4 0
0 1
x = 1
5 1
4 8
4 1
4 8
0 1
y = x
Write-Read Check: W_x ⊑ VC_A ?
4 8 ⊑ 5 1 ?
No
O(n) time

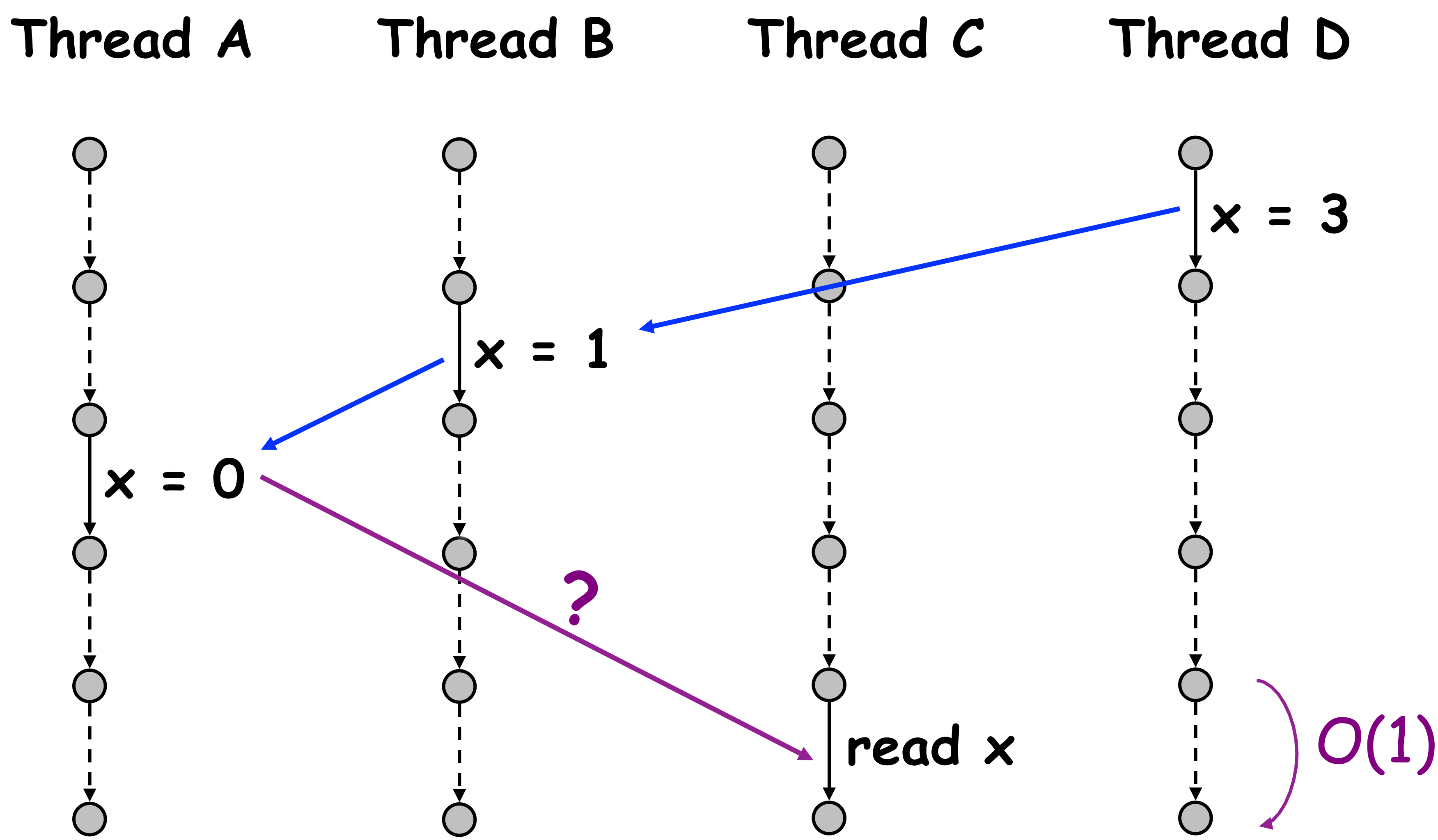
Write-Write and Write-Read Races

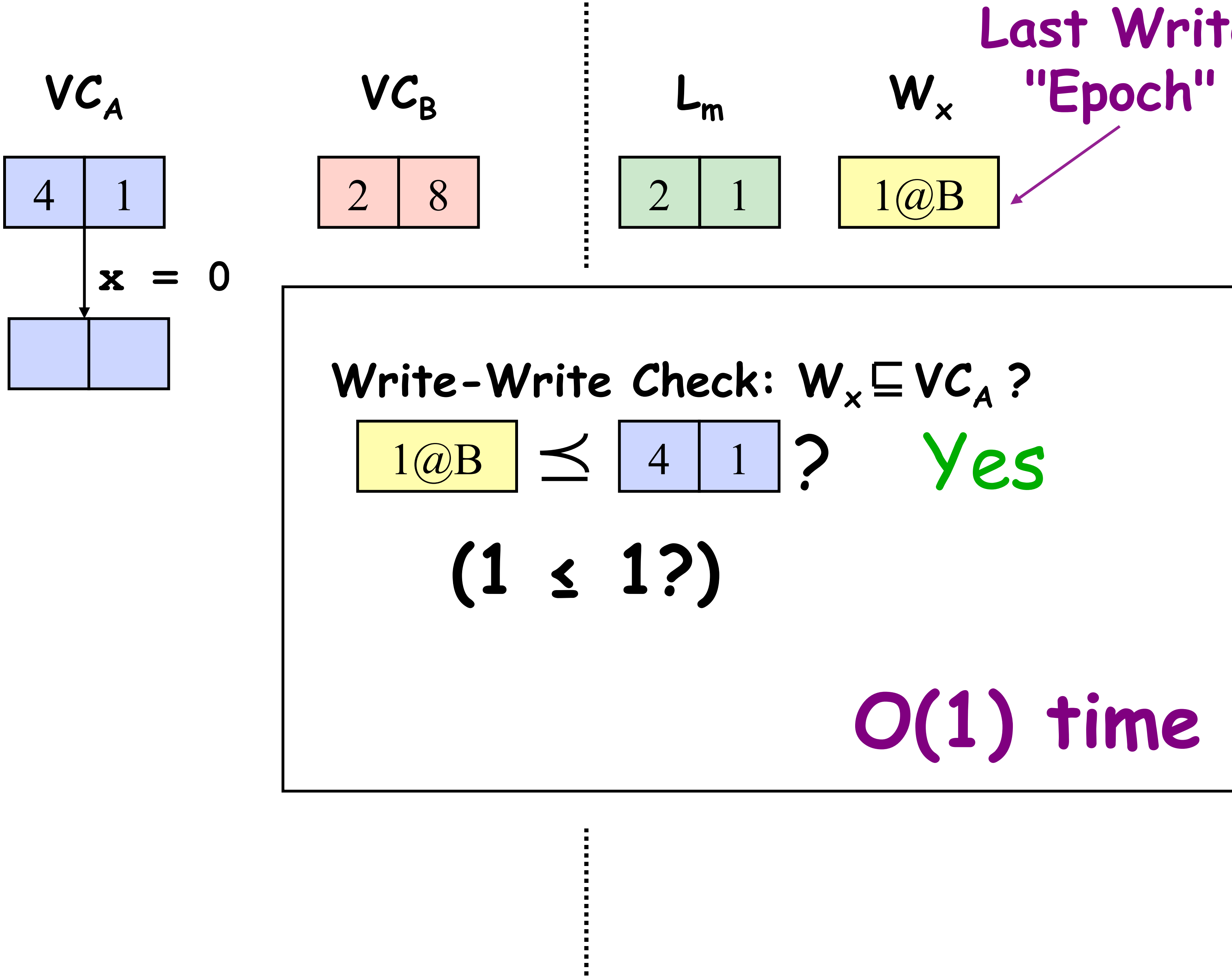


No Races Yet: Writes Totally Ordered!

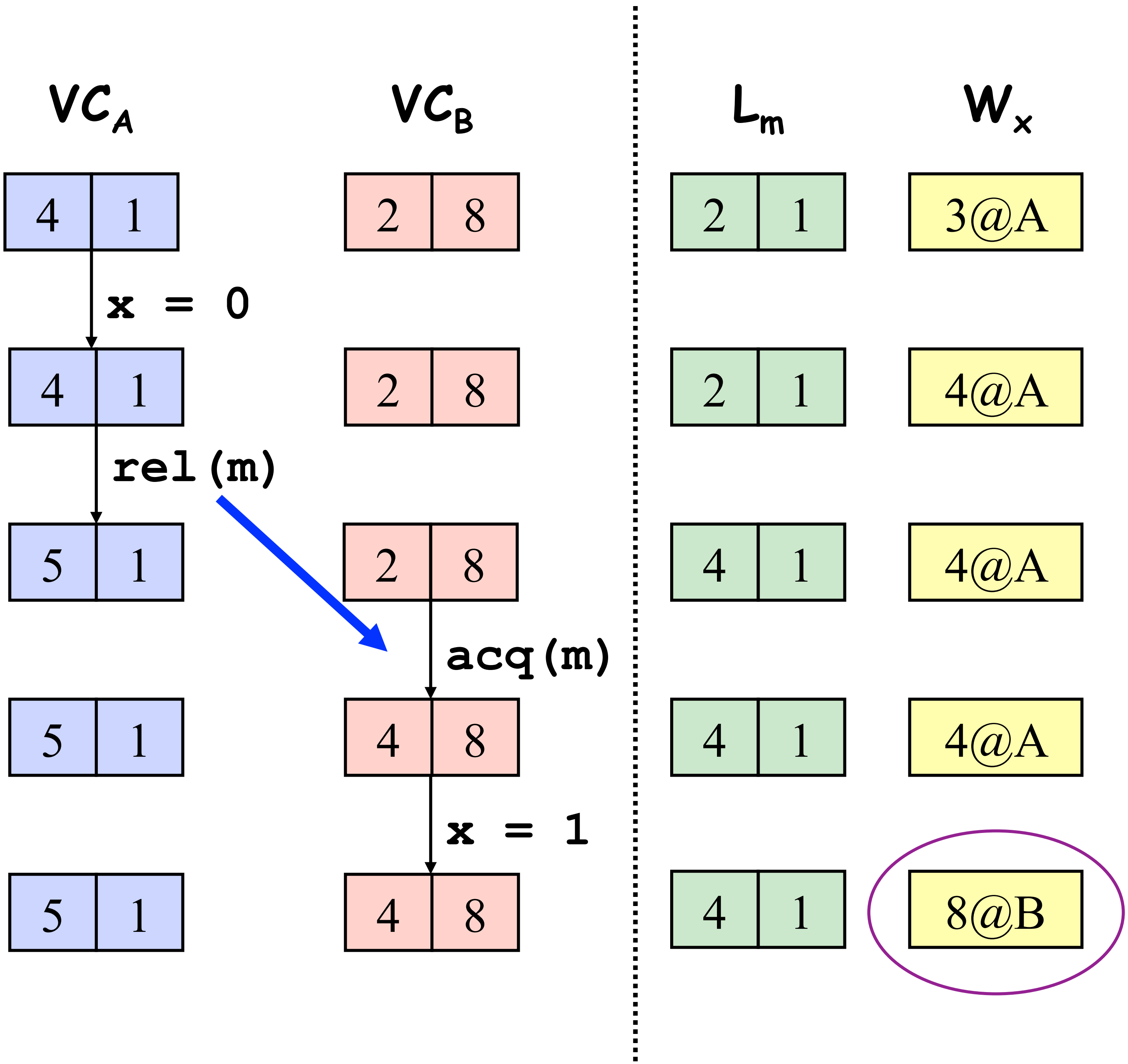


No Races Yet: Writes Totally Ordered!

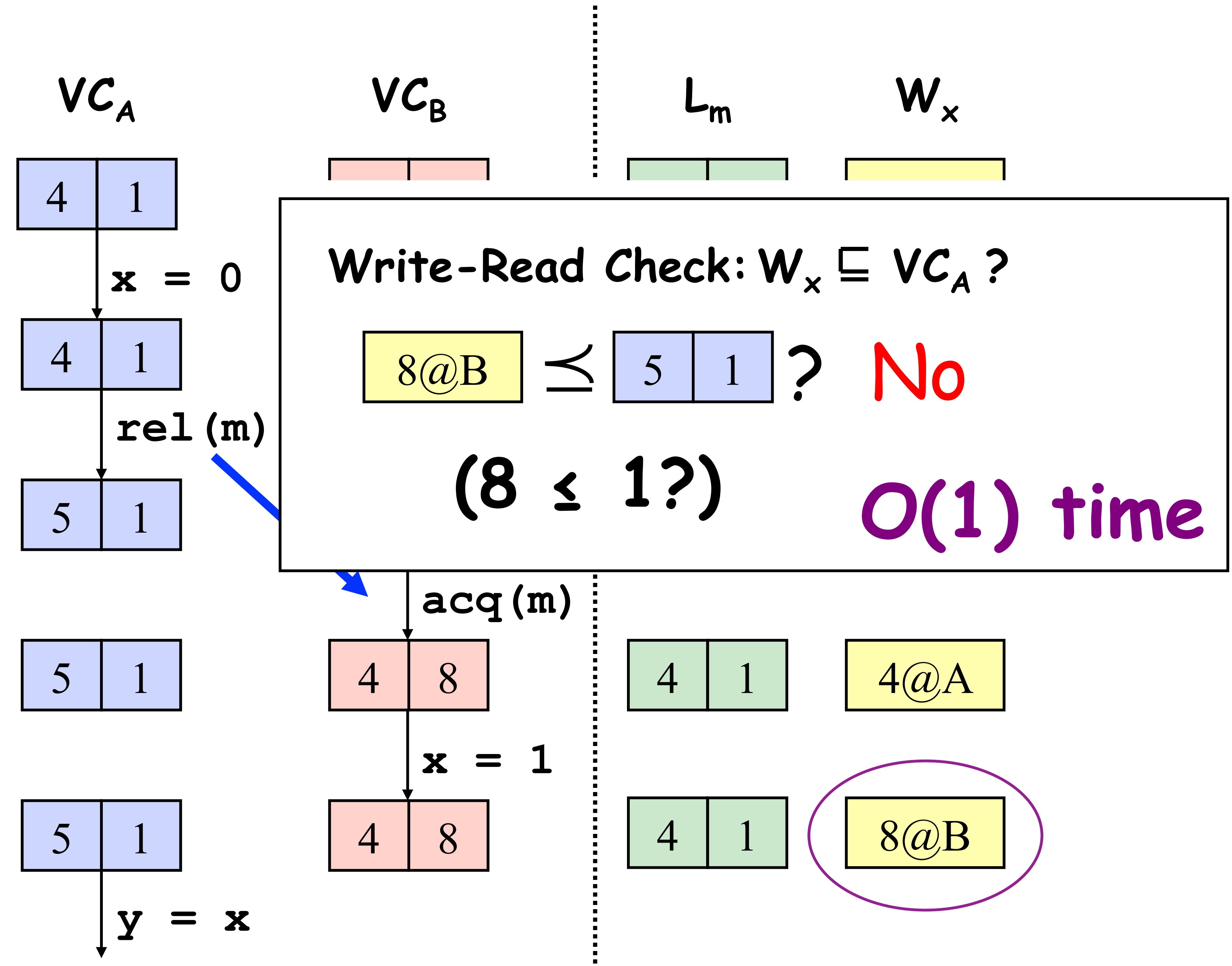




Last Write "Epoch"
VC_A
VC_B
L_m
W_x
4
1
2
8
2
1
1@B
x = 0
Write-Write Check: W_x ⊑ VC_A ?
1@B ≼ 4 1 ?
Yes
(1 ≤ 1?)
O(1) time

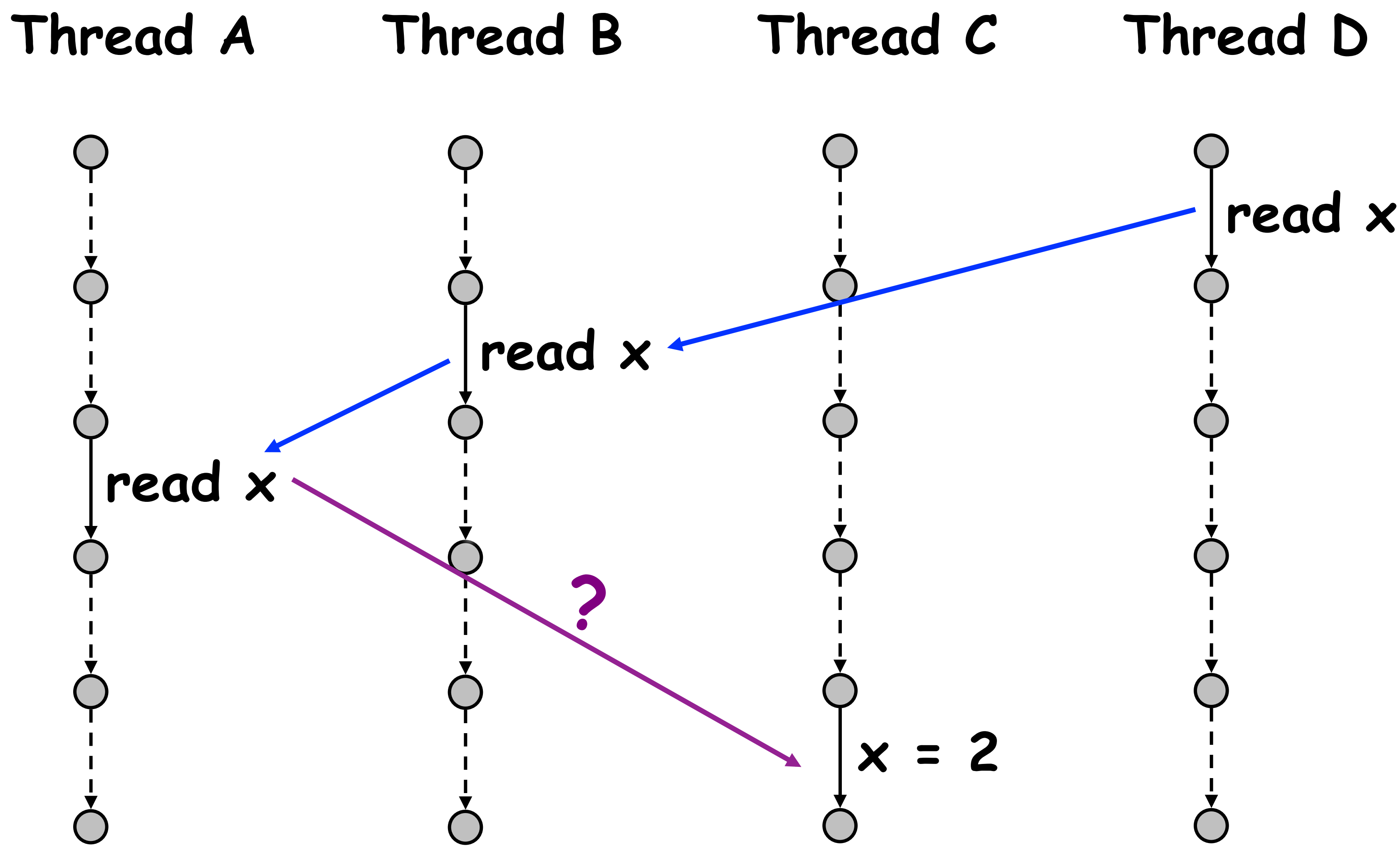


VC_A
VC_B
L_m
W_x
4 1
2 8
2 1
3@A
x = 0
4 1
2 8
2 1
4@A
rel(m)
5 1
2 8
4 1
4@A
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B



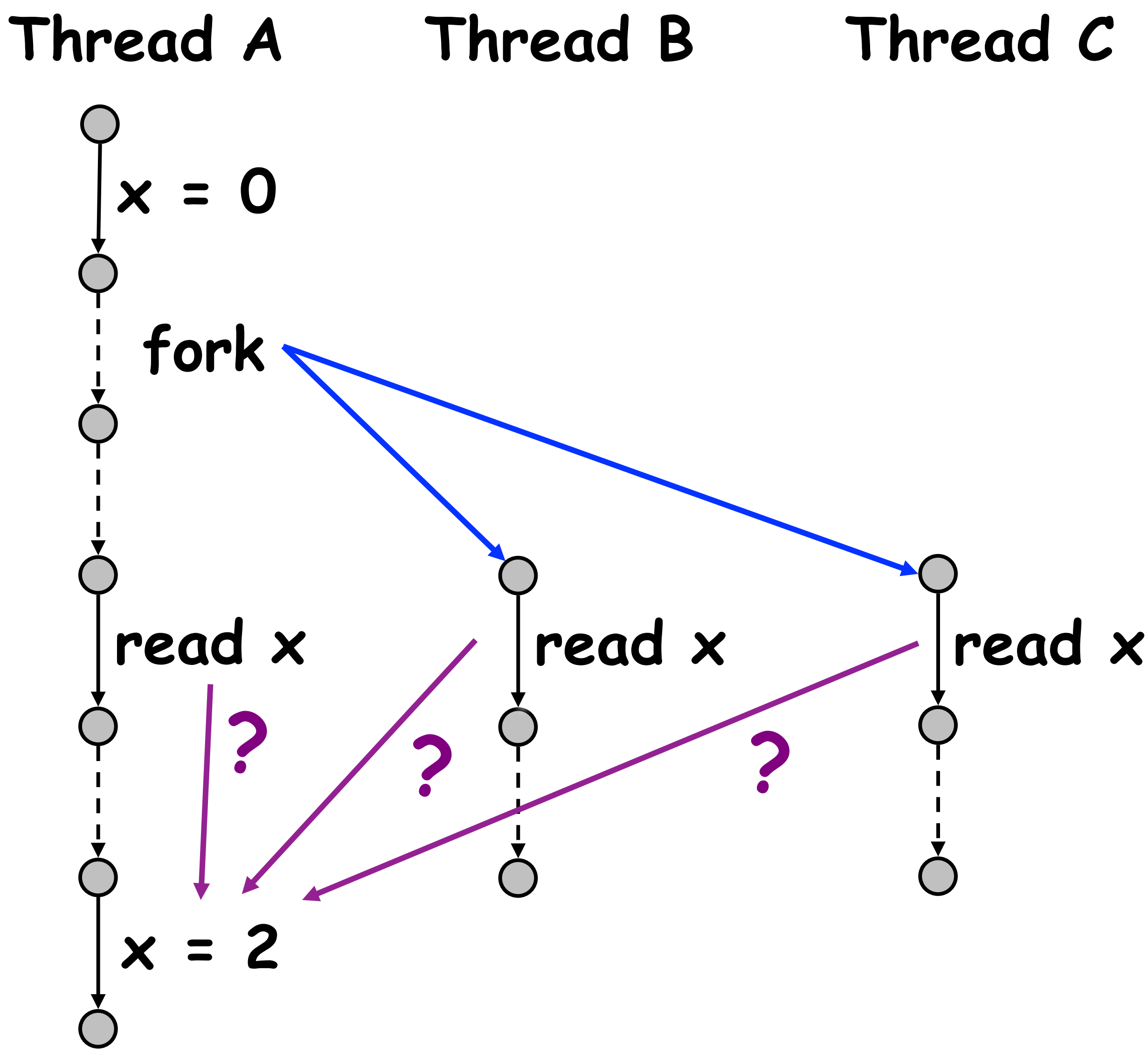
VC_A
VC_B
L_m
W_x
4 1
x = 0
4 1
rel(m)
5 1
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B
y = x
Write-Read Check: $W_x \sqsubseteq VC_A$?
8@B $\preceq$ 5 1 ? No
$(8 \leq 1?)$
O(1) time

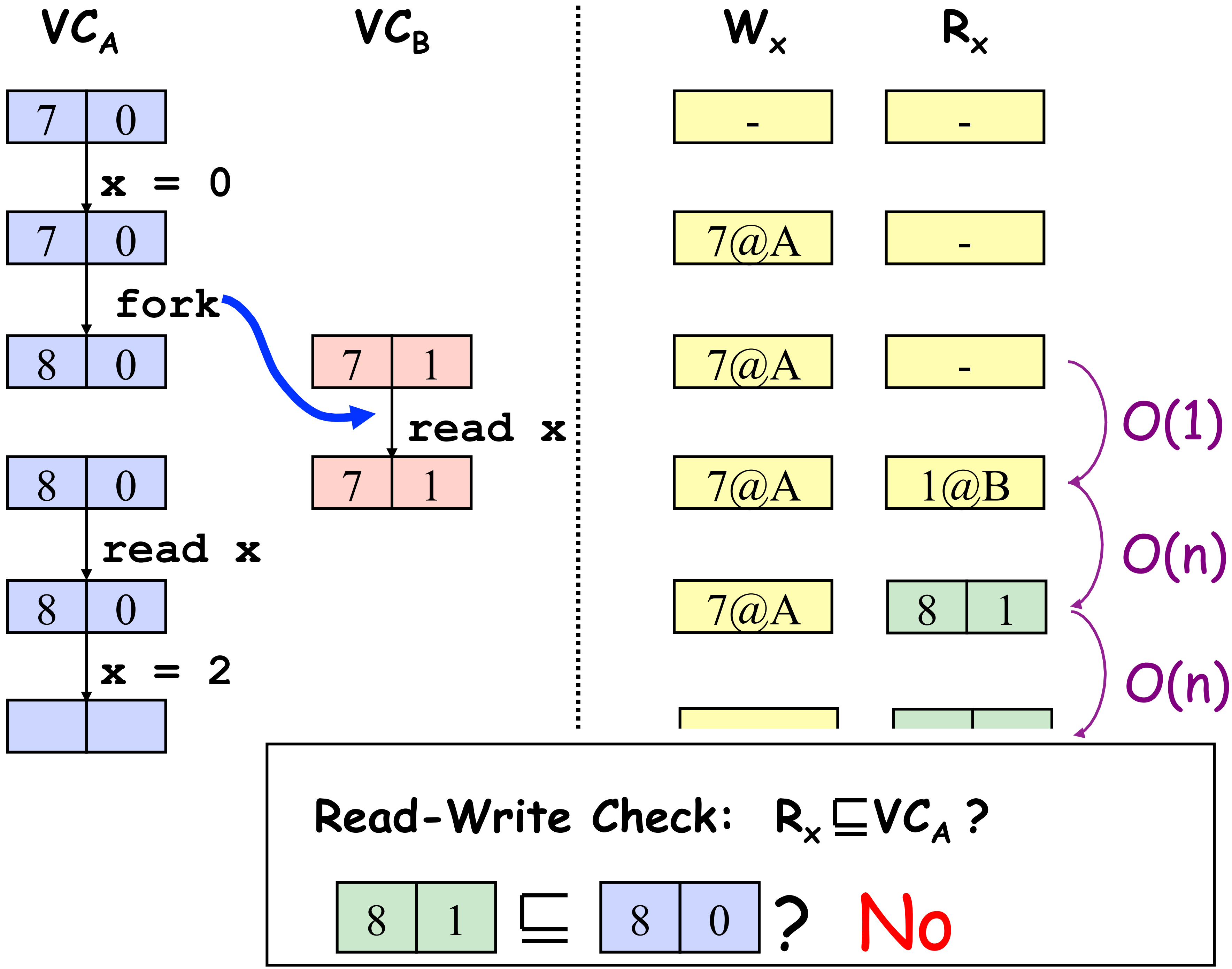
Read-Write Races -- Ordered Reads

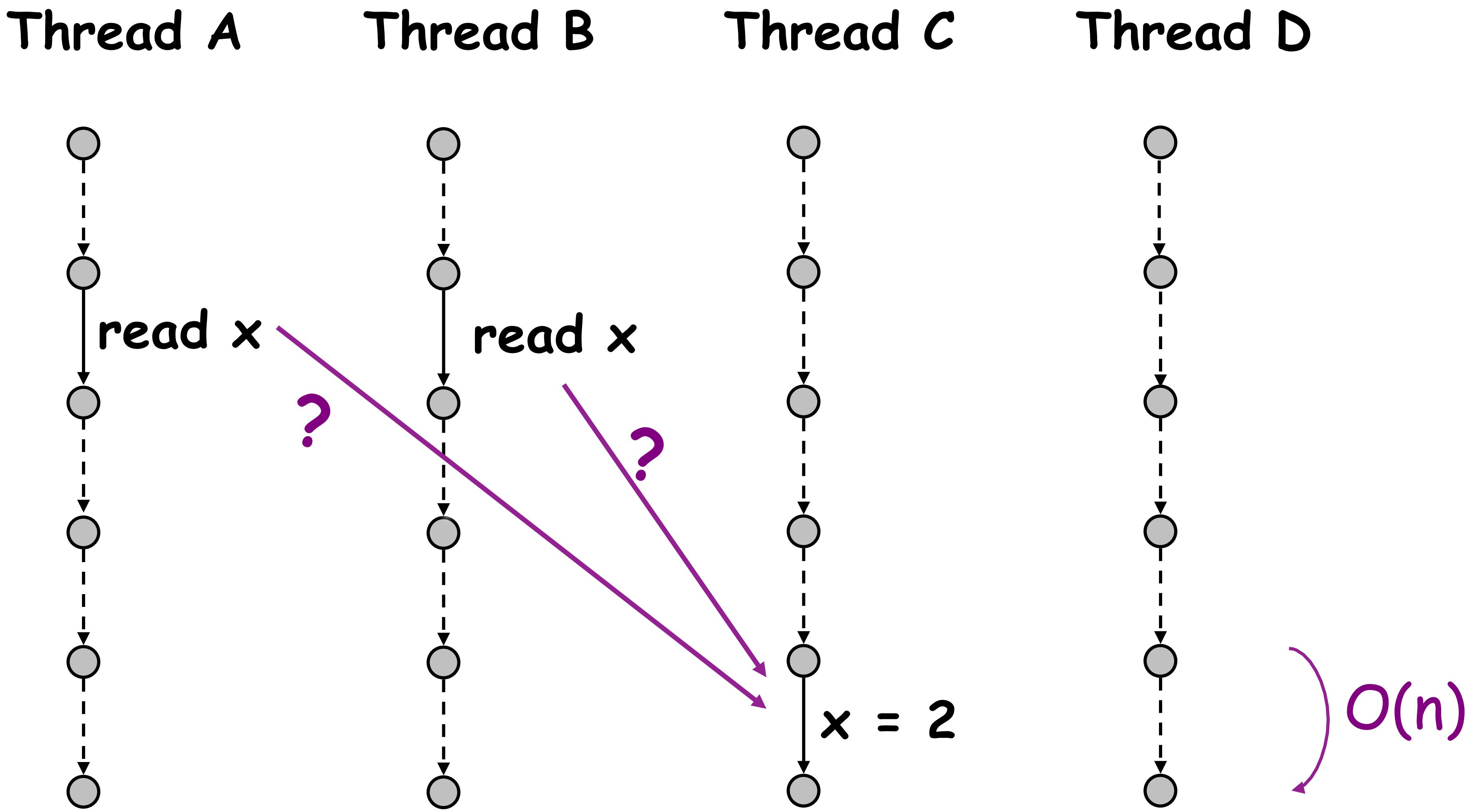


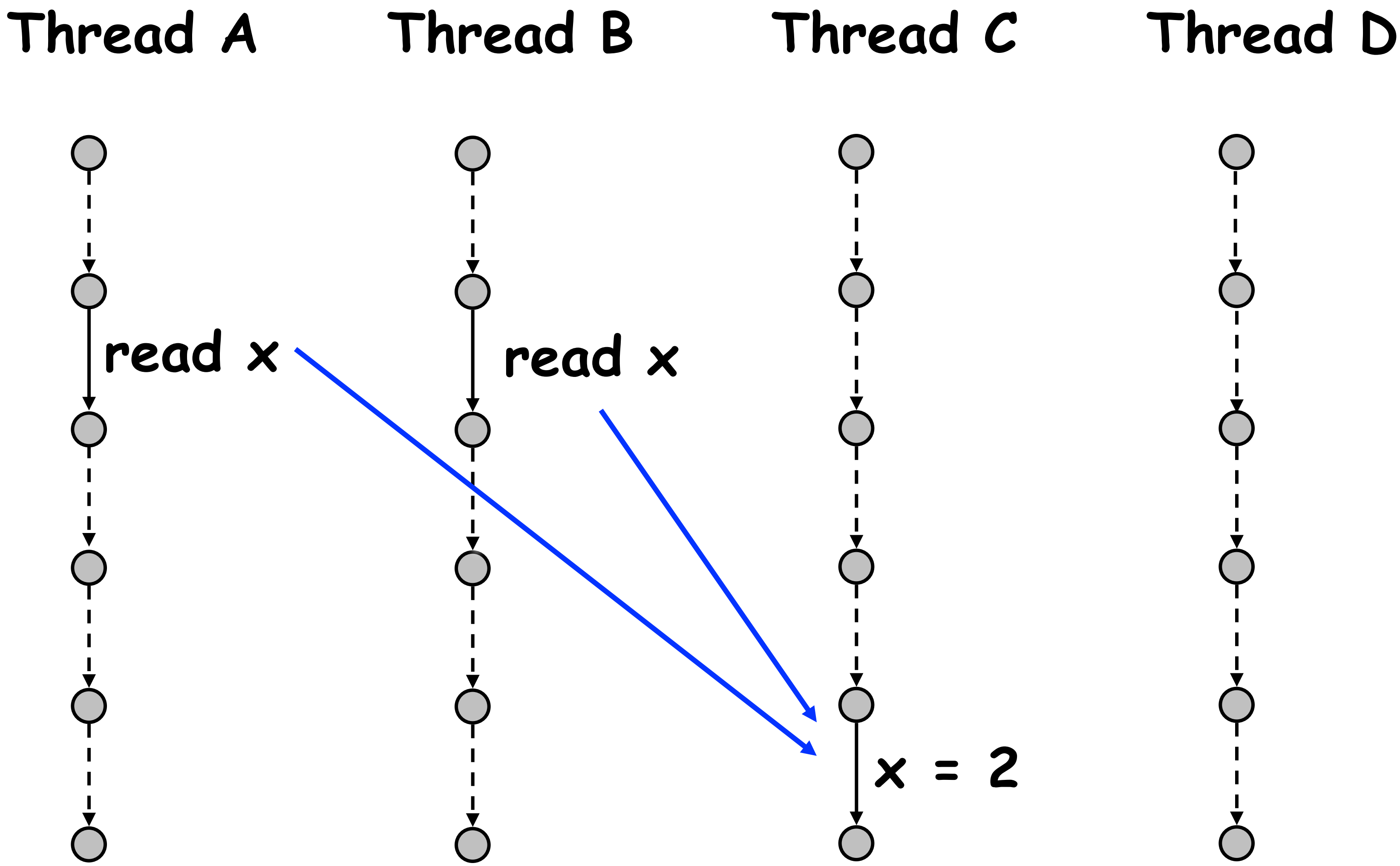
Most common case: thread-local, lock-protected, ...

Read-Write Races -- Unordered Reads

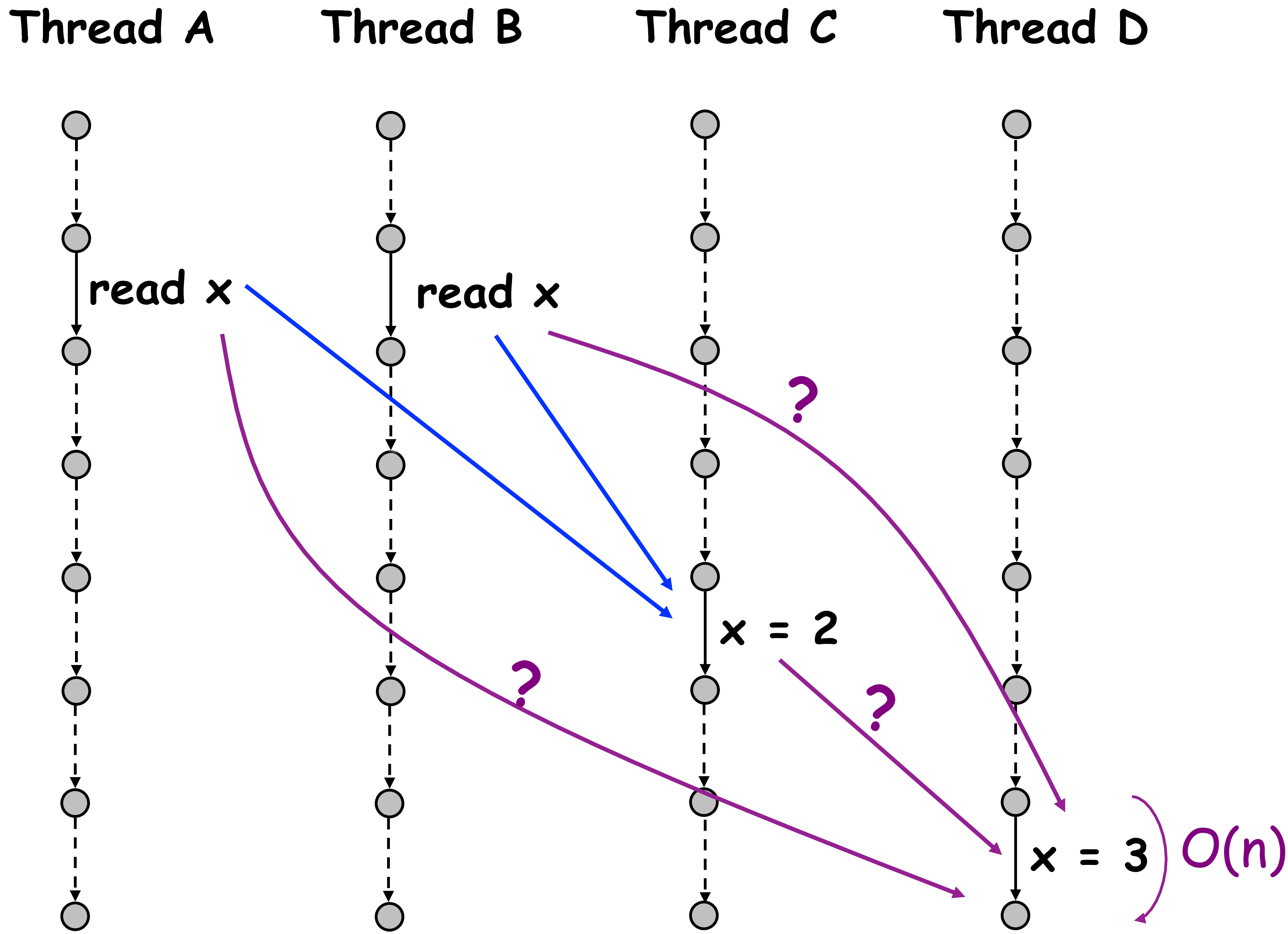



VC_A
VC_B
W_x
R_x
7 0
x = 0
7 0
fork
8 0
7 1
read x
8 0
7 1
read x
8 0
x = 2
-
-
7@A
-
7@A
-
7@A
1@B
7@A
8 1
O(1)
O(n)
O(n)
Read-Write Check: R_x ⊑ VC_A ?
8 1 ⊑ 8 0 ? No

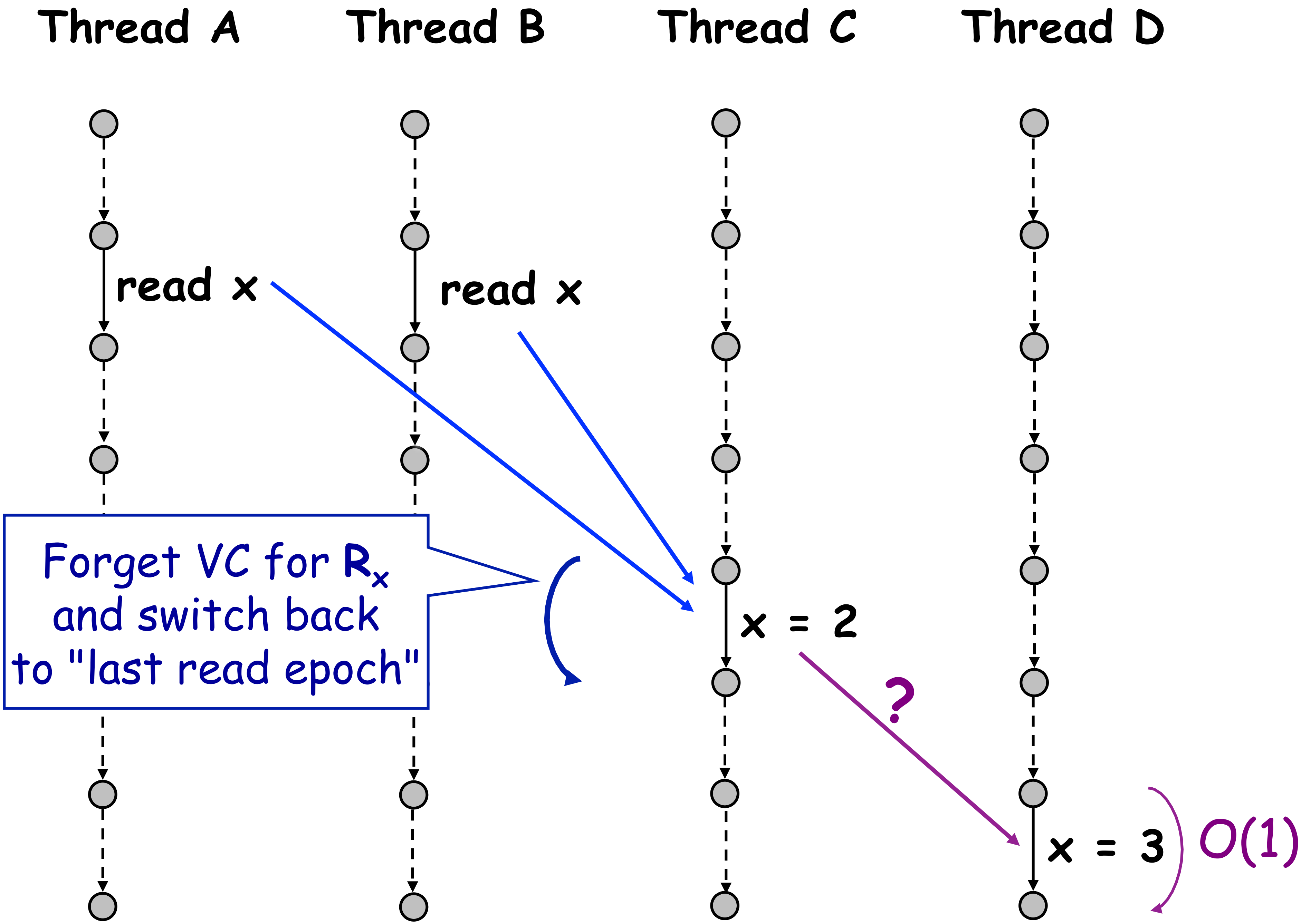

Thread A
Thread B
Thread C
Thread D
read x
read x
?
?
x = 2
O(n)



Thread A
Thread B
Thread C
Thread D
read x
read x
x = 2

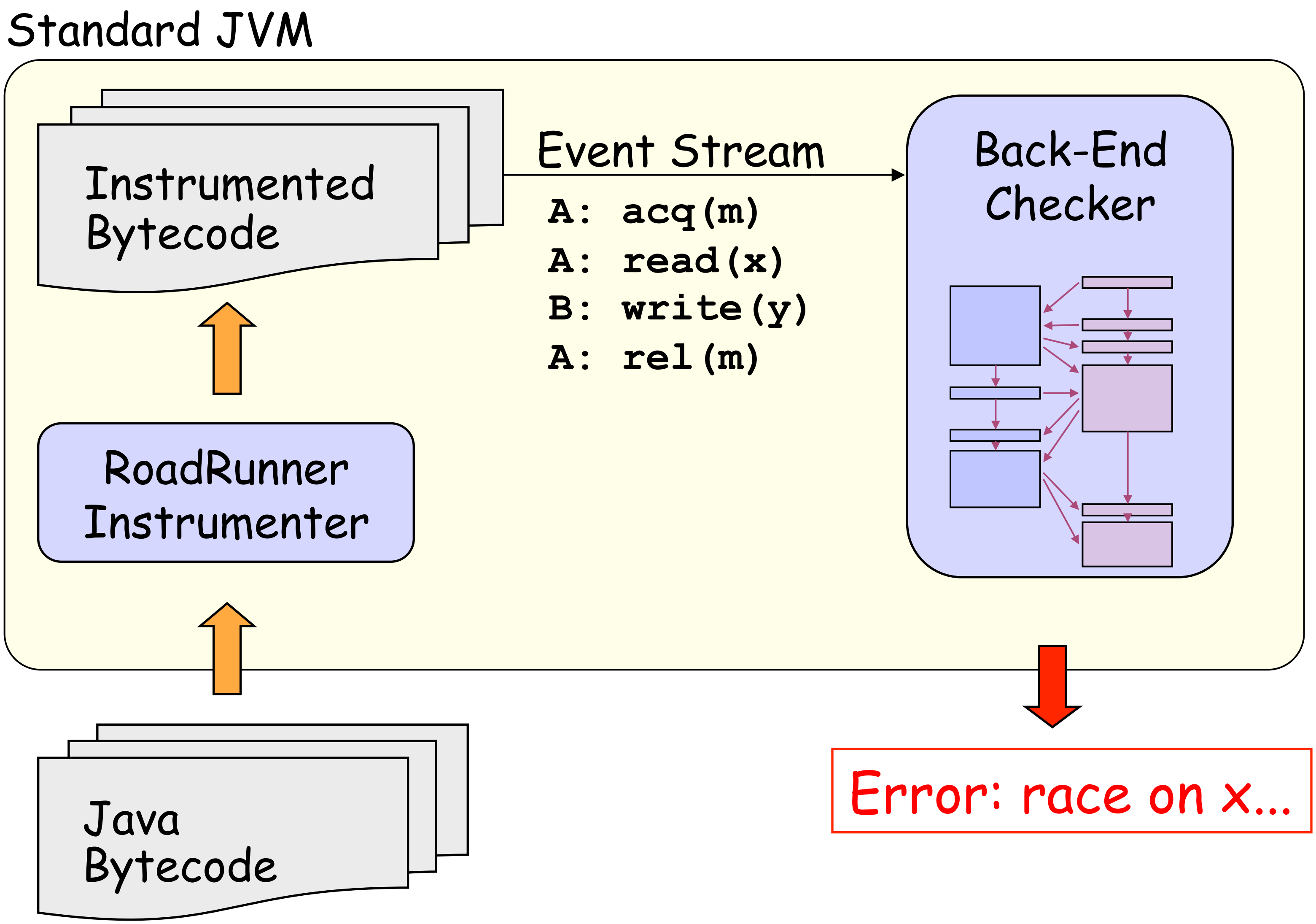


Thread A
Thread B
Thread C
Thread D
read x
read x
?
?
?
x = 2
x = 3
O(n)



Thread A
Thread B
Thread C
Thread D
read x
read x
Forget VC for R_x and switch back to "last read epoch"
x = 2
?
x = 3
O(1)

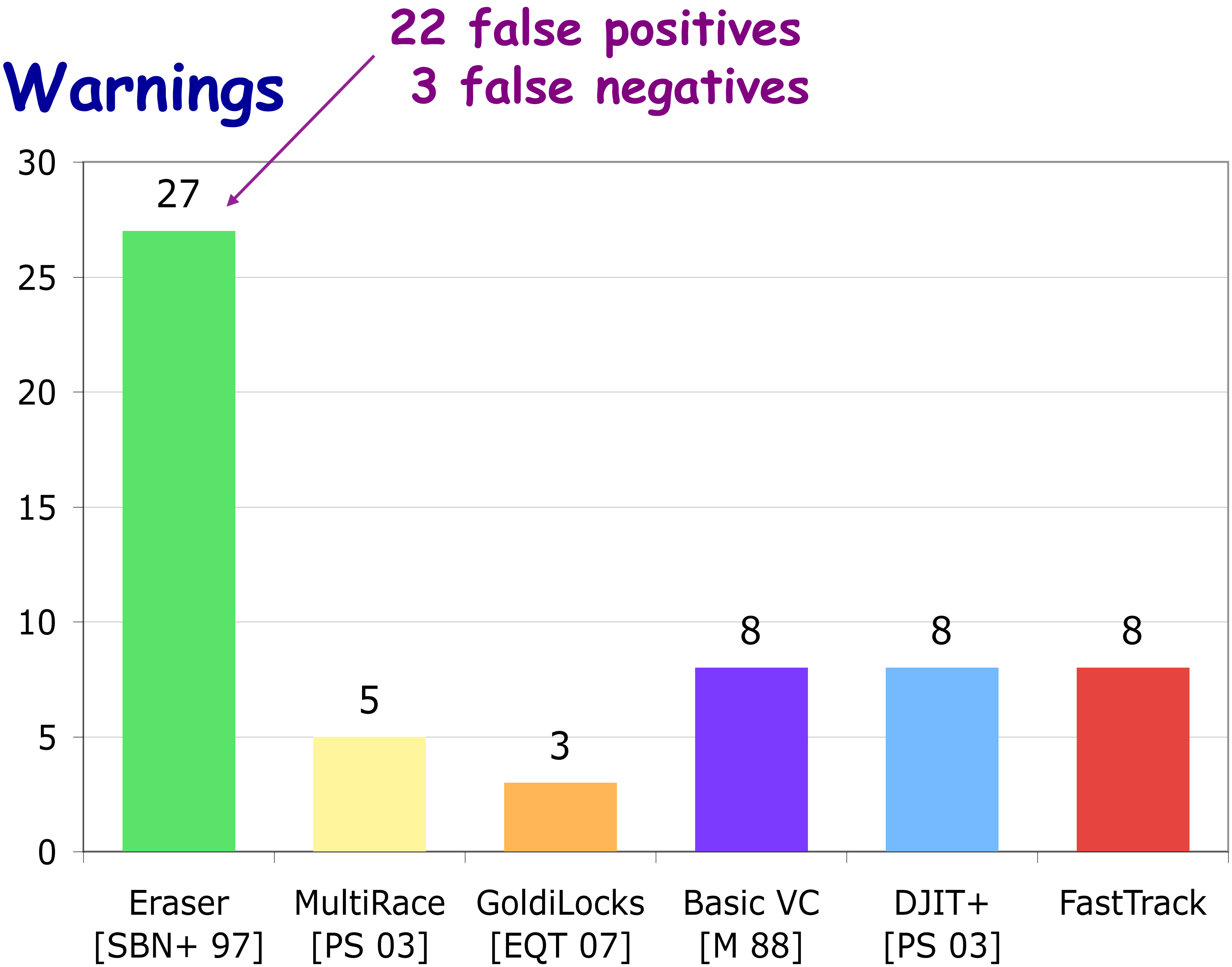
RoadRunner Architecture



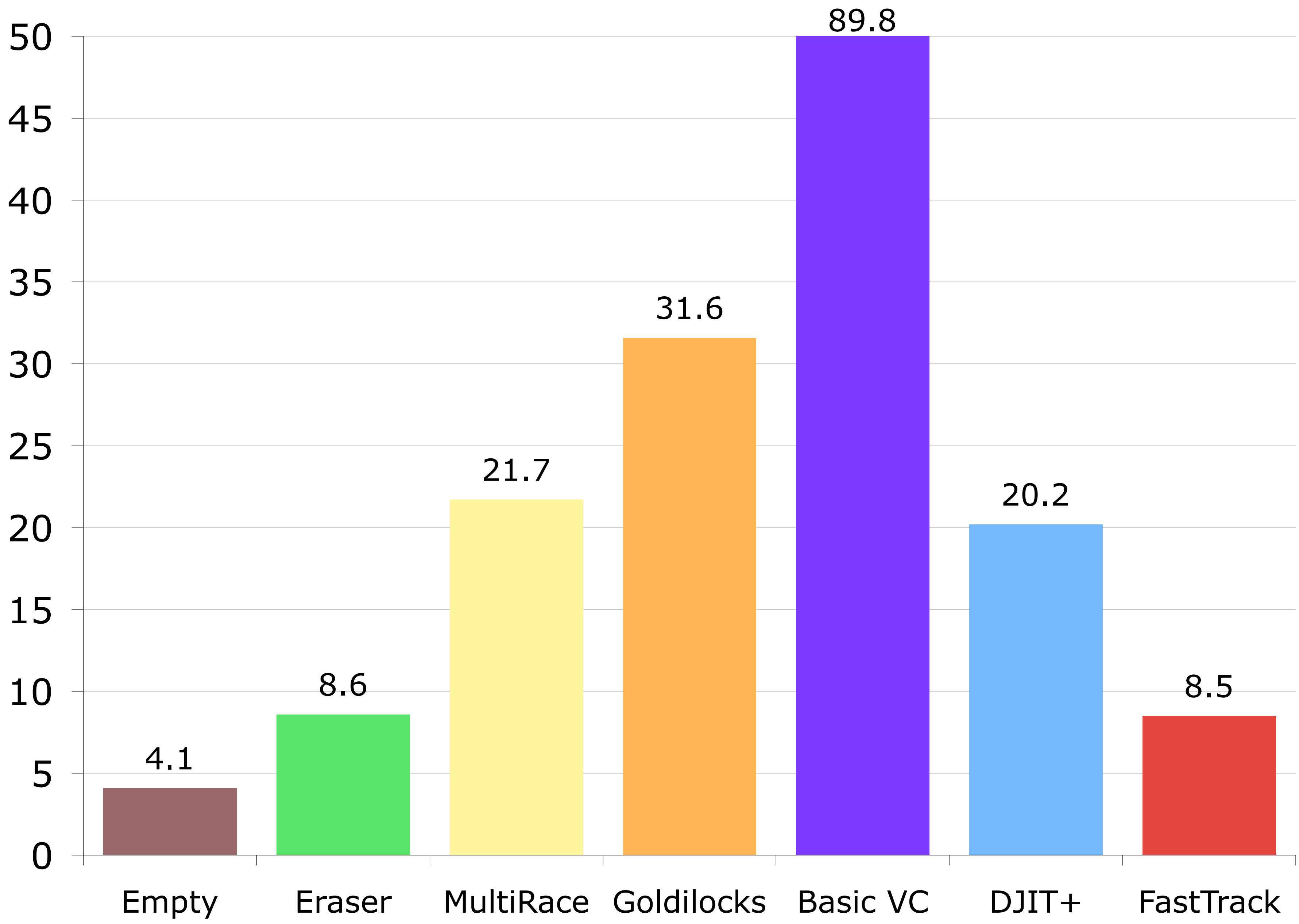
Validation

- Six race condition checkers
  - all use RoadRunner
  - share common components (eg, VectorClock)
  - profiled and optimized
- Further optimization opportunities
  - unsound extensions, dynamic escape analysis, static analysis, implement inside JVM, hardware support, ...
- 15 Benchmarks
  - 250 KLOC
  - locks, wait/notify, fork/join, barriers, ...

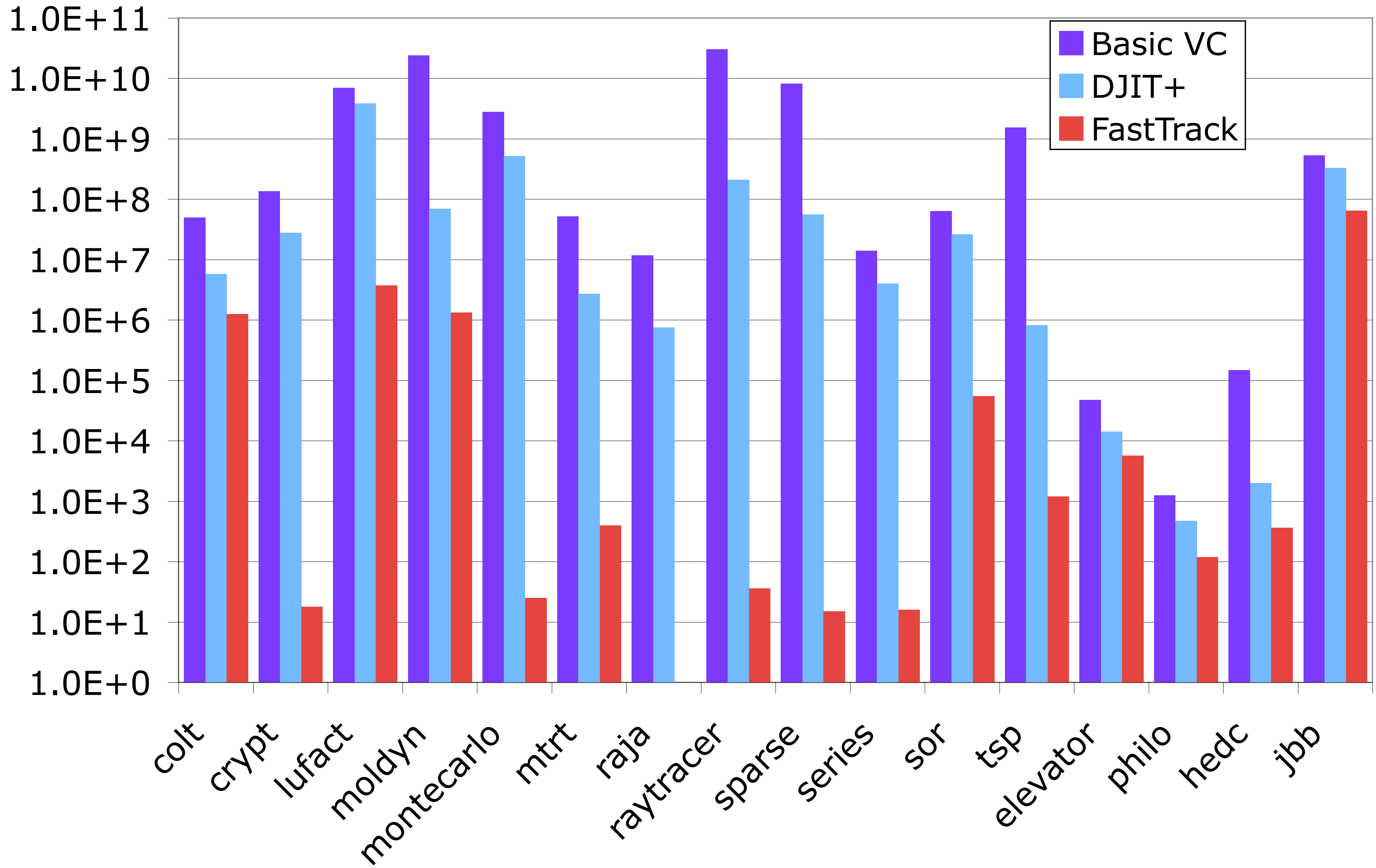
Warnings


22 false positives
3 false negatives
30
25
20
15
10
5
0
27
5
3
8
8
8
Eraser
[SBN+ 97]
MultiRace
[PS 03]
GoldiLocks
[EQT 07]
Basic VC
[M 88]
DJIT+
[PS 03]
FastTrack

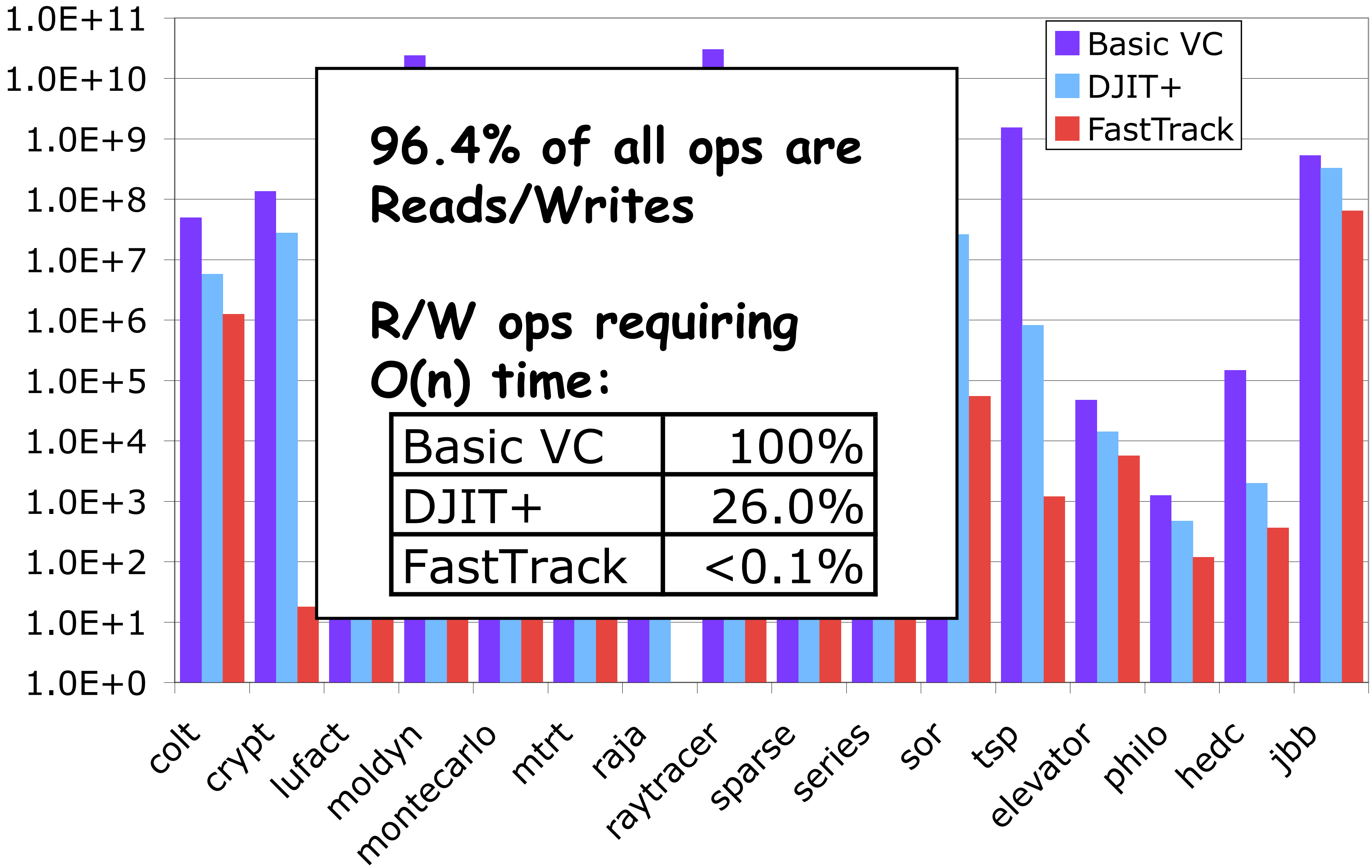
Slowdown (x Base Time)



O(n) Vector Clock Operations



O(n) Vector Clock Operations



96.4% of all ops are Reads/Writes

R/W ops requiring O(n) time:

Basic VC	100%
DJIT+	26.0%
FastTrack	<0.1%

Memory Usage

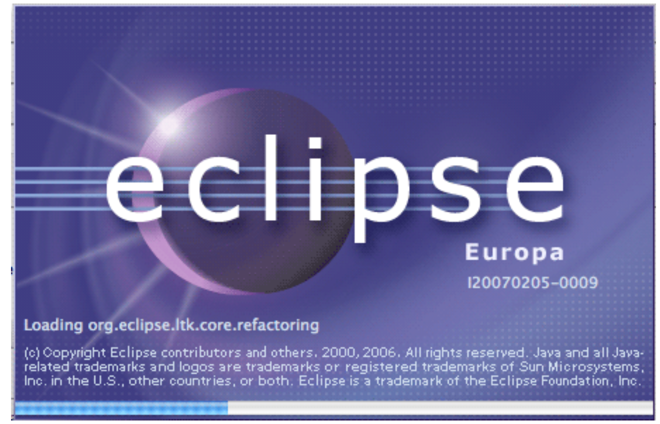
- FastTrack allocated ~200x fewer VCs

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

(Note: VCs for dead objects can be garbage collected)

- Improvements
  - accordion clocks [CB 01]
  - analysis granularity [PS 03, YRC 05] (see paper)

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

Beyond Detecting Race Conditions

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

```
class Point {
  double x, y;
  static Point p;

  Point() { x = 1.0; y = 1.0; }

  static Point get() {
    Point t = p;
    if (t != null) return t;
    synchronized (Point.class) {
      if (p==null) p = new Point();
      return p;
    }
  }

  static double slope() {
    return get().x / get().y;
  }

  public static void main(String[] args) {
    fork { System.out.println( slope() ); }
    fork { System.out.println( slope() ); }
  }
}
```

Thread 0	Thread 1	Thread 2
`p = null`		
`px = 0`		
`py = 0`		
`fork 1,2`		
		`read p // null`
		`acquire`
		`read p // null`
		`p = new Point`
		`px = 1`
		`py = 1`
		`release`
		`read px // get 1`
		`read py // get 1`
	`read p // non-null`	
	`read px // ?`	

Thread 0

```
p = null
px = 0
py = 0
fork 1,2
```

Thread 1

```
read p // non-null
read px // ?
```

Thread 2

```
read p // null
acquire
read p // null
p = new Point
px = 1
py = 1
release
read px // get 1
read py // get 1
```

Thread 0

```
p = null
px = 0
py = 0
fork 1,2
```

Thread 1

```
read p // non-null
read px // ?
```

Thread 2

```
read p // null
acquire
read p // null
p = new Point
px = 1
py = 1
release
read px // get 1
read py // get 1
```

- Race: can return either write (mm non-determinism)
- Typical JVM: mostly sequentially consistent
- Adversarial memory
  - use heuristics to return older stale values

Adversarial Memory

- Record history of all writes (plus VCs) to racy variables
- At read
  - determine all visible writes legal under JMM
  - heuristically pick one likely to crash target program
- Six heuristics:
  - Sequentially consistent: return last write
  - Oldest: return “most stale” value
  - Oldest-but-different: never return same val twice
    - if (p != null) p.draw()
  - Random, Random-but-different

Experimental Results

Program	Field	Erroneous Behavior Observation Rate (%)						Destructive Race?
		No Jumble	JUMBLE configurations					
			Sequentially Consistent	Oldest	Oldest but Different	Random	Random but Different	
Figure 1	x	**0**	**0**	**0**	**0**	**28**	**57**	**Yes**
Figure 2	p	0	0	0	0	0	0	No
	p.x	**0**	**0**	**60**	**52**	**32**	**30**	**Yes**
	p.y	**0**	**0**	**48**	**53**	**27**	**30**	**Yes**
hedc	Task.thread	**0**	**0**	**0**	**96**	**24**	**43**	**Yes**
	MetaSearchResult.results	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
	MetaSearchResult.completed	**0**	**0**	**33**	**36**	**25**	**26**	**Yes**
	MetaSearchResult.request	0	0	0	0	0	0	No
	Task.valid	0	0	0	0	0	0	No
jbb	Company.elapsed_time	**0**	**0**	**100**	**0**	**15**	**5**	**Yes**
	Company.mode	**0**	**0**	**100**	**100**	**95**	**98**	**Yes**
montecarlo	Universal.UNIVERSAL_DEBUG	0	0	0	0	0	0	No
mtrt	RayTracer.threadCount	0	0	0	0	0	0	No
raytracer	JGFRayTracerBench.checksum1	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
tsp	TspSolver.MinTourLen	0	0	100	100	100	100	QoS
sor	array index [0] and [1]	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
lufact	array index [0] and [1]	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**
moldyn	array index [0] and [1]	**0**	**0**	**100**	**100**	**100**	**100**	**Yes**