Detecting Data Races in Parallel Programs (Part 2)

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu

Detecting Data Races in Cilk Programs that use Locks

Guang-Ien Cheng, Mingdong Feng,

Charles Leiserson, Keith Randall,

Andrew Stark

Mutual Exclusion in Cilk: Locks

cilk_lock(L)

critical section

cilk_unlock(L)

Assumptions about Locking

- **Lock/unlock pair is contained in a single thread**
- **Holding a lock across a parallel control construct is forbidden**

Terminology

- **"Lock set" of an access: set of locks held when access is performed**
- **Lock set of several accesses: intersection of individual sets**

A Cilk Program with a Data Race

```
int x;
Cilk_lockvar A, B;

cilk void foo1() {
  Cilk_lock(&A);
  Cilk_lock(&B);
  x += 5;
  Cilk_unlock(&B);
  Cilk_unlock(&A);
}

cilk void foo2() {
  Cilk_lock(&A);
  x -= 3;
  Cilk_unlock(&A);
}

cilk void foo3() {
  Cilk_lock(&B);
  x++;
  Cilk_unlock(&B);
}

cilk int main() {
  Cilk_lock_init(&A);
  Cilk_lock_init(&B);
  x = 0;
  spawn foo1();
  spawn foo2();
  spawn foo3();
  sync;
  printf("%d", x);
}
```

- **Conflicting accesses: at least one is a WRITE**
- **No ordering by happens before <u>and</u> no common lock**

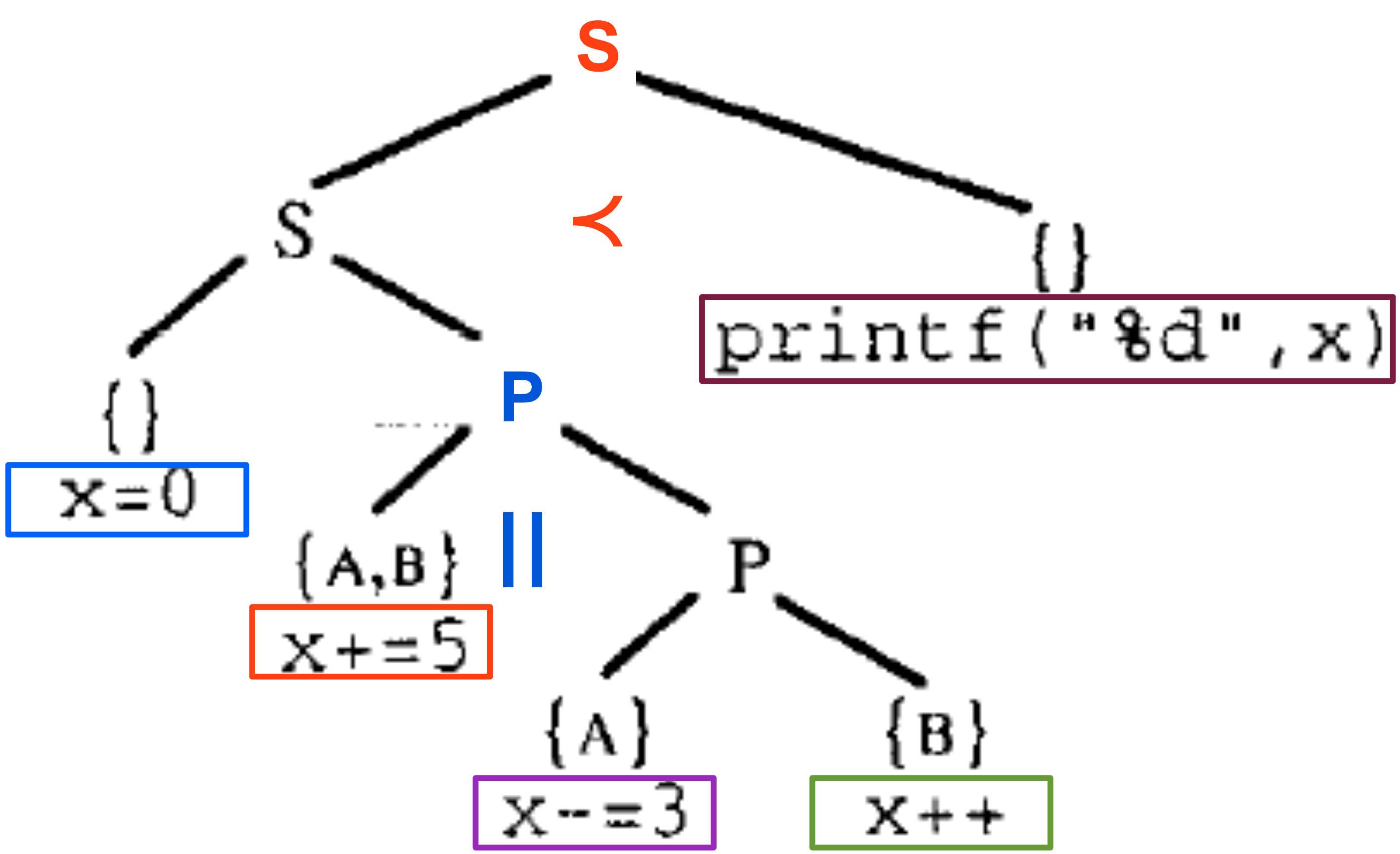
SP-Parse Tree

```
int x;
Cilk_lockvar A, B;

cilk void foo1() {
  Cilk_lock(&A);
  Cilk_lock(&B);
  x += 5;
  Cilk_unlock(&B);
  Cilk_unlock(&A);
}

cilk void foo2() {
  Cilk_lock(&A);
  x -= 3;
  Cilk_unlock(&A);
}
cilk void foo3() {
  Cilk_lock(&B);
  x++;
  Cilk_unlock(&B);
}

cilk int main() {
  Cilk_lock_init(&A);
  Cilk_lock_init(&B);
  x = 0;
  spawn foo1();
  spawn foo2();
  spawn foo3();
  sync;
  printf("%d", x);
}
```



Apparent vs. Feasible Races

initial condition: x = 0

<u>T1</u>
z = 1
lock(L)
x = 2
unlock(L)

<u>T2</u>
lock(L)
y = x
unlock(L)
if (y == 2) ... = z

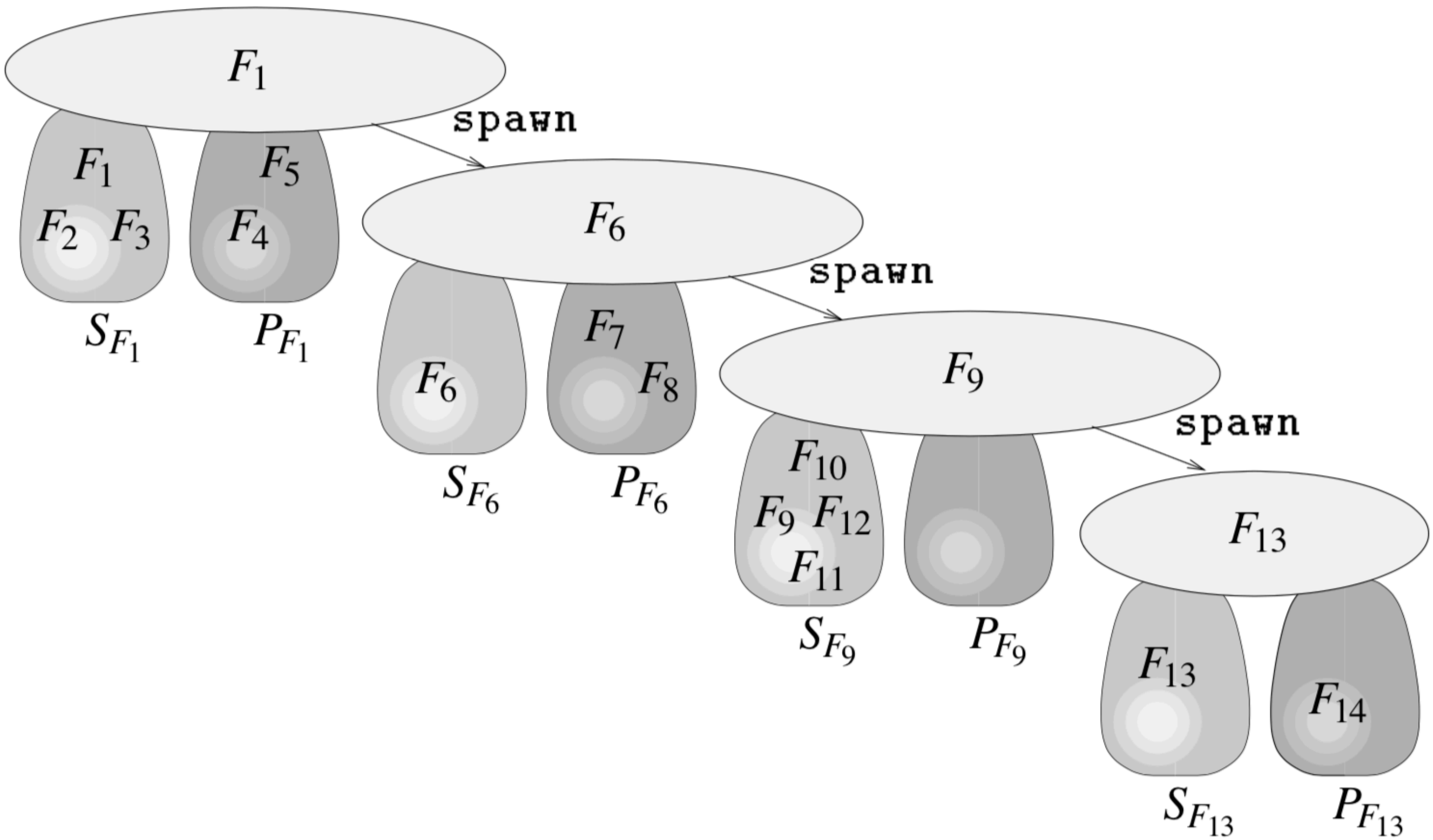
Detecting Races in Cilk

- Data race if the lock set for two parallel accesses to the same location is empty and at least one is a WRITE
- Problem: "At least one is a WRITE" is cumbersome
- Simplification
  - — introduce a fake R-LOCK
    - – as if implicitly acquired and held for the duration of a read
    - – for race detector: R-LOCK behaves as regular lock
  - — if the lock set of two parallel accesses to the same location is empty, then a data race exists

Two Algorithms for Race Detection

- **ALL-SETS - general serial race detection algorithm**
- **BRELLY - faster serial race detection algorithm limited to "umbrella locking discipline"**

ALL-SETS uses SP-Bags Representation



Use SP-Bags to determine concurrency relationship

ALL-SETS Protocol

ACCESS(l) in thread e with lock set H

for each $\langle e', H' \rangle \in lockers[l]$
  do if $e' \parallel e$ and $H' \cap H = \{\}$
    then declare a data race

redundant $\leftarrow$ FALSE

for each $\langle e', H' \rangle \in lockers[l]$
  do if $e' \prec e$ and $H' \supseteq H$
    then $lockers[l] \leftarrow lockers[l] - \{\langle e', H' \rangle\}$

    if $e' \parallel e$ and $H' \subseteq H$
      then *redundant* $\leftarrow$ TRUE

if *redundant* = FALSE
  then $lockers[l] \leftarrow lockers[l] \cup \{\langle e, H \rangle\}$

```
Cilk_lock(&A); Cilk_lock(&B);
READ(l)              {A,B,R-LOCK}
Cilk_unlock(&B); Cilk_unlock(&A);
Cilk_lock(&B); Cilk_lock(&C);
WRITE(l)             {B,C}
Cilk_unlock(&C); Cilk_unlock(&B);
```

check for race:
  parallel accesses
  non-overlapping lock sets

prune redundant lock sets
  precedes & larger set

add new lock set if not redundant

lockers(L): set of tuples <thread, lock set>

set of locks held by previous access to L by thread

ALL-SETS Detects Races

Detects a race in a Cilk execution based on a given input if and only if a data race exists in the execution.

- if: any race reported between accesses by ALL-SETS meets the condition for a race: no common lock
- only if: if a race between accesses A and C exists in the computation, a race will be reported
  - —if lock set for A was not added to lockers, there must be another parallel access with a smaller lock set. a race will be reported.
  - —what if there was an intervening non-racing access B that caused a lock set for A to be removed from the lock set?
    - – there can be no such access B
      - B must have a larger lock set if it doesn't race
      - a lock set will be removed only if its lock set is larger than B's
      - thus, the A won't have its lock set removed

ALL-SETS Properties

- Cilk program executes in time T
- Uses V variables
- Uses a total of n locks; no more than k simultaneously
- Let L = max number of distinct lock sets used for any location

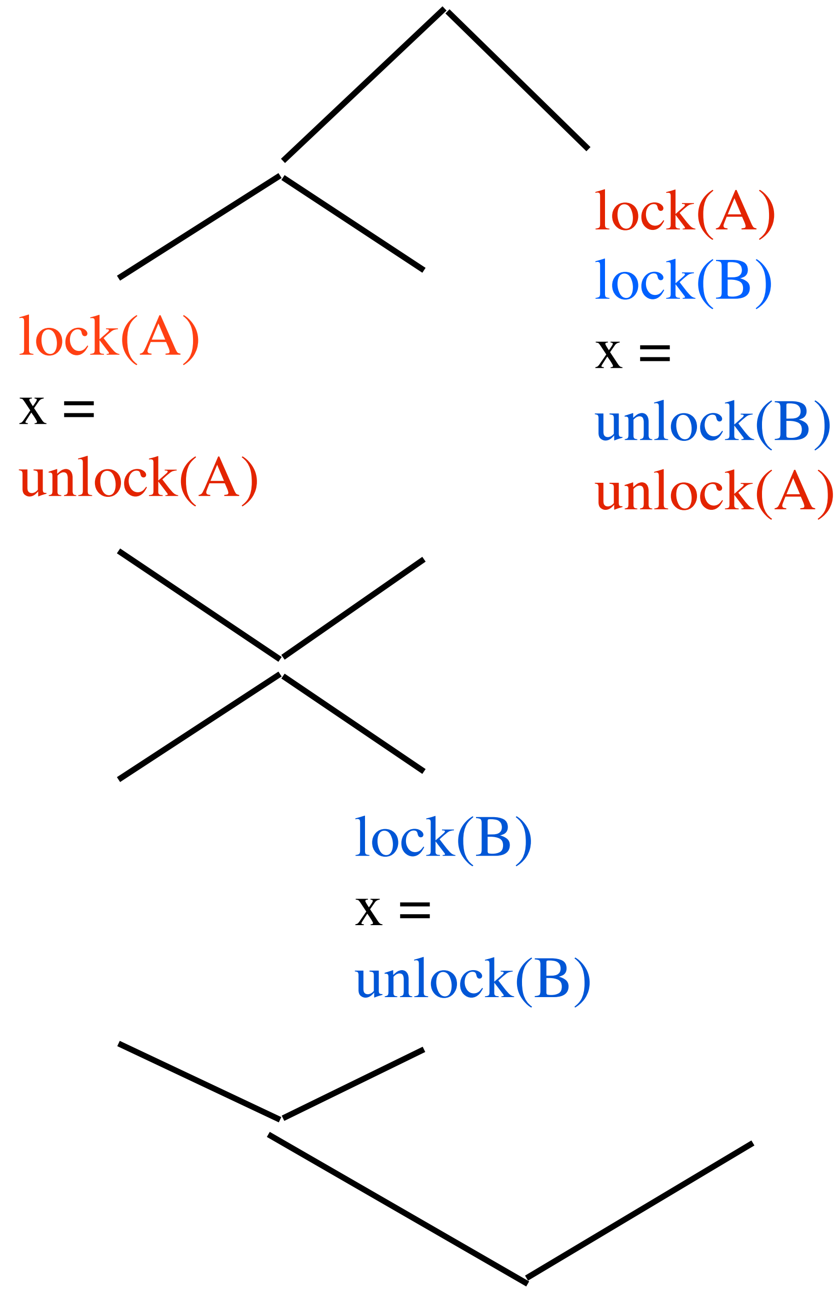
- Time: O(TL(k + α(V,V))
  - — loose upper bound for L: L ≤ sum of n choose i, i = 0, k = $O(n^k/k!)$
  - — at most 2L series/parallel tests (lines 2, 6) at cost of O(α(V,V))
  - — lock set comparisons take at most O(k) time
- Space: O(kLV)
  - — each lock set takes at most k space

ALL-SETS vs. BRELLY

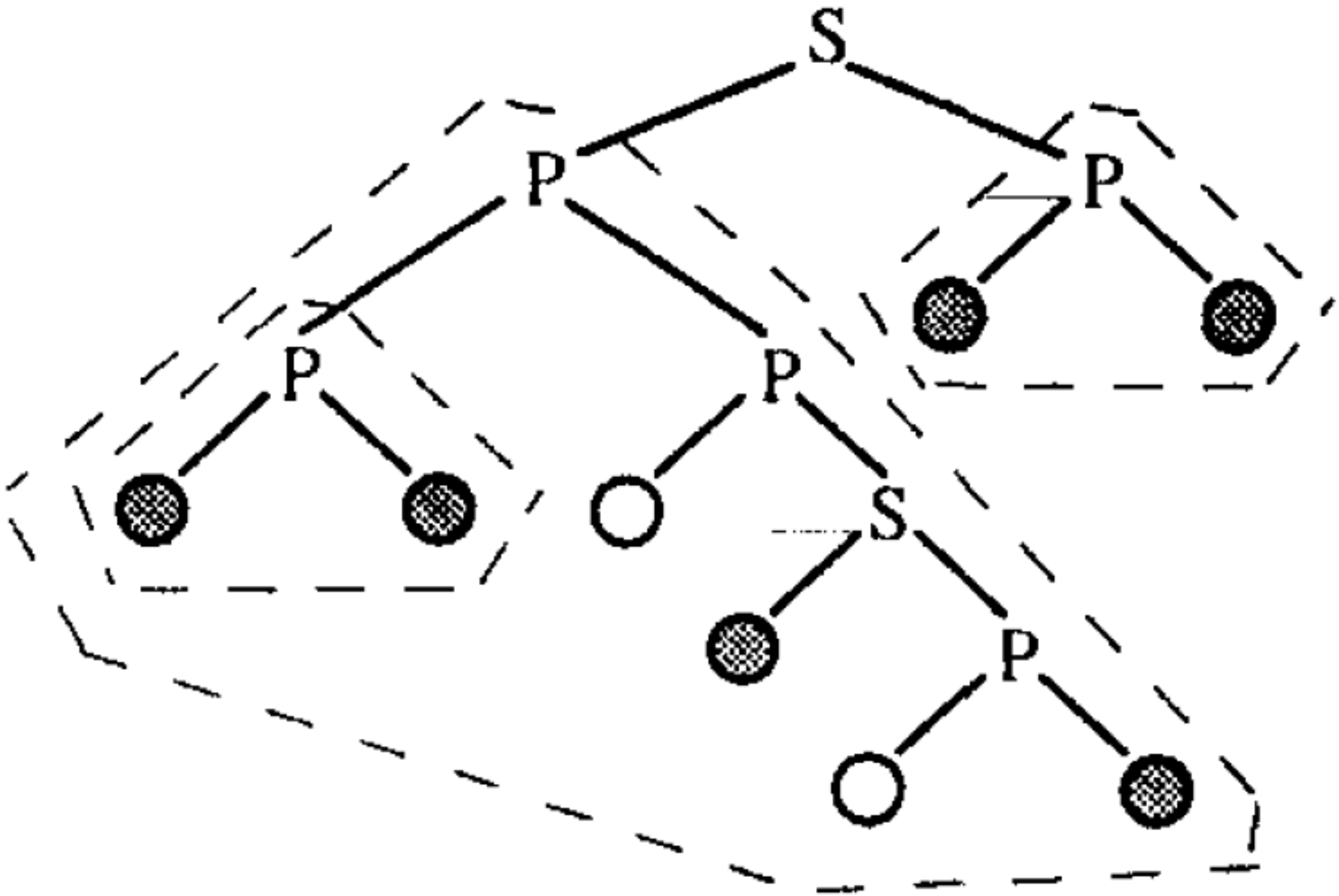
- **ALL-SETS detects data races directly**
  - — **but at asymptotically high cost: factor of n^k slower than SP-bags protocol**
- **Umbrella locking discipline**
  - — **requires each that each location be protected by <u>the same lock</u> within every parallel subcomputation**
  - — **threads in series may use different locks (or none)**
- **BRELLY only detects violations of the "umbrella" locking discipline, which precludes races**
  - — **more restrictive locking discipline than ALL-SETS requires**

What's Not in the Umbrella Discipline?

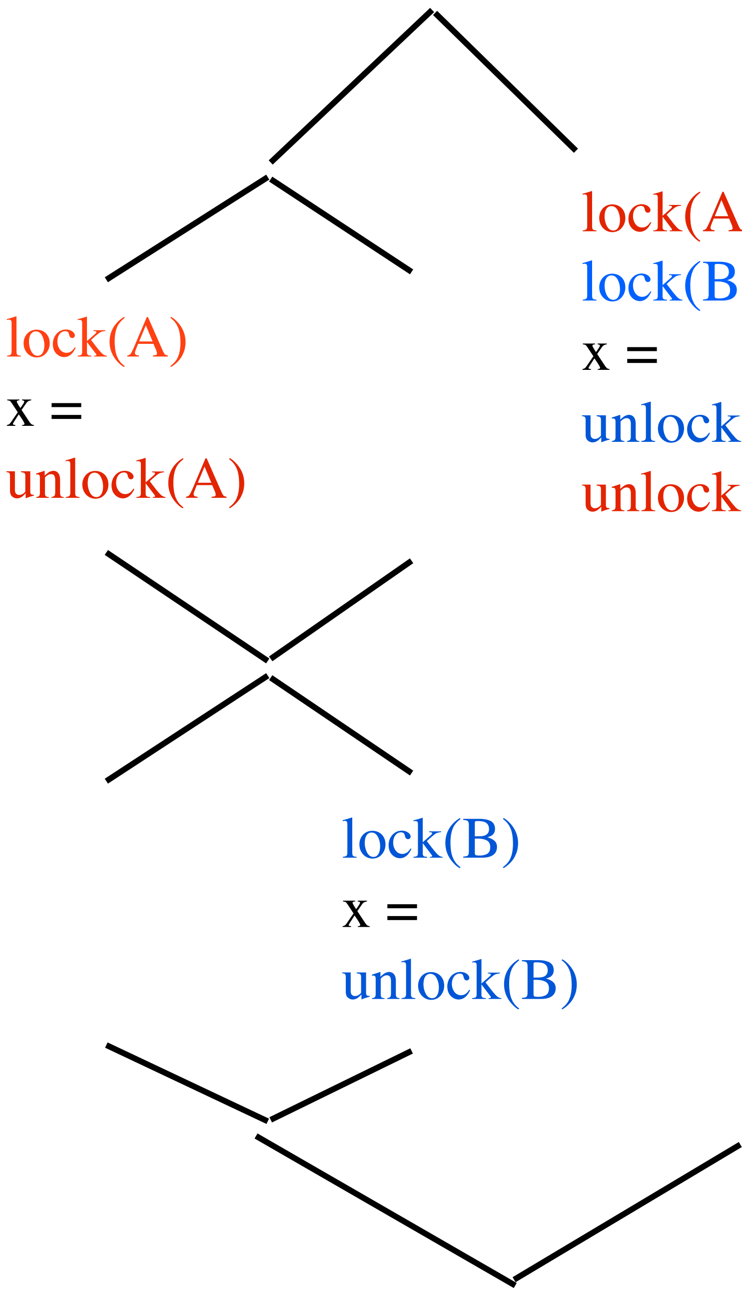
- Umbrella discipline requires that all sections in a parallel subcomputation use the <u>same</u> lock for a variable
- One thread uses A&B
- Two serial computations in parallel with first use
  - only A
  - only B

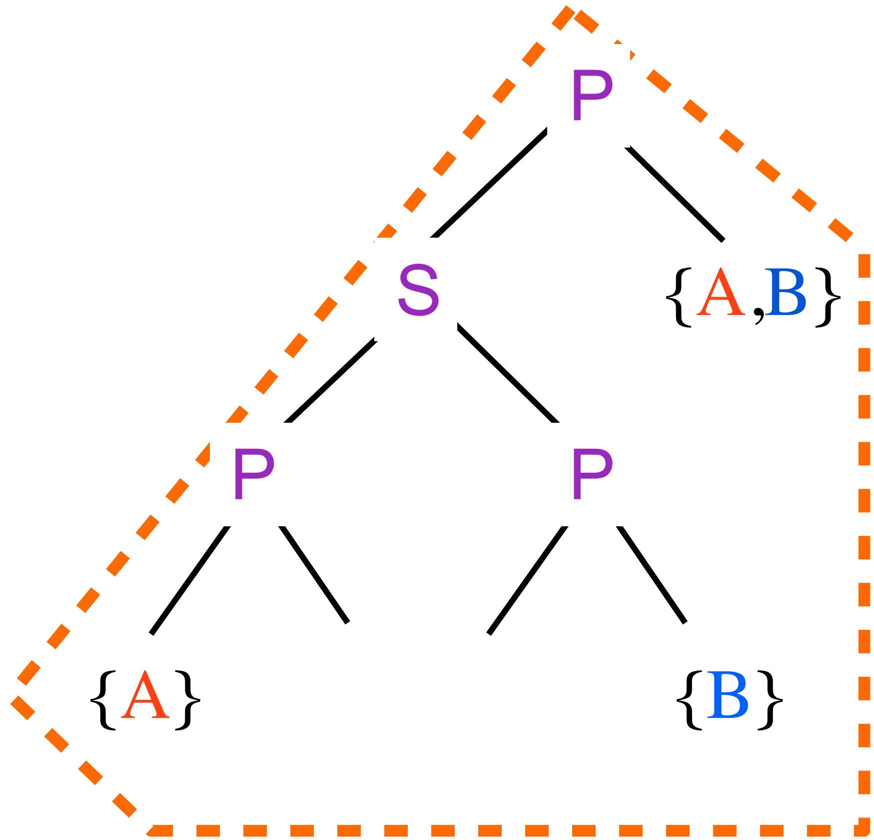


Umbrellas in SP-Parse Tree



Understanding our Example with its SP-Parse





Umbrellas and Races

A Cilk computation with a data race violates the umbrella discipline

- **Any two threads involved in a race must have a P-node as their LCA in the SP-Parse**
- **The LCA P-node is the root of an unprotected umbrella**
  - **—both threads access the same location**
  - **—their lock sets are disjoint**

BRELLY Protocol

Simplication: unlike ALL-SETS, keep only single lock set per location

ACCESS(l) in thread e with lock set H

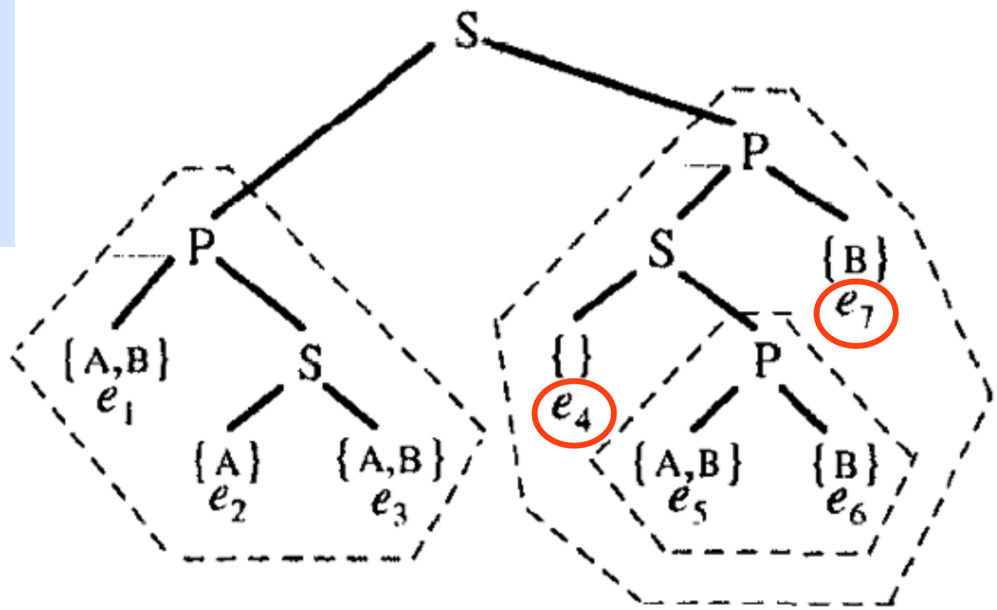
```
if accessor[l] ≺ e
   then ▷ serial access
        locks[l] ← H, leaving nonlocker[h] with its old
          nonlocker if it was already in locks[l] but
          setting nonlocker[h] ← accessor[l] otherwise
        for each lock h ∈ locks[l]
           do alive[h] ← TRUE
        accessor[l] ← e
   else ▷ parallel access
        for each lock h ∈ locks[l] − H
           do if alive[h] = TRUE
                 then alive[h] ← FALSE
                      nonlocker[h] ← e
        for each lock h ∈ locks[l] ∩ H
           do if alive[h] = TRUE and nonlocker[h] ∥ e
                 then alive[h] ← FALSE
        if no locks in locks[l] are alive (or locks[l] = {})
          then report violation on l involving
                 e and accessor[l]
                 for each lock h ∈ H ∩ locks[l]
                    do report access to l without h
                          by nonlocker[h]
```

Tag lock h in the lock set for L with

- nonlocker[h] - a thread accessing L without holding h
- alive[h] - whether h should be considered as belonging to the umbrella
  - kill h rather than removing from lock set to improve precision of race reports

BRELLY at Work

<u>Notation</u>
A(x) : x is non-locker of A
$\underline{A}$: A is not alive



- **e_7 finds itself in parallel with non-locker e_4 for B**
- **kills lock B leaving no live locks**
- **causes a data race to be detected**

thread	*accessor*[*l*]	*locks*[*l*]	access type
initial	e_0	{ }	
e_1	e_1	$\{A(e_0), B(e_0)\}$	serial
e_2	e_1	$\{A(e_0), \underline{B}(e_2)\}$	parallel
e_3	e_1	$\{A(e_0), \underline{B}(e_2)\}$	parallel
e_4	e_4	{ }	serial
e_5	e_5	$\{A(e_4), B(e_4)\}$	serial
e_6	e_5	$\{\underline{A}(e_6), B(e_4)\}$	parallel
e_7	e_5	$\{\underline{A}(e_6), \underline{B}(e_4)\}$	parallel

BRELLY Properties

- Cilk program executes in time T
- Uses V variables
- Uses a total of n locks; no more than k simultaneously

- Time: $O(kT\alpha(V,V))$
  - — tests if nonlocker[h] || e dominate running time
  - — at most k series/parallel tests at cost of $O(\alpha(V,V))$ each
- Space: $O(kV)$
  - — at most k locks per variable

Cilkscreen

- Detects and reports data races when program terminates
  - finds all data races even those by third-party or system libraries

```
// code with a data race
int sum = 0;
cilk_for (int i = 0; i < n; i++) {
      sum += a[i];
}
```

- Does not report determinacy races
  - e.g. two concurrent strands use a lock to access a queue
    - enqueue & dequeue operations could occur in different order potentially leads to different result

Race Detection Strategies in Cilkscreen

- Lock covers
  - two conflicting accesses to a variable don't race if some lock L is held while each of the accesses is performed by a strand
- Happens-before
  - two conflicting accesses do not race if one must happen before the other
    - access A is by a strand X, which precedes the spawn of strand Y which performs access B
    - access A is performed by strand X, which precedes a sync that is an ancestor of strand Y

Cilkscreen Race Example

```
#include <stdio.h>
#include <cilk++/cilk_mutex.h>

int sum = 0;
cilk::mutex m;

#ifdef SYNCH
#define LOCK m.lock()
#define UNLOCK m.unlock()
#else
#define LOCK
#define UNLOCK
#endif
```

```
void do_accum(int l, int u)
{
        if (u == l) { LOCK; sum += l; UNLOCK; }
        else {
          int mid = (u+l)/2;
          cilk_spawn do_accum(l, mid);
          do_accum(mid+1, u);
        }
}
int cilk_main()
{
        do_accum(0, 1000);
        printf("sum = %d\n", sum);

        int ssum = 0;
        for (int i = 0; i <= 1000; i++) ssum +=i;
        printf("serial sum = %d\n", ssum);
}
```

Cilkscreen Limitations

- Only detects races between Cilk++ strands
  - depends upon their strict fork/join paradigm
- Only detects races that occur given the input provided
  - does not prove the absence of races for other inputs
  - choose your testing inputs carefully!
- Cilkscreen runs serially, 15-30x slower
- Cilkscreen increases the memory footprint of an application
  - could cause an error if too large
- If you build your program with debug information, cilkscreen will associate races with source line numbers

Cilkscreen Output

Race on location 0x6033c0 between

/users/johnmc/tests/race.cilk:17: _Z8do_accumii+0x31 (eip=0x40167d)

and

/users/johnmc/tests/race.cilk:17: _Z8do_accumii+0x31 (eip=0x40167d)

/users/johnmc/tests/race.cilk:21: _Z8do_accumii+0x6a (eip=0x4016b6) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

/users/johnmc/tests/race.cilk:20: _Z8do_accumii+0x5c (eip=0x4016a8) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

/users/johnmc/tests/race.cilk:20: _Z8do_accumii+0x5c (eip=0x4016a8) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

/users/johnmc/tests/race.cilk:20: _Z8do_accumii+0x5c (eip=0x4016a8) called from here

/users/johnmc/tests/race.cilk:20: __cilk_spawn_do_accum_000+0x79 (eip=0x40161d) called from here

...

SigRace: Signature-based Race Detection

Abdullah Muzahid, Dario Suarez,

Shanxiang Qi, Josep Torrellas

The Big Picture

- People like shared-memory models for parallel programming
- Data races are a significant problem
  — most people don't write programs in languages like Ct or NESL
- Software-only data race detection is slow
  — perhaps as much as 50x
- Every 18 months: 2x transistors on a chip

Hardware Support for Race Detection

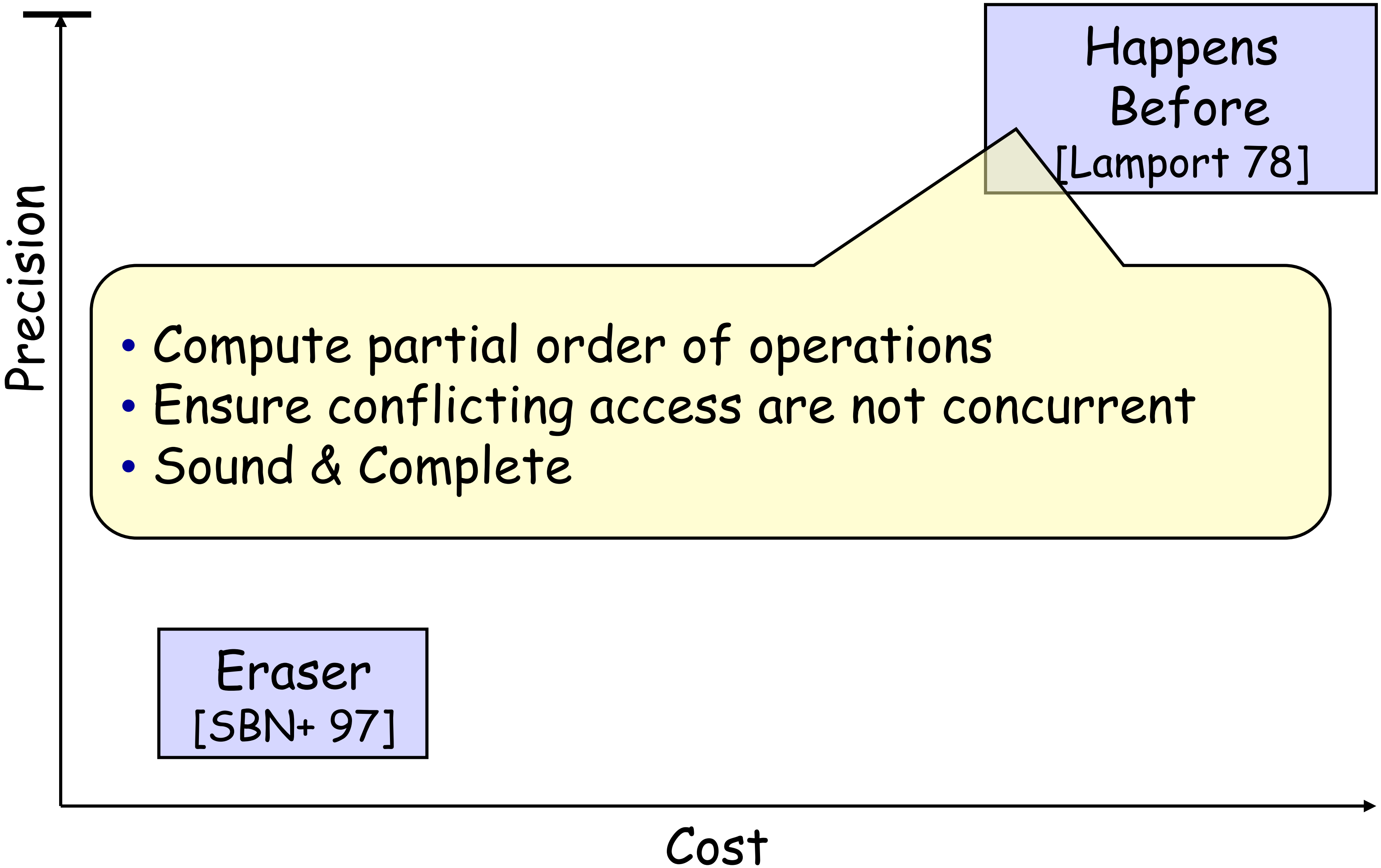
- Monitor accesses in hardware and detect races
- Typical approach
  - tag data in caches with timestamps as accesses occur
  - piggyback tags & race detection on cache coherence protocol
    - invalidation, external read of a dirty line
- Specific approaches
  - happened-before (ReEnact, CORD, Min & Choi)
  - locksets (HARD)
- SigRace approach
  - don’t require changes to L1 cache!
  - don’t change the coherence protocol

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

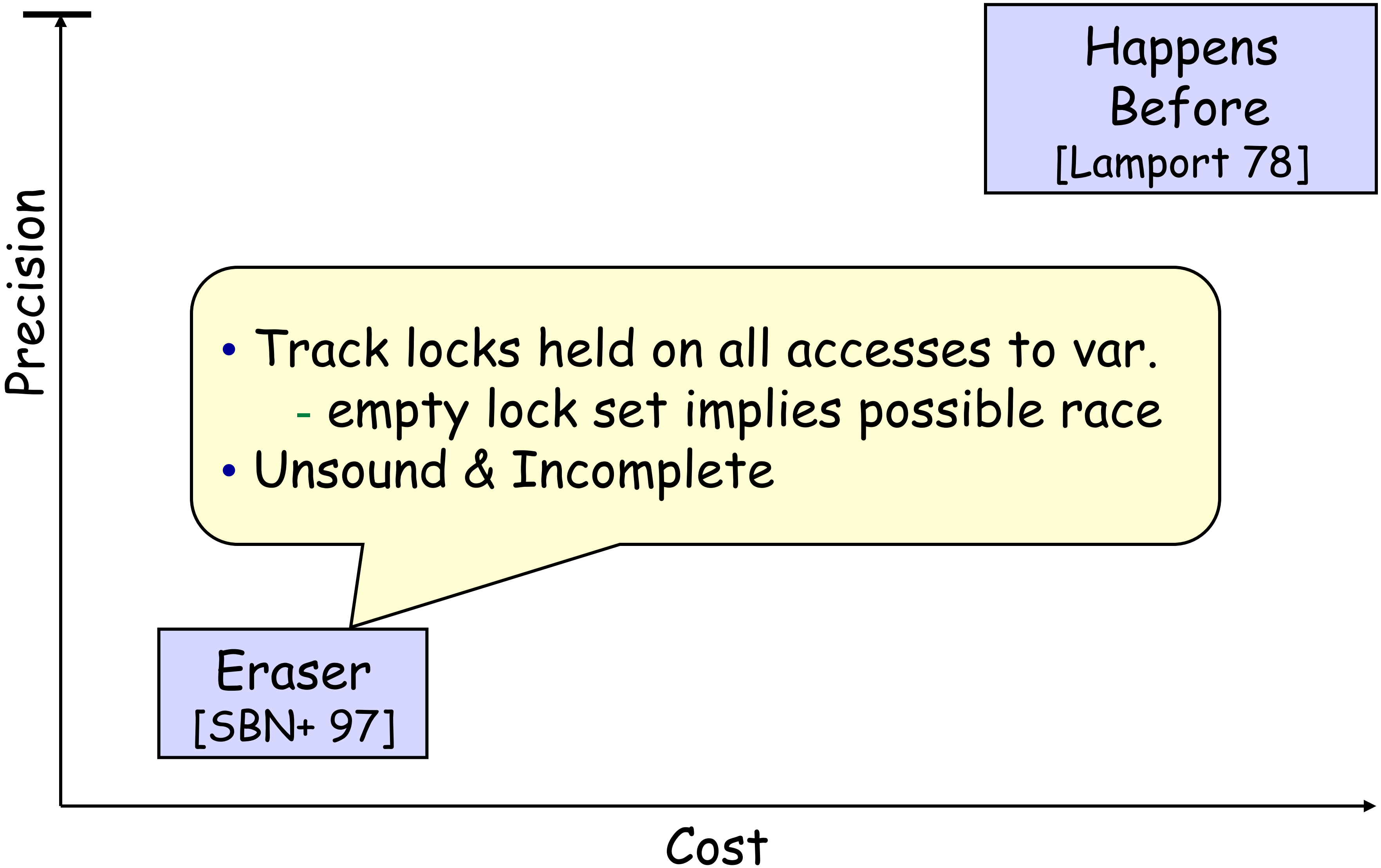
Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College

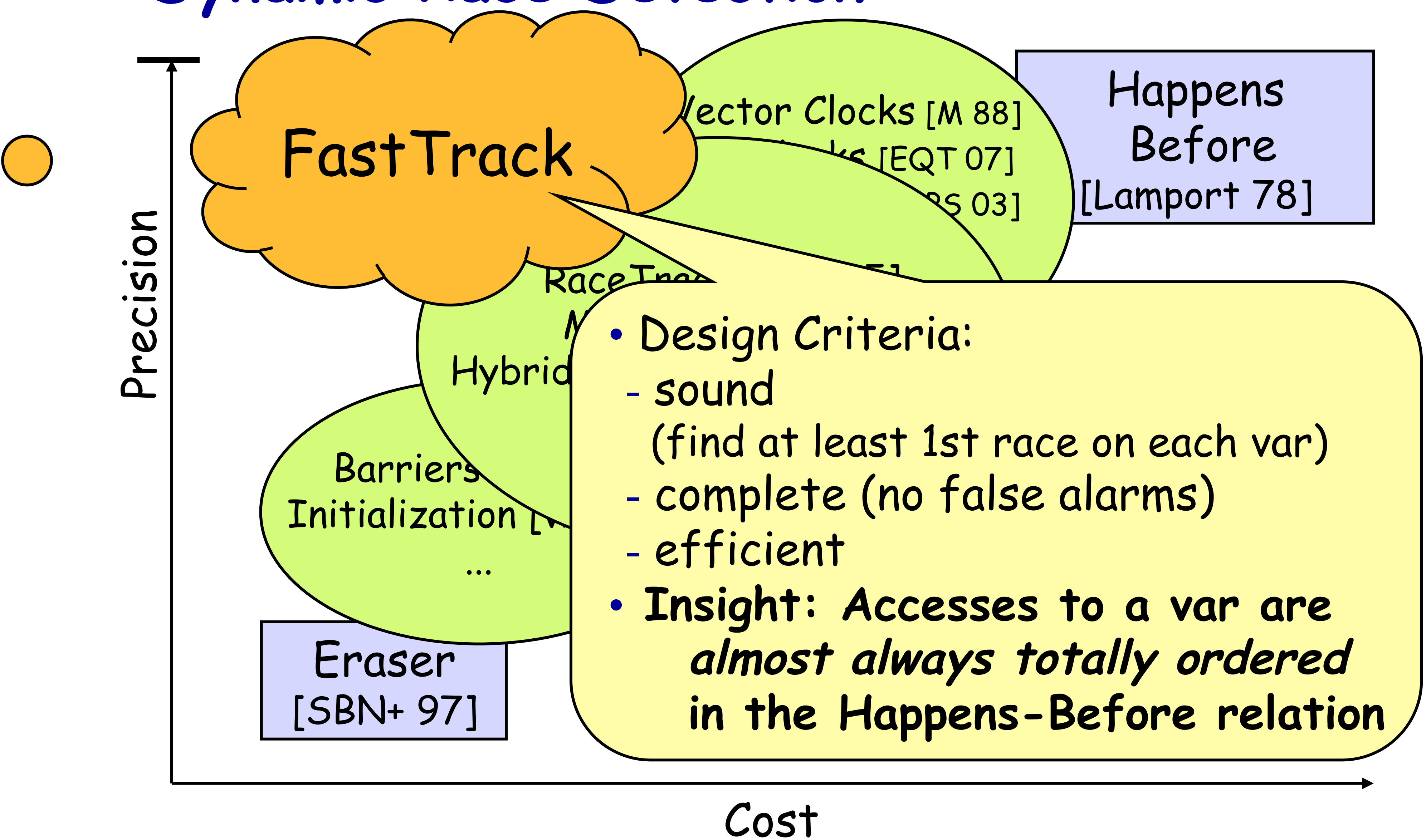
Dynamic Race Detection



Dynamic Race Detection



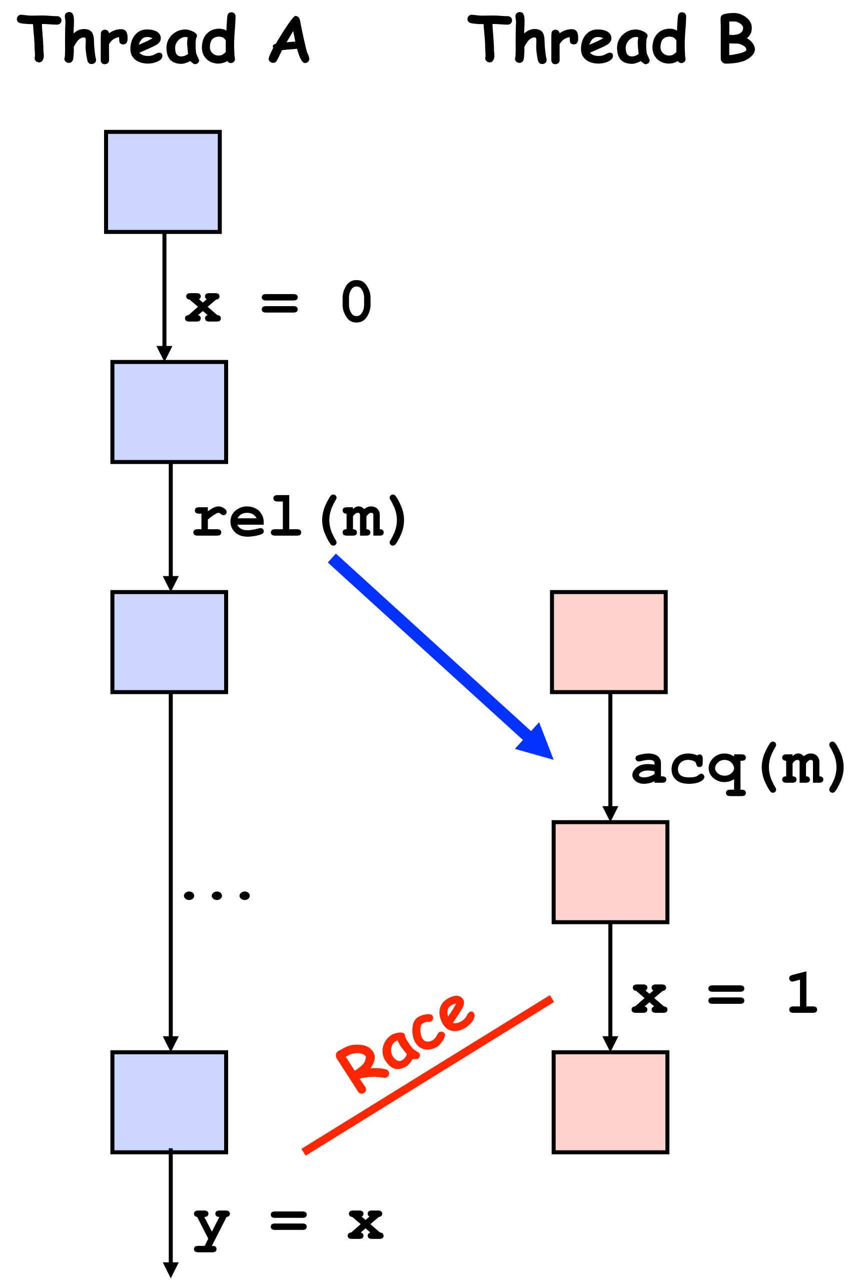
Dynamic Race Detection



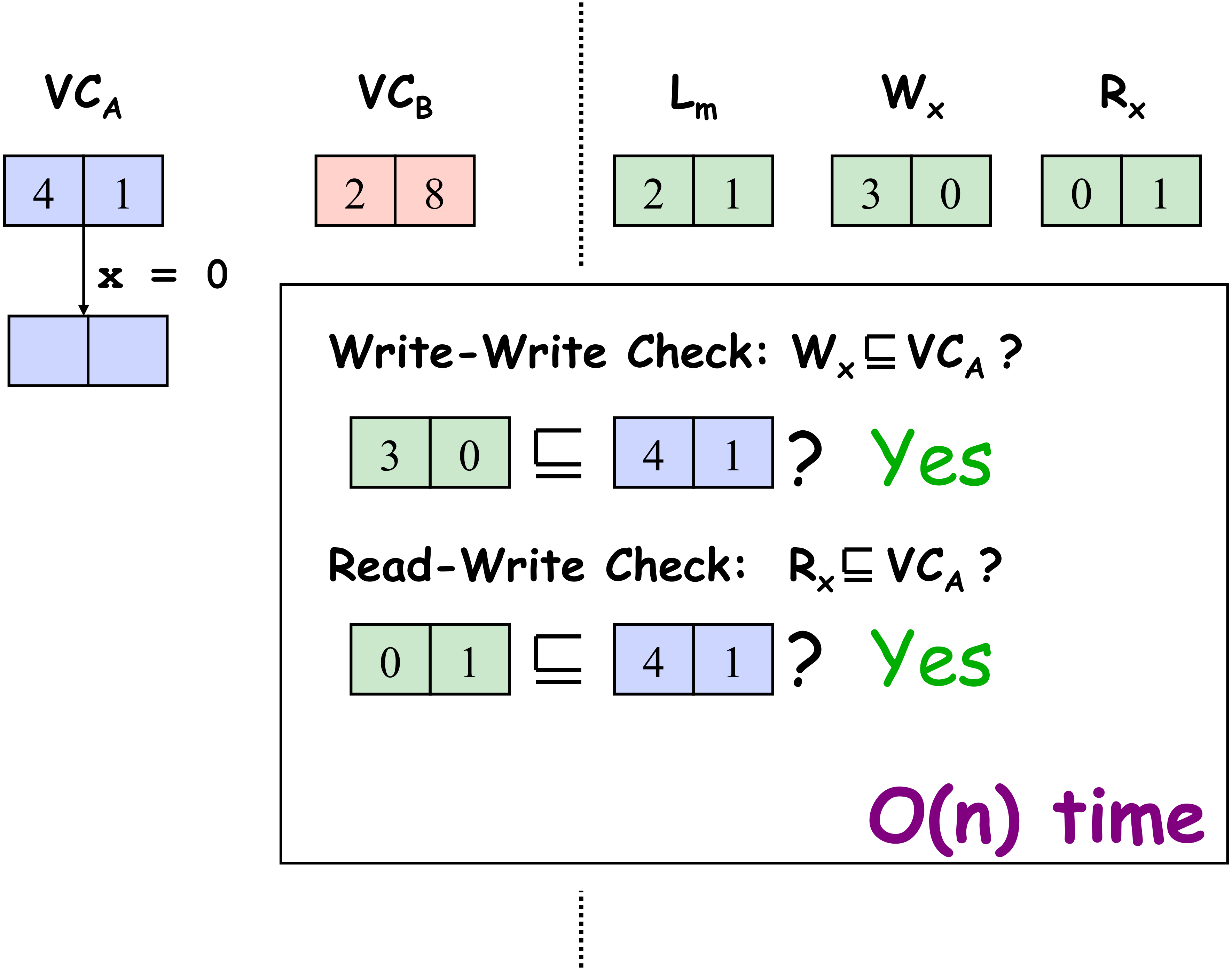
Happens-Before

- Event Ordering:
  - program order
  - synchronization order

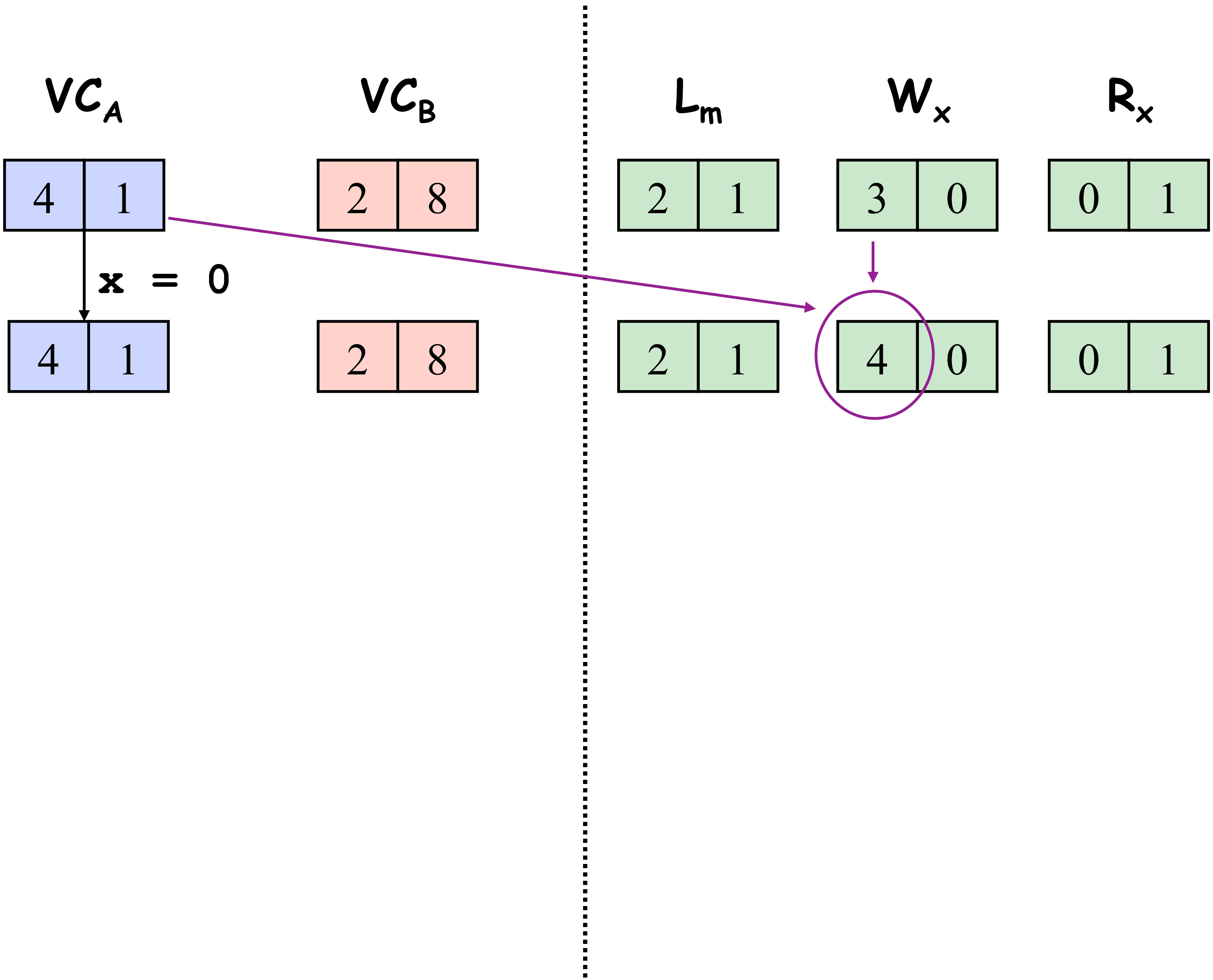
- Types of Races:
  - Write-Write
  - Write-Read
    - (write before read)
  - Read-Write
    - (read before write)



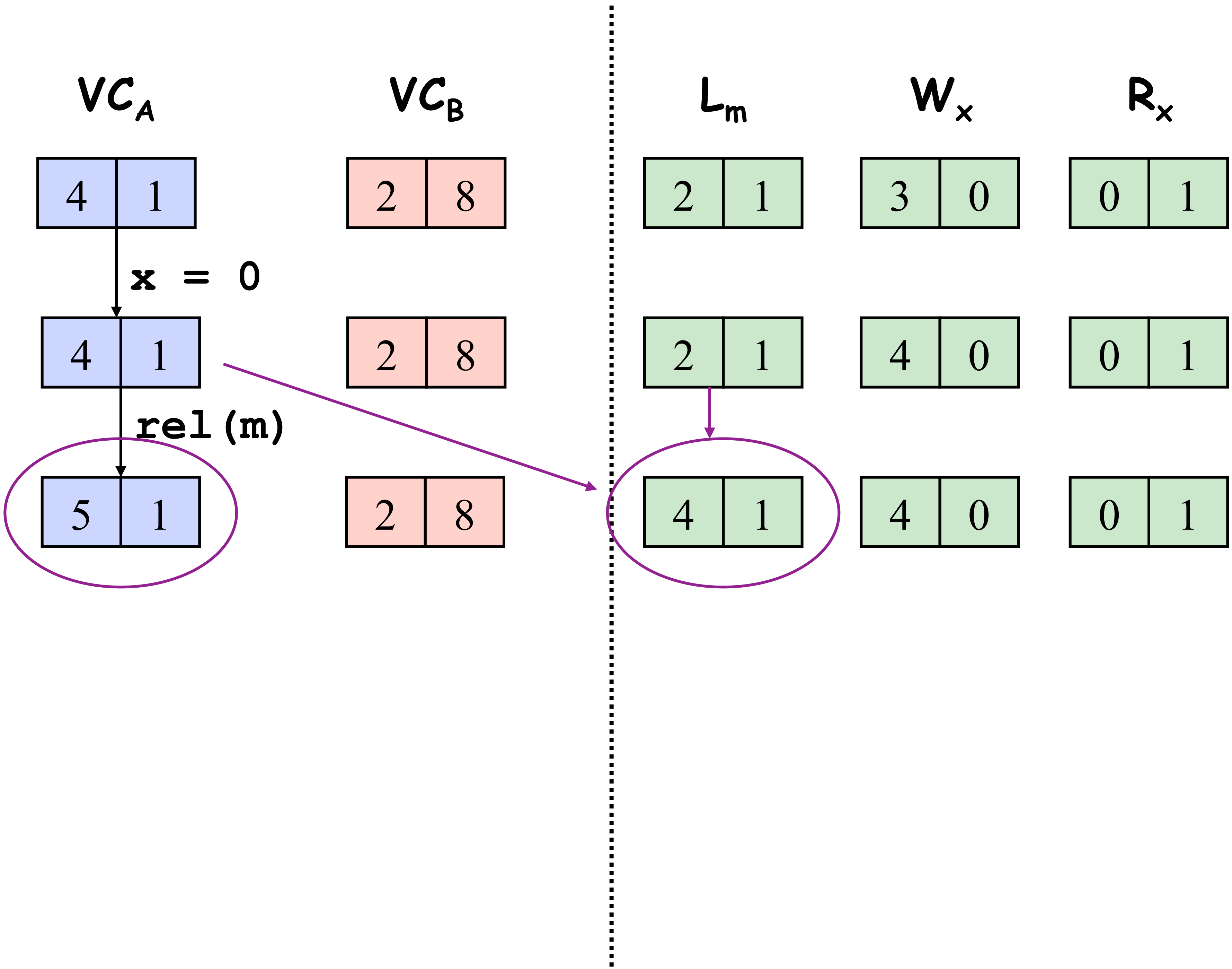
	A	B
VC_A	4	1
VC_B	2	8
L_m	2	1
W_x	3	0
R_x	0	1



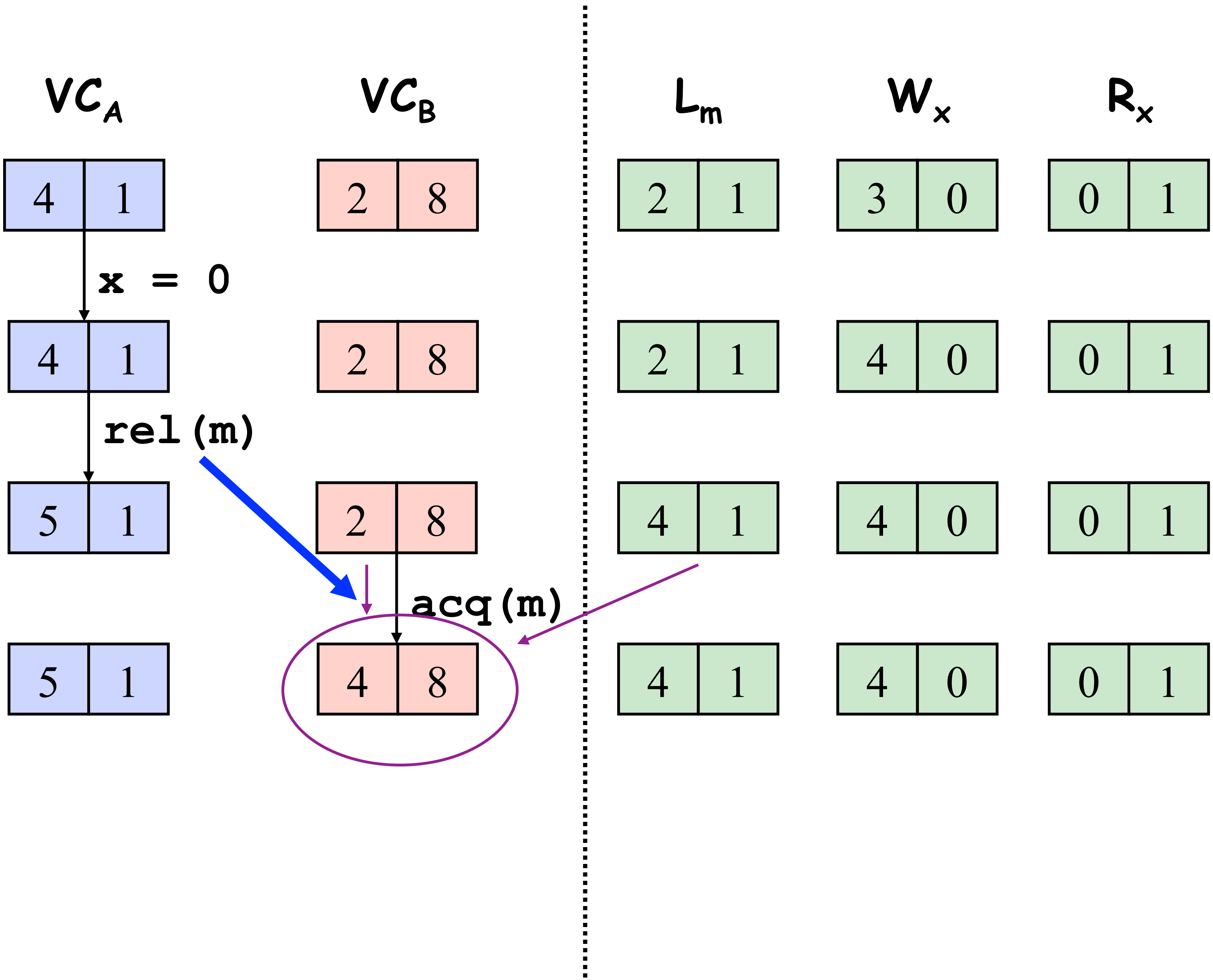
VC_A
4 1
x = 0
VC_B
2 8
L_m
2 1
W_x
3 0
R_x
0 1
Write-Write Check: W_x ⊑ VC_A ?
3 0 ⊑ 4 1 ? Yes
Read-Write Check: R_x ⊑ VC_A ?
0 1 ⊑ 4 1 ? Yes
O(n) time



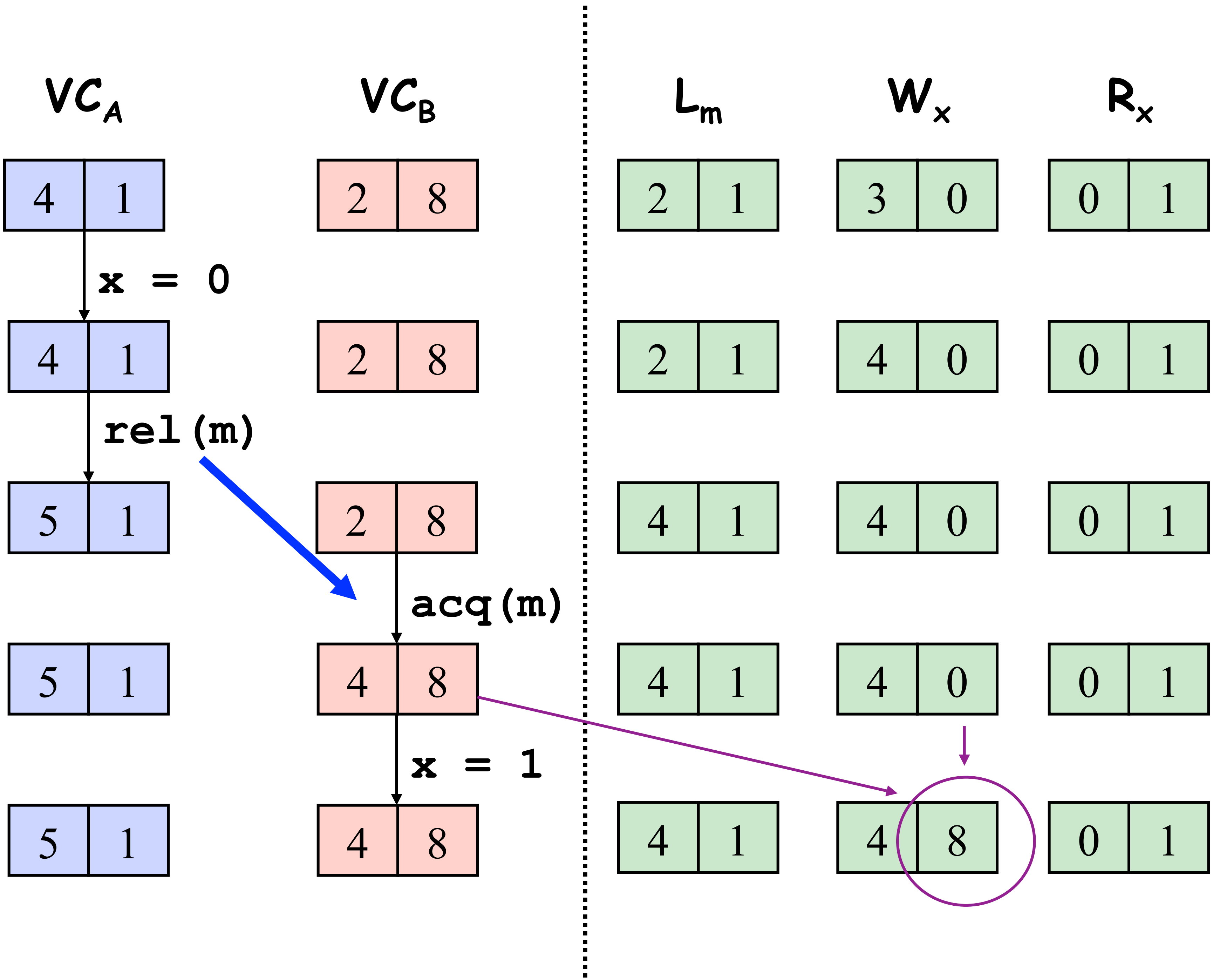
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1



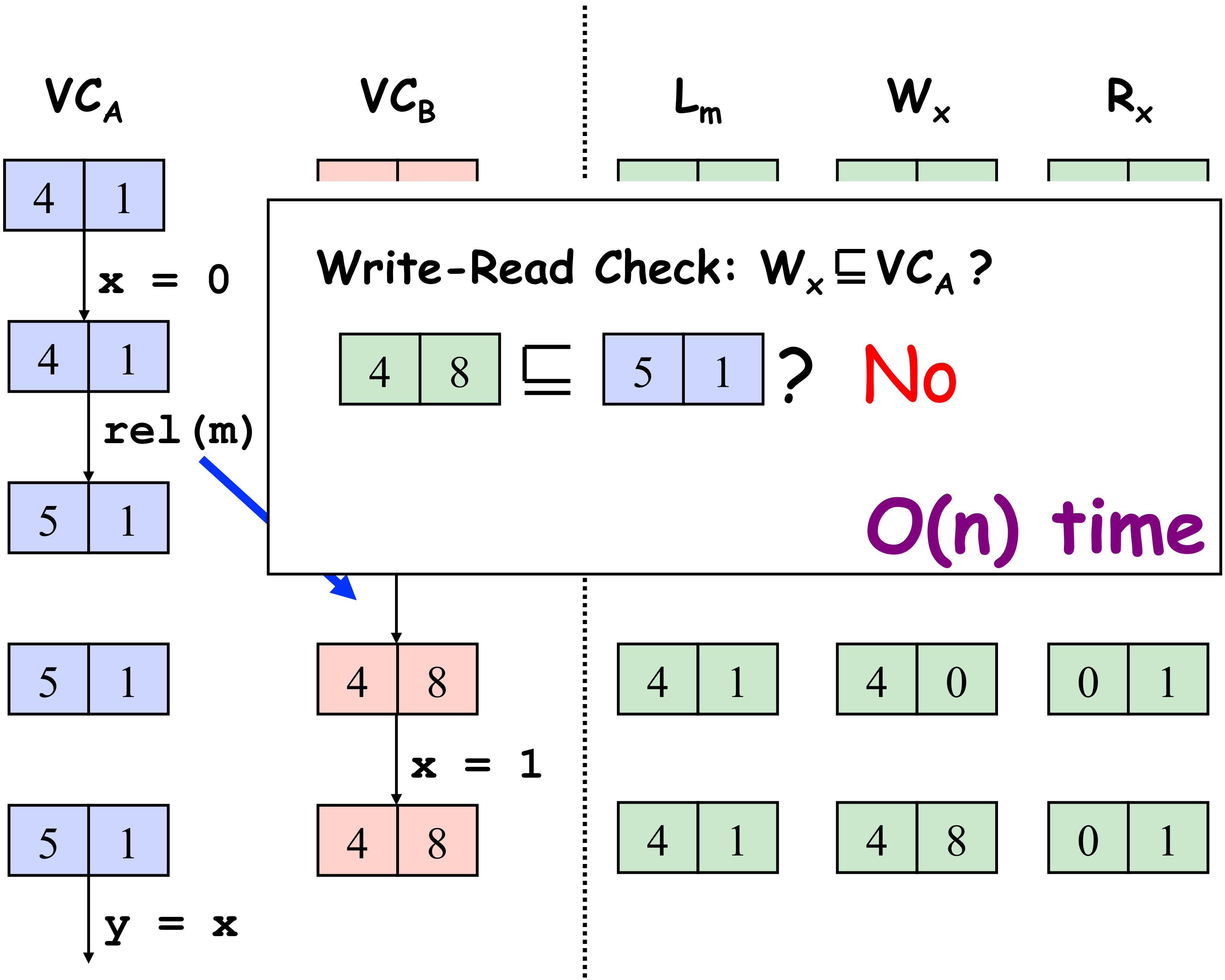
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1



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

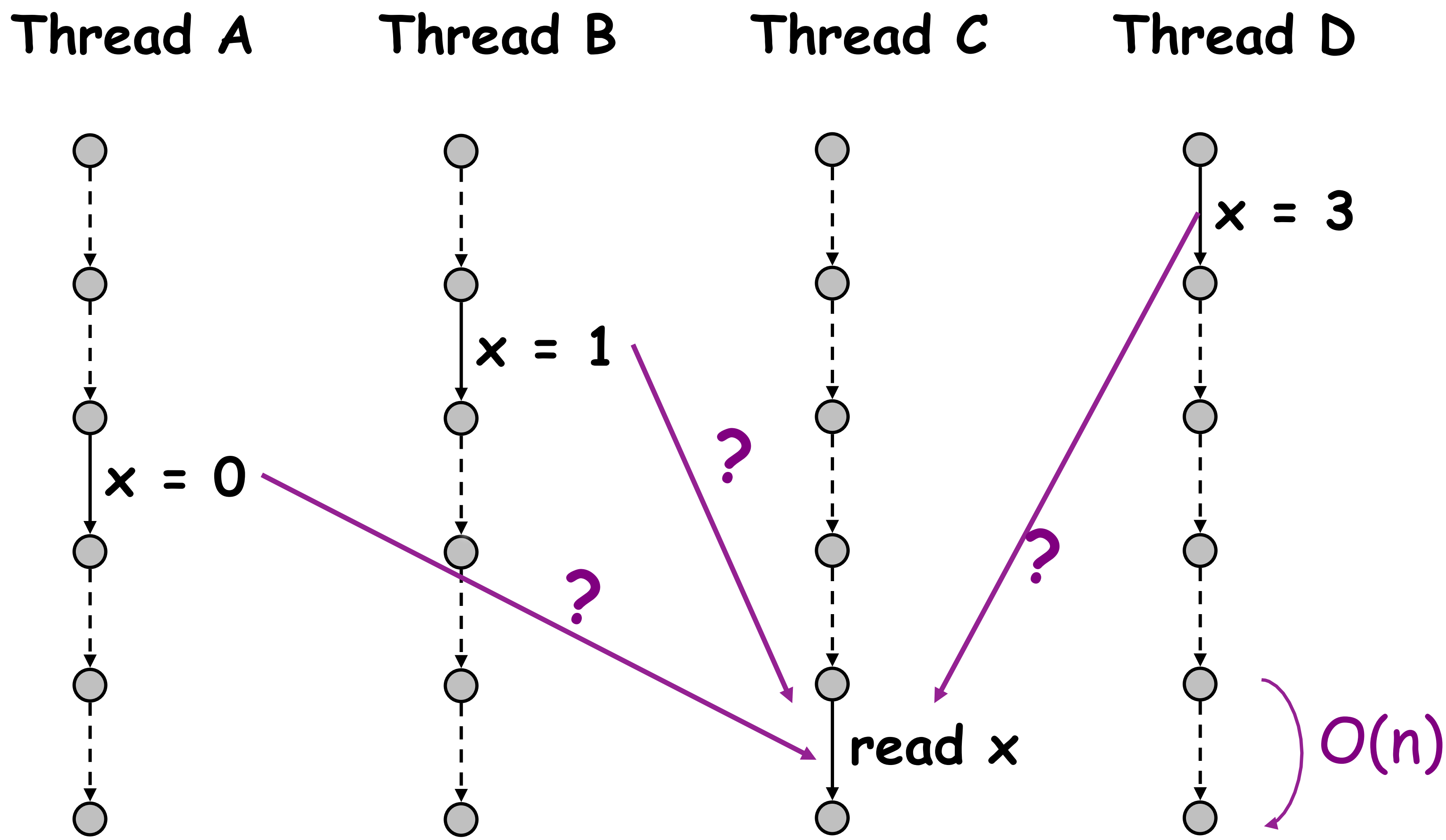


VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1
acq(m)
5 1
4 8
4 1
4 0
0 1
x = 1
5 1
4 8
4 1
4 8
0 1

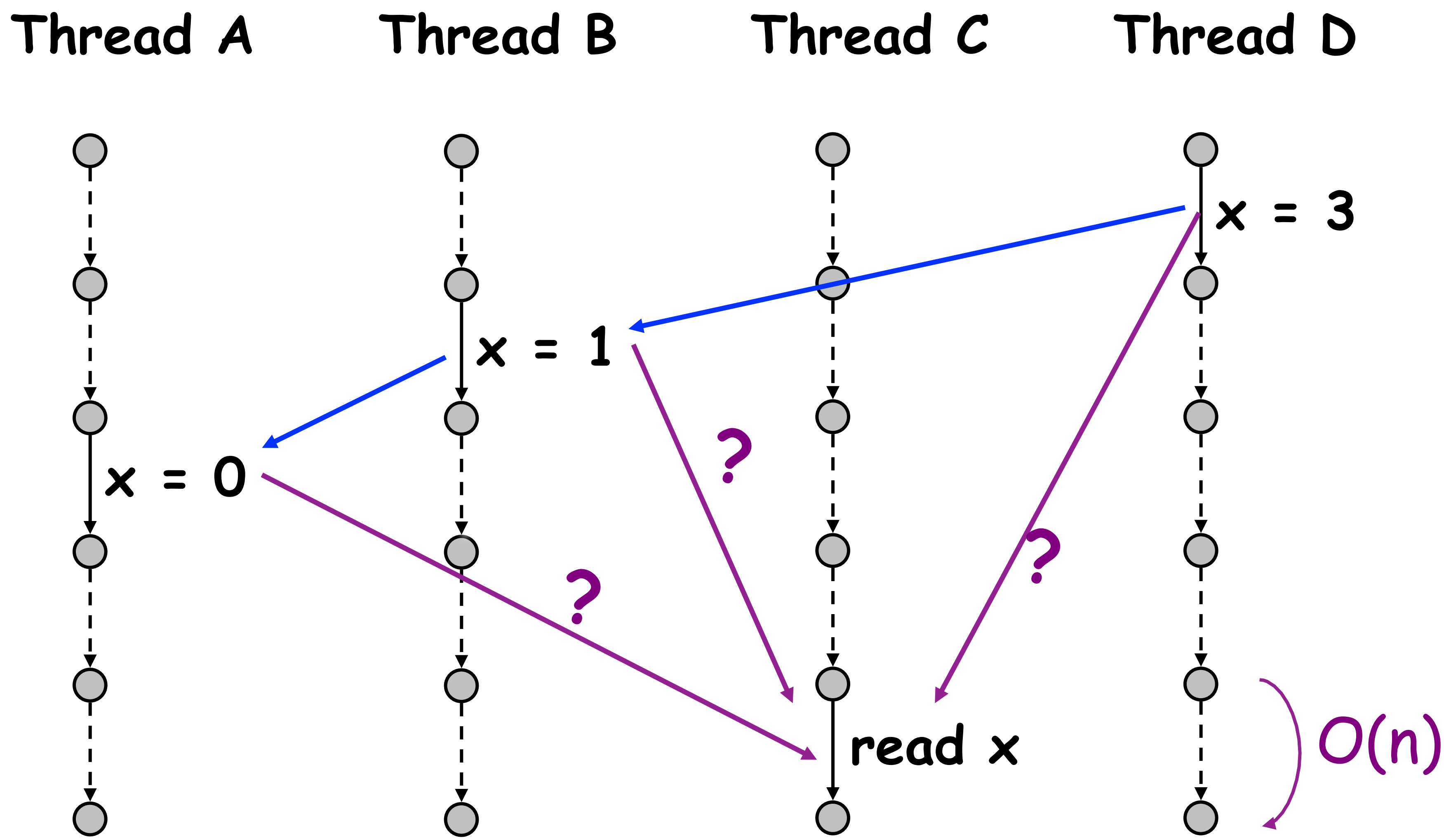


VC_A
VC_B
L_m
W_x
R_x
4 1
x = 0
4 1
rel(m)
5 1
5 1
4 8
4 1
4 0
0 1
x = 1
5 1
4 8
4 1
4 8
0 1
y = x
Write-Read Check: W_x ⊑ VC_A ?
4 8 ⊑ 5 1 ?
No
O(n) time

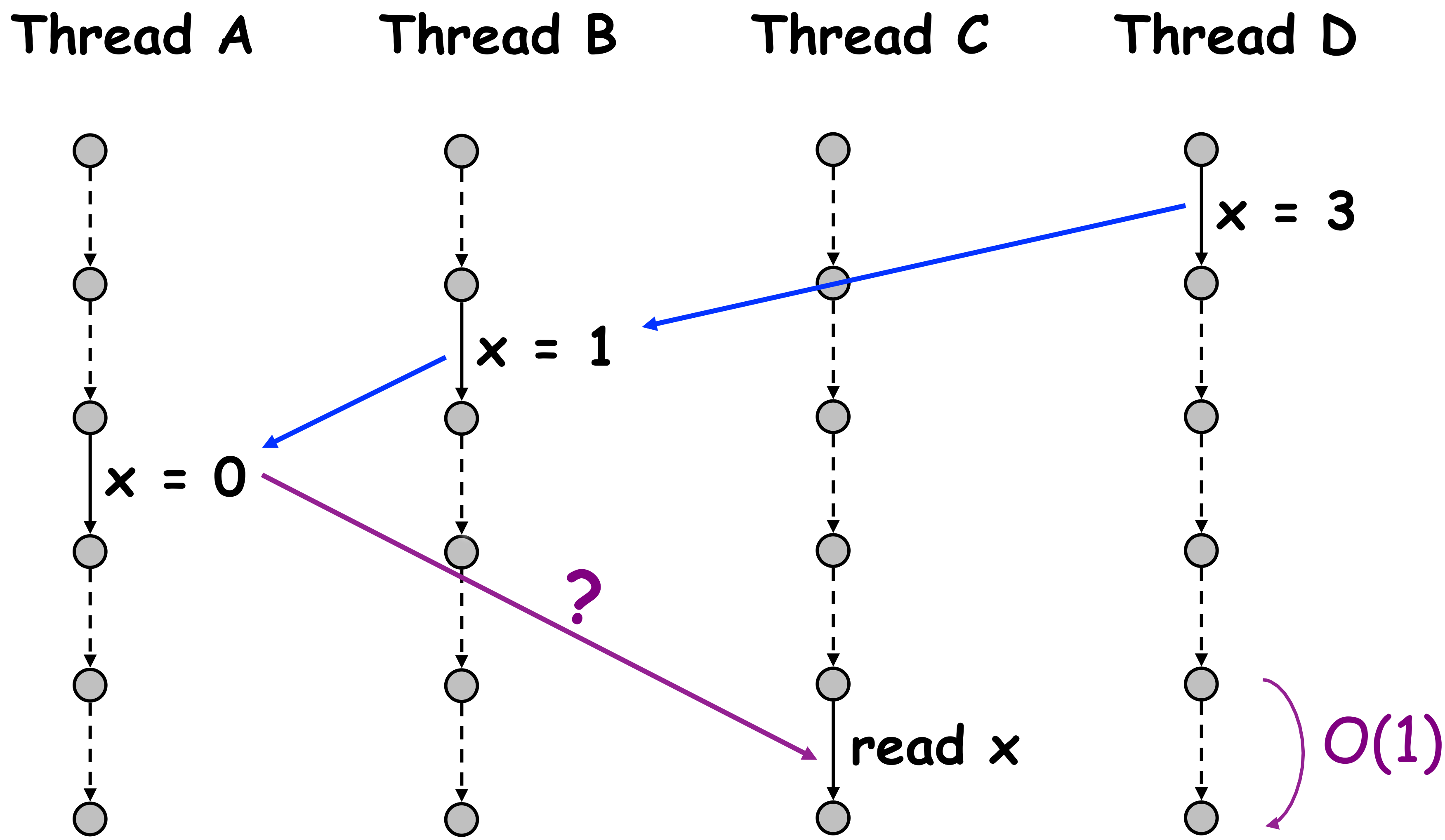
Write-Write and Write-Read Races

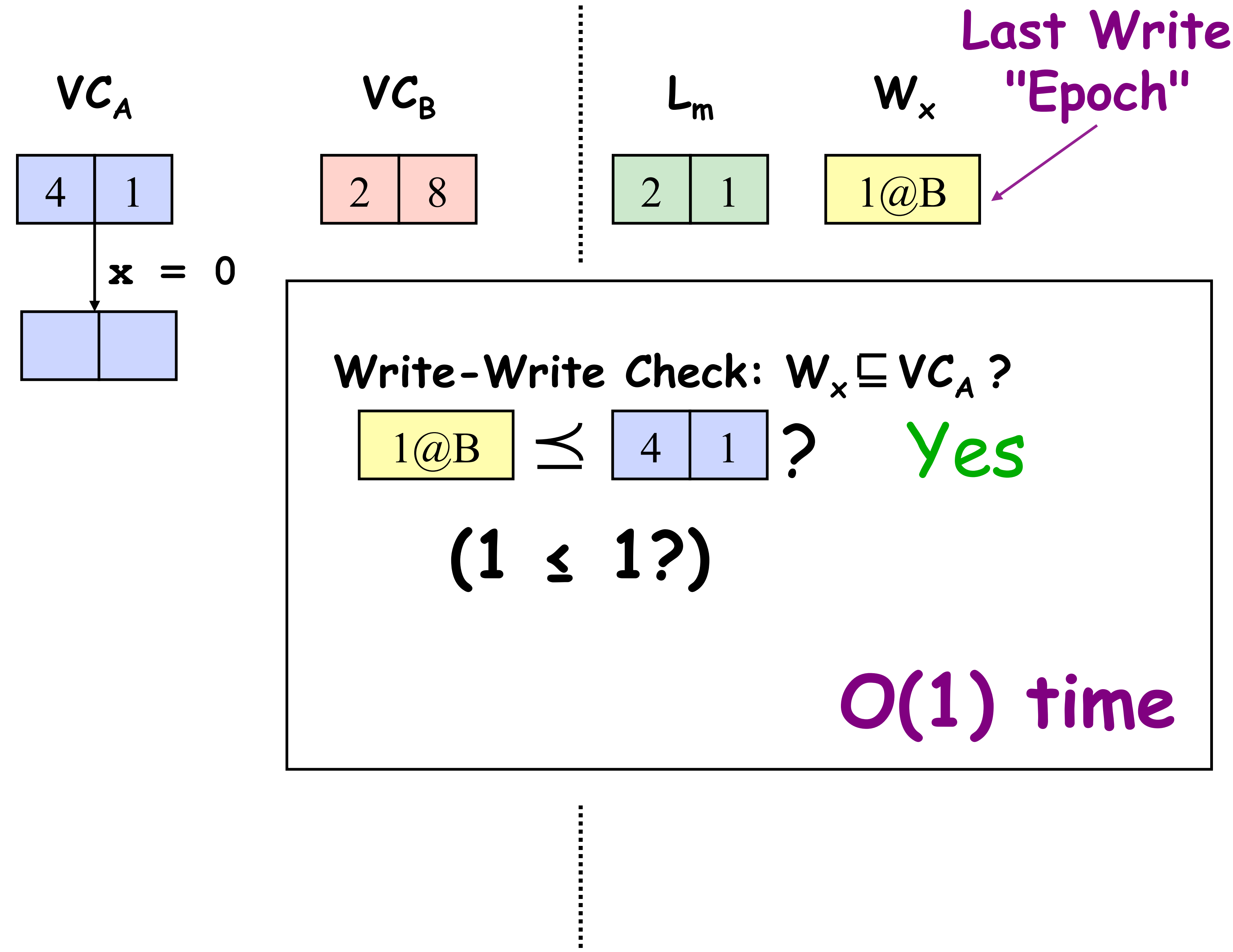


No Races Yet: Writes Totally Ordered!

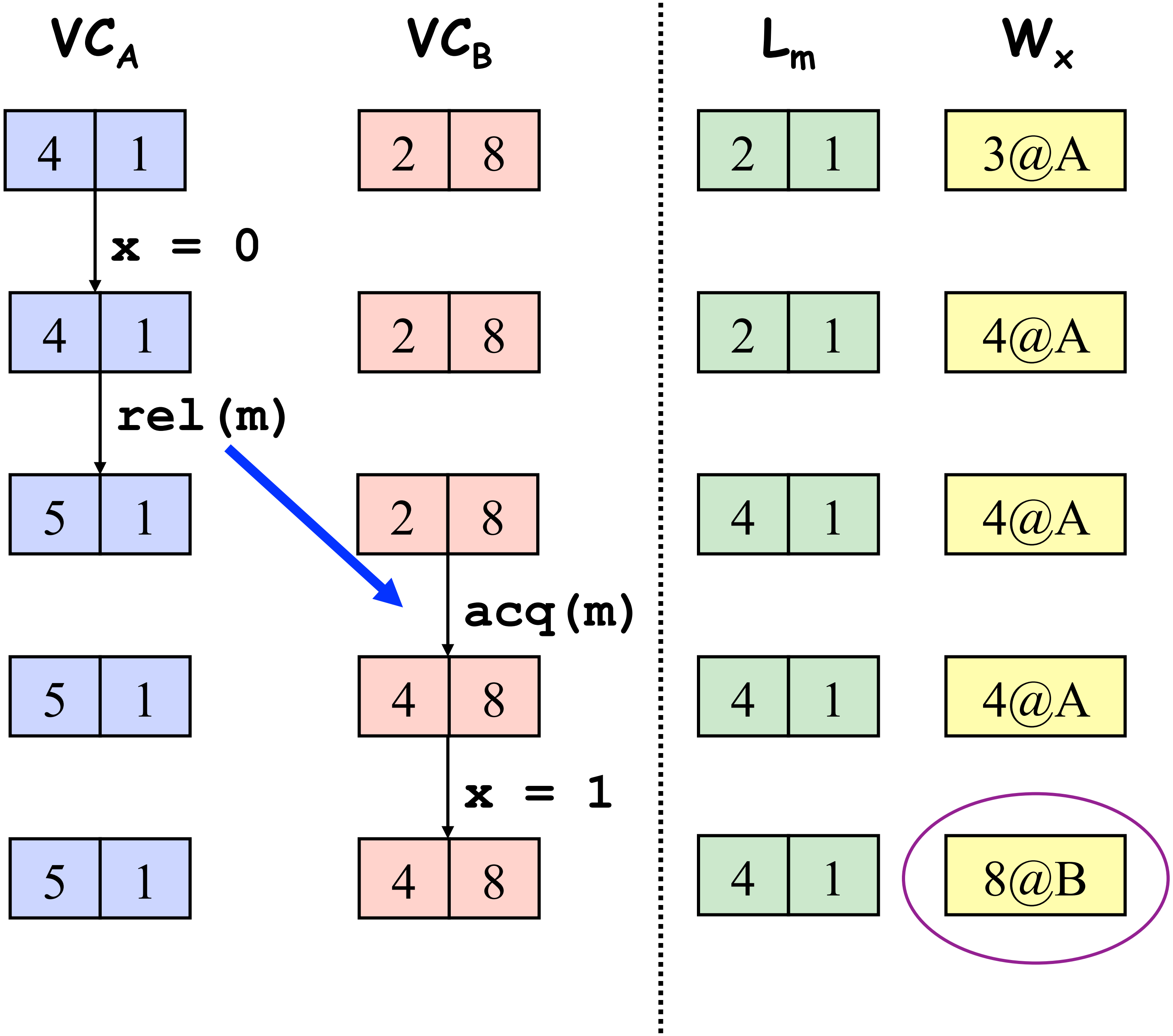


No Races Yet: Writes Totally Ordered!

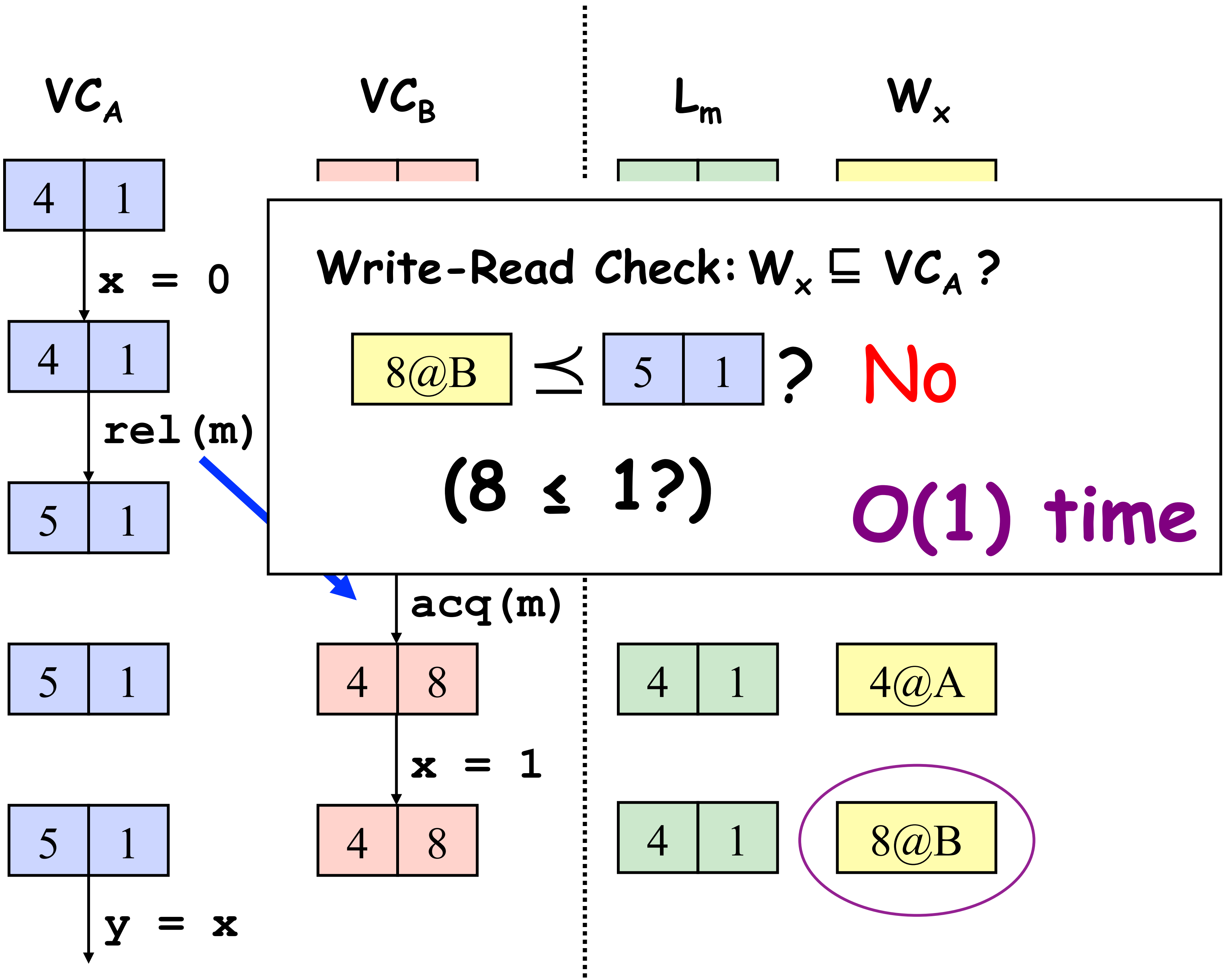




Last Write "Epoch"
VC_A
4 1
x = 0
VC_B
2 8
L_m
2 1
W_x
1@B
Write-Write Check: W_x ⊑ VC_A ?
1@B ≼ 4 1 ?
Yes
(1 ≤ 1?)
O(1) time

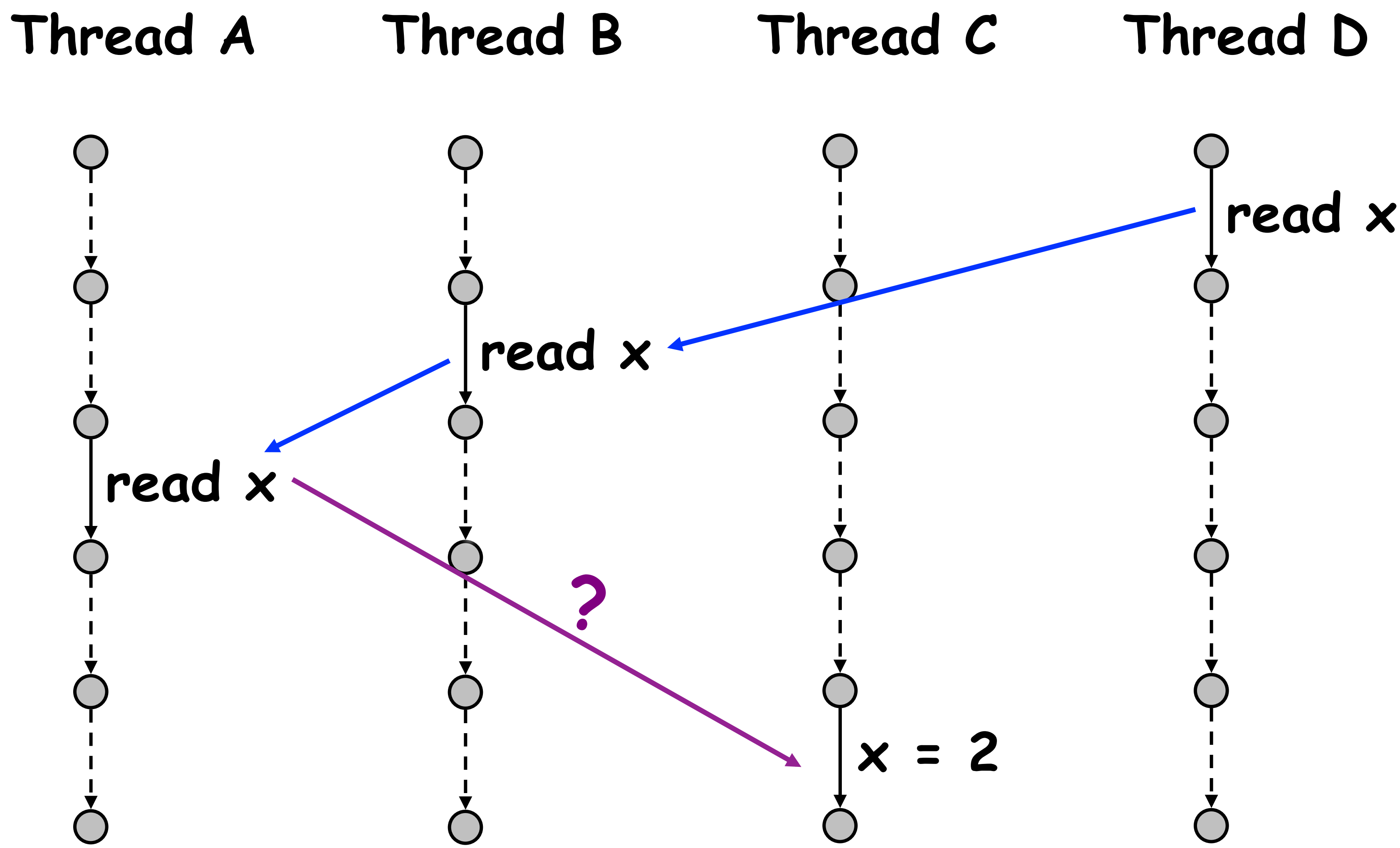


VC_A
VC_B
L_m
W_x
4 1
2 8
2 1
3@A
x = 0
4 1
2 8
2 1
4@A
rel(m)
5 1
2 8
4 1
4@A
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B



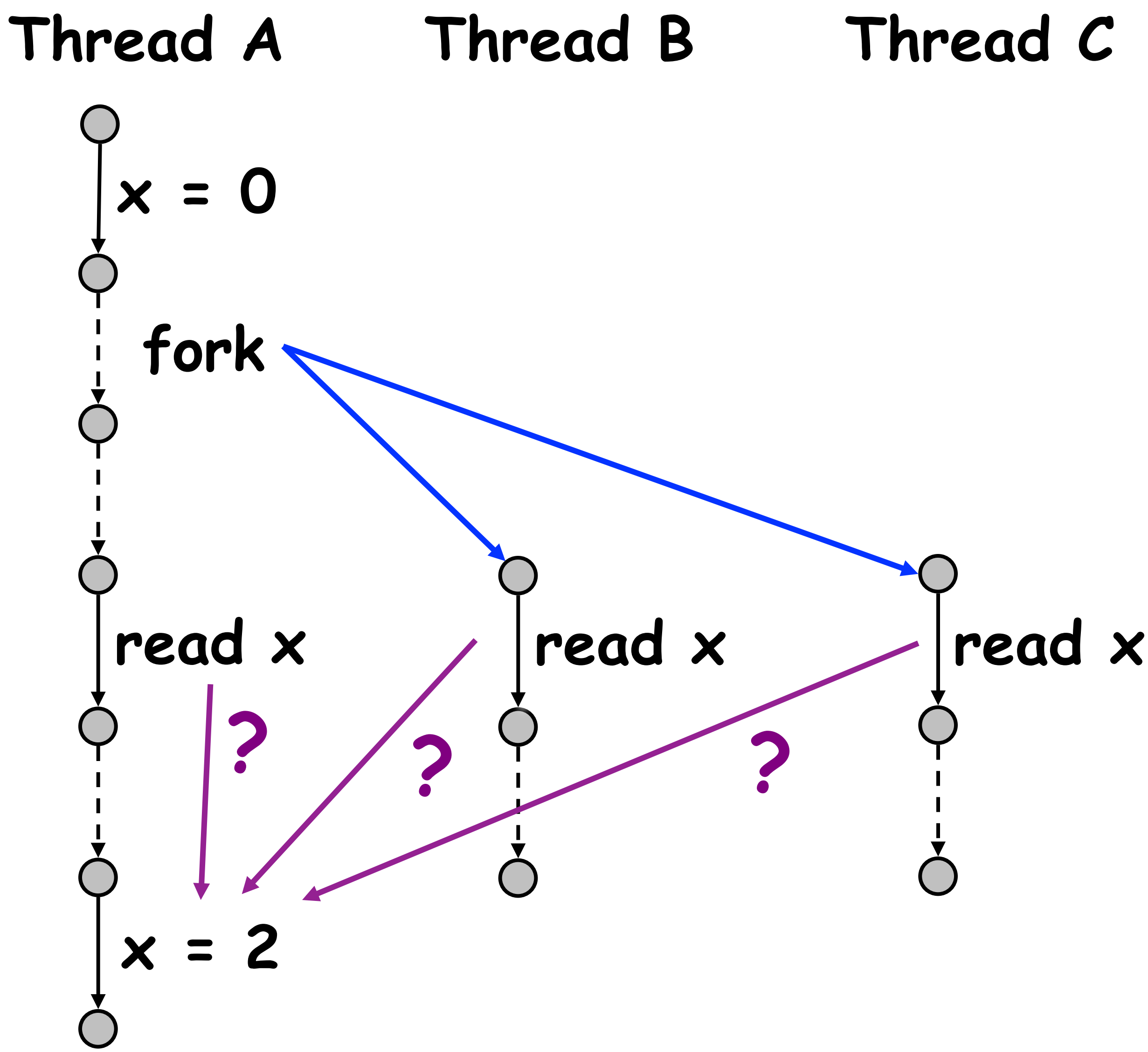
VC_A
VC_B
L_m
W_x
4 1
x = 0
4 1
rel(m)
5 1
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B
y = x
Write-Read Check: W_x ⊑ VC_A ?
8@B ≼ 5 1 ? No
(8 ≤ 1?)
O(1) time

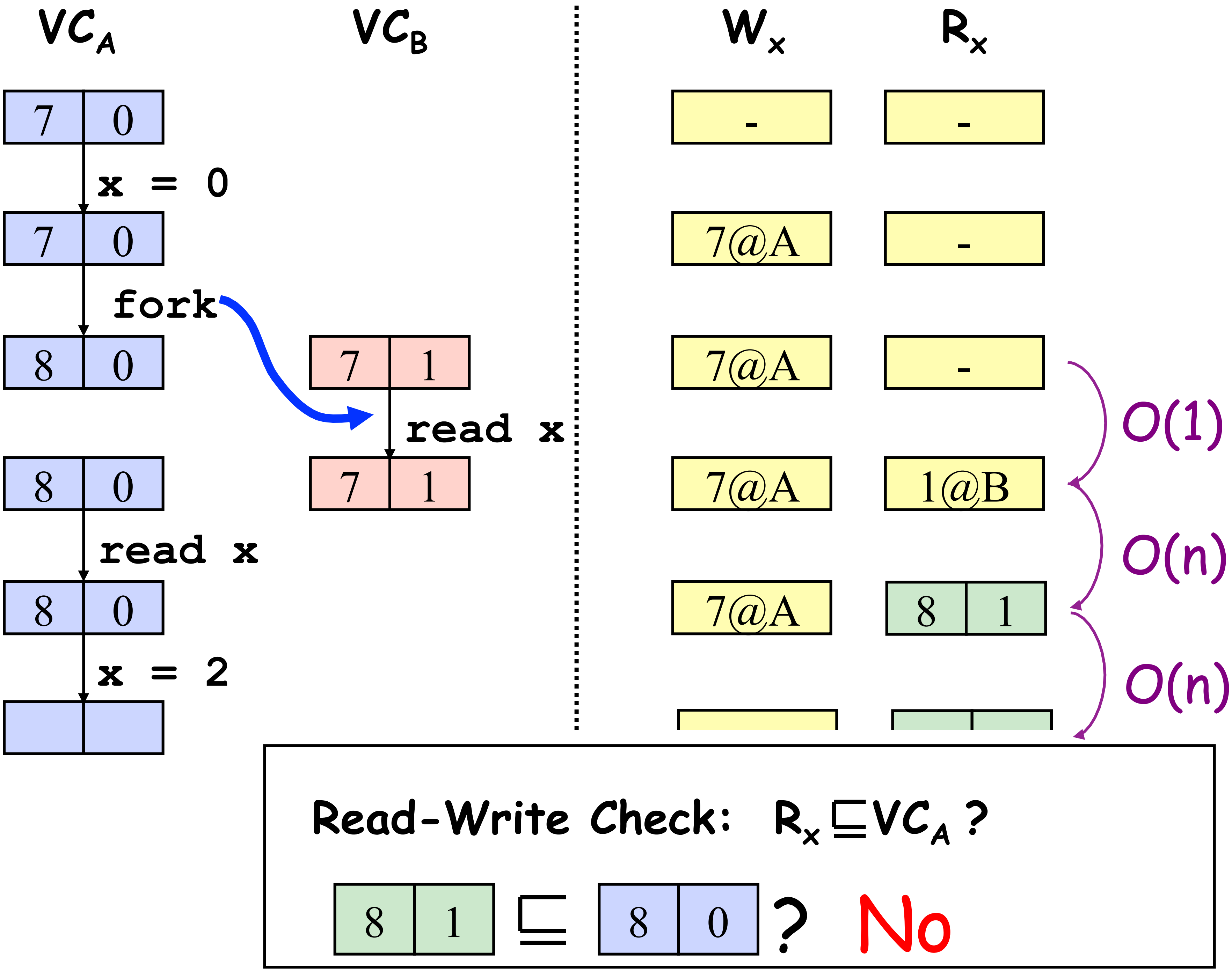
Read-Write Races -- Ordered Reads



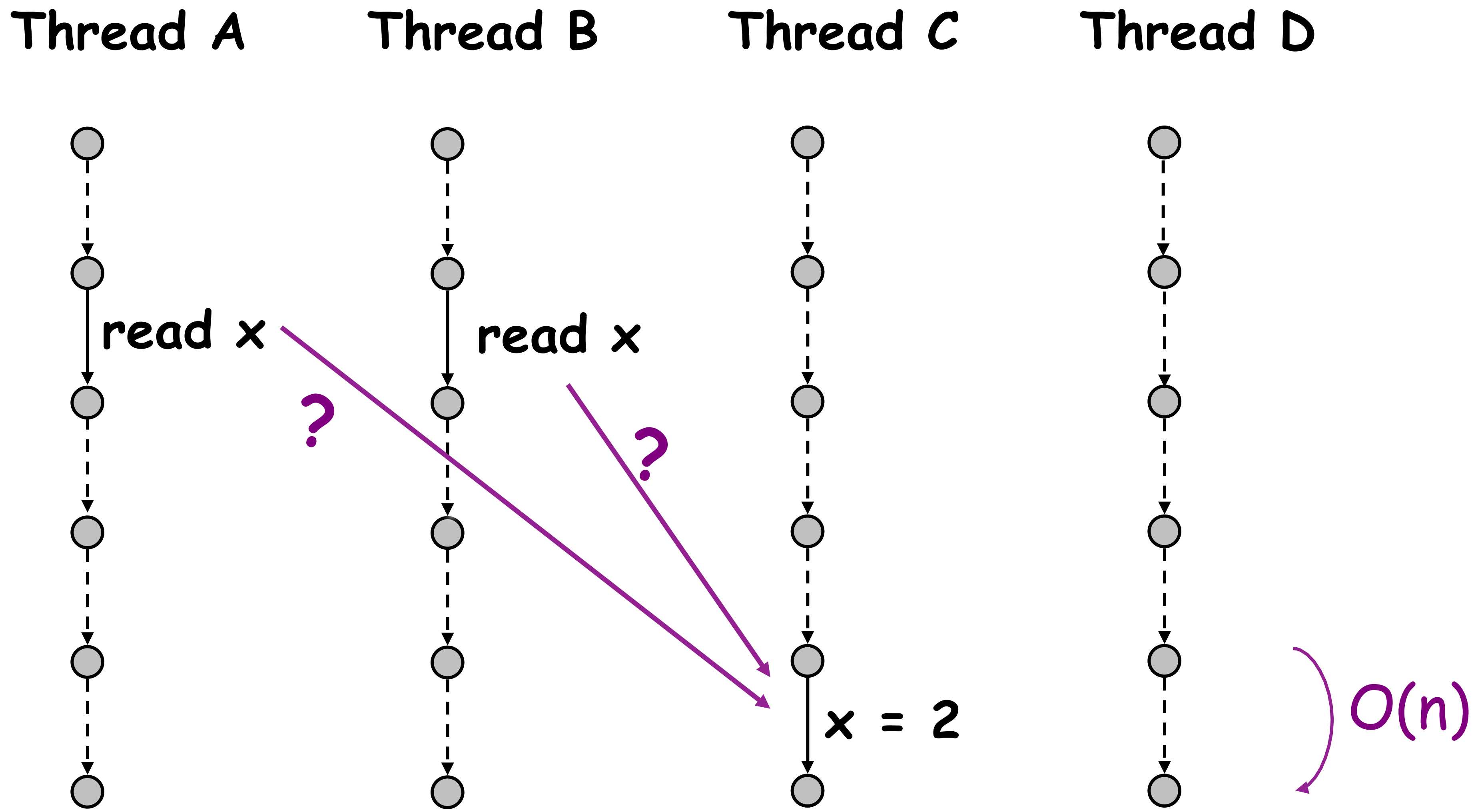
Most common case: thread-local, lock-protected, ...

Read-Write Races -- Unordered Reads

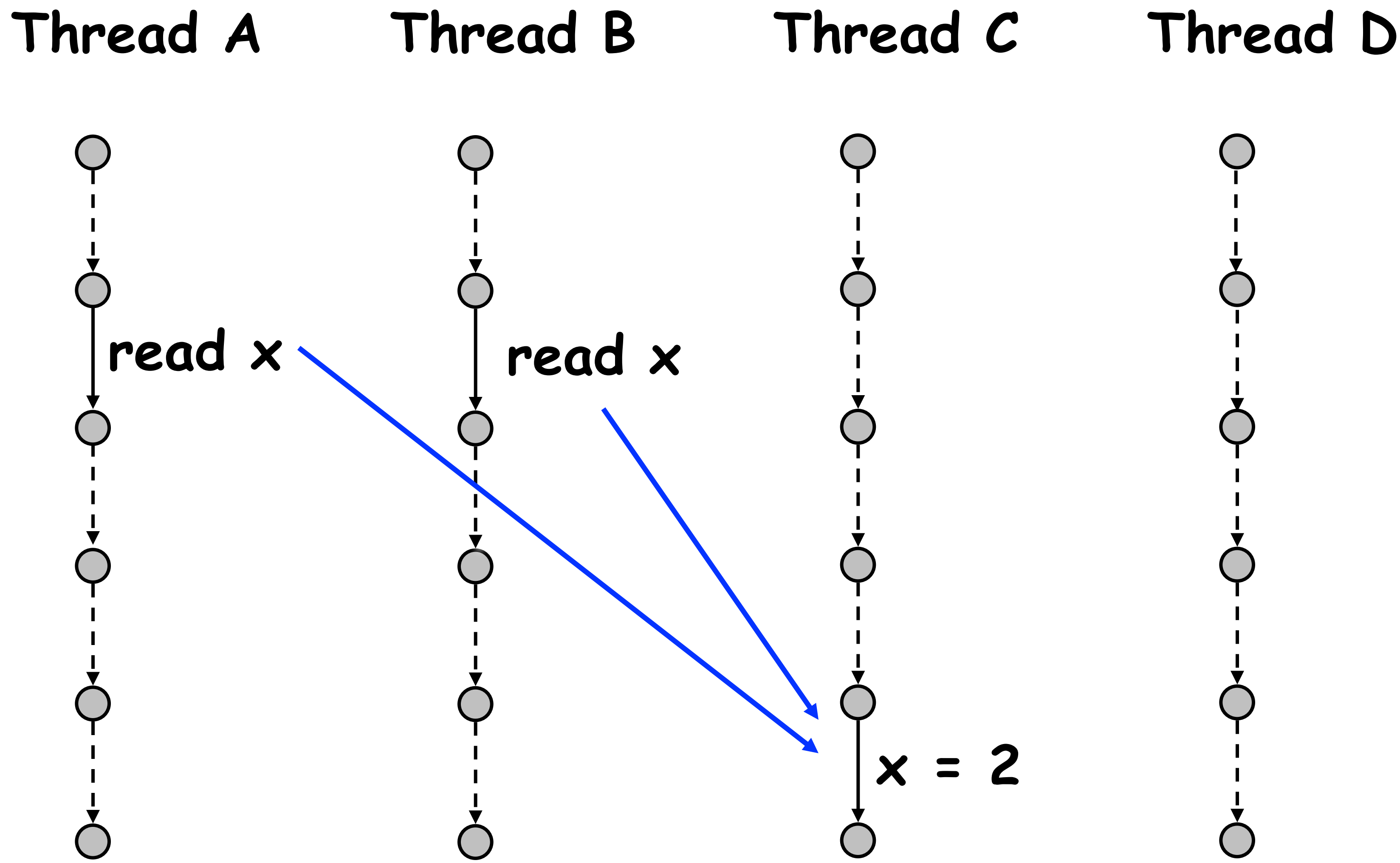




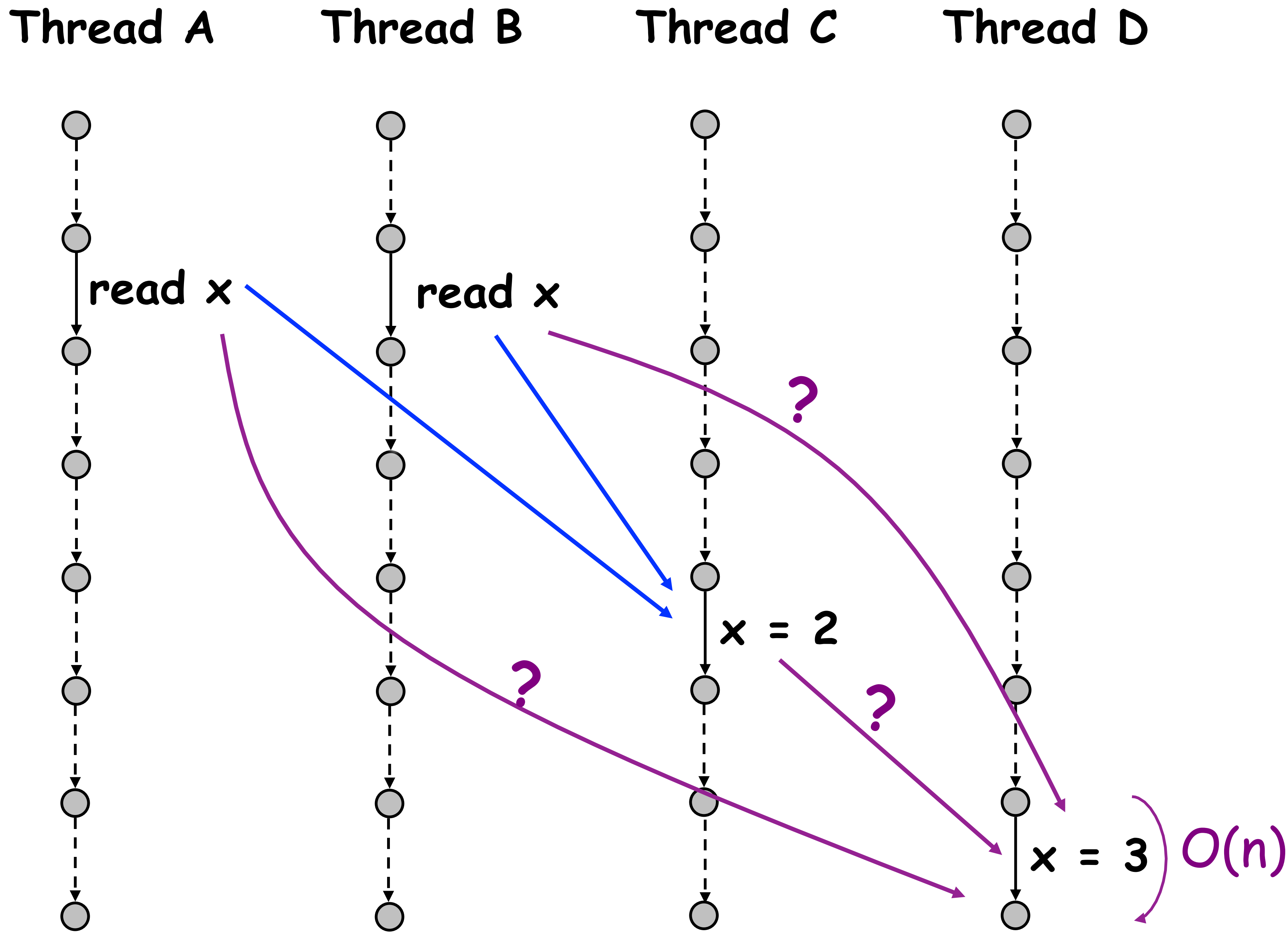
VC_A
VC_B
W_x
R_x
7 0
x = 0
7 0
fork
8 0
7 1
read x
8 0
7 1
read x
8 0
x = 2
-
-
7@A
-
7@A
-
7@A
1@B
7@A
8 1
O(1)
O(n)
O(n)
Read-Write Check: R_x ⊑ VC_A ?
8 1 ⊑ 8 0 ? No

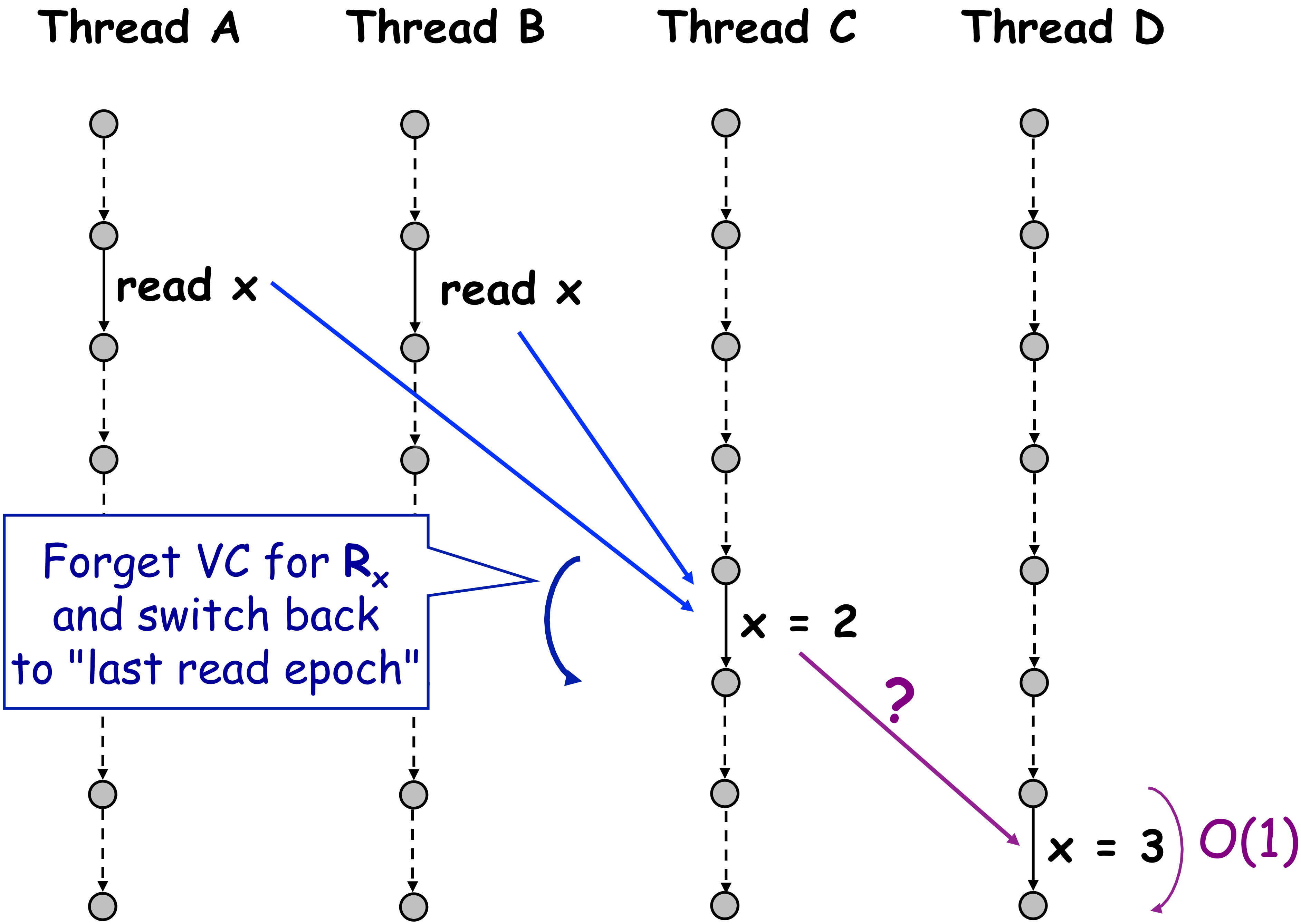


Thread A
Thread B
Thread C
Thread D
read x
read x
?
?
x = 2
O(n)

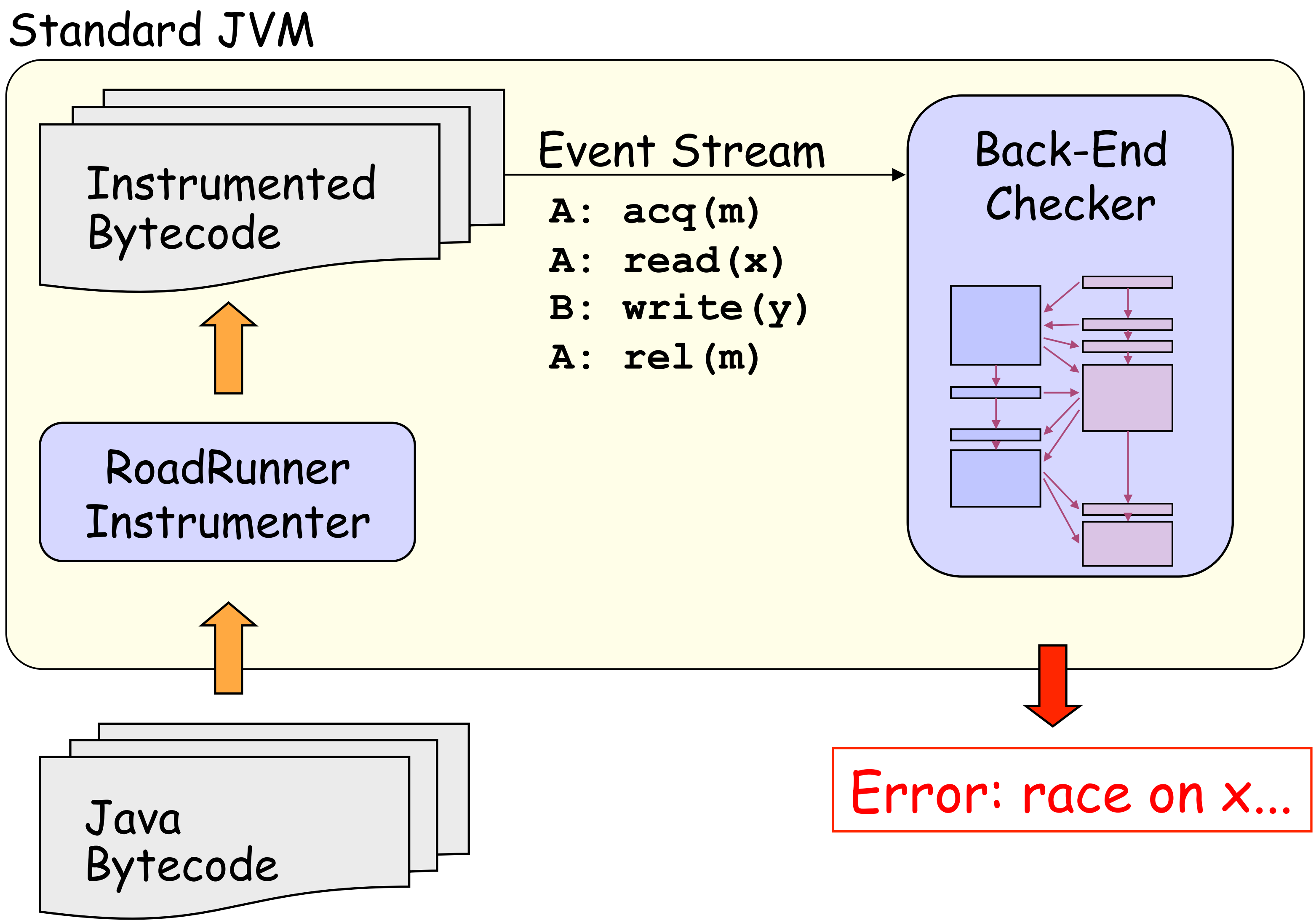


Thread A
Thread B
Thread C
Thread D
read x
read x
x = 2


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

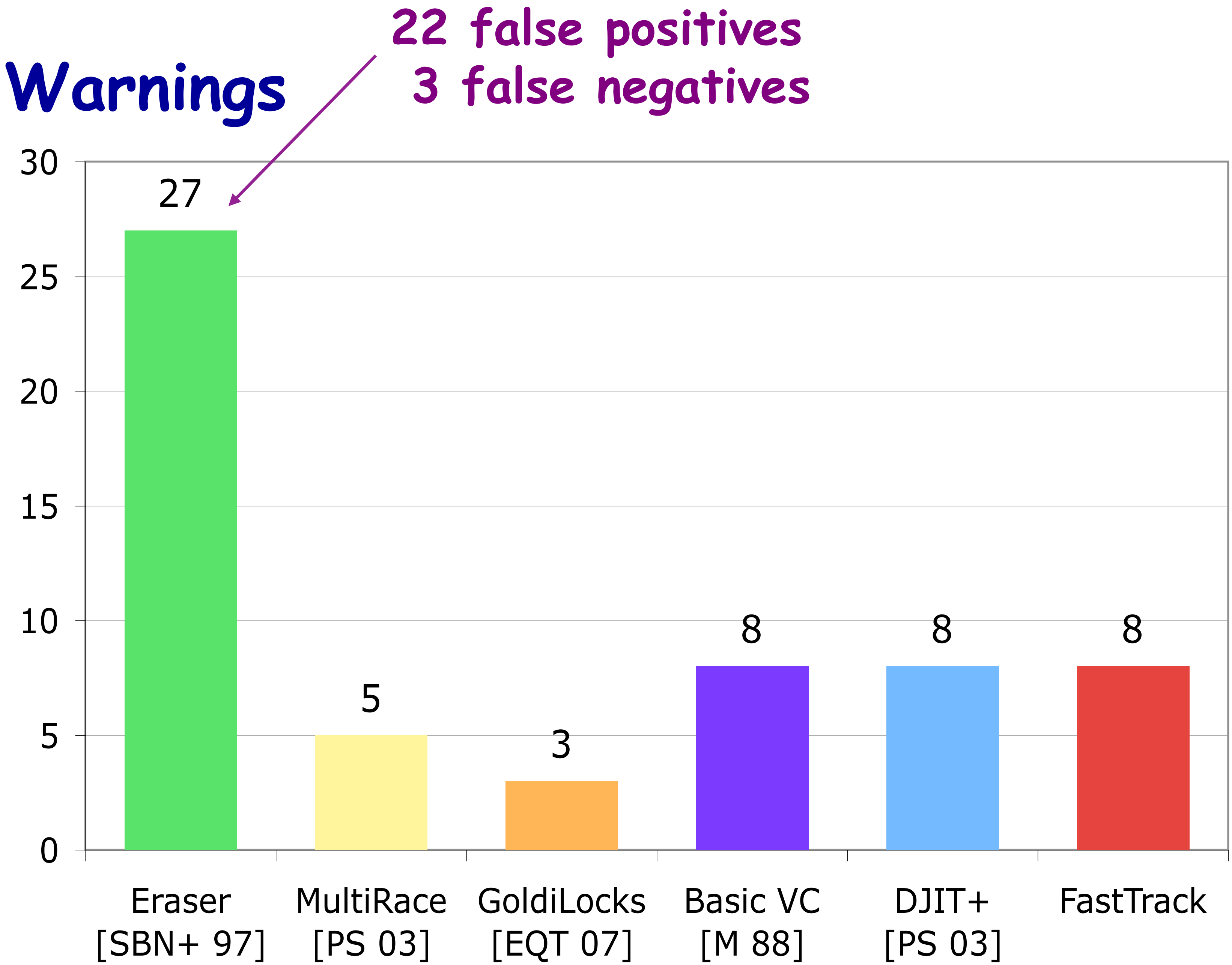

Thread A
Thread B
Thread C
Thread D
read x
read x
Forget VC for R_x and switch back to "last read epoch"
x = 2
?
x = 3
O(1)

RoadRunner Architecture



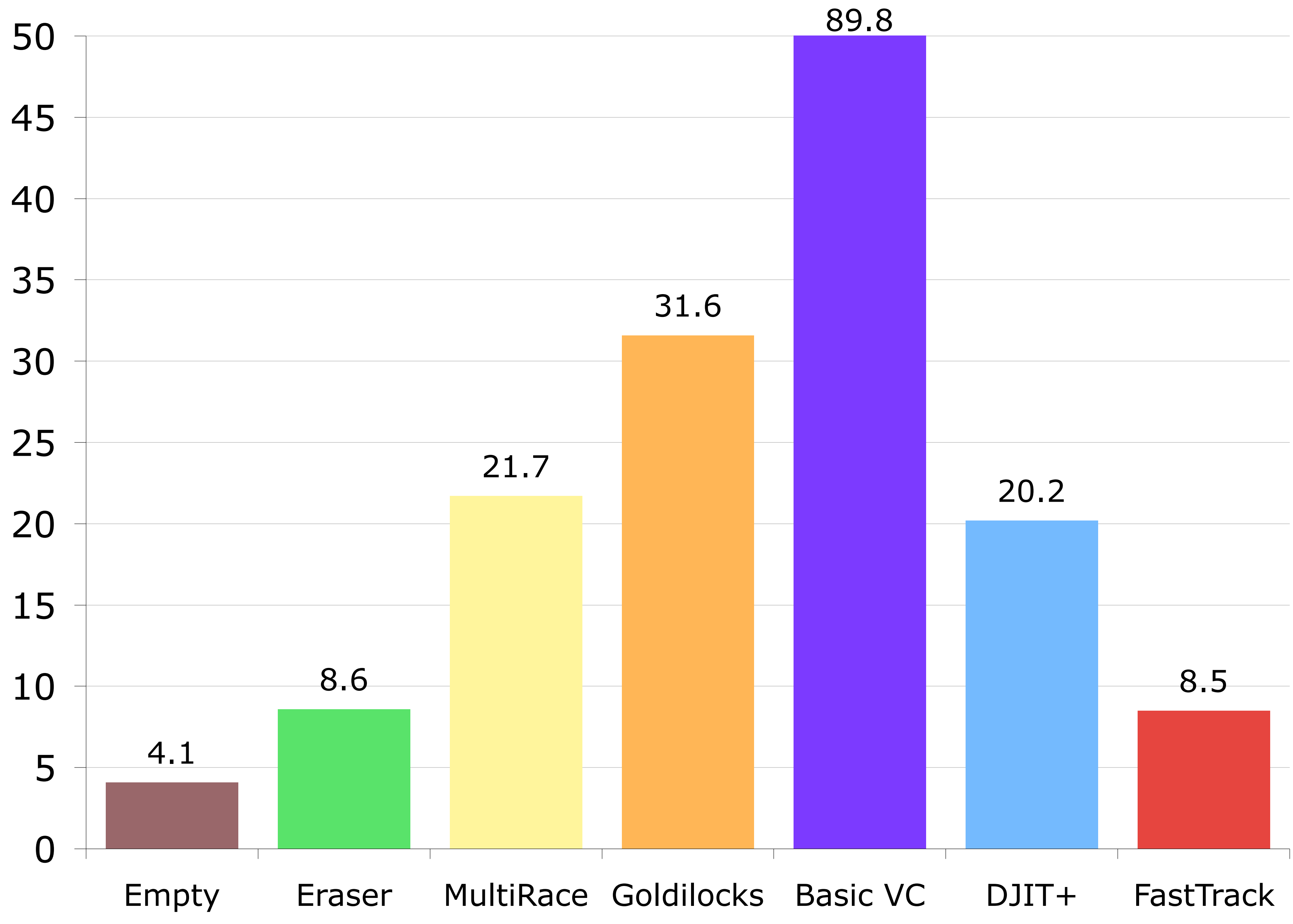
Validation

- 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, ...

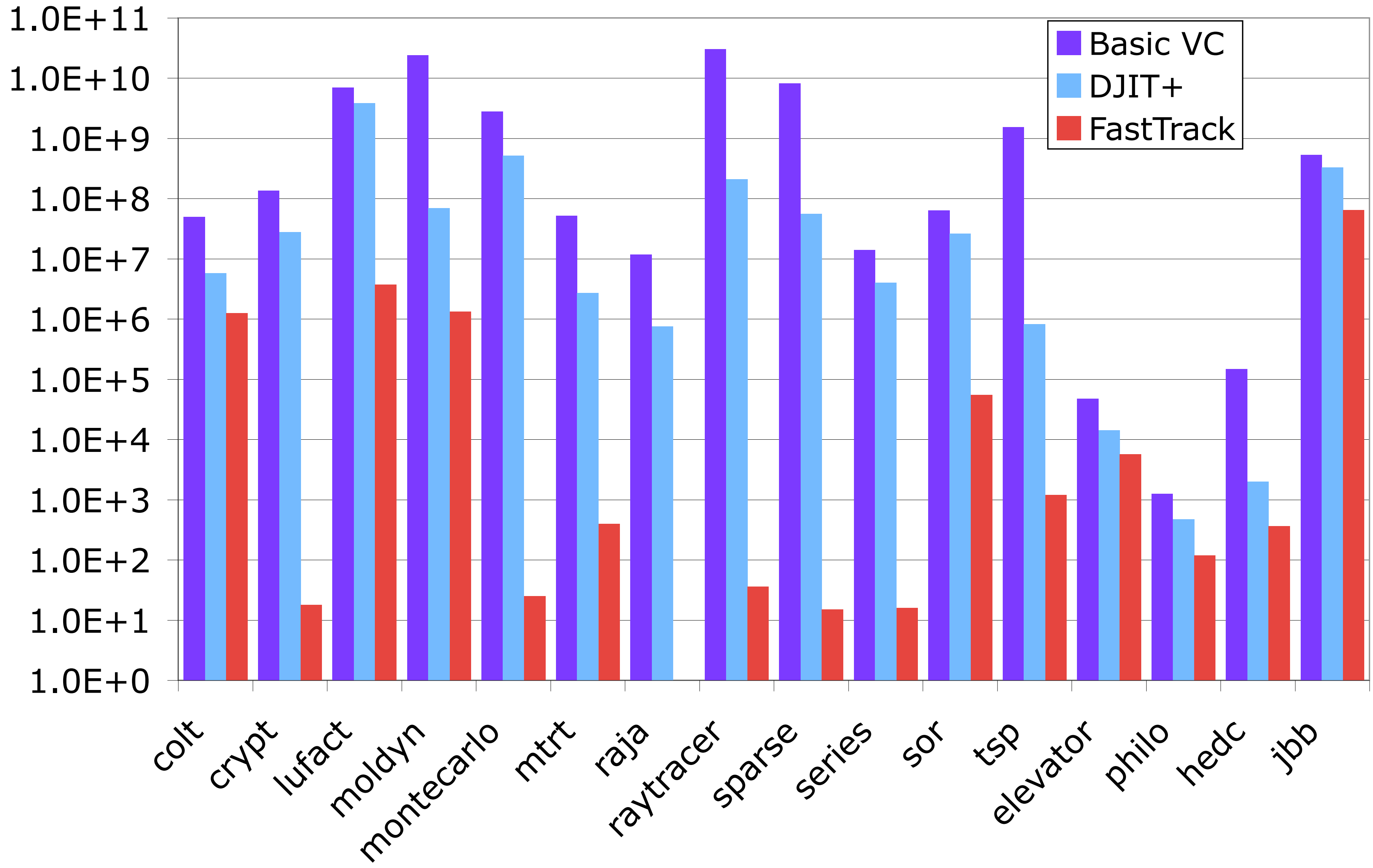


Warnings
22 false positives
3 false negatives
30
25
20
15
10
5
0
27
5
3
8
8
8
Eraser
[SBN+ 97]
MultiRace
[PS 03]
GoldiLocks
[EQT 07]
Basic VC
[M 88]
DJIT+
[PS 03]
FastTrack

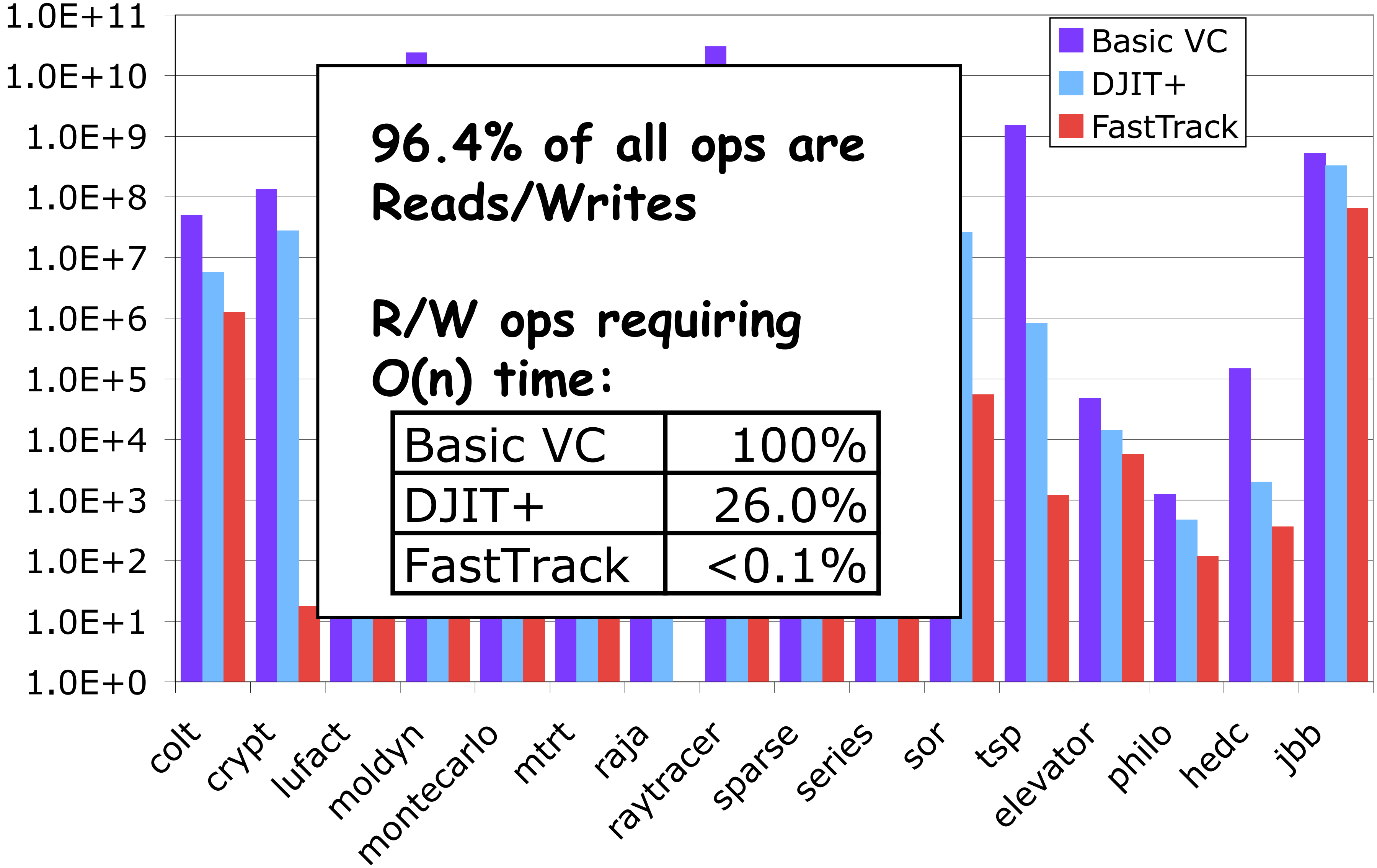
Slowdown (x Base Time)



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

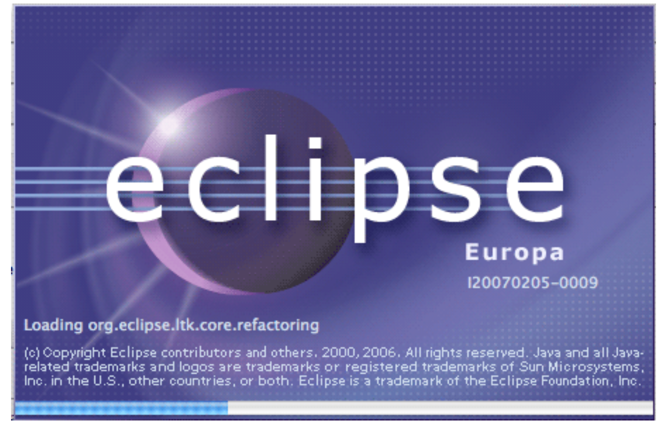
- FastTrack allocated ~200x fewer VCs

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

ThreadSanitizer, ~~MemorySanitizer~~

Scalable run-time detection of uninitialized memory reads and data races with LLVM instrumentation

Timur Iskhodzhanov, Alexander Potapenko,
Alexey Samsonov, Kostya Serebryany,
Evgeniy Stepanov, Dmitry Vyukov

LLVM developers' meeting, Nov 8 2012

ThreadSanitizer

data races

ThreadSanitizer v1

- Race detector based on Valgrind
- Used since early 2009
- Slow (20x–300x slowdown)
  - Still, found thousands races
  - Faster & more usable than others
    - Helgrind (Valgrind)
    - Intel Parallel Inspector (PIN)
- WBIA'09

ThreadSanitizer v2 overview

- Simple compile-time instrumentation
  - ~400 LOC
- Redesigned run-time library
  - Fully parallel
  - No expensive atomics/locks on fast path
  - Scales to huge apps
  - Predictable memory footprint
  - Informative reports

TSan report example: data race

```
void Thread1() { Global = 42; }
int main() {
  pthread_create(&t, 0, Thread1, 0);
  Global = 43;
  ...
% clang -fsanitize=thread -g a.c -fPIE -pie && ./a.out
WARNING: ThreadSanitizer: data race (pid=20373)
  Write of size 4 at 0x7f... by thread 1:
    #0 Thread1 a.c:1
  Previous write of size 4 at 0x7f... by main thread:
    #0 main a.c:4
  Thread 1 (tid=20374, running) created at:
    #0 pthread_create ??:0
    #1 main a.c:3
```

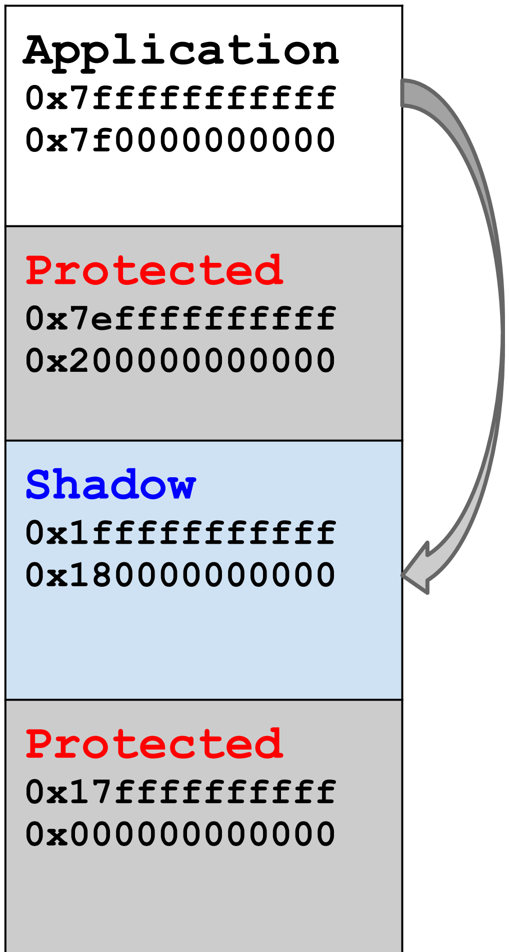
Compiler instrumentation

```
void foo(int *p) {
  *p = 42;
}
```

```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_address(0));
  __tsan_write4(p);
  *p = 42;
  __tsan_func_exit()
}
```

Direct shadow mapping (64-bit Linux)

```
Shadow = 4 * (Addr & kMask);
```

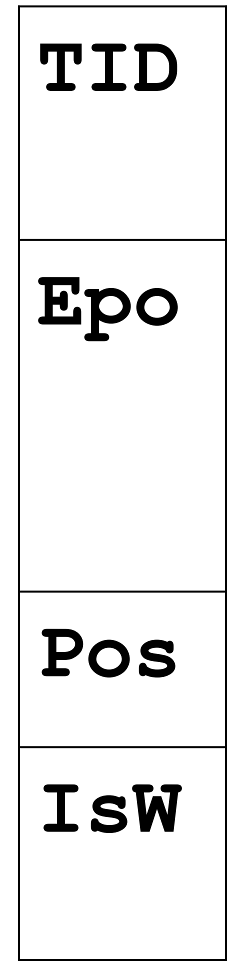


Shadow cell

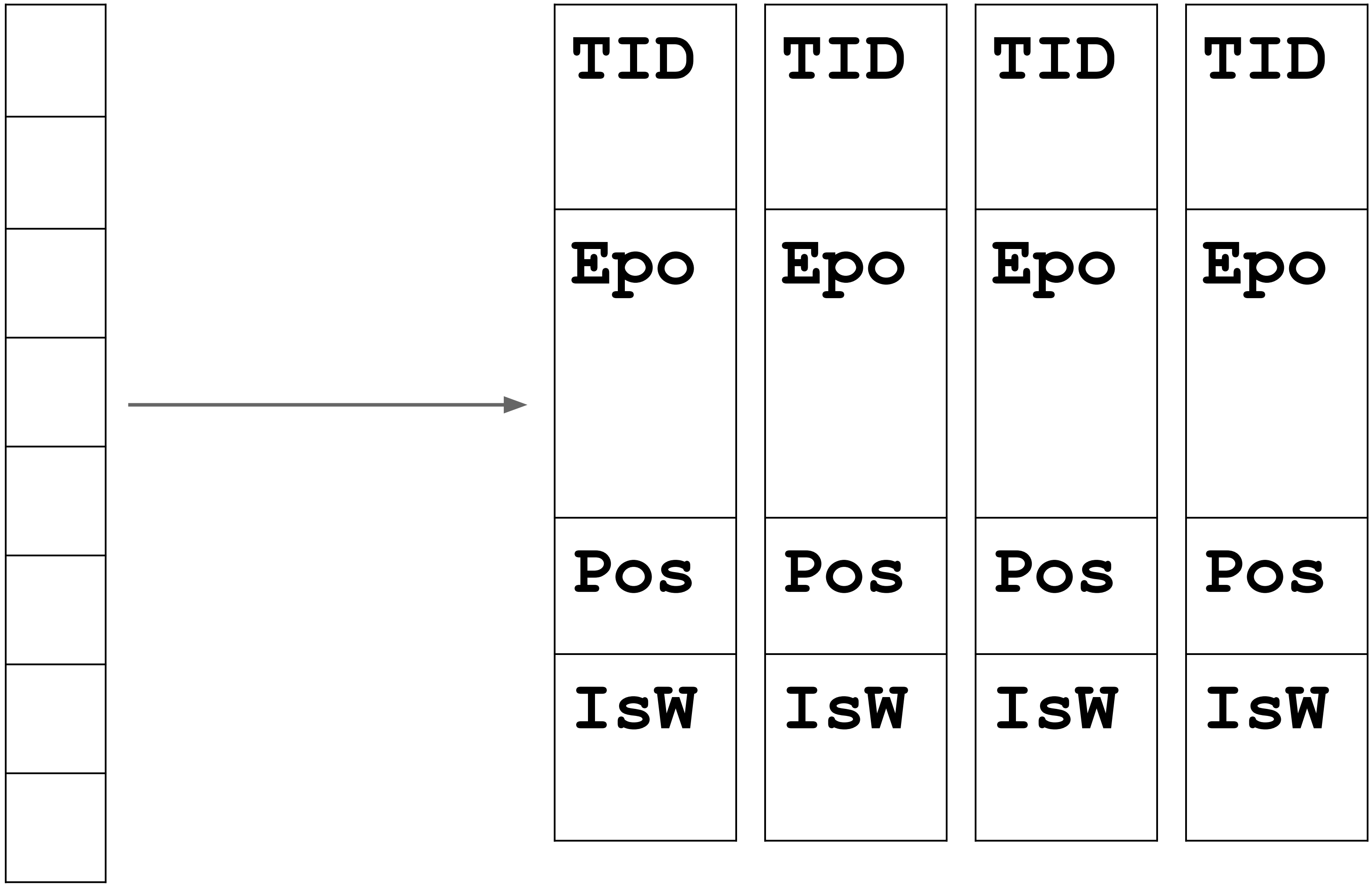
An 8-byte shadow cell represents one memory access:

- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

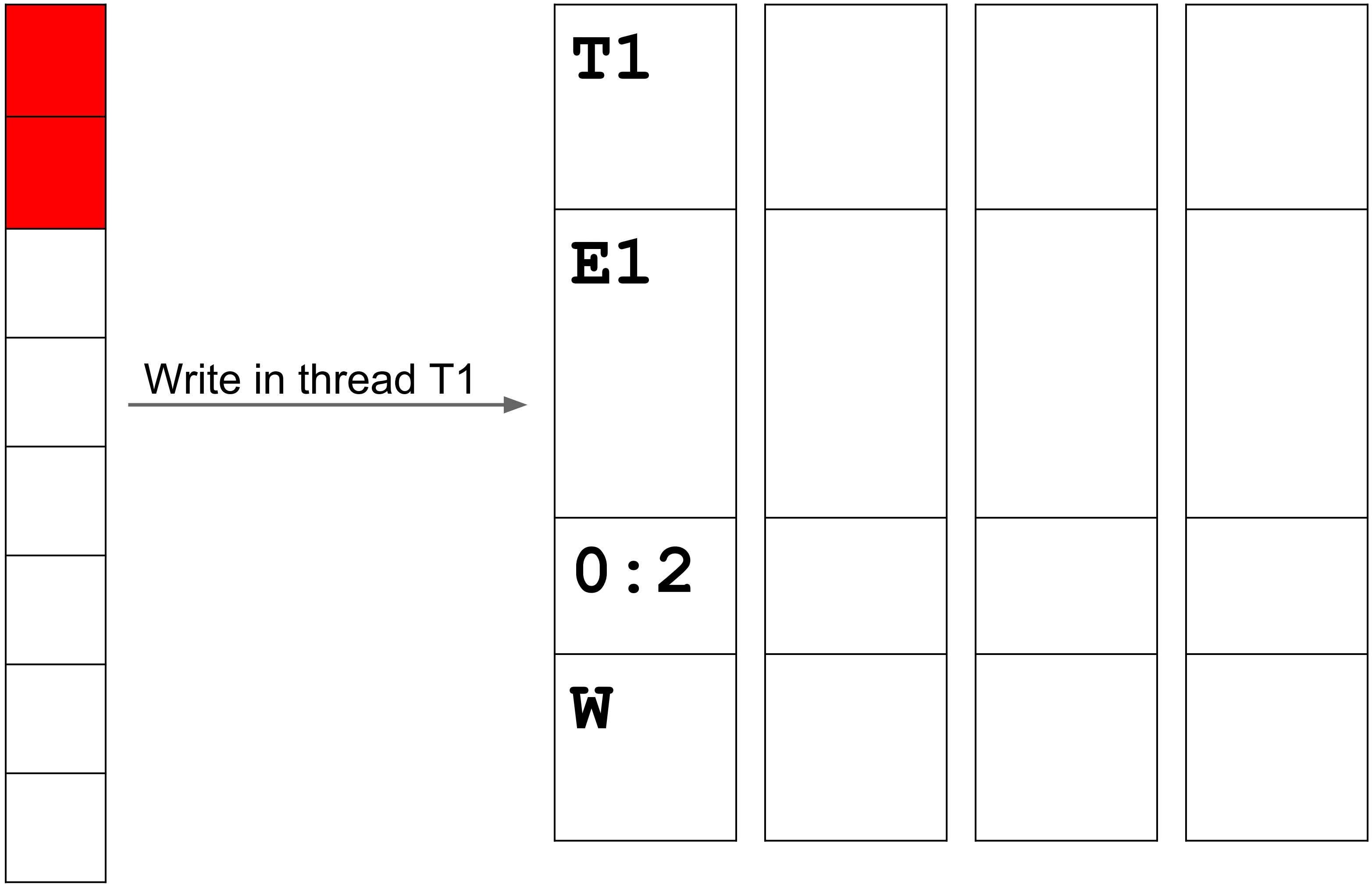
Full information (no more dereferences)



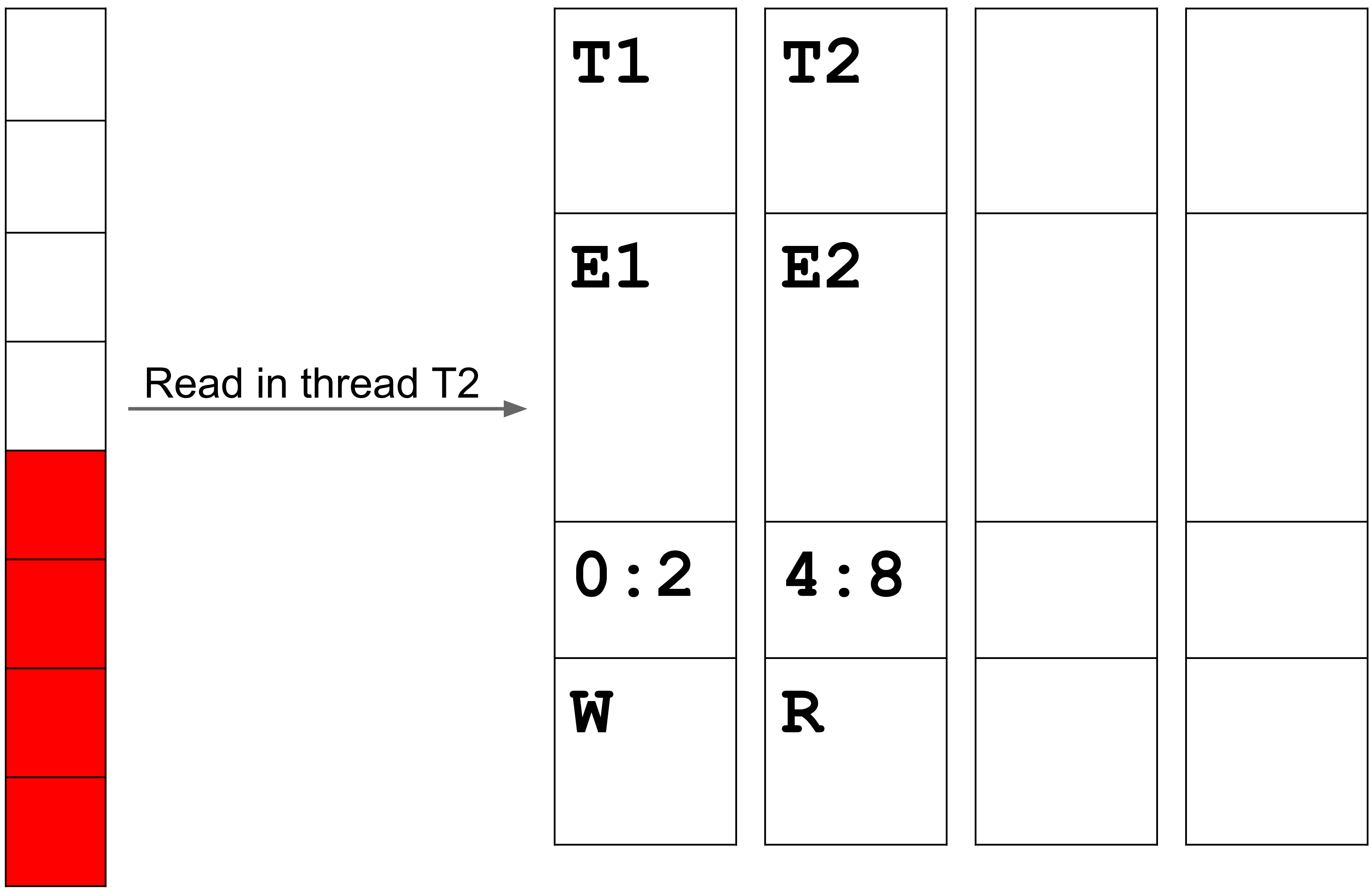
4 shadow cells per 8 app. bytes



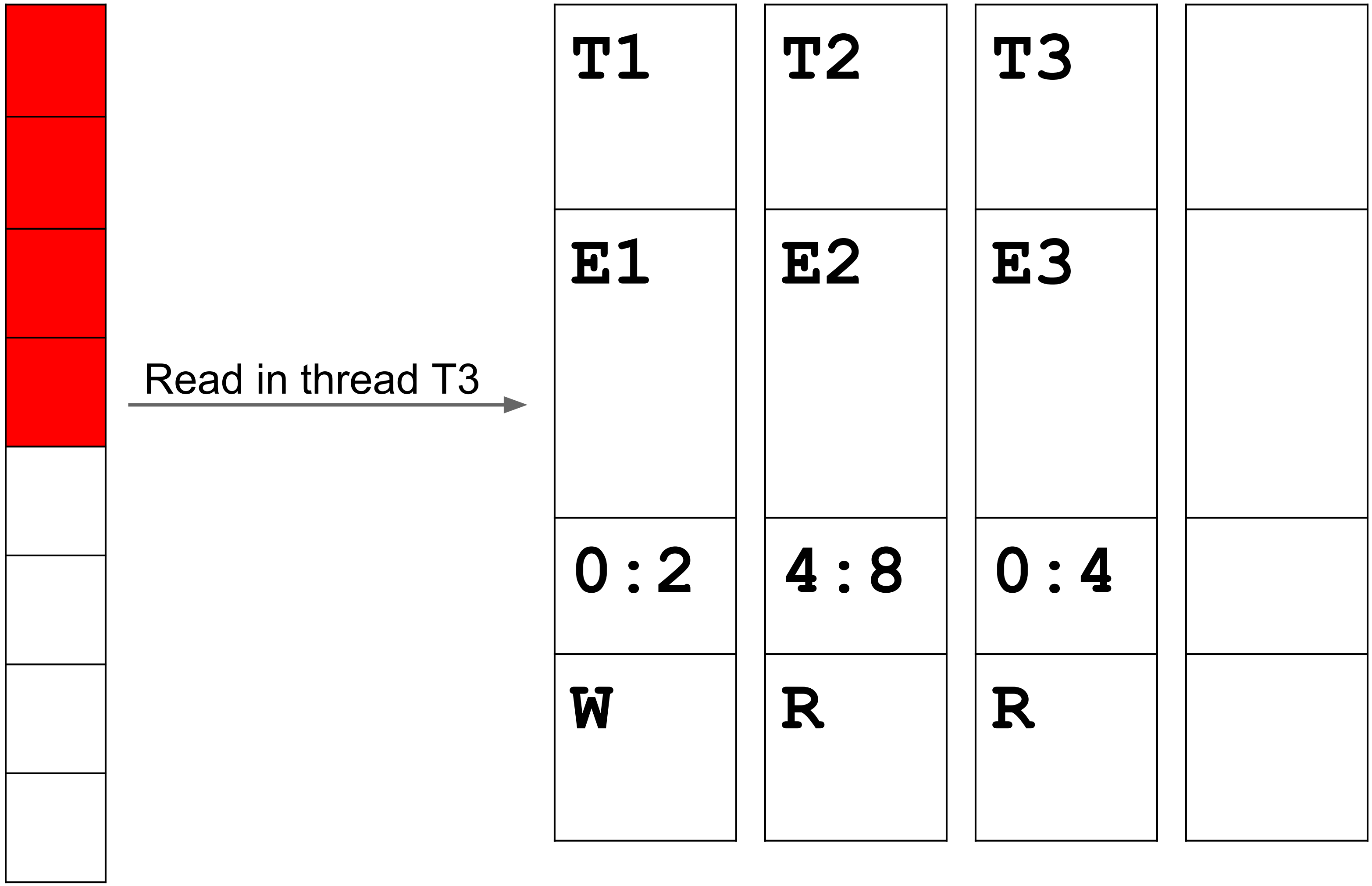
Example: first access



Example: second access



Example: third access



Example: race?

Race if **E1** does not "happen-before" **E3**

T1	T2	T3	
E1	E2	E3	
0:2	4:8	0:4	
W	R	R	

Fast happens-before

- Constant-time operation
  - Get TID and Epoch from the shadow cell
  - 1 load from thread-local storage
  - 1 comparison
- Similar to FastTrack (PLDI'09)

Shadow word eviction

- When all shadow cells are filled, one random cell is replaced

Informative reports

- Stack traces for two memory accesses:
  - current (easy)
  - previous (hard)

- TSan1:
  - Stores fixed number of frames (default: 10)
  - Information is never lost
  - Reference-counting and garbage collection

Stack trace for previous access

- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
  - Buffer size is configurable
- Replay the event buffer on report
  - Unlimited number of frames

Function interceptors

- 100+ interceptors
  - malloc, free, ...
  - pthread_mutex_lock, ...
  - strlen, memcmp, ...
  - read, write, ...

Atomics

- LLVM atomic instructions are replaced with __tsan_* callbacks

```
%0 = load atomic i8* %a acquire, align 1
```

```
%0 = call i8
@__tsan_atomic8_load(i8* %a, i32 504)
```

TSan slowdown vs clang -O1

Application	TSan1	TSan2	TSan1/TSan2
RPC benchmark	40x	7x	5.5x
Web server test	25x	2.5x	10x
String util test (1 thread)	50x	6x	8.5x

Trophies

- 200+ races in Google server-side apps (C++)
- 80+ races in Go programs
  - 25+ bugs in Go stdlib
- Several races in OpenSSL
  - 1 fixed, ~5 'benign'
- More to come
  - We've just started testing Chrome :)

Key advantages

- Speed
  - > 10x faster than other tools
- Native support for atomics
  - Hard or impossible to implement with binary translation (Helgrind, Intel Inspector)

Limitations

- Only 64-bit Linux
- Hard to port to 32-bit platforms
    - Small address space
    - Relies on atomic 64-bit load/store
- Heavily relies on TLS
    - Slow TLS on some platforms
- Does not instrument:
    - pre-built libraries
    - inline assembly

ThreadSanitizer, MemorySanitizer

Scalable run-time detection of uninitialized memory reads and data races with LLVM instrumentation

Timur Iskhodzhanov, Alexander Potapenko,
Alexey Samsonov, Kostya Serebryany,
Evgeniy Stepanov, Dmitry Vyukov

LLVM developers' meeting, Nov 8 2012

Agenda

- AddressSanitizer (aka ASan)
  - recap from 2011
  - detects use-after-free and buffer overflows (C++)
- ThreadSanitizer (aka TSan)
  - detects data races (C++ & Go)
- MemorySanitizer (aka MSan)
  - detects uninitialized memory reads (C++)
- Similar tools, find different kinds of bugs

AddressSanitizer (recap from 2011)

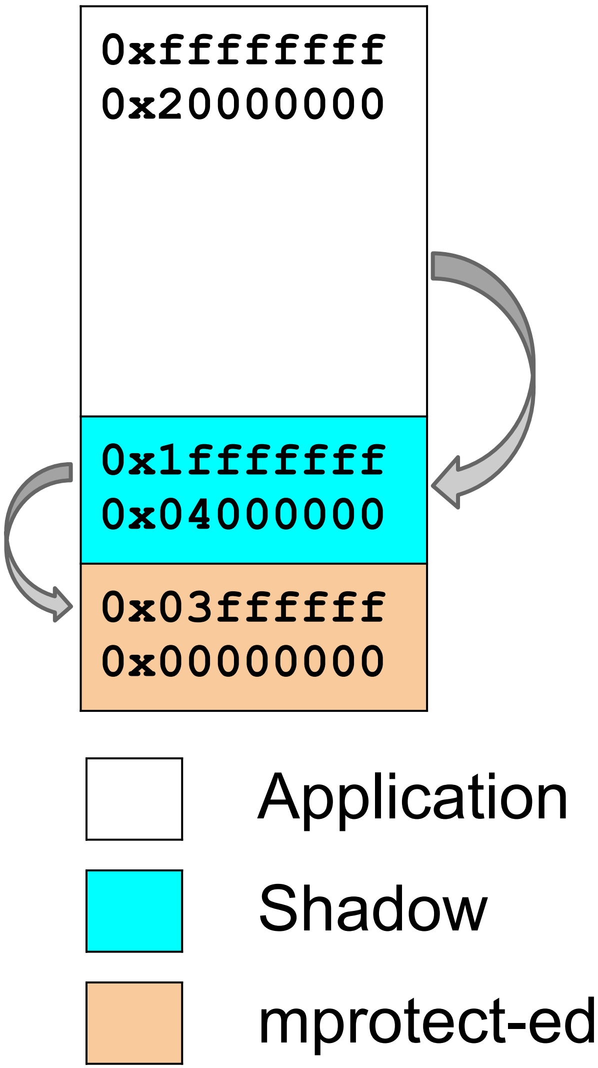
- Finds
  - buffer overflows (stack, heap, globals)
  - use-after-free
  - some more
- LLVM compiler module (~1KLOC)
  - instruments all loads/stores
  - inserts red zones around Alloca and GlobalVariables
- Run-time library (~10KLOC)
  - malloc replacement (redzones, quarantine)
  - Bookkeeping for error messages

ASan report example: use-after-free

```
int main(int argc, char **argv) {
 int *array = new int[100];
 delete [] array;
 return array[argc];  } // BOOM
% clang++ -O1 -fsanitize=address a.cc && ./a.out
==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
   #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte region
freed by thread T0 here:
   #0 0x4058fd in operator delete[](void*) _asan_rtl_
   #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
   #0 0x405579 in operator new[](unsigned long) _asan_rtl_
   #1 0x4042f3 in main a.cc:2
```

ASan shadow memory

Virtual address space



Instrumentation

```
*a = ...
```

```
char *shadow
 = addr >> 3;
if (*shadow)
  ReportError(a);
*a = ...
```

ASan *marketing* slide

- 2x slowdown (Valgrind: 20x and more)
- 1.5x-4x memory overhead
- 500+ bugs found in Chrome in 1.5 years
  - Used for tests and fuzzing, 2000+ machines 24/7
  - 100+ bugs by external researchers
- 1000+ bugs everywhere else
  - Firefox, FreeType, FFmpeg, WebRTC, libjpeg-turbo, Perl, Vim, LLVM, GCC, MySQL

Plea to hardware vendors

Trivial hardware support may reduce the overhead from 2x to 20%

ThreadSanitizer

data races

ThreadSanitizer v1

- Race detector based on Valgrind
- Used since early 2009
- Slow (20x–300x slowdown)
  - Still, found thousands races
  - Faster & more usable than others
    - Helgrind (Valgrind)
    - Intel Parallel Inspector (PIN)
- WBIA'09

ThreadSanitizer v2 overview

- Simple compile-time instrumentation
  - ~400 LOC
- Redesigned run-time library
  - Fully parallel
  - No expensive atomics/locks on fast path
  - Scales to huge apps
  - Predictable memory footprint
  - Informative reports

TSan report example: data race

```
void Thread1() { Global = 42; }
int main() {
  pthread_create(&t, 0, Thread1, 0);
  Global = 43;
  ...
% clang -fsanitize=thread -g a.c -fPIE -pie && ./a.out
WARNING: ThreadSanitizer: data race (pid=20373)
  Write of size 4 at 0x7f... by thread 1:
    #0 Thread1 a.c:1
  Previous write of size 4 at 0x7f... by main thread:
    #0 main a.c:4
  Thread 1 (tid=20374, running) created at:
    #0 pthread_create ??:0
    #1 main a.c:3
```

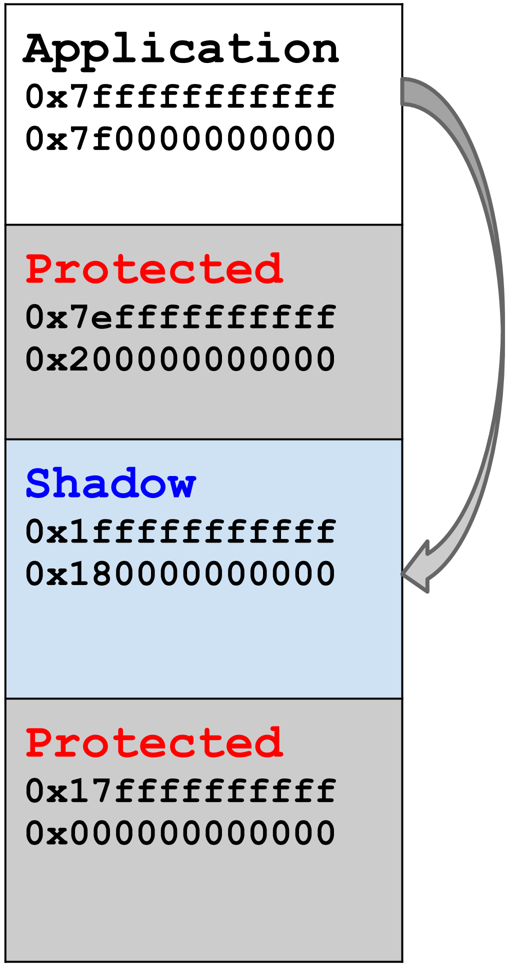
Compiler instrumentation

```
void foo(int *p) {
  *p = 42;
}
```

```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_address(0));
  __tsan_write4(p);
  *p = 42;
  __tsan_func_exit()
}
```

Direct shadow mapping (64-bit Linux)

```
Shadow = 4 * (Addr & kMask);
```

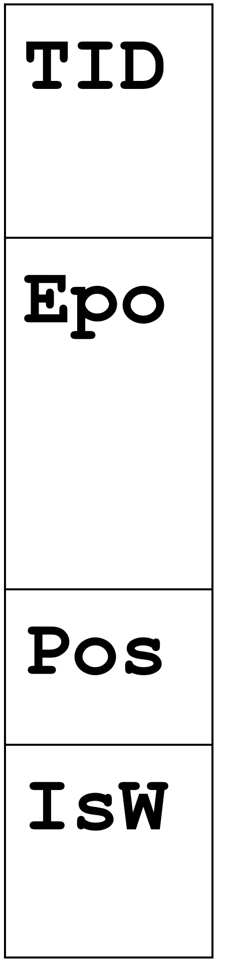


Shadow cell

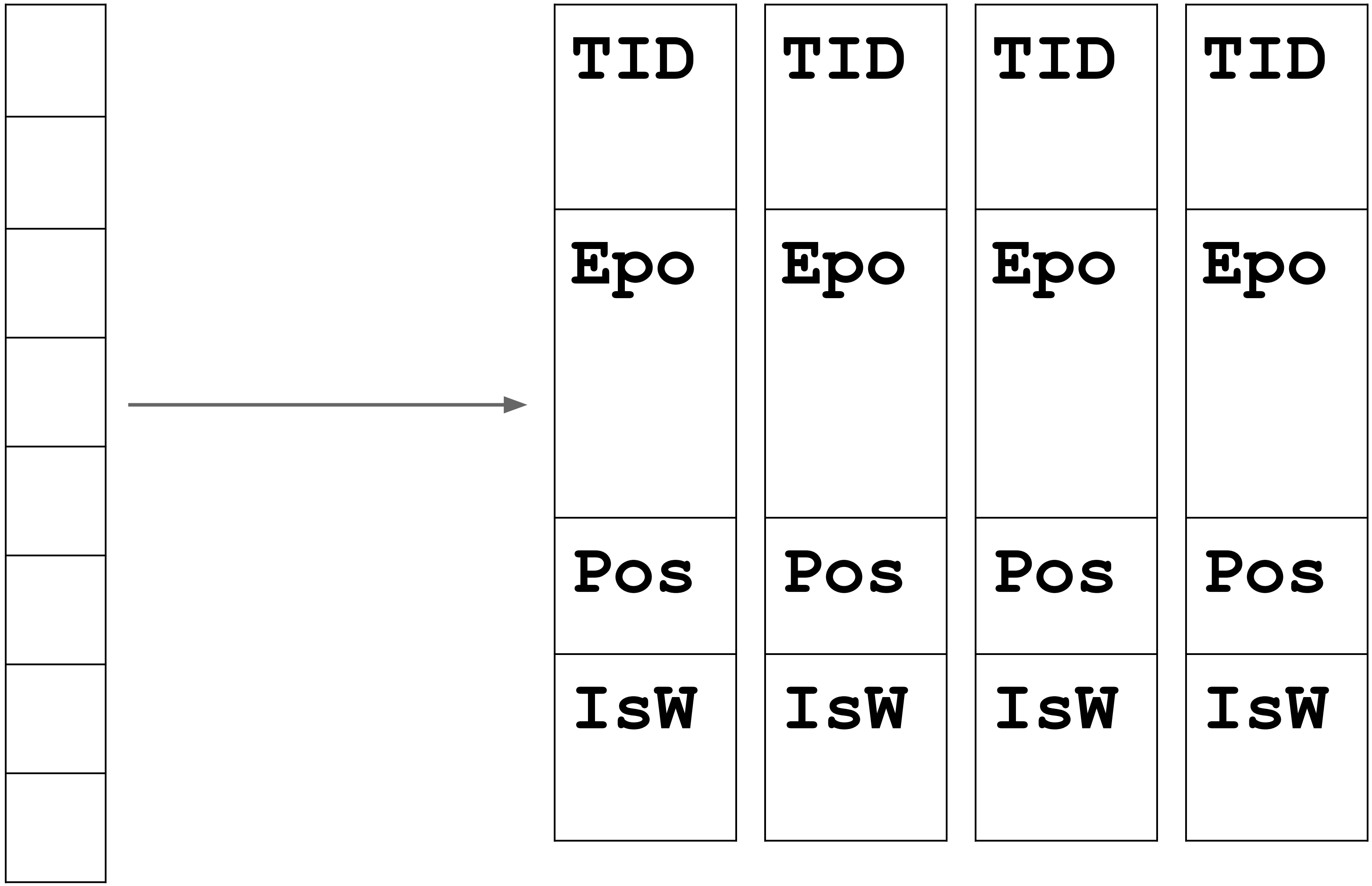
An 8-byte shadow cell represents one memory access:

- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

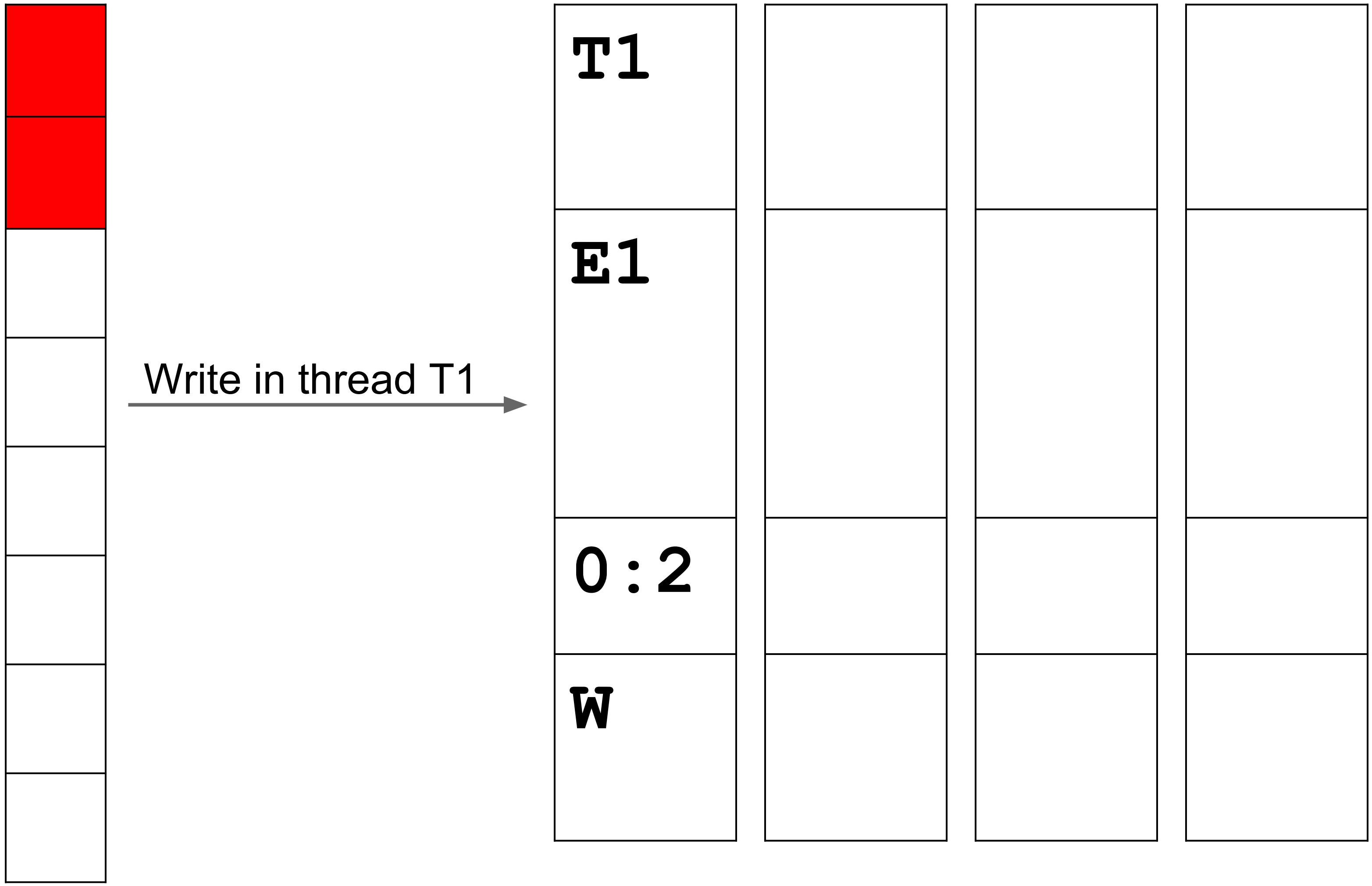
Full information (no more dereferences)



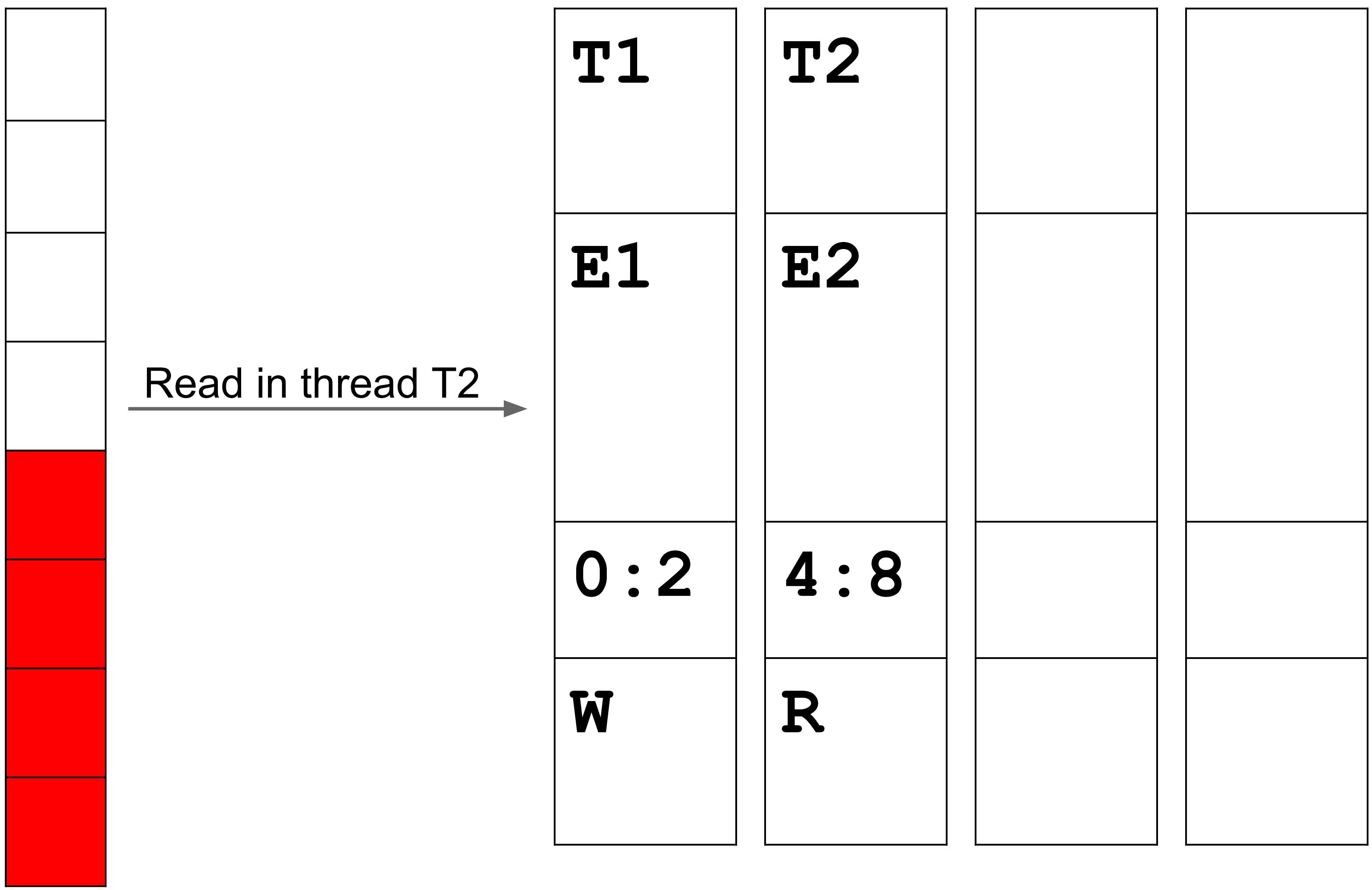
4 shadow cells per 8 app. bytes



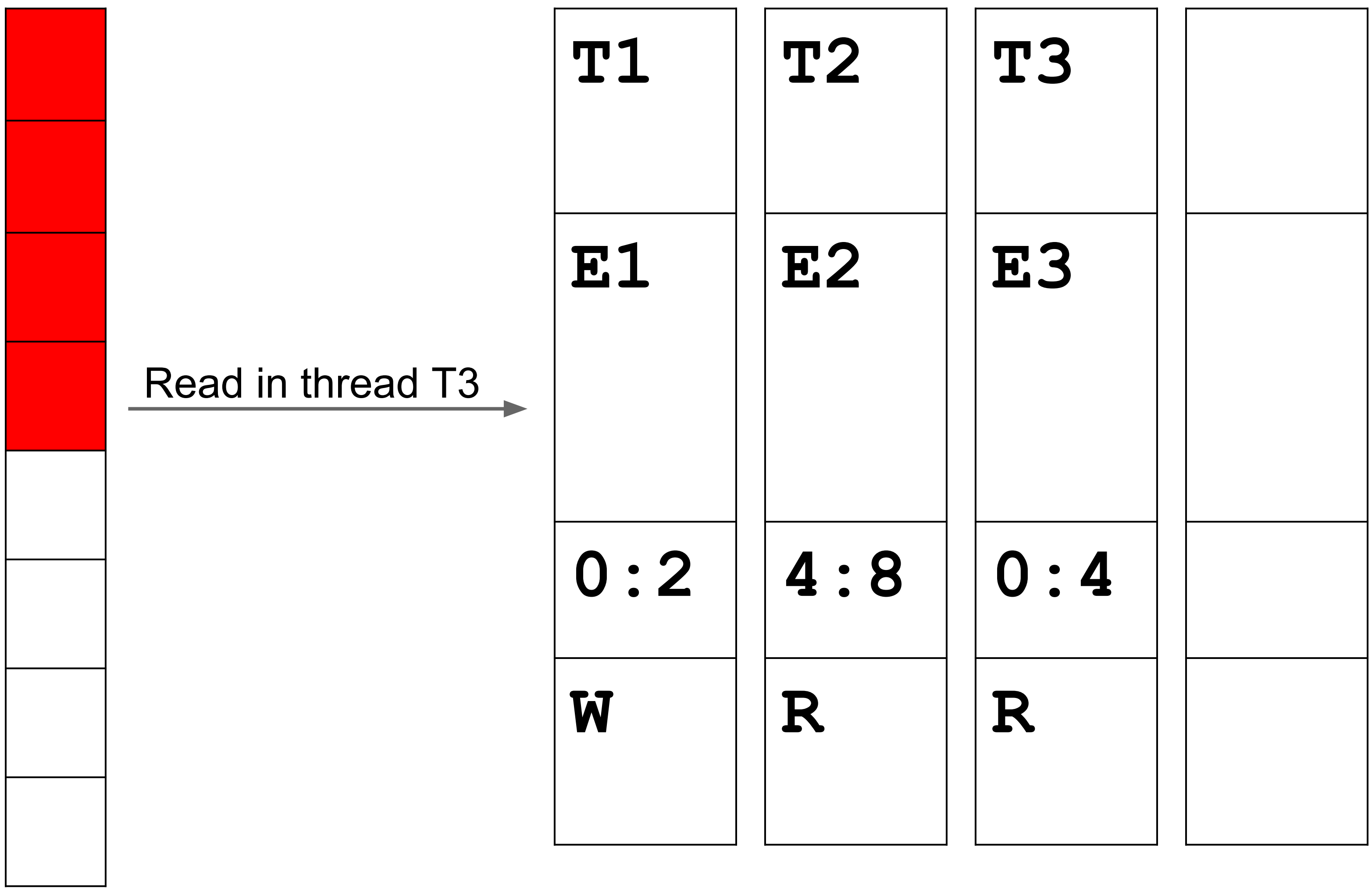
Example: first access



Example: second access



Example: third access



Example: race?

Race if **E1** does not "happen-before" **E3**

T1	**T2**	**T3**	
E1	**E2**	**E3**	
0:2	**4:8**	**0:4**	
W	**R**	**R**	

Fast happens-before

- Constant-time operation
  - Get TID and Epoch from the shadow cell
  - 1 load from thread-local storage
  - 1 comparison
- Similar to FastTrack (PLDI'09)

Shadow word eviction

- When all shadow cells are filled, one random cell is replaced

Informative reports

- Stack traces for two memory accesses:
  - current (easy)
  - previous (hard)

- TSan1:
  - Stores fixed number of frames (default: 10)
  - Information is never lost
  - Reference-counting and garbage collection

Stack trace for previous access

- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
  - Buffer size is configurable
- Replay the event buffer on report
  - Unlimited number of frames

Function interceptors

- 100+ interceptors
    - malloc, free, ...
    - pthread_mutex_lock, ...
    - strlen, memcmp, ...
    - read, write, ...

Atomics

- LLVM atomic instructions are replaced with __tsan_* callbacks

```
%0 = load atomic i8* %a acquire, align 1
```

⇩

```
%0 = call i8
@__tsan_atomic8_load(i8* %a, i32 504)
```

TSan slowdown vs clang -O1

Application	TSan1	TSan2	TSan1/TSan2
RPC benchmark	40x	7x	5.5x
Web server test	25x	2.5x	10x
String util test (1 thread)	50x	6x	8.5x

Trophies

- 200+ races in Google server-side apps (C++)
- 80+ races in Go programs
  - 25+ bugs in Go stdlib
- Several races in OpenSSL
  - 1 fixed, ~5 'benign'
- More to come
  - We've just started testing Chrome :)

Key advantages

- Speed
  - > 10x faster than other tools
- Native support for atomics
  - Hard or impossible to implement with binary translation (Helgrind, Intel Inspector)

Limitations

- Only 64-bit Linux
- Hard to port to 32-bit platforms
  - Small address space
  - Relies on atomic 64-bit load/store
- Heavily relies on TLS
  - Slow TLS on some platforms
- Does not instrument:
  - pre-built libraries
  - inline assembly

MemorySanitizer

uninitialized memory reads (UMR)

MSan report example: UMR

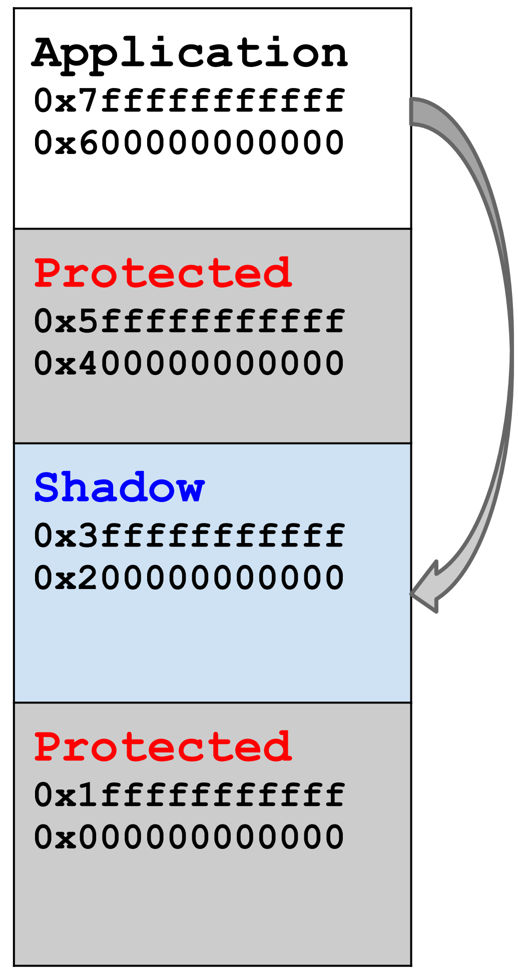
```
int main(int argc, char **argv) {
  int x[10];
  x[0] = 1;
  if (x[argc]) return 1;
  ...
% clang -fsanitize=memory -fPIE -pie  a.c -g
% ./a.out
WARNING: MemorySanitizer: UMR (uninitialized-memory-read)
    #0 0x7ff6b05d9ca7 in main stack_umr.c:4
  ORIGIN: stack allocation: x@main
```

Shadow memory

- Bit to bit shadow mapping
    - 1 means 'poisoned' (uninitialized)
- Uninitialized memory:
    - Returned by malloc
    - Local stack objects (poisoned at function entry)
- Shadow is propagated through arithmetic operations and memory writes
- Shadow is unpoisoned when constants are stored

Direct 1:1 shadow mapping

```
Shadow = Addr - 0x4000000000000;
```



Shadow propagation

- Reporting UMR on first read causes false positives
  - E.g. copying **`struct {char x; int y;}`**
- Report UMR only on some uses (branch, syscall, etc)
  - That's what Valgrind does
- Propagate shadow values through expressions
  - A = B + C: A' = B' | C'
  - A = B & C: A' = (B' & C') | (~B & C') | (B' & ~C)
  - Approximation to minimize false positives/negatives
  - Similar to Valgrind
- Function parameter/retval: shadow is stored in TLS
  - Valgrind shadows registers/stack instead

Tracking origins

- Where was the poisoned memory allocated?

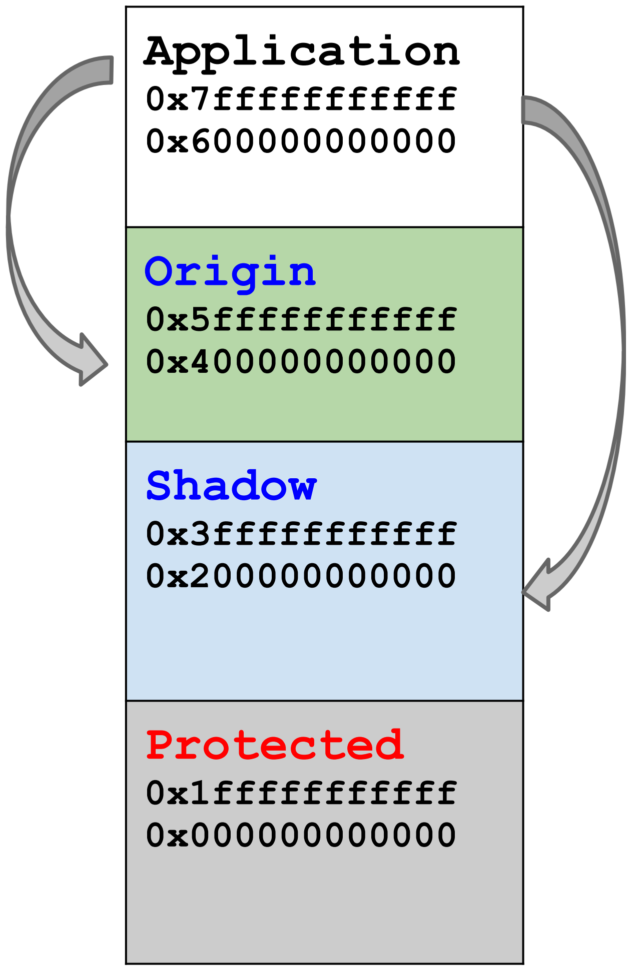
  `a = malloc() ...`
  `b = malloc() ...`
  `c = *a + *b  ...`
  `if (c) ...  //` UMR. Is 'a' guilty or 'b'?

- Valgrind **`--track-origins`**: propagate the origin of the poisoned memory alongside the shadow

- MemorySanitizer: secondary shadow
  - Origin-ID is 4 bytes, 1:1 mapping
  - 2x additional slowdown

Secondary shadow (origin)

```
Origin = Addr - 0x200000000000;
```



MSan overhead

- Without origins:
  - CPU: 3x
  - RAM: 2x
- With origins:
  - CPU: 6x
  - RAM: 3x + malloc stack traces

Tricky part :(

- Missing any write instruction causes false reports
- Must monitor ALL stores in the program
  - libc, libstdc++, syscalls, etc

Solutions:

- Instrumented libc++, wrappers for libc
  - Works for many "console" apps, e.g. LLVM
- Instrument libraries at run-time
  - DynamoRIO-based prototype (SLOW)
- Instrument libraries statically (is it possible?)
- Compile everything, wrap syscalls
  - Will help AddressSanitizer/ThreadSanitizer too

MSan trophies

- Proprietary console app, 1.3 MLOC in C++
  - Not tested with Valgrind previously
  - 20+ unique bugs in < 2 hours
  - Valgrind finds the same bugs in 24+ hours
  - MSan gives better reports for stack memory
- 1 Bug in LLVM
  - LLVM bootstraps, ready to set regular runs
- A few bugs in Chrome (just started)
  - Have to use DynamoRIO module (MSanDR)
  - 7x faster than Valgrind

Summary (all 3 tools)

- AddressSanitizer (memory corruption)
  - A "must use" for everyone (C++)
  - Supported on Linux, OSX, CrOS, Android,
  - WIP: iOS, Windows, *BSD (?)
- ThreadSanitizer (races)
  - A "must use" if you have threads (C++, Go)
  - Only x86_64 Linux
- MemorySanitizer (uses of uninitialized data)
  - WIP, usable for "console" apps (C++)
  - Only x86_64 Linux

Q&A

http://code.google.com/p/address-sanitizer/

http://code.google.com/p/thread-sanitizer/

http://code.google.com/p/memory-sanitizer/

ASan/MSan vs Valgrind (Memcheck)

	Valgrind	ASan	MSan
Heap out-of-bounds	YES	YES	NO
Stack out-of-bounds	NO	YES	NO
Global out-of-bounds	NO	YES	NO
Use-after-free	YES	YES	NO
Use-after-return	NO	Sometimes	NO
Uninitialized reads	YES	NO	YES
CPU Overhead	10x-300x	1.5x-3x	3x

Why not a single tool?

- Slowdowns will add up
  - Bad for interactive or network apps
- Memory overheads will multiply
  - ASan redzone vs TSan/MSan large shadow
- Not trivial to implement