DYNAMIC ANALYSES FOR DATA RACE DETECTION

John Erickson
Microsoft

Stephen Freund
Williams College

Madan Musuvathi
Microsoft Research

Introductions…

Tutorial Goals

- What is (and is not) a data race
- State of the art techniques in dynamic data race detection
- Implementation insights
- Open research problems

Tutorial Structure

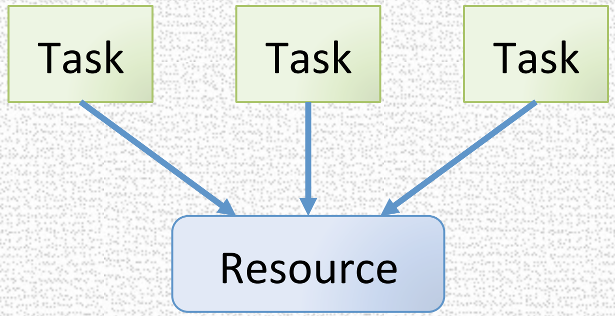
- Background
- Lockset and Happens-before Algorithms
- FastTrack – In Depth
- Implementation Frameworks
- RoadRunner – In Depth
- DataCollider – Sampling for Data-race Detection
- Advanced Schedule Perturbation
  - Cuzz
  - Adversarial Memory Models

BACKGROUND

Concurrency vs Parallelism

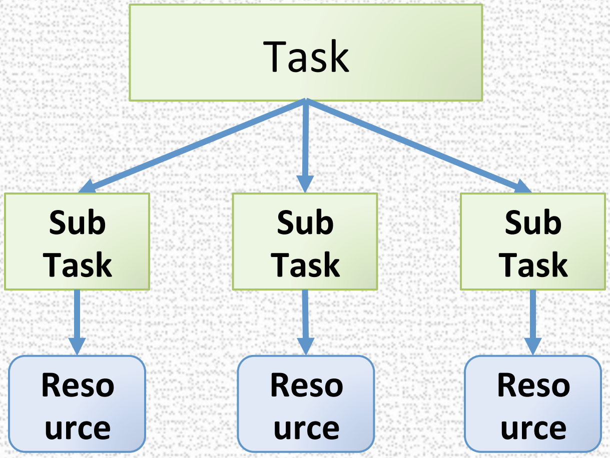
Concurrency

Manage access to shared resources, correctly and efficiently



Parallelism

Use extra resources to solve a problem faster

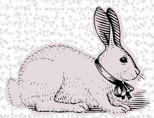


Sometimes, creating parallelism will create concurrency

Threads and Shared Memory

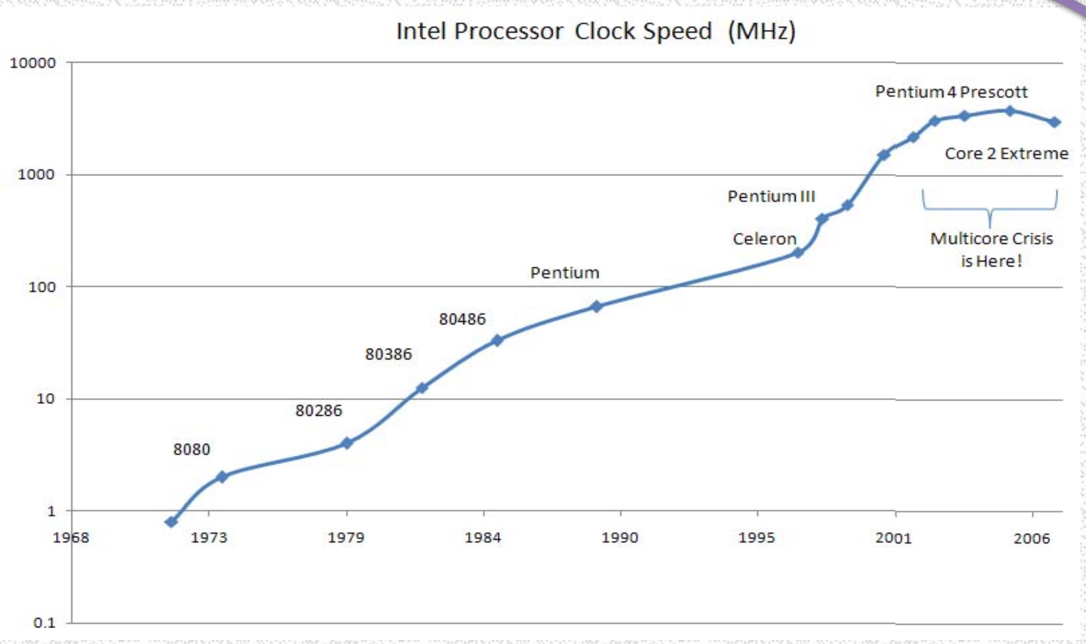
- Common programming model
  - For dealing with parallelism and concurrency
- Tasks vs Threads
  - Subtly different abstractions
  - Threads are usually managed by the operating system
  - Tasks are lightweight, and are usually provided by the language runtime
- Data races can occur
  - In both threads and task based programming
  - When dealing with both parallelism and concurrency

Moore’s law



Moore’s law
make computers faster

Moore’s law now
produces more cores



Open Research Problems

- Make concurrency and parallelism accessible to all
- Other Programming models
- How to write efficient multi-threaded code
- How to write correct multi-threaded code ⟵ This tutorial

First things First
Assigning Semantics to Concurrent Programs

int X = F = 0;

```
X = 1;
F = 1;
```

```
t = F;
u = X;
```

- What does this program mean?
- Sequential Consistency [Lamport ‘79]

  Program behavior = set of its thread interleavings

Sequential Consistency Explained

int X = F = 0; // F = 1 implies X is initialized

```
X = 1;
F = 1;
```

```
t = F;
u = X;
```

X = 1;	X = 1;	X = 1;	t = F;	t = F;	t = F;
F = 1;	t = F;	t = F;	u = X;	X = 1;	X = 1;
t = F;	F = 1;	u = X;	X = 1;	u = X;	F = 1;
u = X;	u = X;	F = 1;	F = 1;	F = 1;	u = X;
t=1, u=1	t=0, u=1	t=0, u=1	t=0, u=0	t=0, u=1	t=0, u=1

t=1 implies u=1

Naturalness of Sequential Consistency

- Sequential Consistency provides two crucial abstractions
- Program Order Abstraction
  - Instructions execute in the order specified in the program

A ; B

means "Execute A and then B"

- Shared Memory Abstraction
  - Memory behaves as a global array, with reads and writes done immediately
- We implicitly assume these abstractions for sequential programs

In this Tutorial

- We will assume Sequential Consistency
- Except when explicitly stated otherwise

Common Concurrency Errors

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

Atomicity Violation

- Code that is meant to execute "atomically"
  - That is, without interference from other threads
- Suffers interference from some other thread

Thread 1

```
void Bank::Update(int a)
{
    int t = bal;
    bal = t + a;
}
```

Thread 2

```
void Bank::Withdraw(int a)
{
    int t = bal;
    bal = t - a;
}
```

Ordering Violation

- Incorrect signalling between a producer and a consumer

Thread 1

```
work = null;
CreateThread (Thread 2);
work = new Work();
```

Thread 2

```
ConsumeWork( work );
```

Unintended Sharing

- Threads accidentally sharing objects

Thread 1

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

Thread 2

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

Deadlock / Livelock

Thread 1

```
AcquireLock( X );
AcquireLock( Y );
```

Thread 2

```
AcquireLock( Y );
AcquireLock( X );
```

Deadlock / Livelock

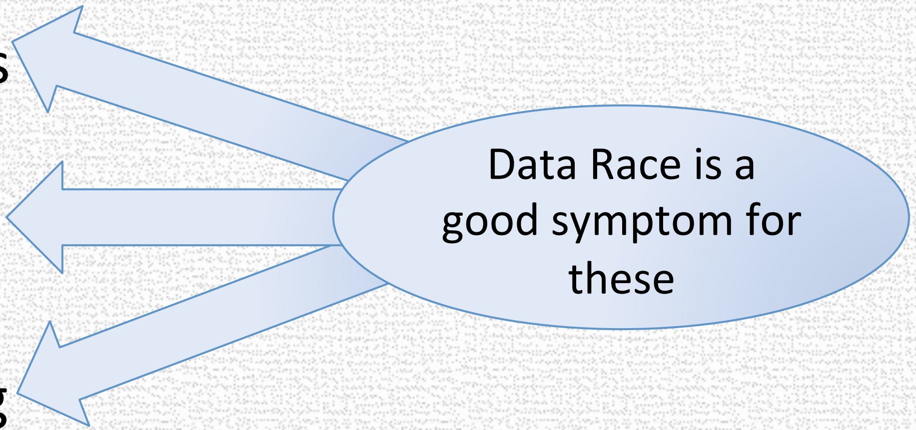
```
Init
x = y = 0;
```

```
Thread 1

while (x == 0) {}

y = 1;
```

```
Thread 2

while (y == 0) {}

x = 1;
```

Common Concurrency Errors

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



WHAT IS A DATA RACE ?

- The term “data race” is often overloaded to mean different things
- Precise definition is important in designing a tool

Data Race

- Two accesses *conflict* if
  - they access the same memory location, and
  - at least one of them is a write

Write X – Write X

Write X – Read X

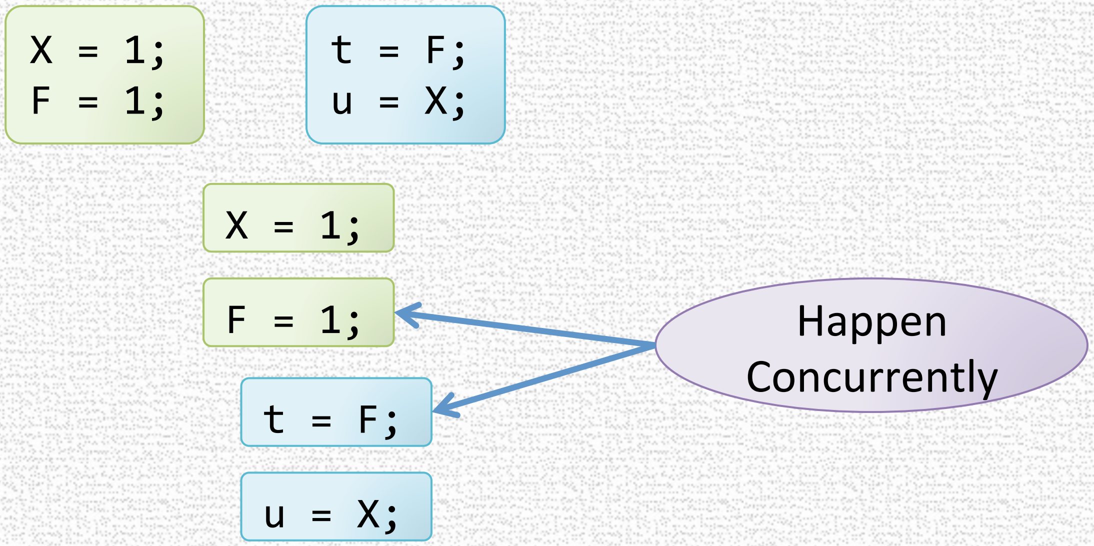
Read X – Write X

Read X – Read X

- A data race is a pair of conflicting accesses that happen concurrently

"Happen Concurrently"

- A and B happen concurrently if
- there exists a sequentially consistent execution in which they happen one after the other



Unintended Sharing

- Threads accidentally sharing objects

Thread 1

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

Thread 2

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

Data Race

Atomicity Violation

- Code that is meant to execute “atomically”
  - That is, without interference from other threads
- Suffers interference from some other thread

Thread 1

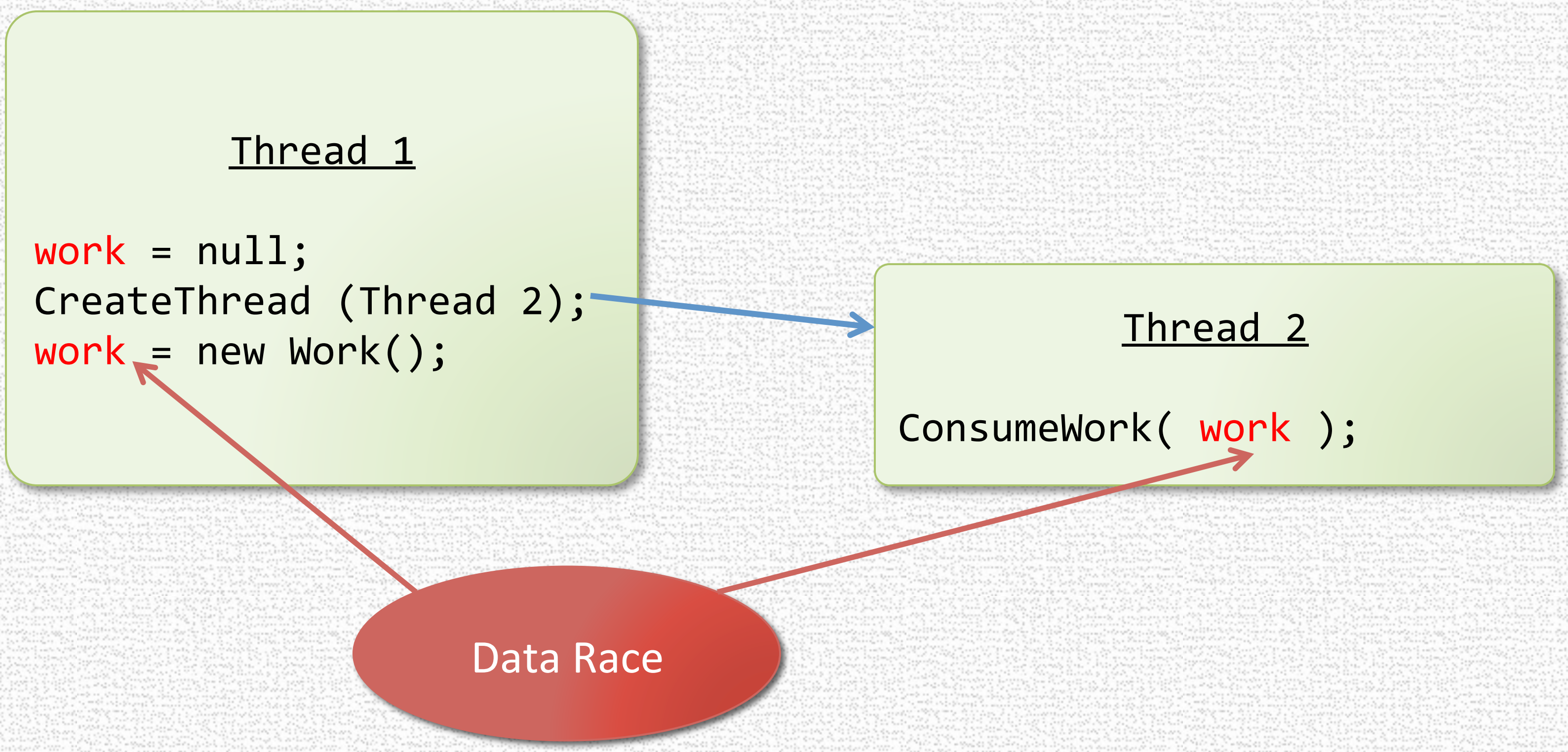
```
void Bank::Update(int a)
{
    int t = bal;
    bal = t + a;
}
```

Thread 2

```
void Bank::Withdraw(int a)
{
    int t = bal;
    bal = t - a;
}
```

Ordering Violation

- Incorrect signalling between a producer and a consumer



But,….

```
AcquireLock(){
     while (lock == 1) {}
     CAS (lock, 0, 1);
}
```

```
ReleaseLock() {
     lock = 0;
}
```

Data Race ?

Acceptable Concurrent Conflicting Accesses

- Implementing synchronization (such as locks) usually requires concurrent conflicting accesses to shared memory
- Innovative uses of shared memory
  - Fast reads
  - Double-checked locking
  - Lazy initialization
  - Setting dirty flag
  - ...
- Need mechanisms to distinguish these from erroneous conflicts

Solution: Programmer Annotation

- Programmer explicitly annotates variables as "synchronization"
  - Java – volatile keyword
  - C++ – std::atomic<> types

Data Race

- Two accesses *conflict* if
  - they access the same memory location, and
  - at least one of them is a write
- A data race is a pair of concurrent conflicting accesses to locations not annotated as synchronization

Data Race vs Race Conditions

- Data Races != Race Conditions
  - Confusing terminology
- Race Condition
  - Any timing error in the program
  - Due to events, device interaction, thread interleaving, …
- Data races are neither sufficient nor necessary for a race condition
  - Data race is a good symptom for a race condition

DATA-RACE-FREEDOM SIMPLIFIES LANGUAGE SEMANTICS

Advantage of Annotating All Data Races

- Defining semantics for concurrent programs becomes surprisingly easy
- In the presence of compiler and hardware optimizations

Can A Compiler Do This?

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

→

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

OK for sequential programs
if X is not modified between L1 and L3

t,u,v are local variables
X,Y are possibly shared

Can Break Sequential Consistent Semantics

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

```
M1: X = 1;
M2: Y = 1;
```

u == 1 ➔ v == 5

Data Race

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

```
M1: X = 1;
M2: Y = 1;
```

possibly u == 1 && v == 0

Can A Compiler Do This?

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

→

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

OK for sequential programs
if X is not modified between L1 and L3

OK for concurrent programs
if there is no data race on X or
if there is no data race on Y

t,u,v are local variables
X,Y are possibly shared

Key Observation [Adve& Hill '90]

- Many sequentially valid (compiler & hardware) transformations also preserve sequential consistency
- Provided the program is data-race free

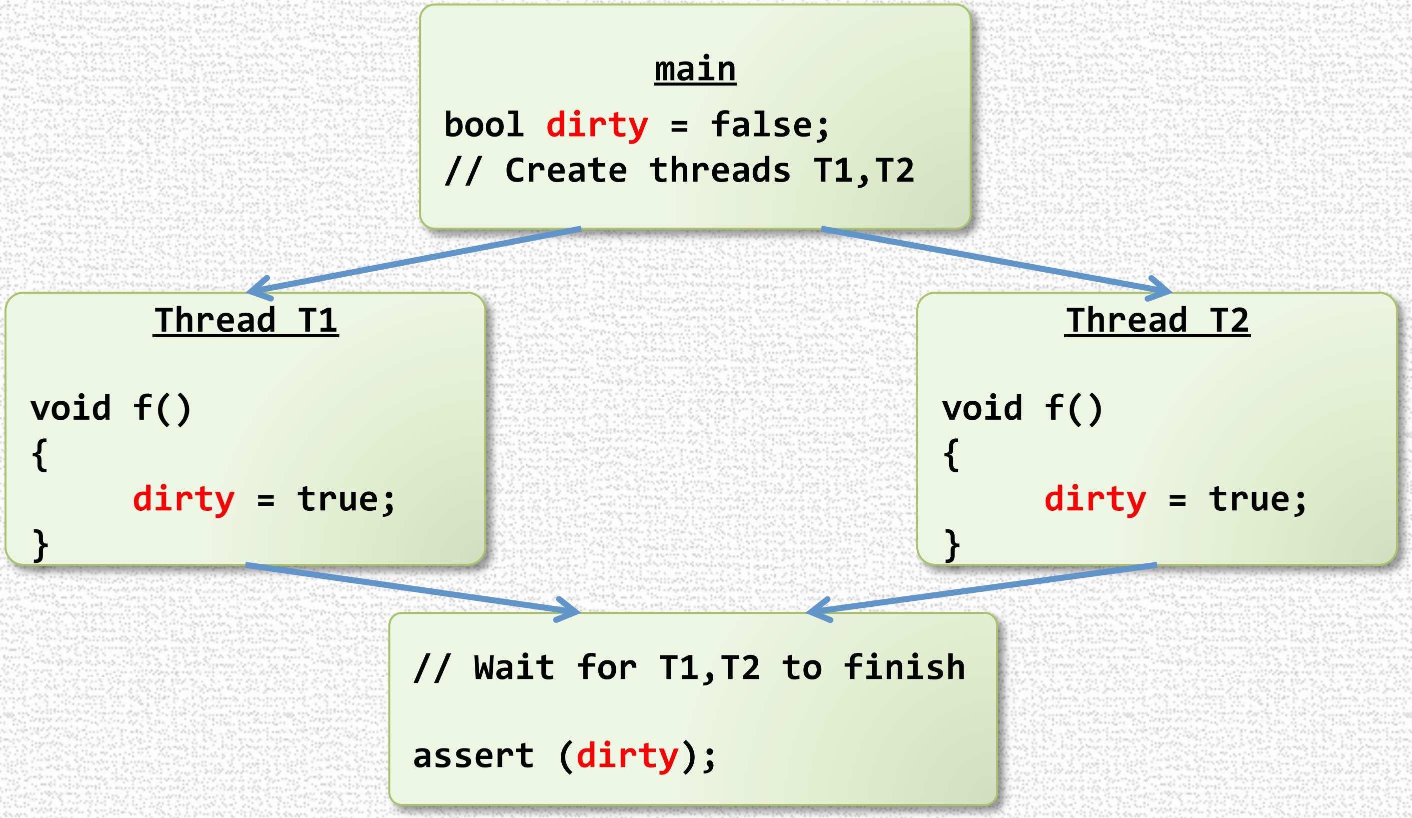
- Forms the basis for modern C++, Java semantics

data-race-free → sequential consistency

otherwise → weak/undefined semantics

A Quiz

- Can the assertion fire in this C++ program?



DATA RACE DETECTION

Overview of Data Race Detection Techniques

- Static data race detection
- Dynamic data race detection
  - Lock-set
  - Happen-before
  - DataCollider

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
  - 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

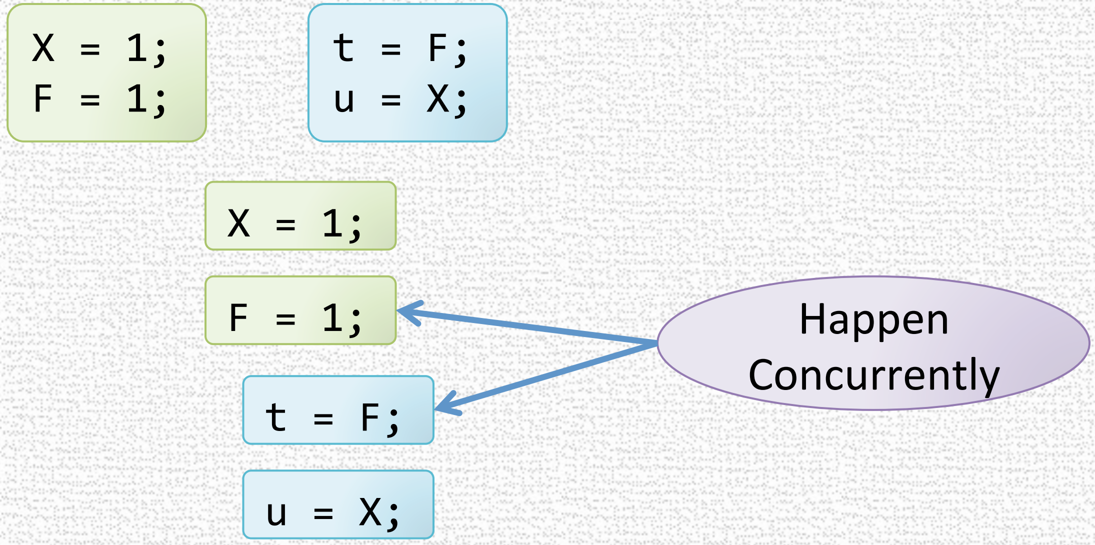
- Advantages
  - Can avoid “false positives”
  - No need for language extensions or sophisticated static analysis
- Disadvantages
  - Run-time overhead (5-20x for best tools)
  - Memory overhead for analysis state
  - Reasons only about observed executions
    - sensitive to test coverage
    - (some generalization possible...)

Tradeoffs: Static vs Dynamic

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

Definition Refresh

- A data race is a pair of concurrent conflicting accesses to unannotated locations



- Problem for dynamic data race detection
  - Very difficult to catch the two accesses executing concurrently

Solution

- Lockset
  - Infer data races through violation of locking discipline
- Happens-before
  - Infer data races by generalizing a trace to a set of traces with the same happens-before relation
- DataCollider
  - Insert delays intelligently to force the data race to occur

LOCKSET ALGORITHM

Eraser [Savage et.al. ‘97]

Lockset Algorithm Overview

- Checks a sufficient condition for data-race-freedom
- Consistent locking discipline
  - Every data structure is protected by a single lock
  - All accesses to the data structure made while holding the lock
- Example:

```
// Remove a received packet
AcquireLock( RecvQueueLk );
pkt = RecvQueue.RemoveTop();
ReleaseLock( RecvQueueLk );

… // process pkt

// Insert into processed
AcquireLock( ProcQueueLk );
ProcQueue.Insert(pkt);
ReleaseLock( ProcQueueLk );
```

RecvQueue is consistently protected by RecvQueueLk

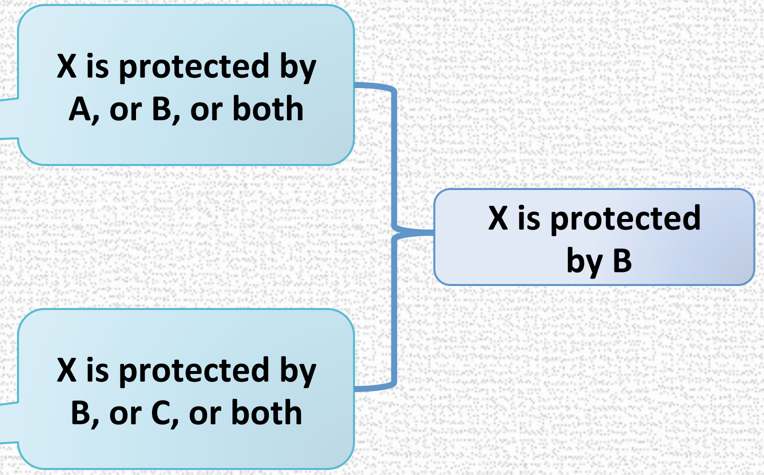
ProcQueue is consistently protected by ProcQueueLk

Inferring the Locking Discipline

- How do we know which lock protects what?
  - Asking the programmer is cumbersome

- Solution: Infer from the program

```
AcquireLock( A );
AcquireLock( B );
x ++;
ReleaseLock( B );
ReleaseLock( A );


AcquireLock( B );
AcquireLock( C );
x ++;
ReleaseLock( C );
ReleaseLock( B );
```



LockSet Algorithm

- Two data structures:
  - LocksHeld(t) = set of locks held currently by thread t
    - Initially set to Empty
  - LockSet(x) = set of locks that could potentially be protecting x
    - Initially set to the universal set

- When thread t acquires lock l
  - $LocksHeld(t) = LocksHeld(t) \cup \{l\}$
- When thread t releases lock l
  - $LocksHeld(t) = LocksHeld(t) - \{l\}$

- When thread t accesses location x
  - $LockSet(x) = LockSet(x) \cap LocksHeld(t)$
  - Report "data race" when LockSet(x) becomes empty

Algorithm Guarantees

- No warnings → no data races on the current execution
  - The program followed consistent locking discipline in this execution
- Warnings does not imply a data race
  - Thread-local initialization

```
// Initialize a packet
pkt = new Packet();
pkt.Consumed = 0

AcquireLock( SendQueueLk );
pkt = SendQueue.Top();
ReleaseLock( SendQueueLk );
```

```
// Process a packet
AcquireLock( SendQueueLk );
pkt = SendQueue.Top();
pkt.Consumed = 1;
ReleaseLock( SendQueueLk );
```

LockSet Algorithm Guarantees

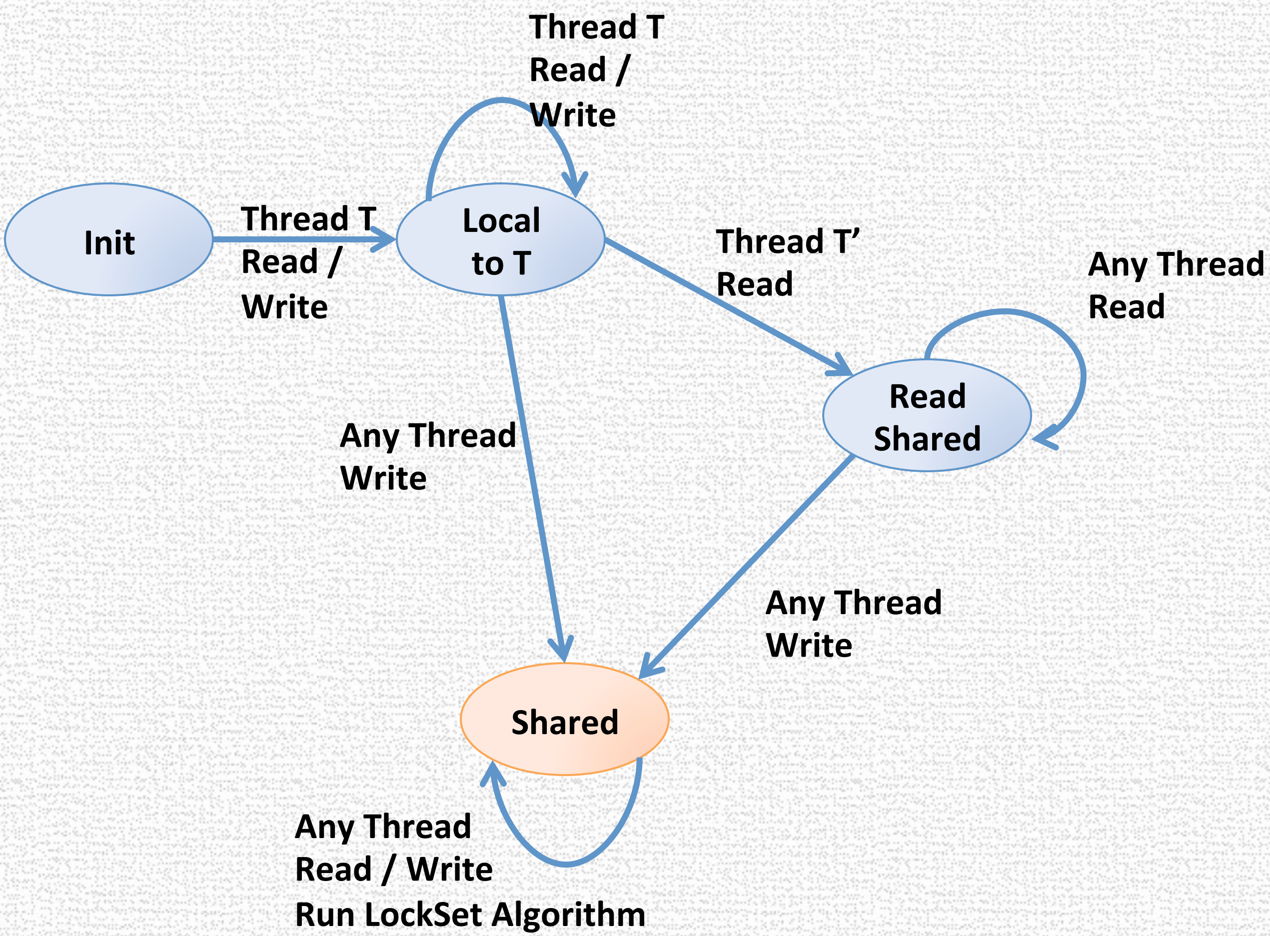
- No warnings → no data races on the current execution
  - The program followed consistent locking discipline in this execution

- Warnings does not imply a data race
  - Object read-shared after thread-local initialization

```
A = new A();
A.f = 0;

// publish A
globalA = A;
```

```
f = globalA.f;
```

Maintain A State Machine Per Location



LockSet Algorithm Guarantees

- State machine misses some data races

```
// Initialize a packet
pkt = new Packet();
pkt.Consumed = 0;

AcquireLock( WrongLk );
pkt = SendQueue.Top();
pkt.Consumed = 1;
ReleaseLock( WrongLk );
```

```
// Process a packet
AcquireLock( SendQueueLk );
pkt = SendQueue.Top();
pkt.Consumed = 1;
ReleaseLock( SendQueueLk );
```

LockSet Algorithm Guarantees

- Does not handle locations consistently protected by different locks during a particular execution

```
// Remove a received packet
AcquireLock( RecvQueueLk );
pkt = RecvQueue.RemoveTop();
ReleaseLock( RecvQueueLk );

… // process pkt

// Insert into processed
AcquireLock( ProcQueueLk );
ProcQueue.Insert(pkt);
ReleaseLock( ProcQueueLk );
```

Pkt is protected by RecvQueueLk

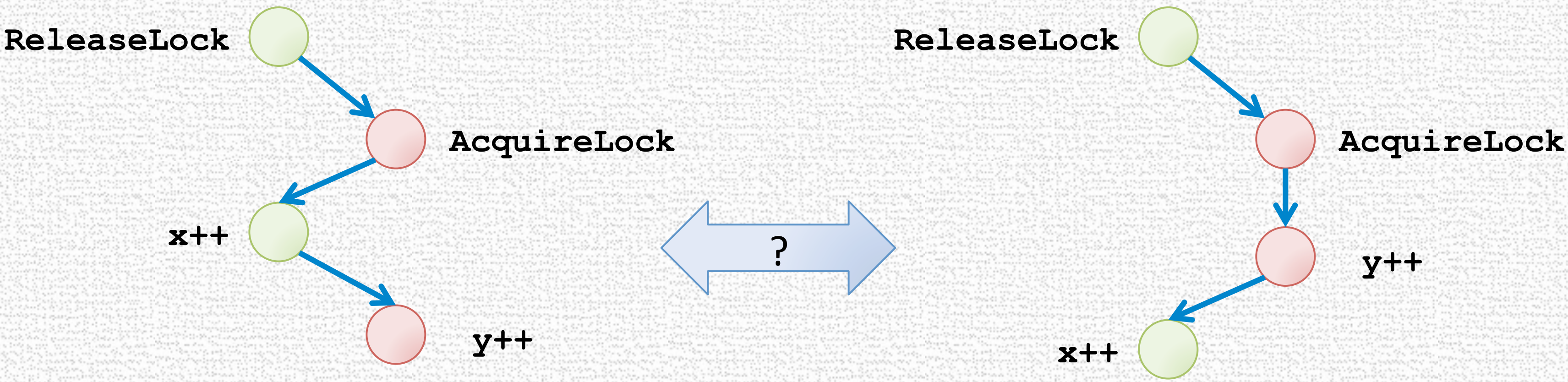
Pkt is thread local

Pkt is protected by ProcQueueLk

HAPPENS-BEFORE

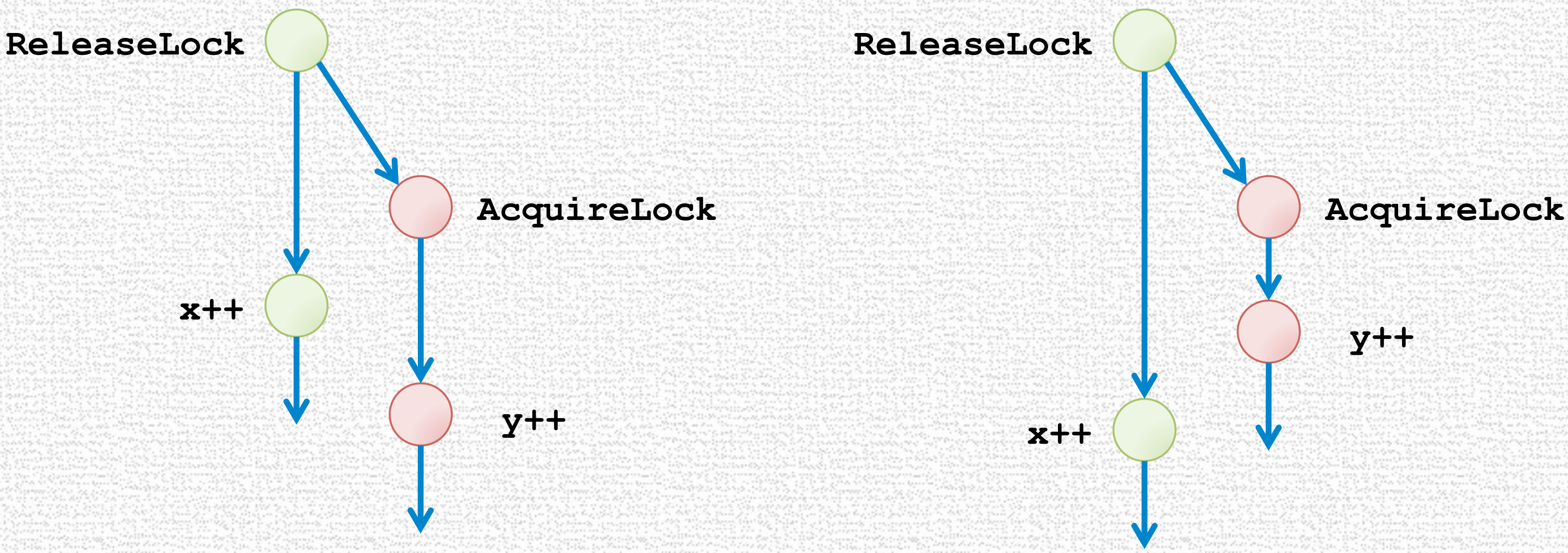
Happens-Before Relation [Lamport '78]

- A concurrent execution is a partial-order determined by communication events
- The program cannot "observe" the order of concurrent non-communicating events



Happens-Before Relation [Lamport '78]

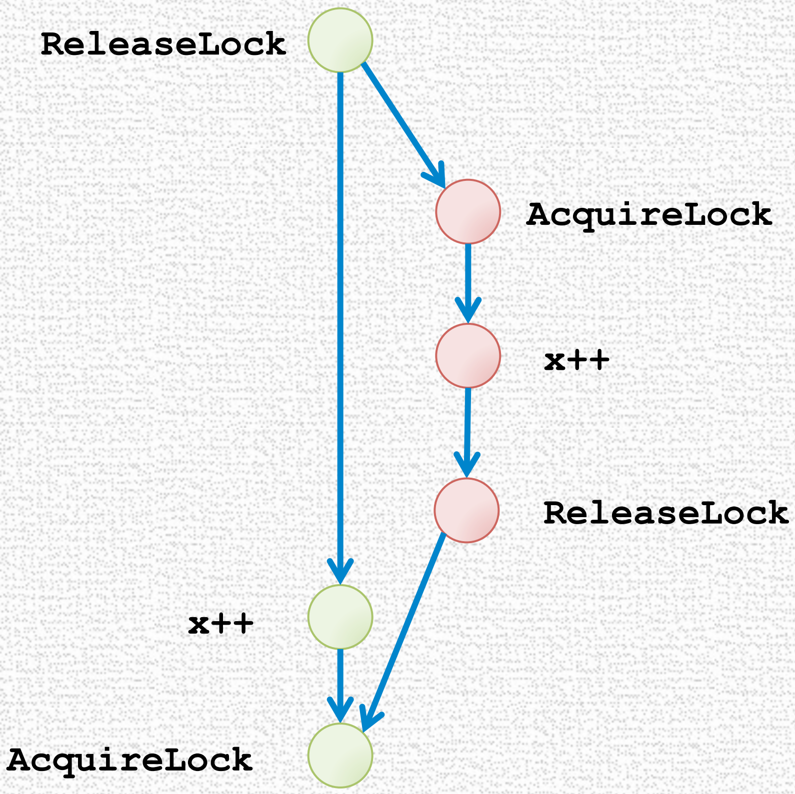
- A concurrent execution is a partial-order determined by communication events
- The program cannot "observe" the order of concurrent non-communicating events



- Both executions form the same happens-before relation

Constructing the Happens-Before Relation

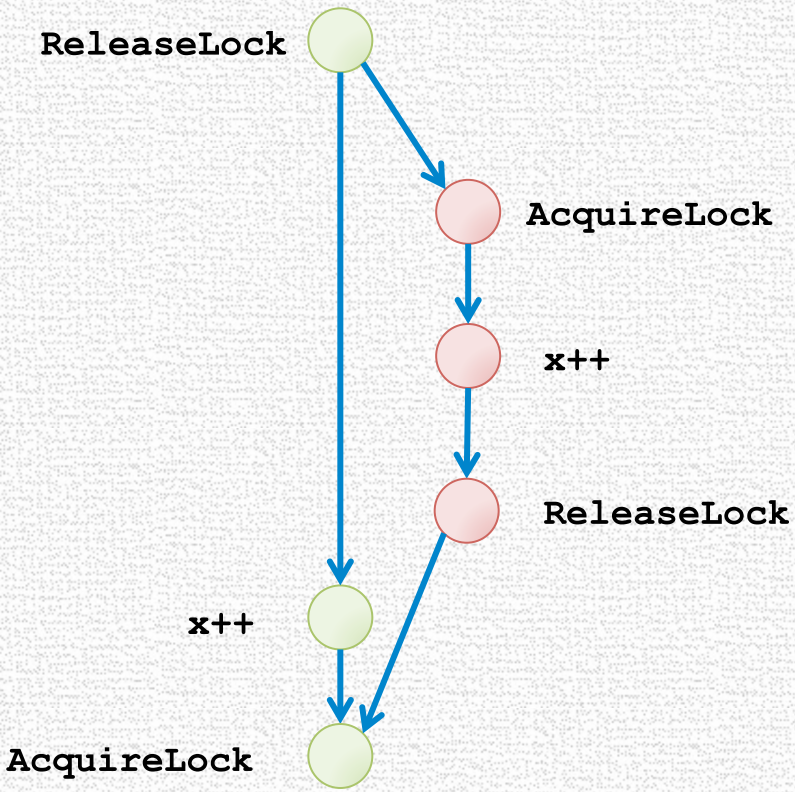
- Program order
  - Total order of thread instructions
- Synchronization order
  - Total order of accesses to the same synchronization



Happens-Before Relation And Data Races

- If all conflicting accesses are ordered by happens-before
- → data-race-free execution
- → All linearizations of partial-order are valid program executions

- If there exists conflicting accesses not ordered
- → a data race



Happens-Before and Data-Races

- Not all unordered conflicting accesses are data races

Init: X = Y = 0;

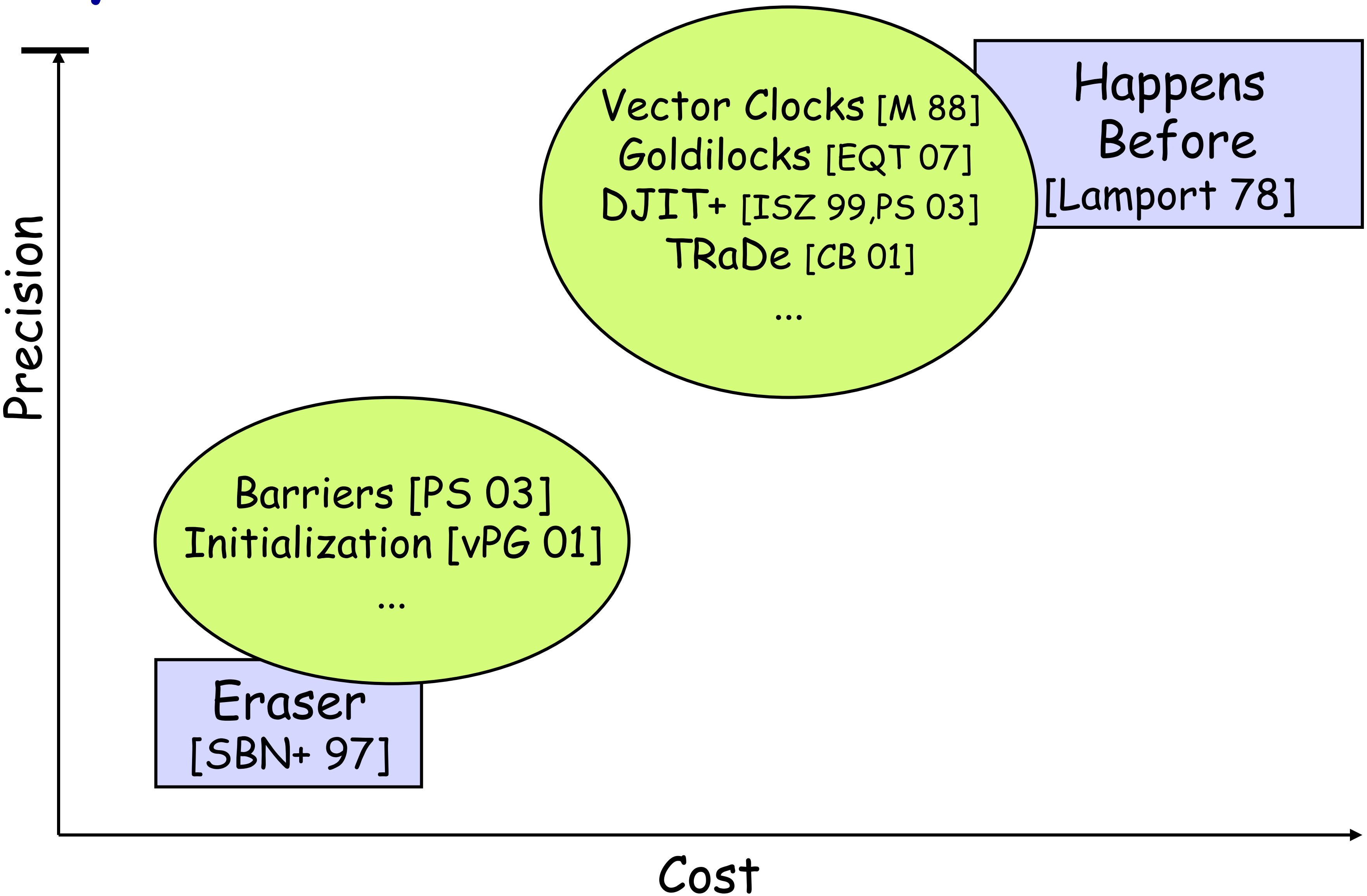
```
X = 1;
Y = 1;
```

```
if( Y == 1 )
  X = 2;
```

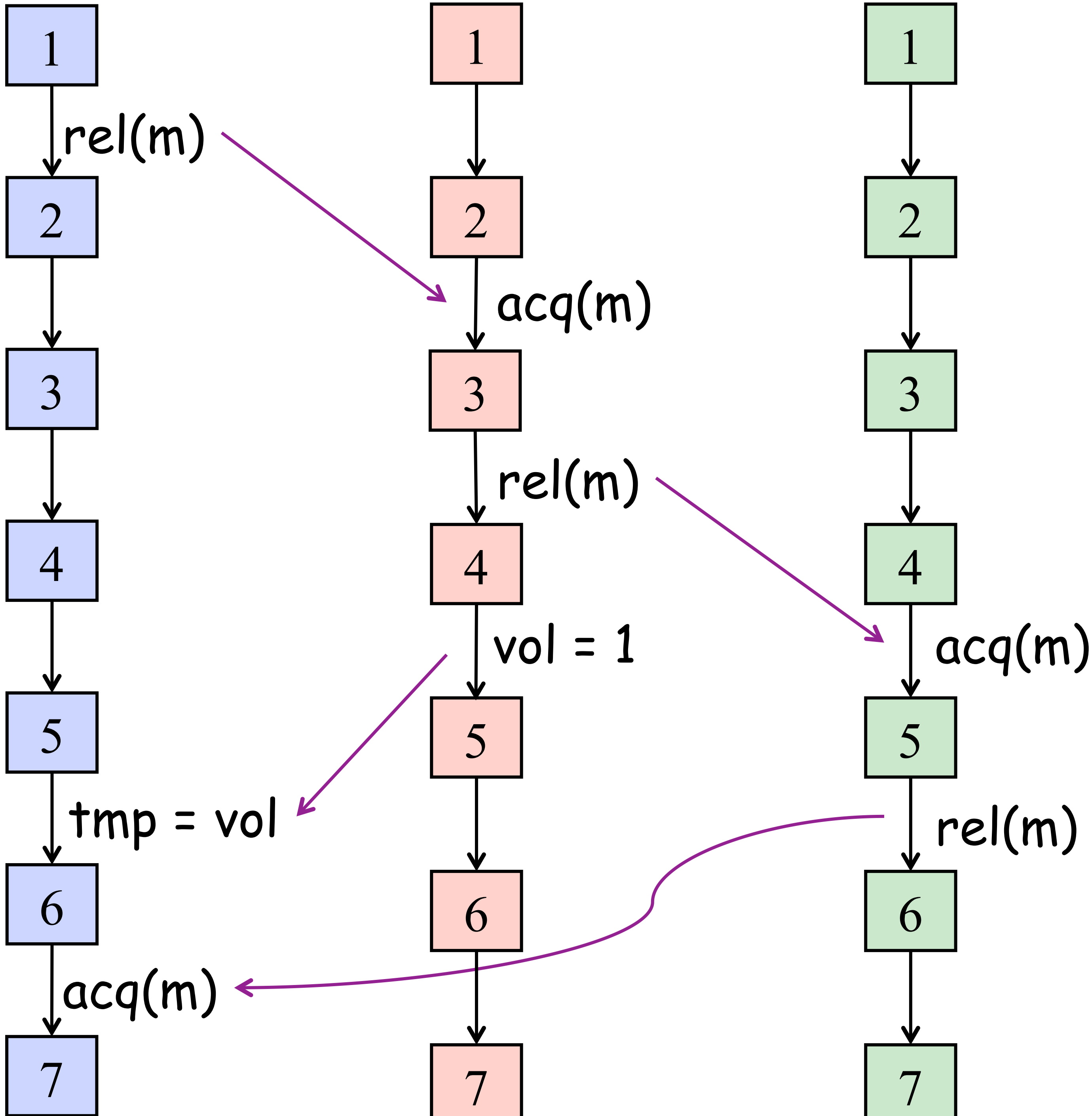
- There is no data race on X
- But, there is a data race on Y
- Remember:
  - Exists unordered conflicting access → Exists data race

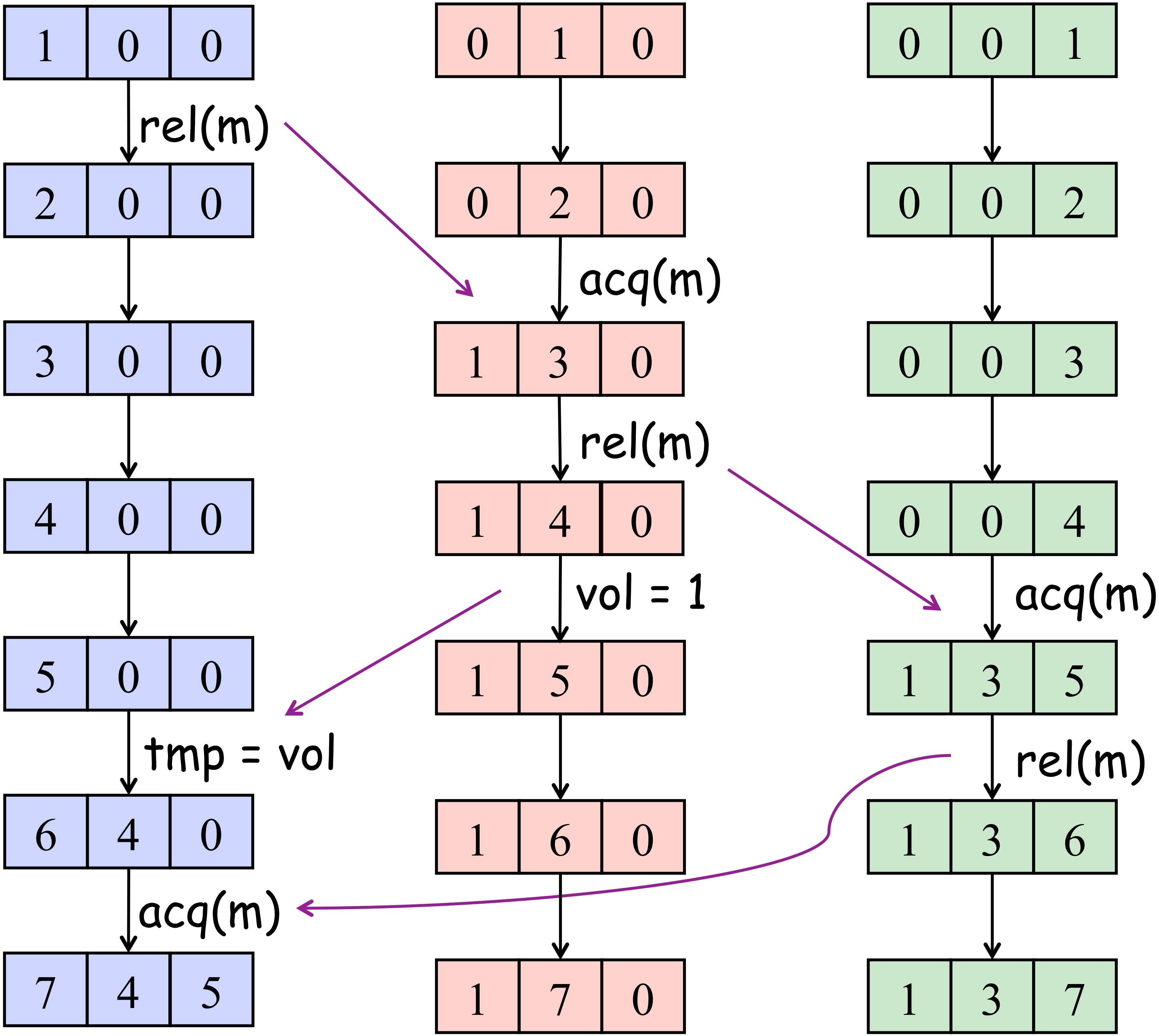
IMPLEMENTING HAPPENS-BEFORE ANALYSES

Dynamic Data-Race Detection

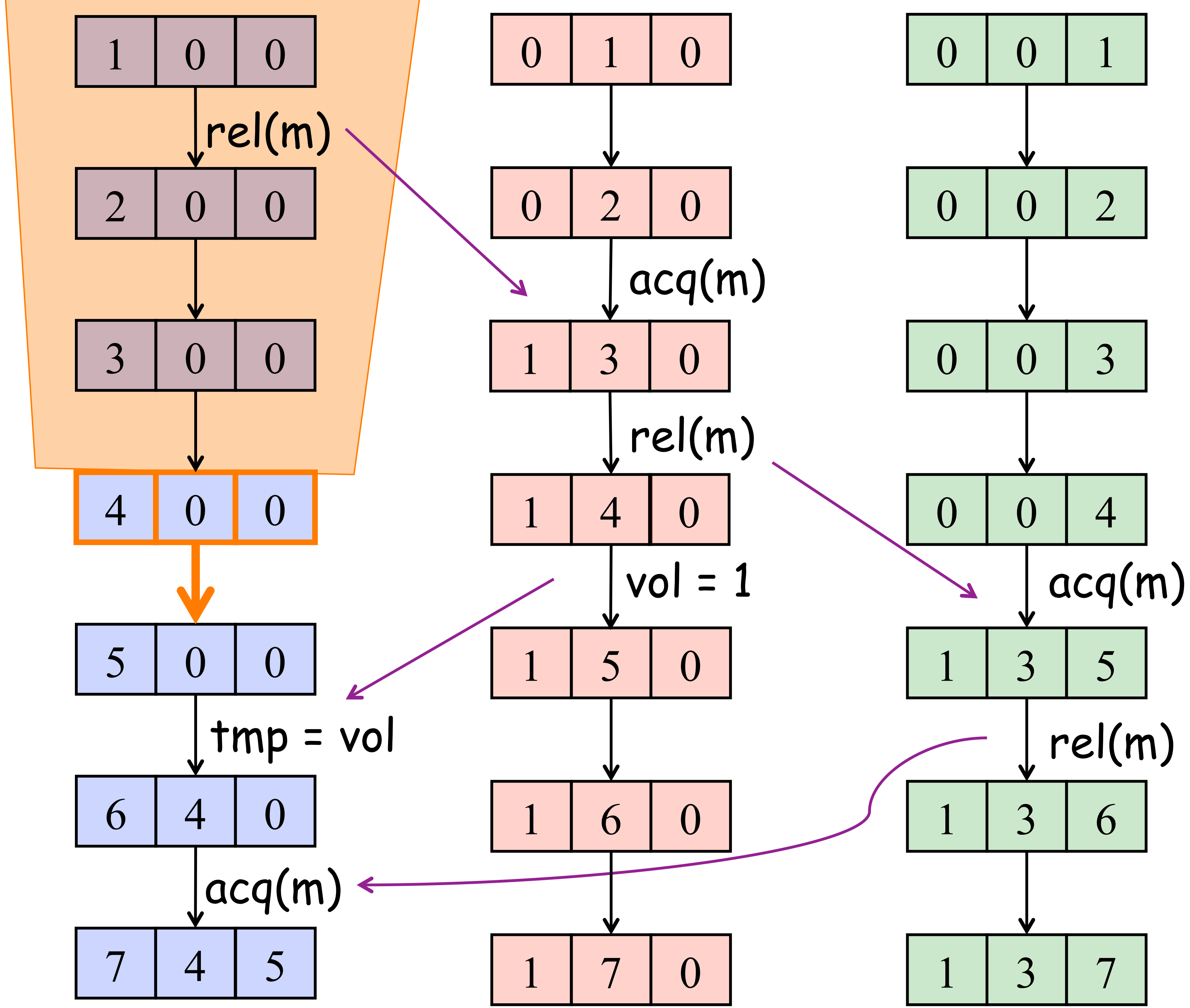


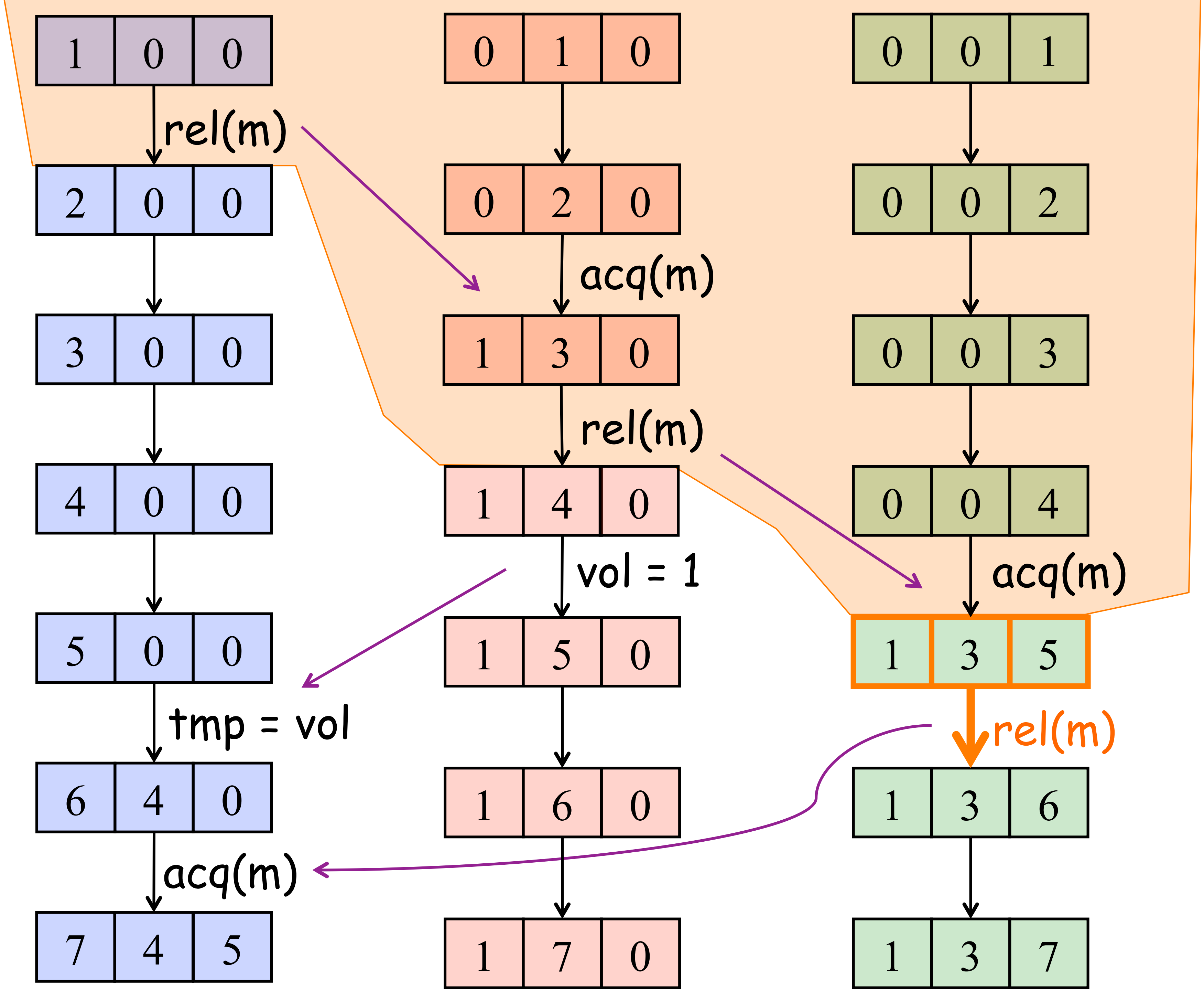
Precise Happens-Before



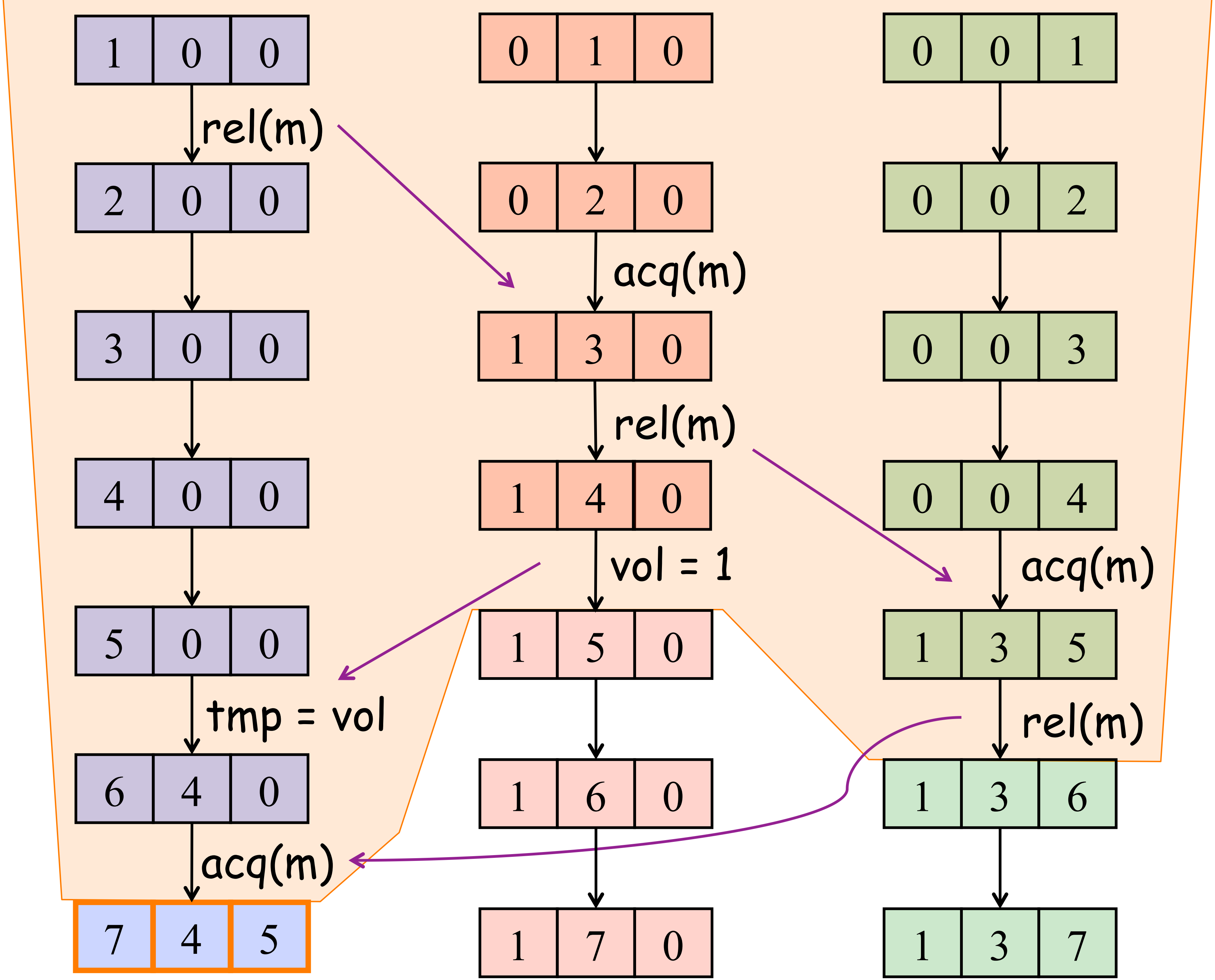


1 0 0
rel(m)
2 0 0
3 0 0
4 0 0
5 0 0
tmp = vol
6 4 0
acq(m)
7 4 5
0 1 0
0 2 0
acq(m)
1 3 0
rel(m)
1 4 0
vol = 1
1 5 0
1 6 0
1 7 0
0 0 1
0 0 2
0 0 3
0 0 4
acq(m)
1 3 5
rel(m)
1 3 6
1 3 7

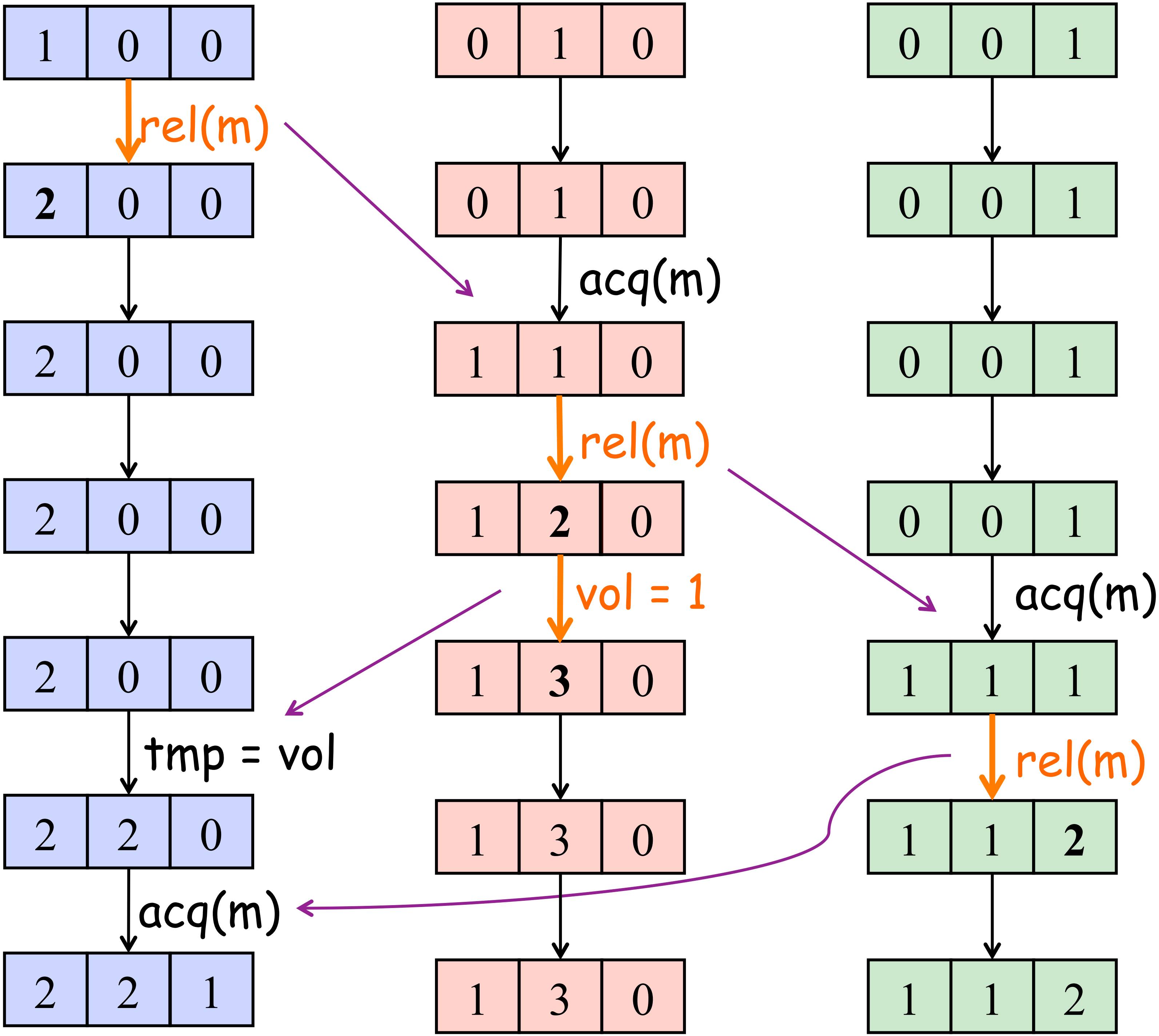

rel(m)
acq(m)
rel(m)
vol = 1
acq(m)
tmp = vol
rel(m)
acq(m)



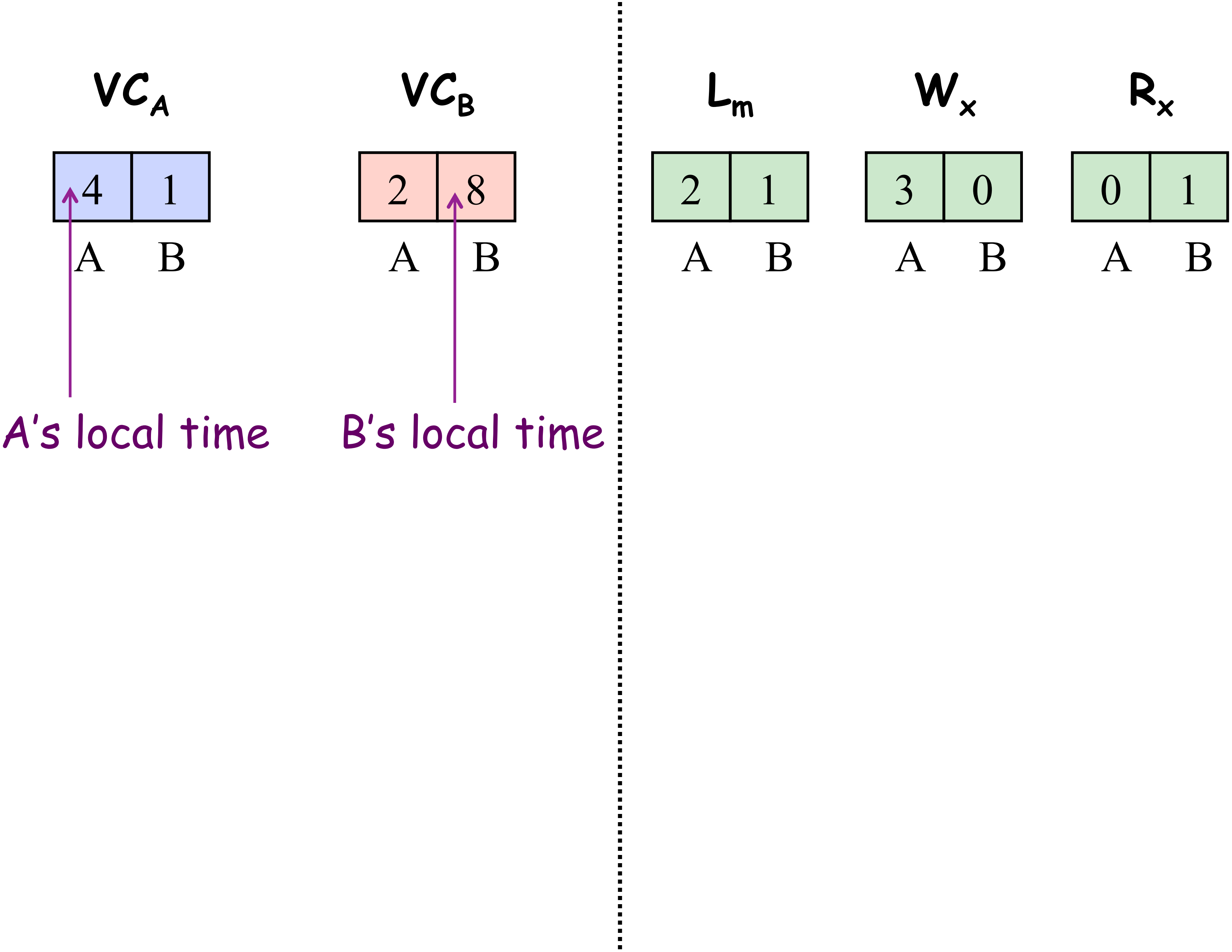
1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 2 0
0 0 2
acq(m)
3 0 0
1 3 0
0 0 3
rel(m)
4 0 0
1 4 0
0 0 4
vol = 1
acq(m)
5 0 0
1 5 0
1 3 5
tmp = vol
rel(m)
6 4 0
1 6 0
1 3 6
acq(m)
7 4 5
1 7 0
1 3 7



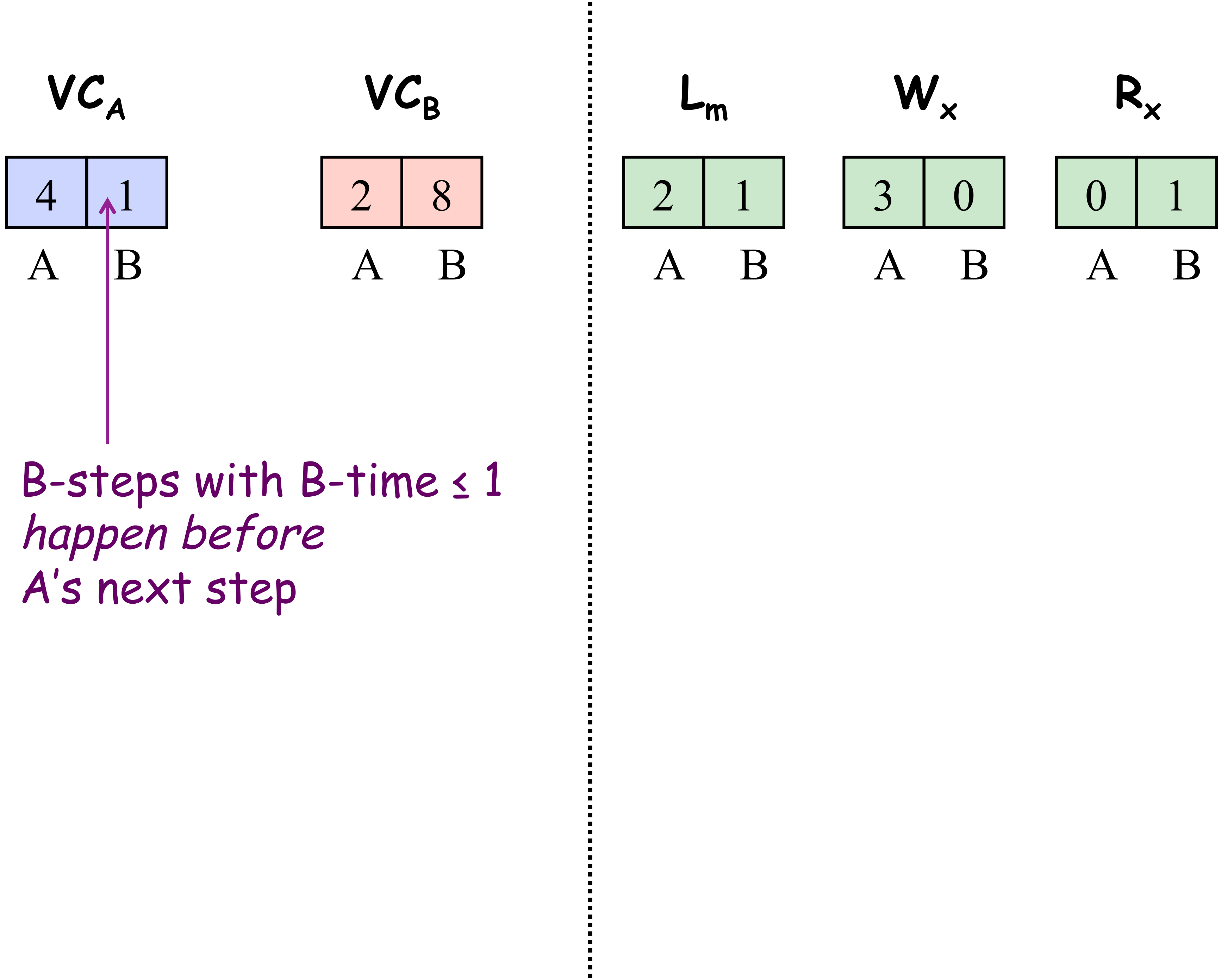
1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 2 0
0 0 2
acq(m)
3 0 0
1 3 0
0 0 3
rel(m)
4 0 0
1 4 0
0 0 4
vol = 1
acq(m)
5 0 0
1 5 0
1 3 5
tmp = vol
rel(m)
6 4 0
1 6 0
1 3 6
acq(m)
7 4 5
1 7 0
1 3 7

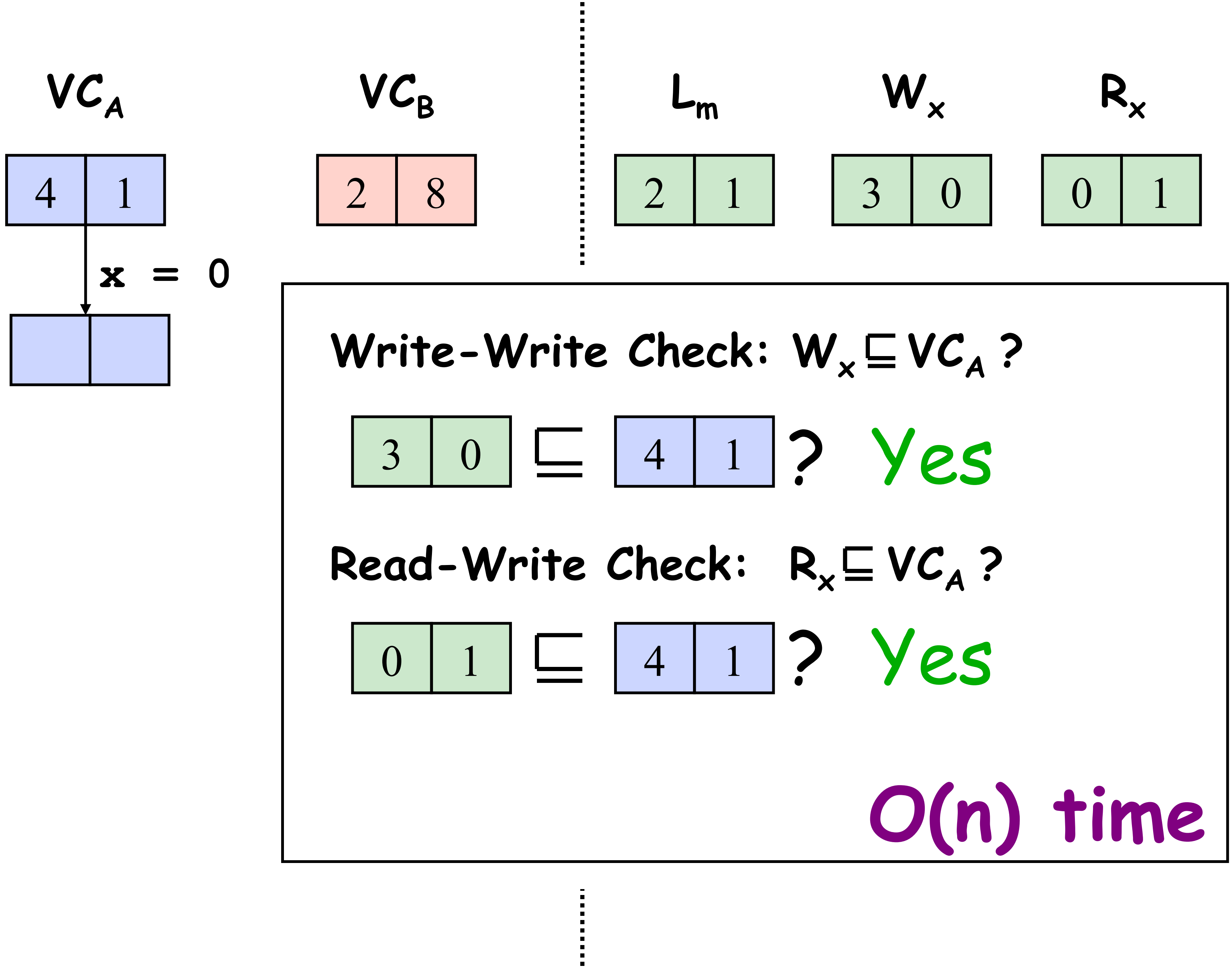


rel(m)
acq(m)
rel(m)
vol = 1
acq(m)
tmp = vol
rel(m)
acq(m)

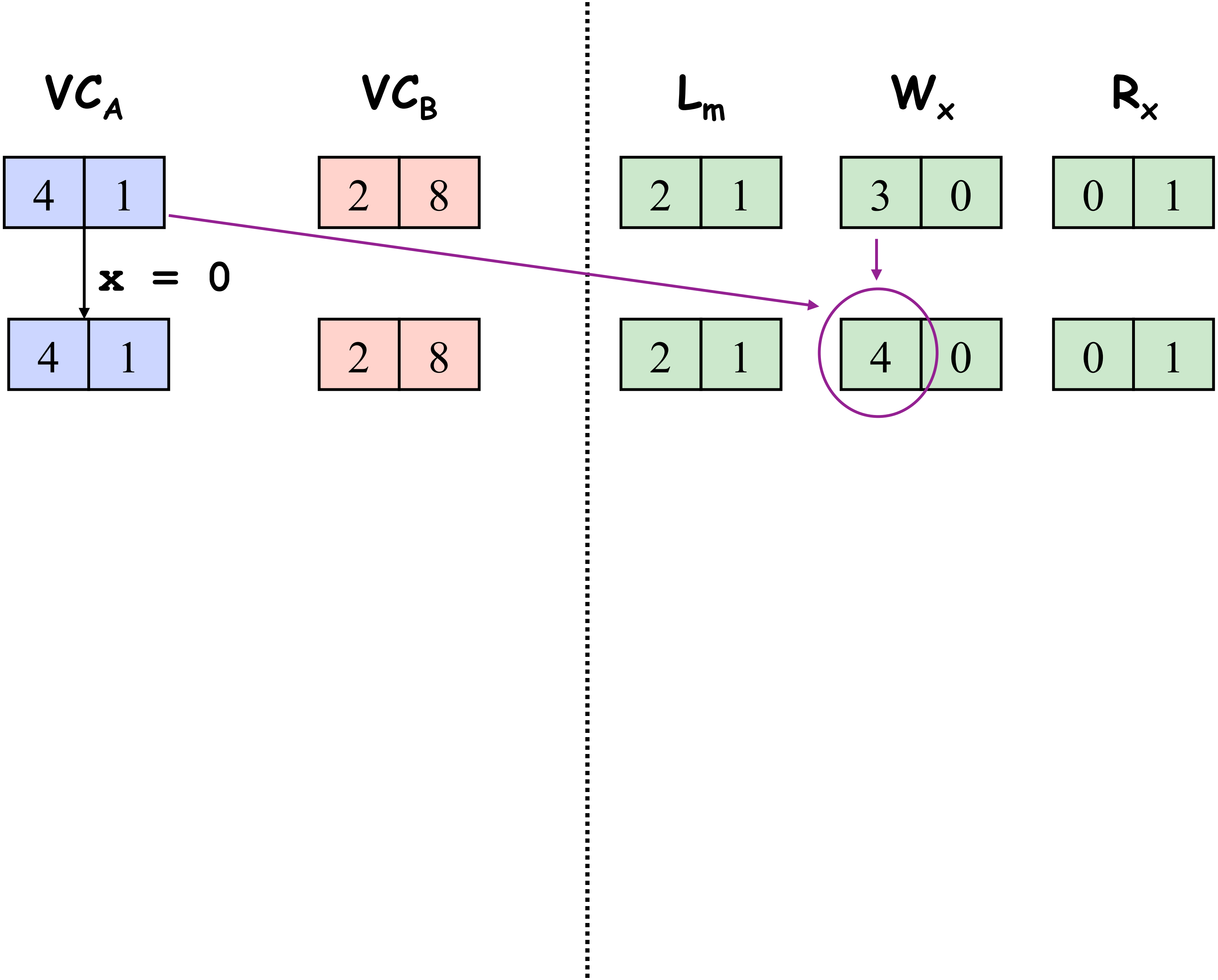


VC_A
4 1
A B
VC_B
2 8
A B
L_m
2 1
A B
W_x
3 0
A B
R_x
0 1
A B
A's local time
B's local time

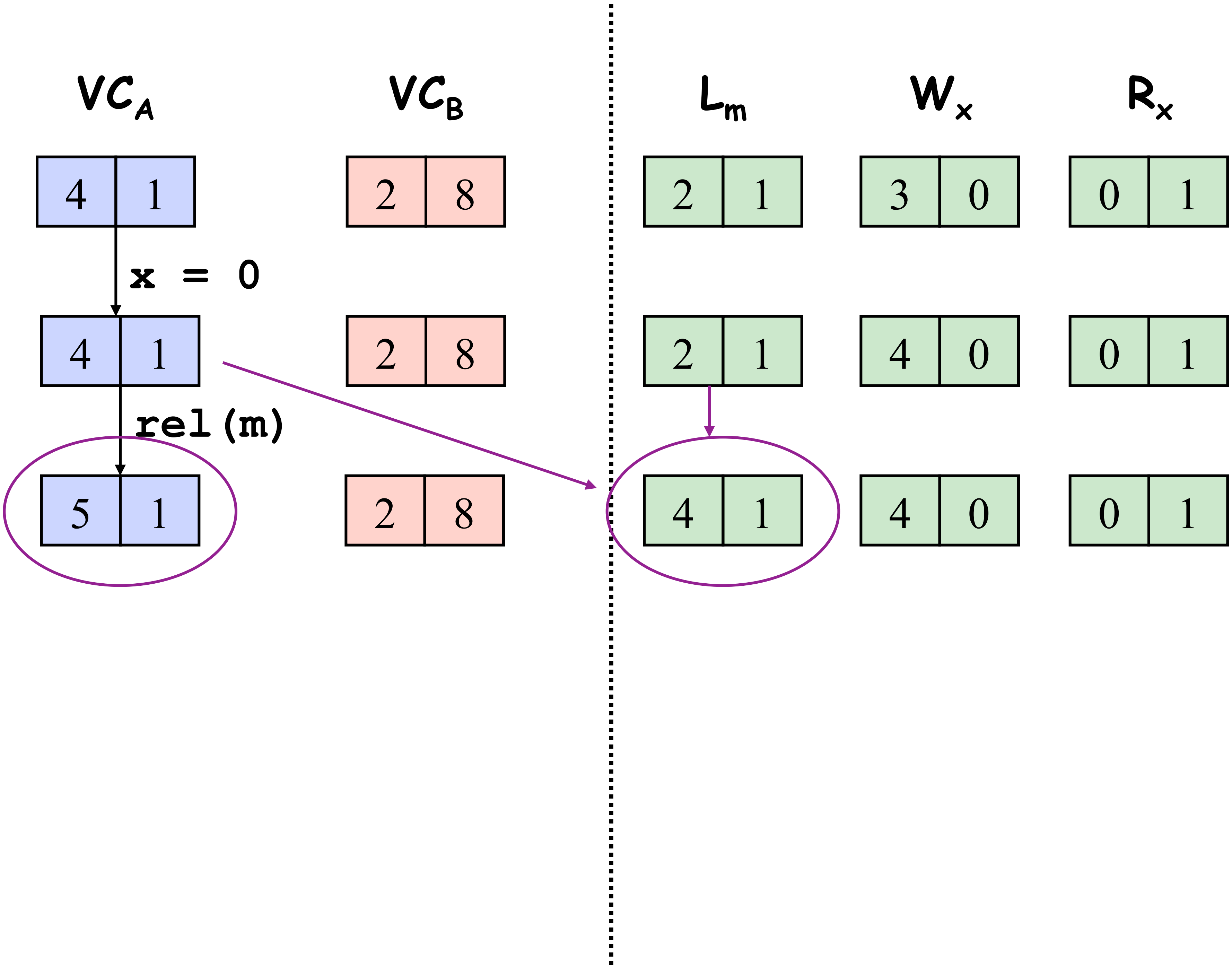

VC_A
4 1
A B
VC_B
2 8
A B
L_m
2 1
A B
W_x
3 0
A B
R_x
0 1
A B
B-steps with B-time ≤ 1
happen before
A's next step



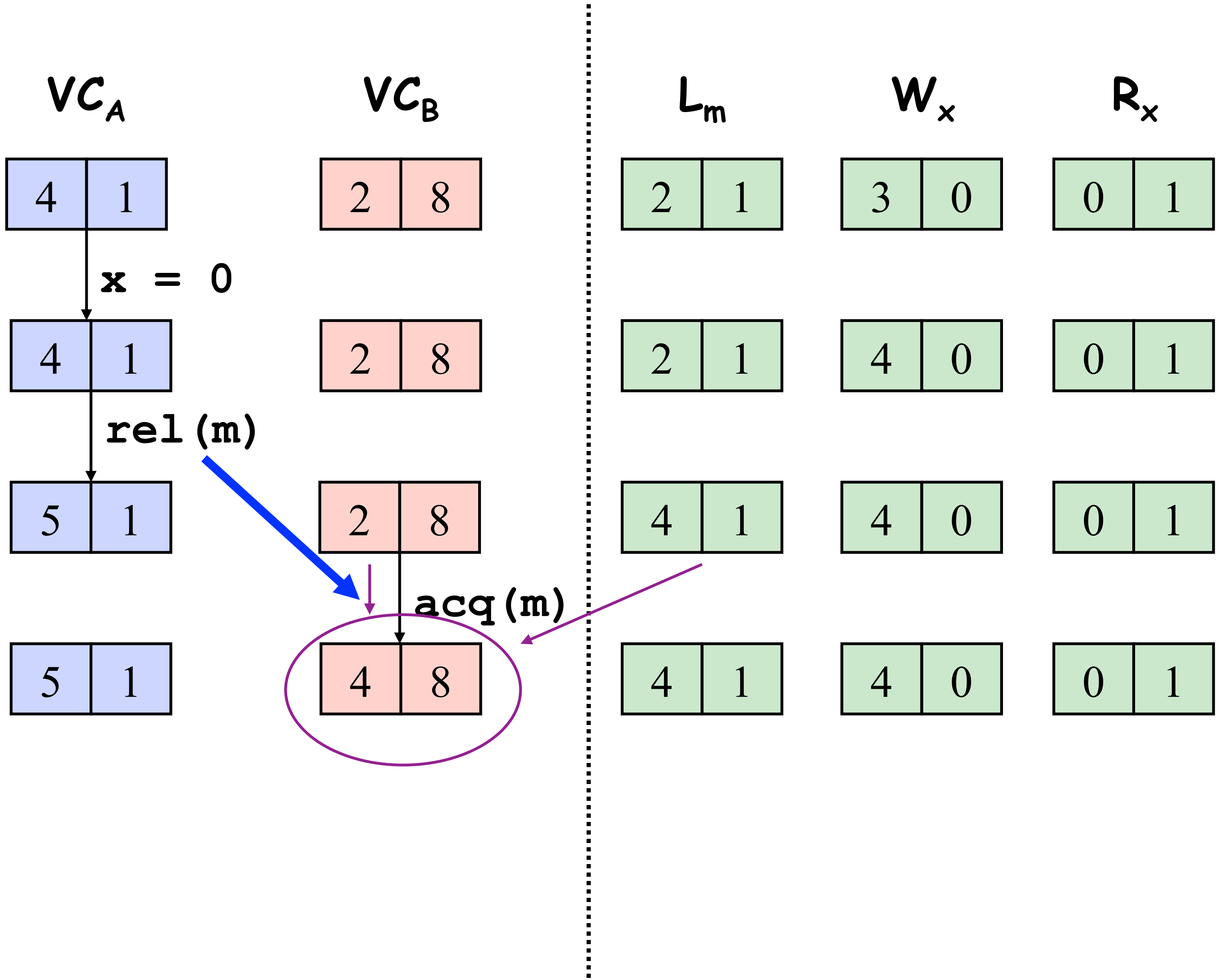
VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
Write-Write Check: $W_x \sqsubseteq VC_A$?
3 0 $\sqsubseteq$ 4 1 ?
Yes
Read-Write Check: $R_x \sqsubseteq VC_A$?
0 1 $\sqsubseteq$ 4 1 ?
Yes
O(n) time

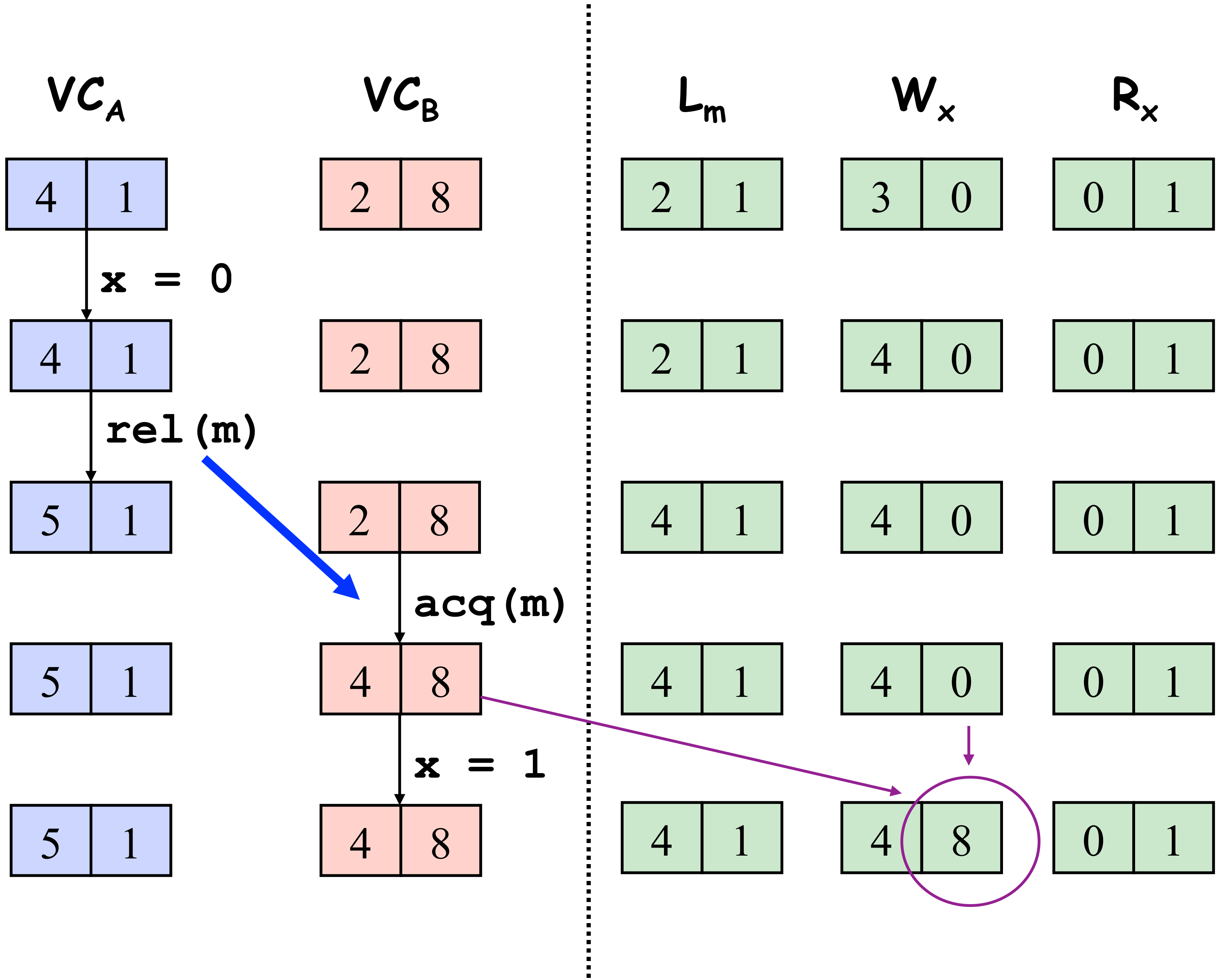


VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1

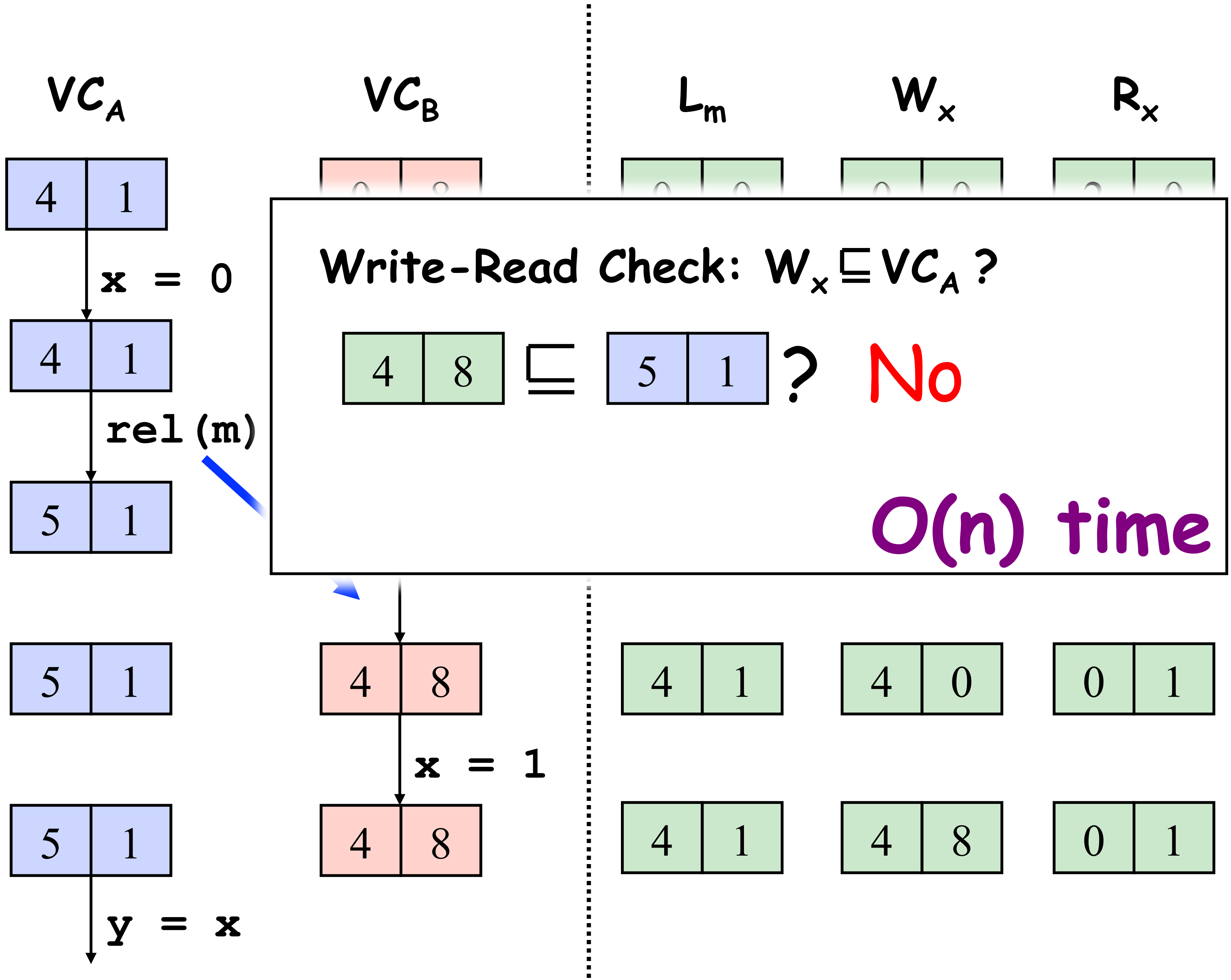


VC_A
VC_B
L_m
W_x
R_x
4 1
2 8
2 1
3 0
0 1
x = 0
4 1
2 8
2 1
4 0
0 1
rel(m)
5 1
2 8
4 1
4 0
0 1


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



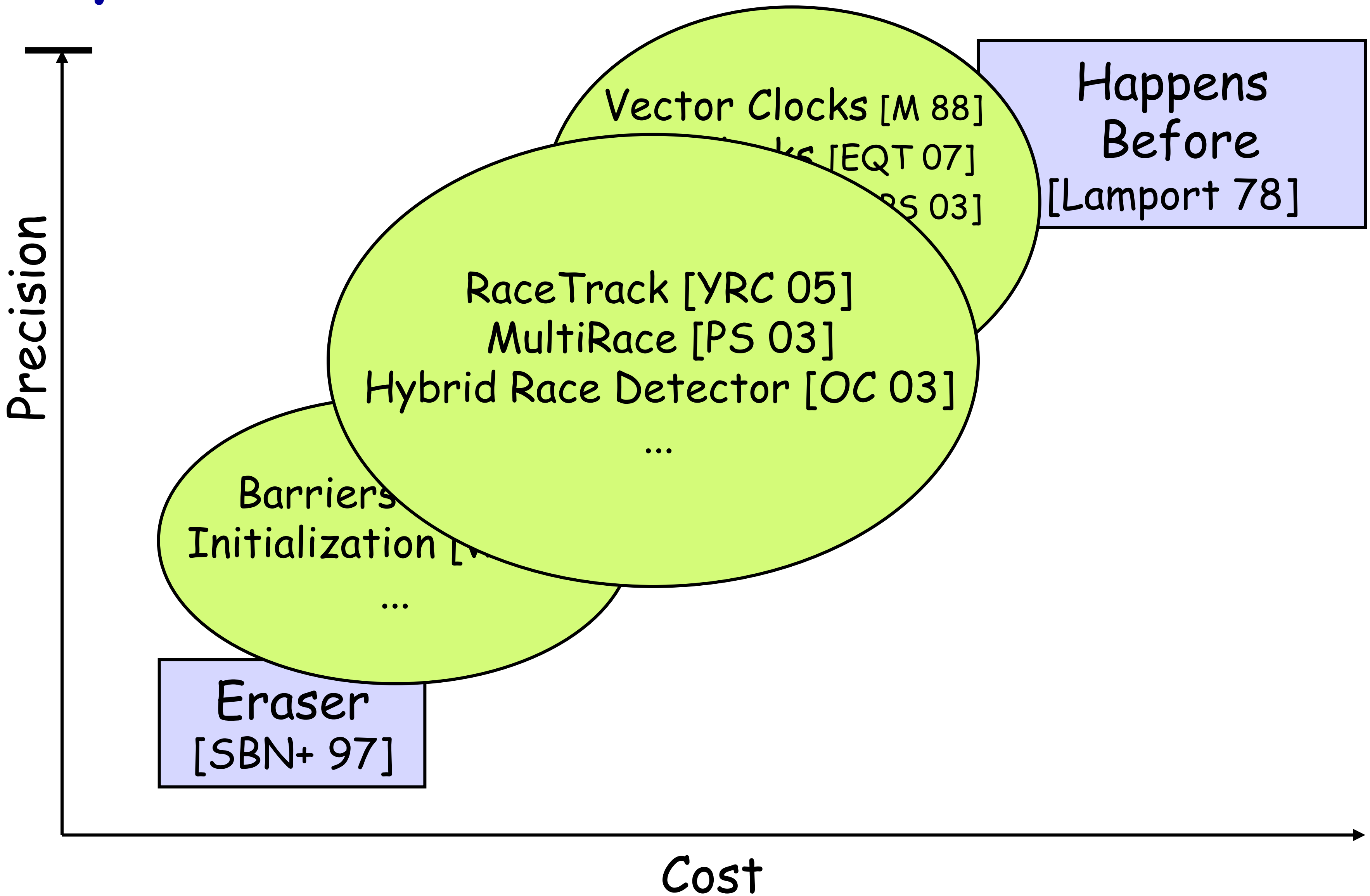
VC_A
VC_B
L_m
W_x
R_x
x = 0
rel(m)
x = 1
y = x
Write-Read Check: W_x ⊑ VC_A ?
4 8 ⊑ 5 1 ?
No
O(n) time

VectorClocks for Data-Race Detection

- Sound
  - No warnings ➔ data-race-free execution
- Complete
  - Warning ➔ data-race exists

- Performance
  - slowdowns > 50x
  - memory overhead

Dynamic Data-Race Detection

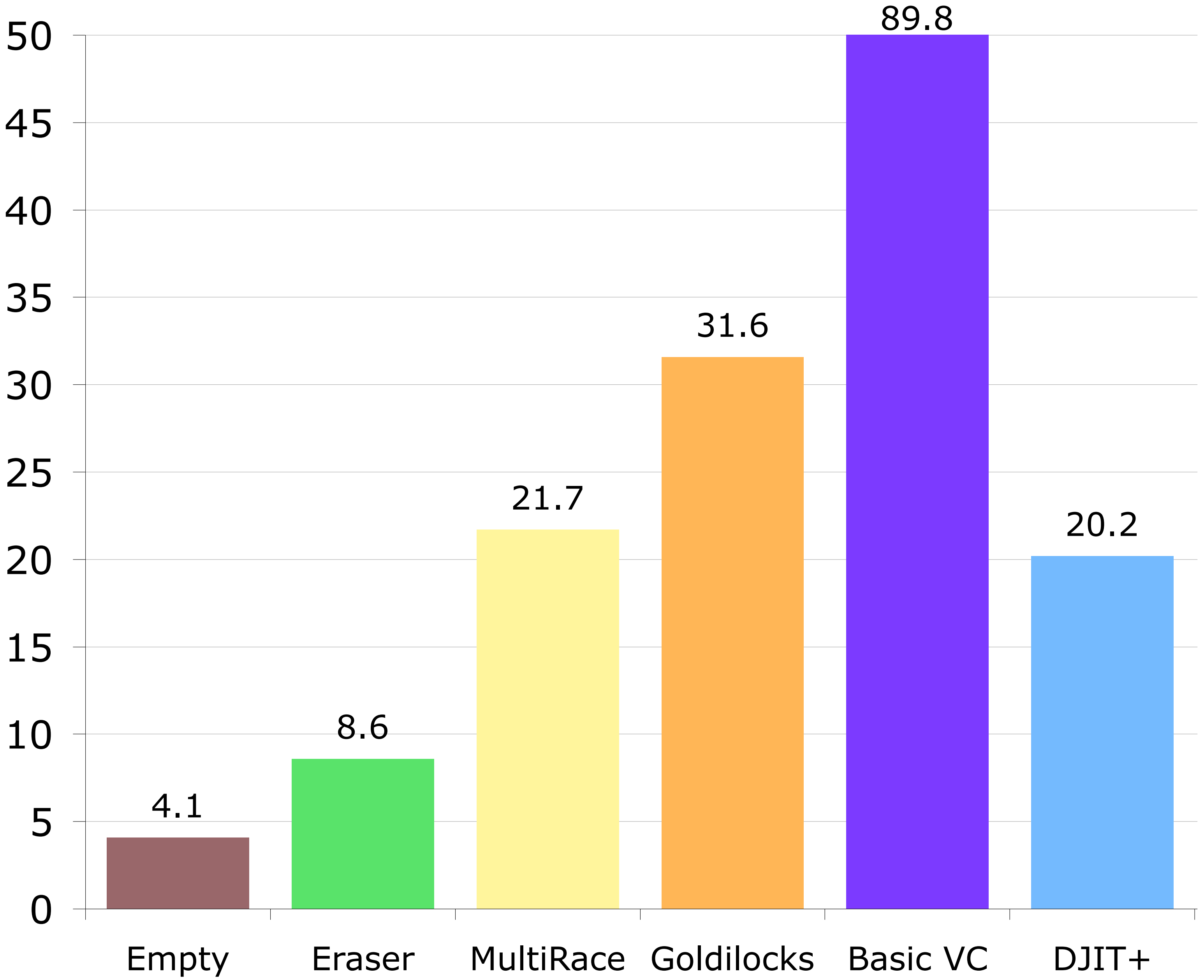


Combined Approaches

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

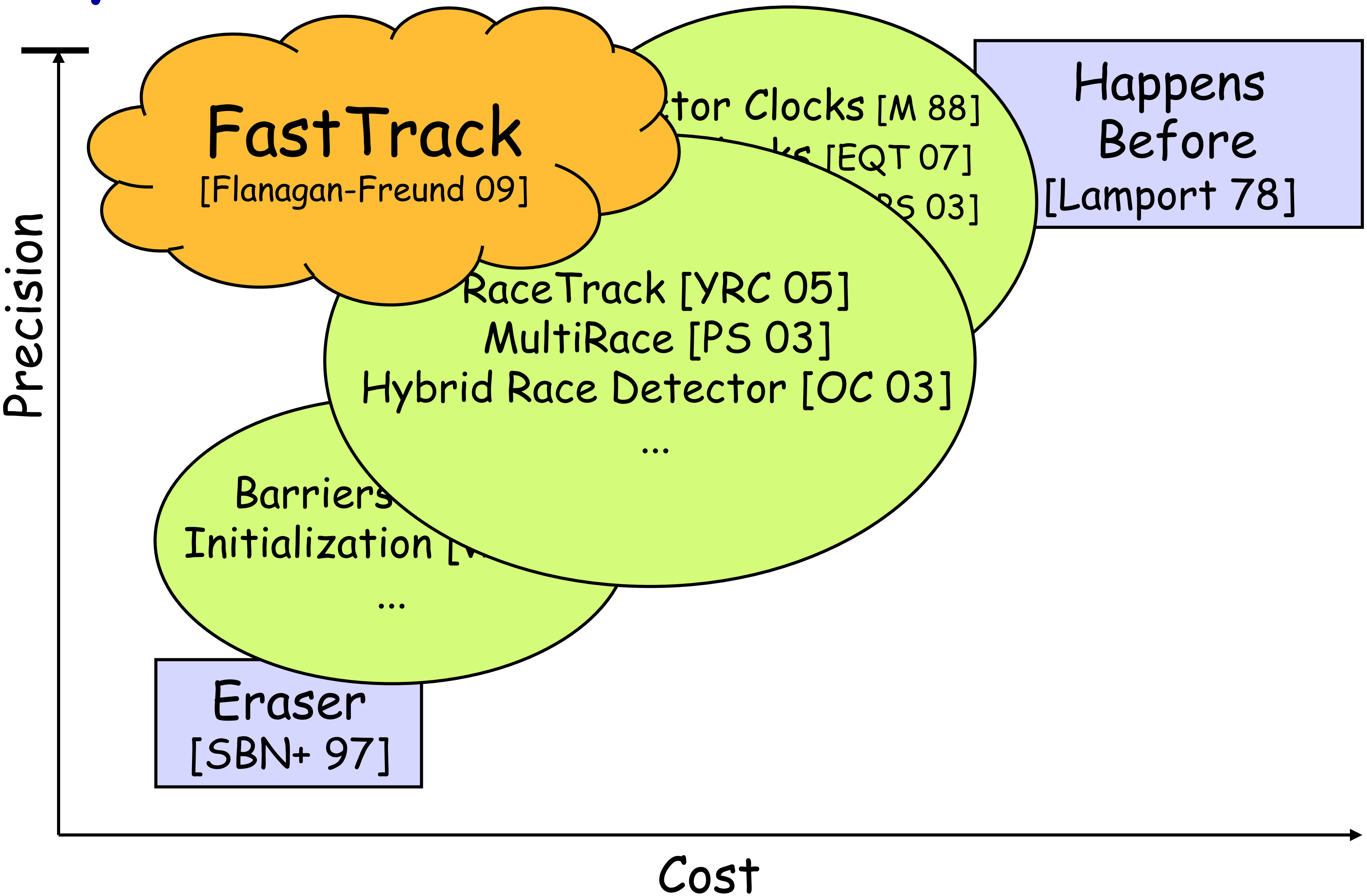
- RaceTrack [YRC 05]
  - Use LockSet for *x* and extended Eraser state machine.
  - Use VCs to reason about fork/join and wait/notify

Slowdown (x Base Time)

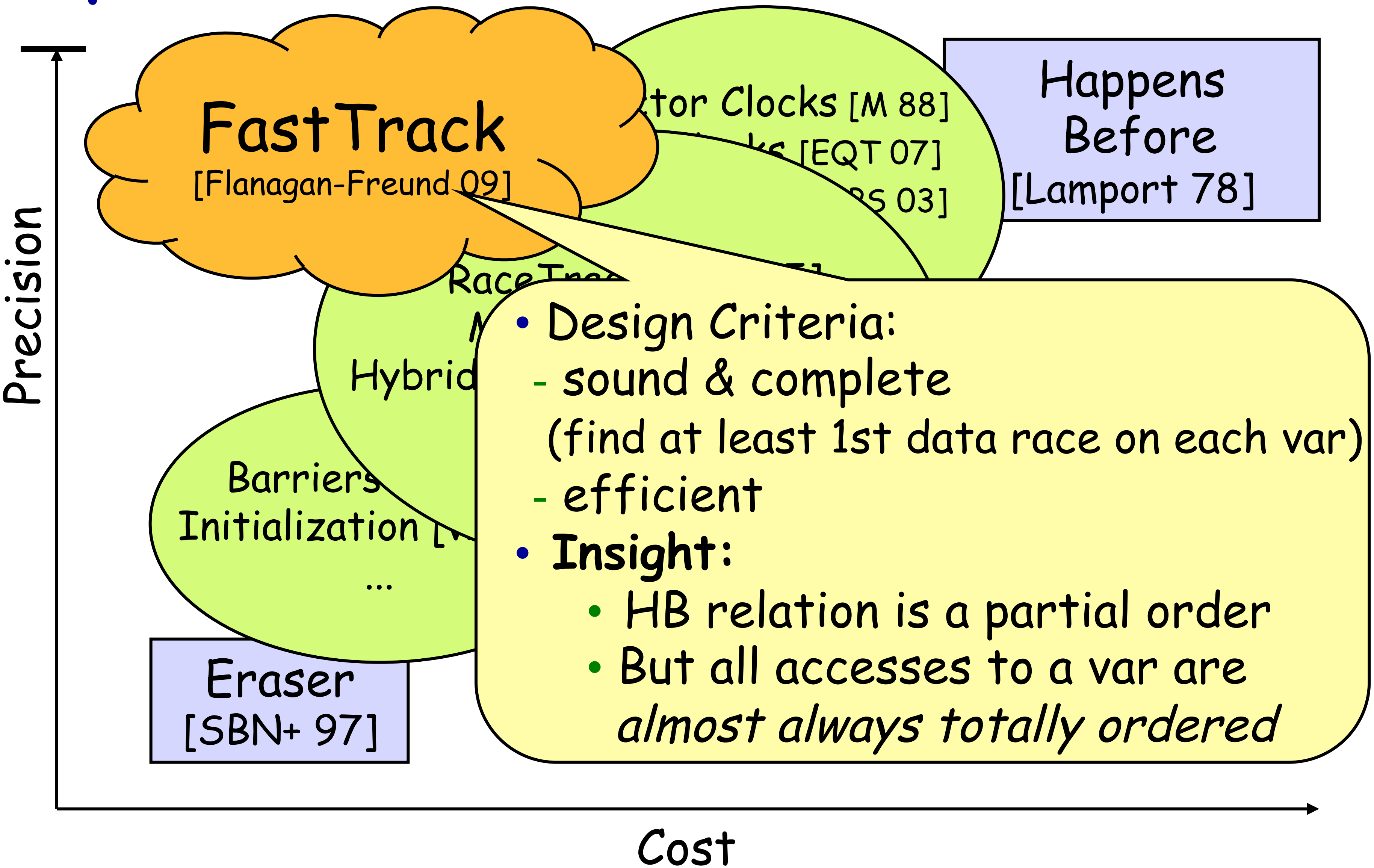


FASTTRACK

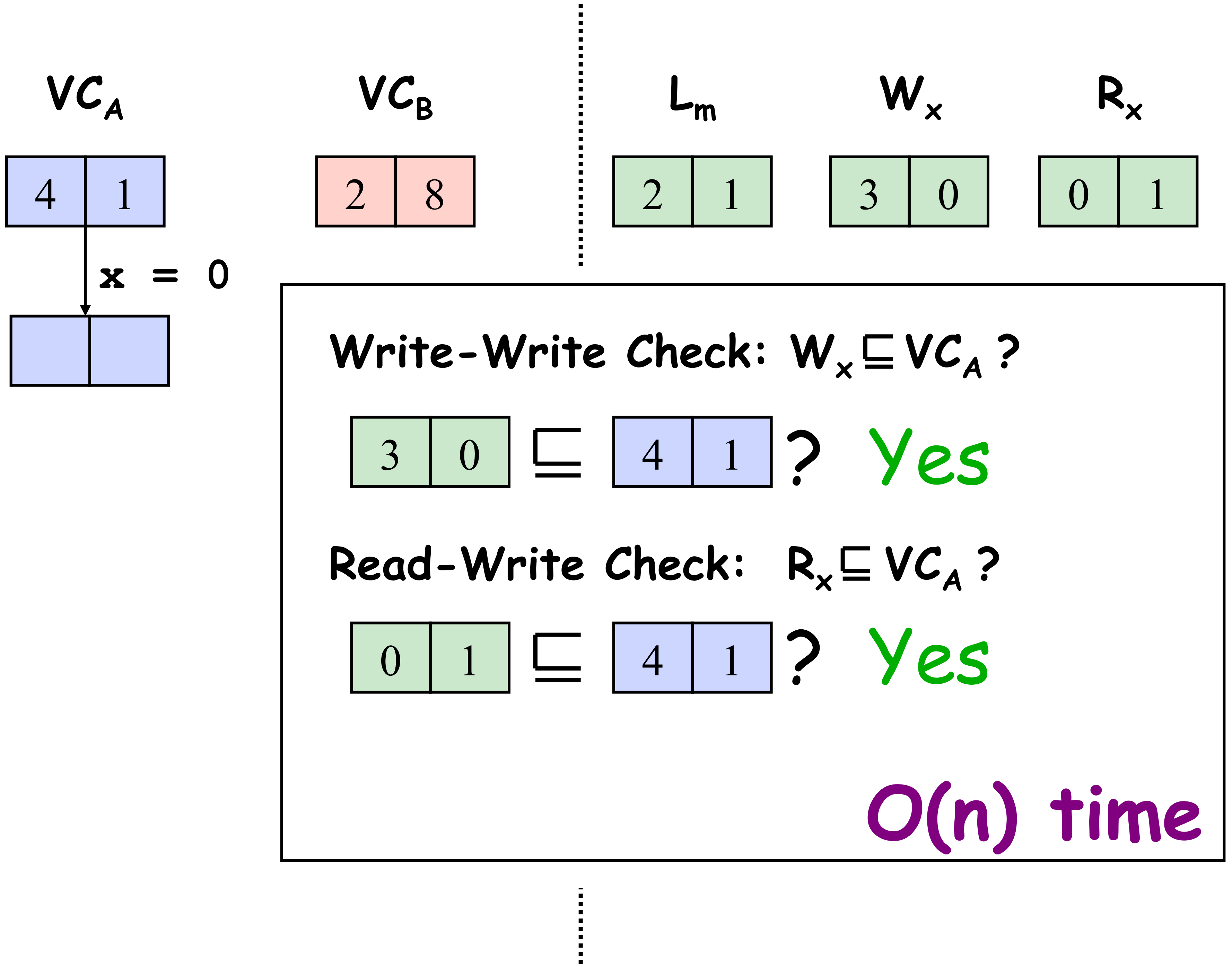
Dynamic Data-Race Detection



Dynamic Data-Race Detection

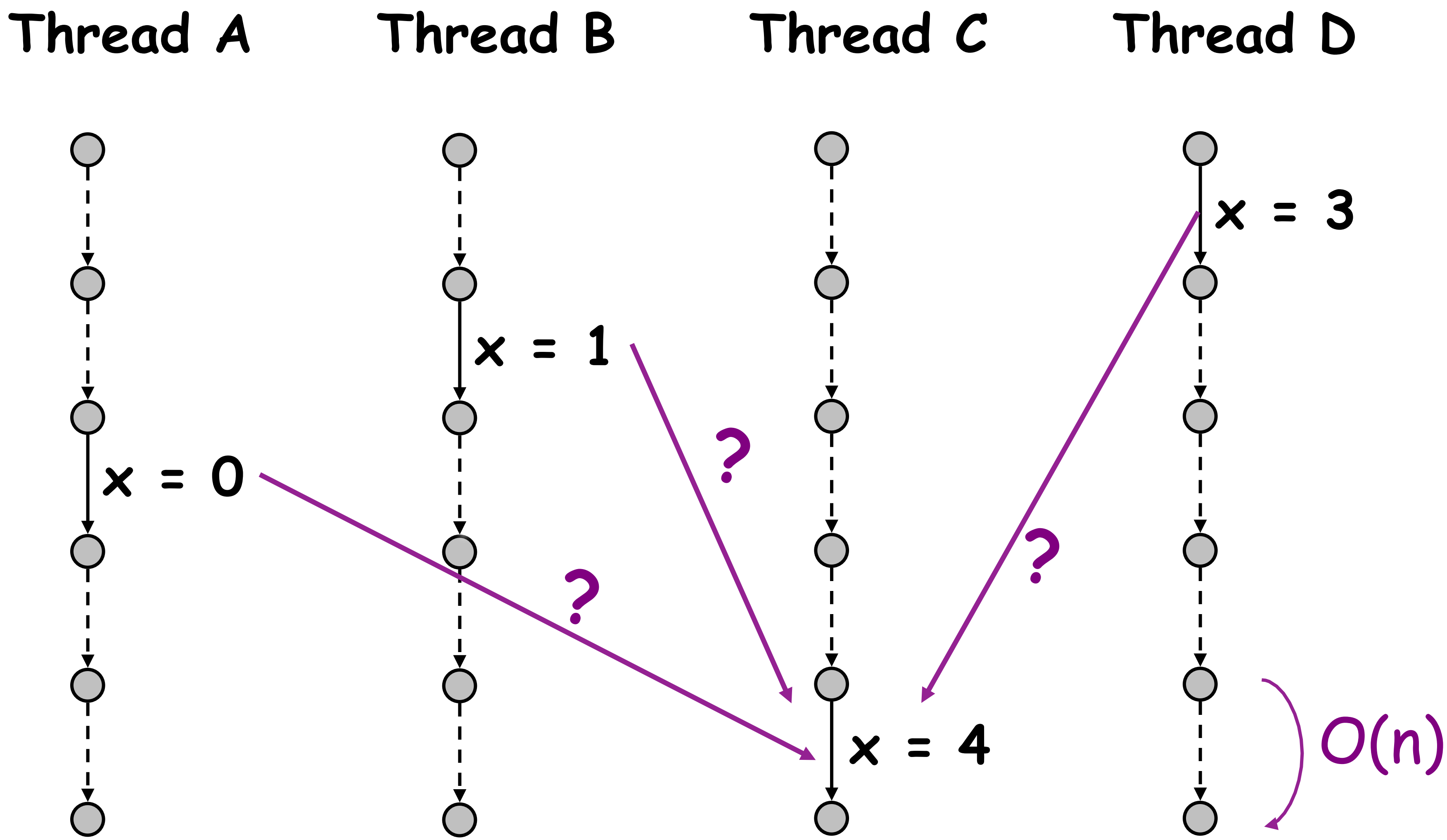


- Design Criteria:
  - sound & complete (find at least 1st data race on each var)
  - efficient
- **Insight:**
  - HB relation is a partial order
  - But all accesses to a var are *almost always totally ordered*

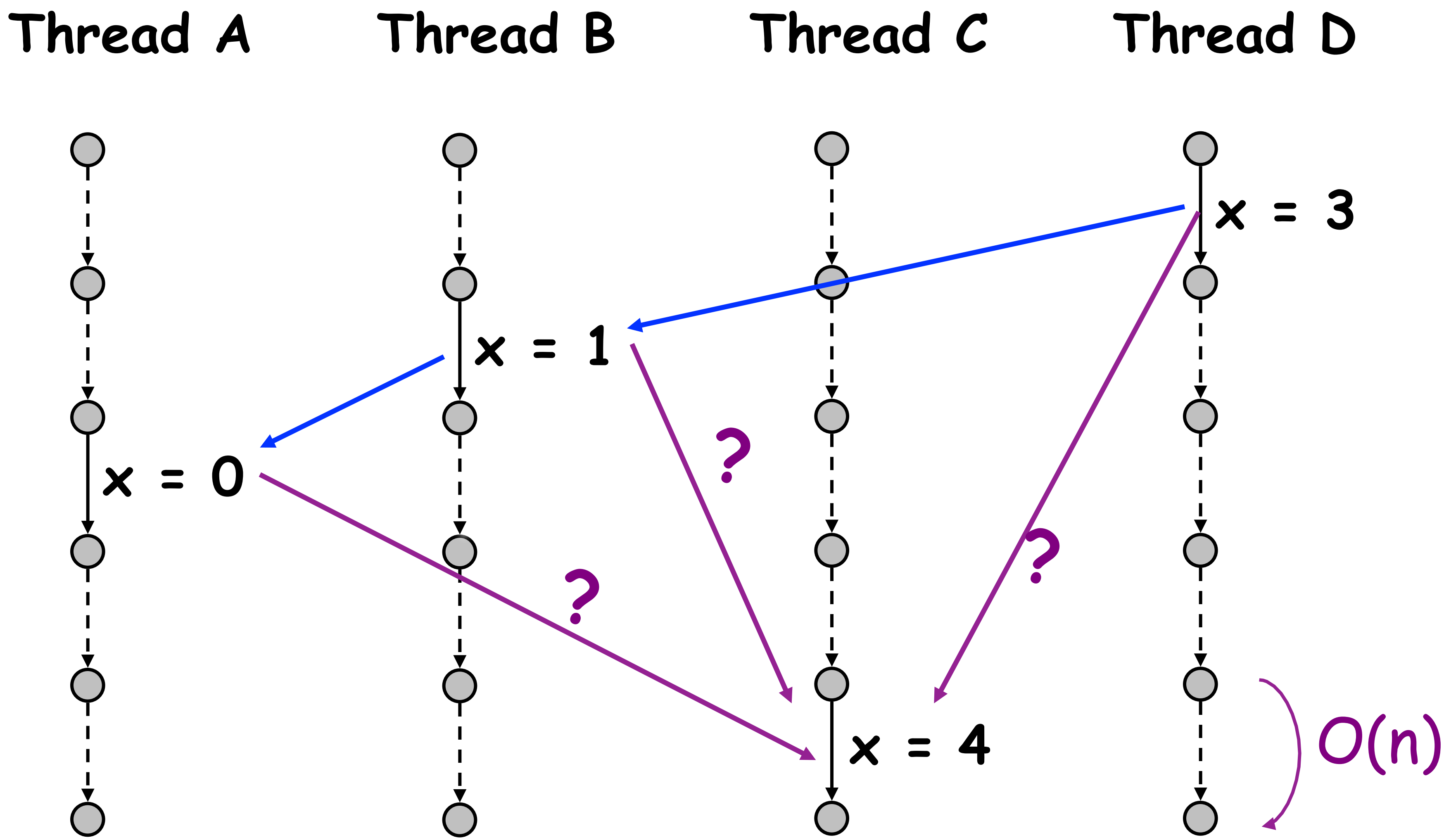


VC_A
4 1
x = 0
VC_B
2 8
L_m
2 1
W_x
3 0
R_x
0 1
Write-Write Check: W_x ⊑ VC_A ?
3 0 ⊑ 4 1 ? Yes
Read-Write Check: R_x ⊑ VC_A ?
0 1 ⊑ 4 1 ? Yes
O(n) time

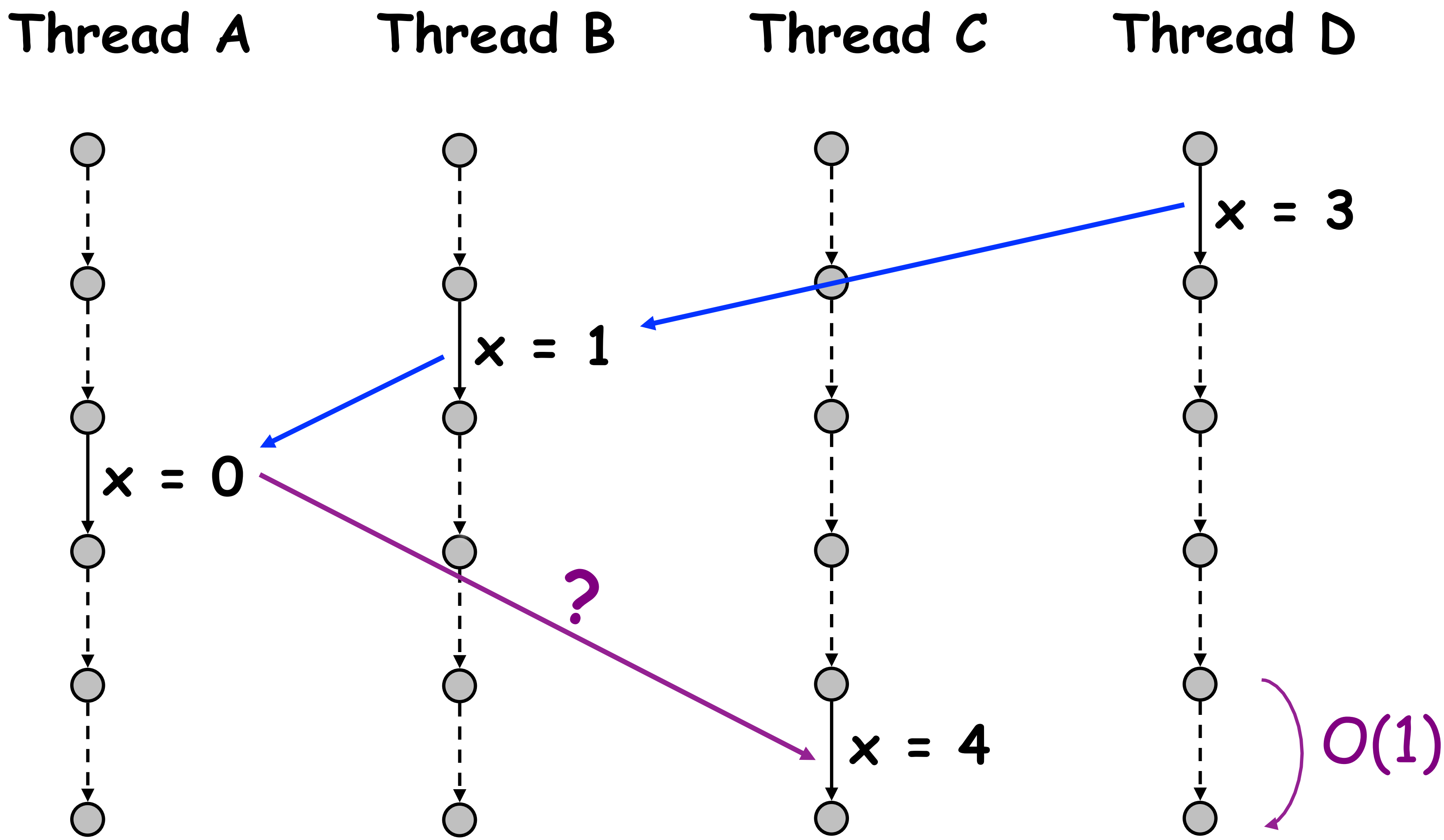
Write-Write and Write-Read Data Races

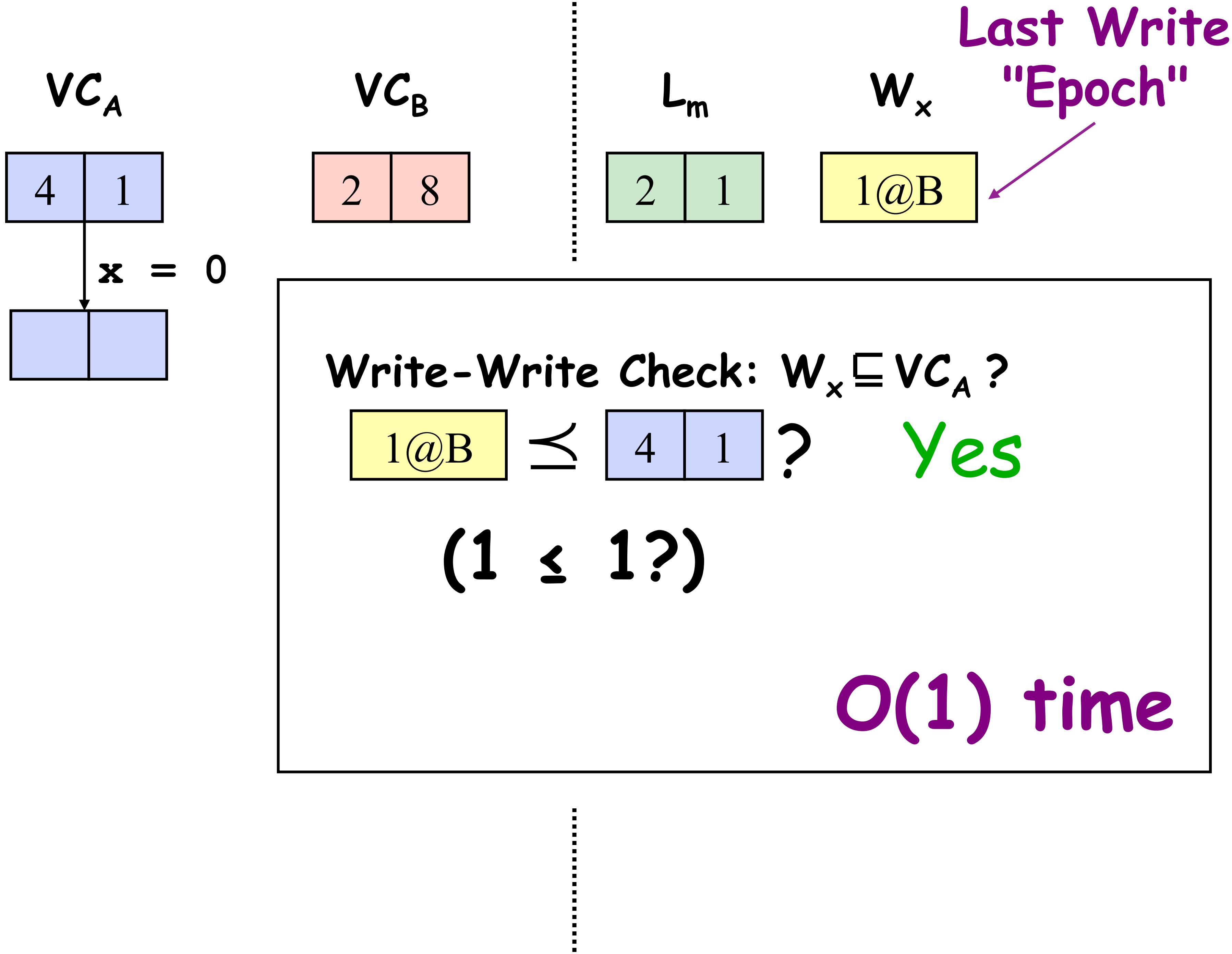


No Data Races Yet: Writes Totally Ordered

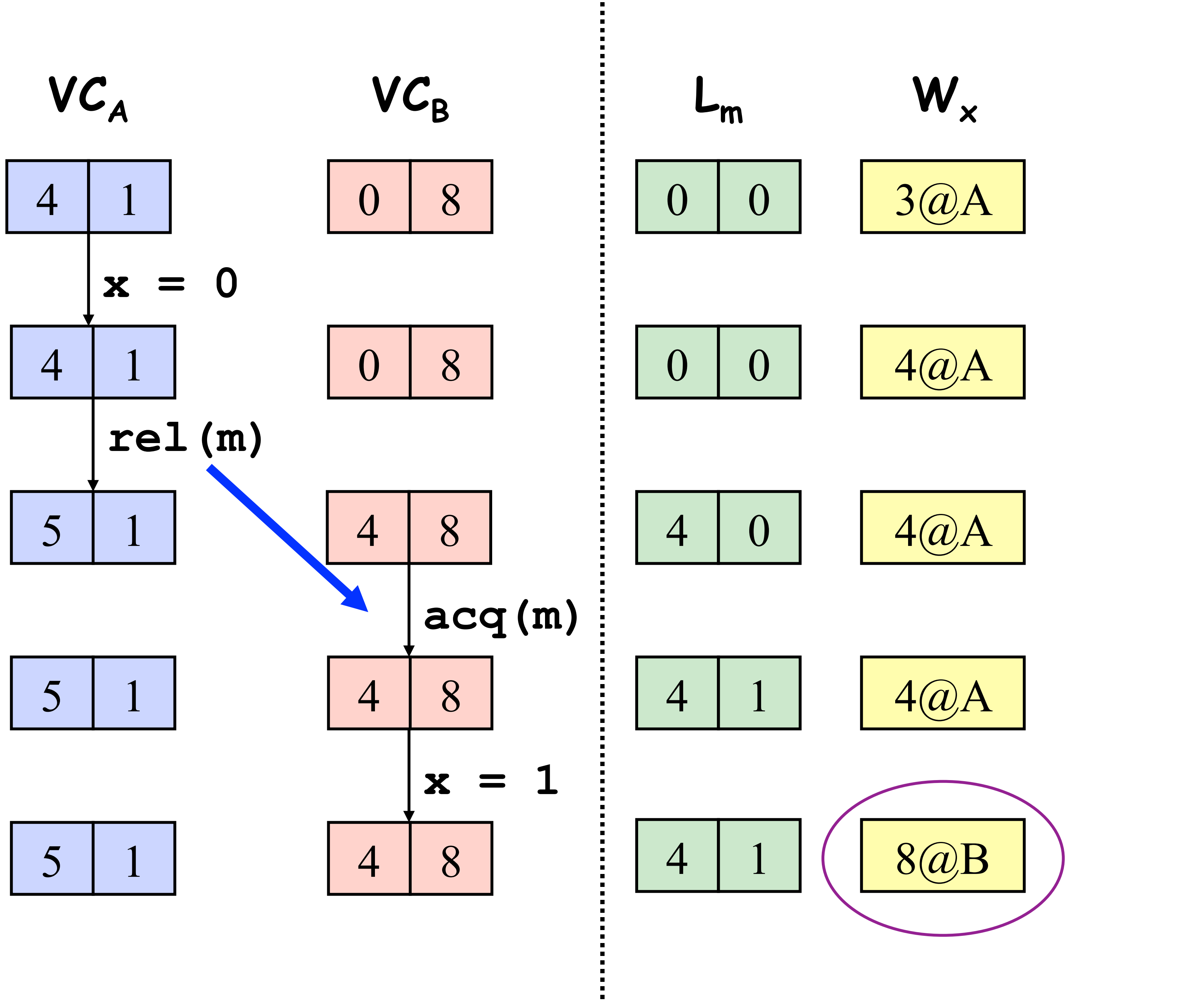


No Data Races Yet: Writes Totally Ordered

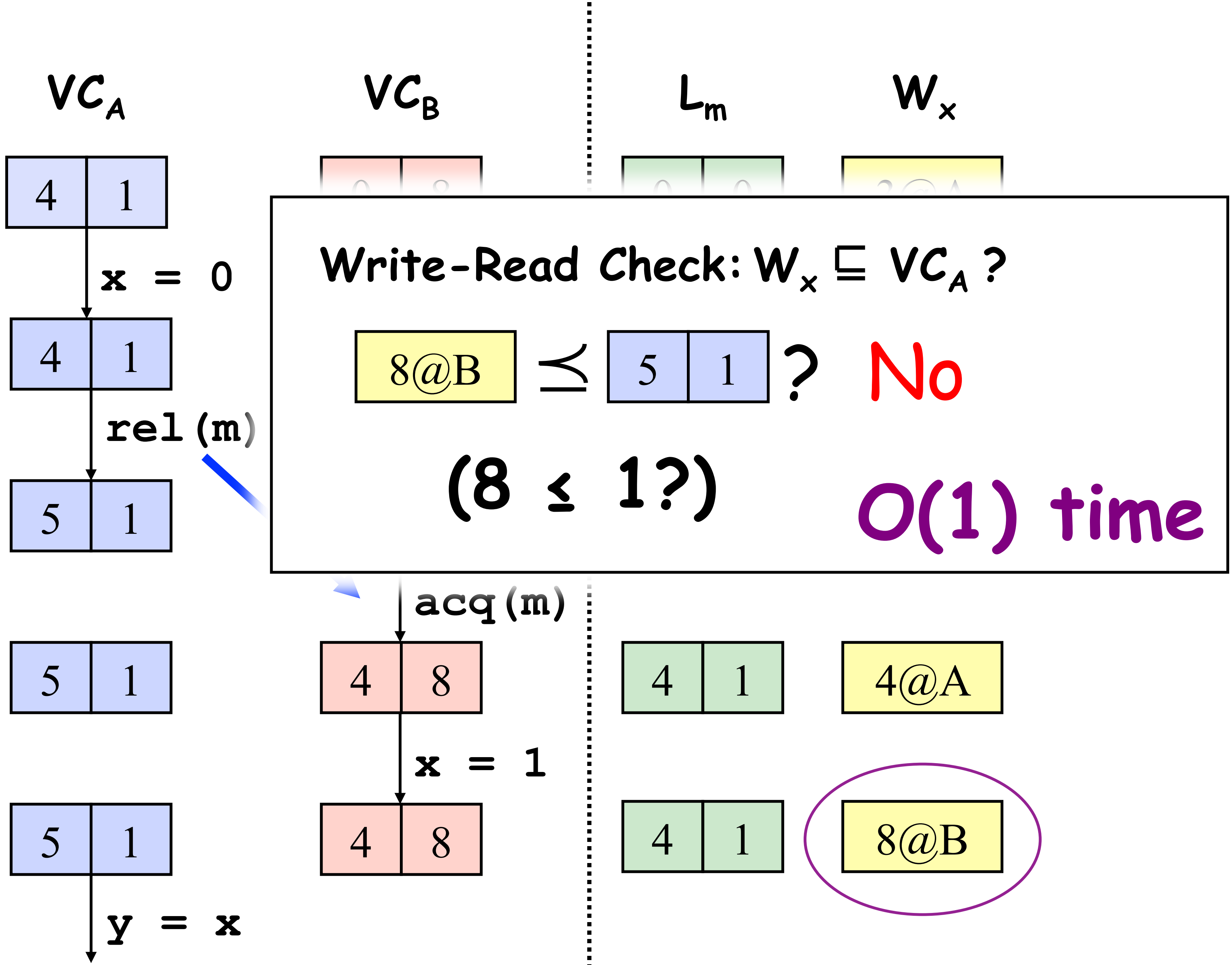




Last Write "Epoch"
VC_A
VC_B
L_m
W_x
4 1
2 8
2 1
1@B
x = 0
Write-Write Check: W_x ⊑ VC_A ?
1@B ⪯ 4 1 ?
Yes
(1 ≤ 1?)
O(1) time

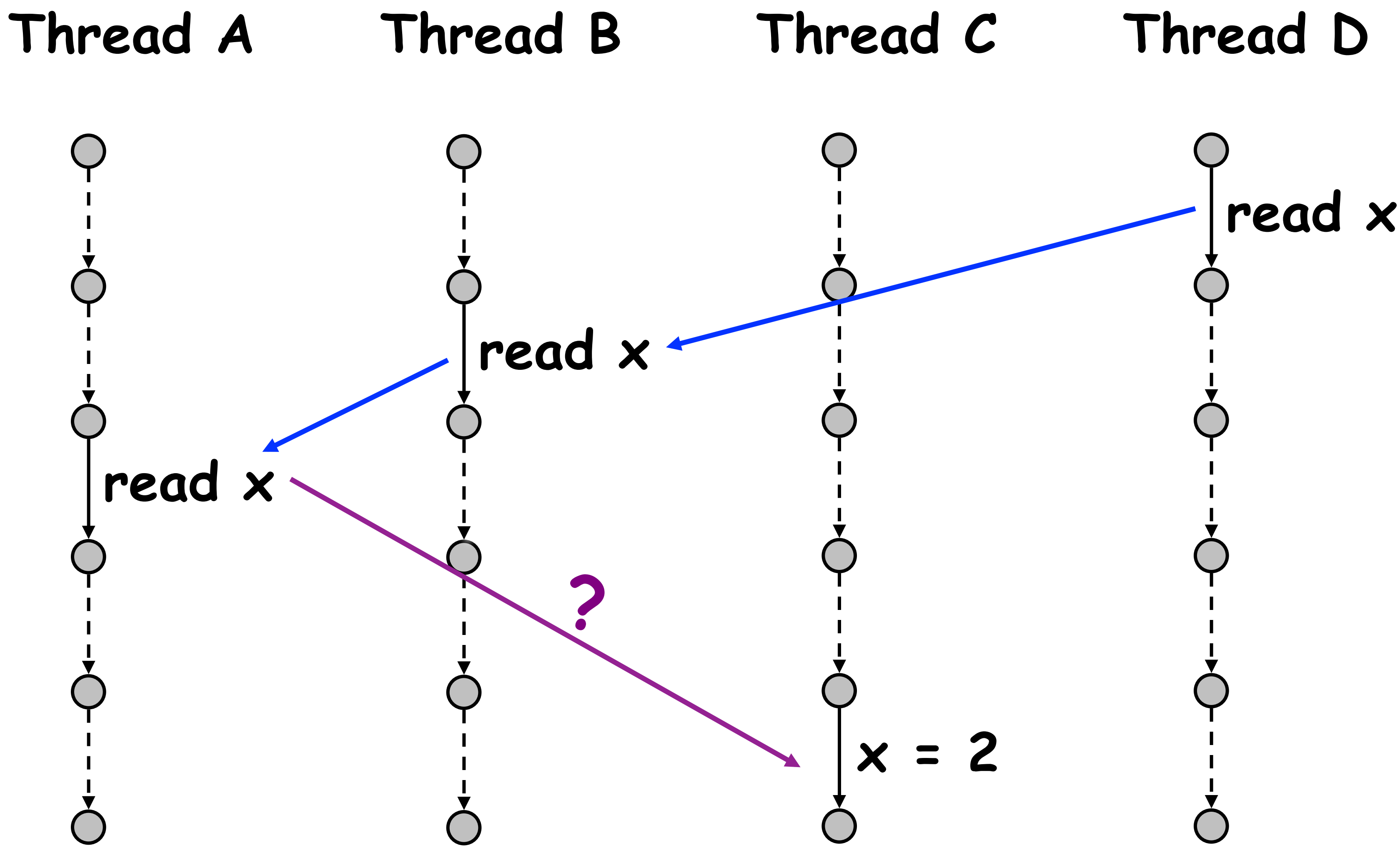


VC_A
VC_B
L_m
W_x
4 1
0 8
0 0
3@A
x = 0
4 1
0 8
0 0
4@A
rel(m)
5 1
4 8
4 0
4@A
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B



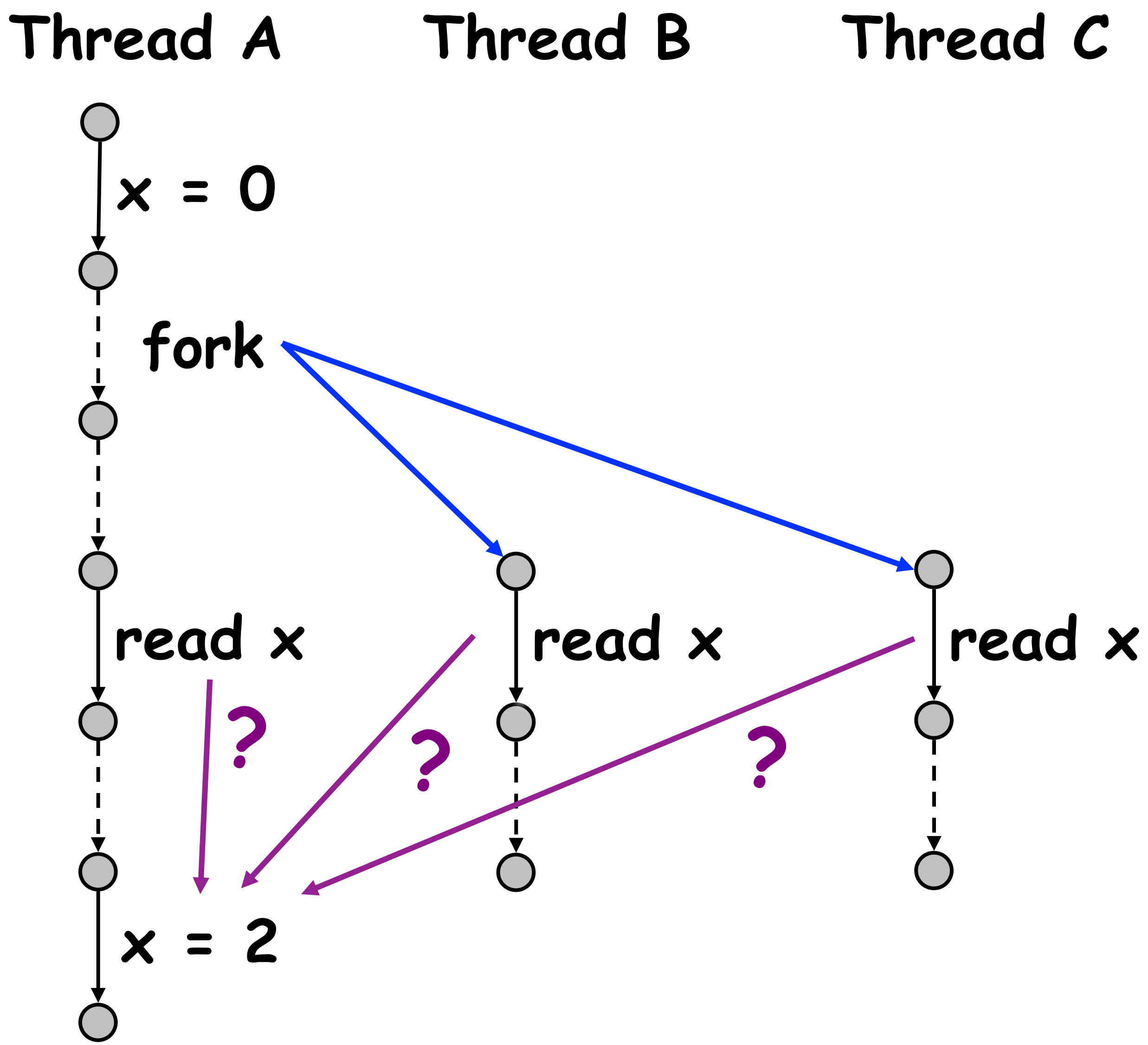
VC_A
VC_B
L_m
W_x
4 1
x = 0
4 1
rel(m)
5 1
acq(m)
5 1
4 8
4 1
4@A
x = 1
5 1
4 8
4 1
8@B
y = x
Write-Read Check: W_x ⊑ VC_A ?
8@B ≼ 5 1 ?
No
(8 ≤ 1?)
O(1) time

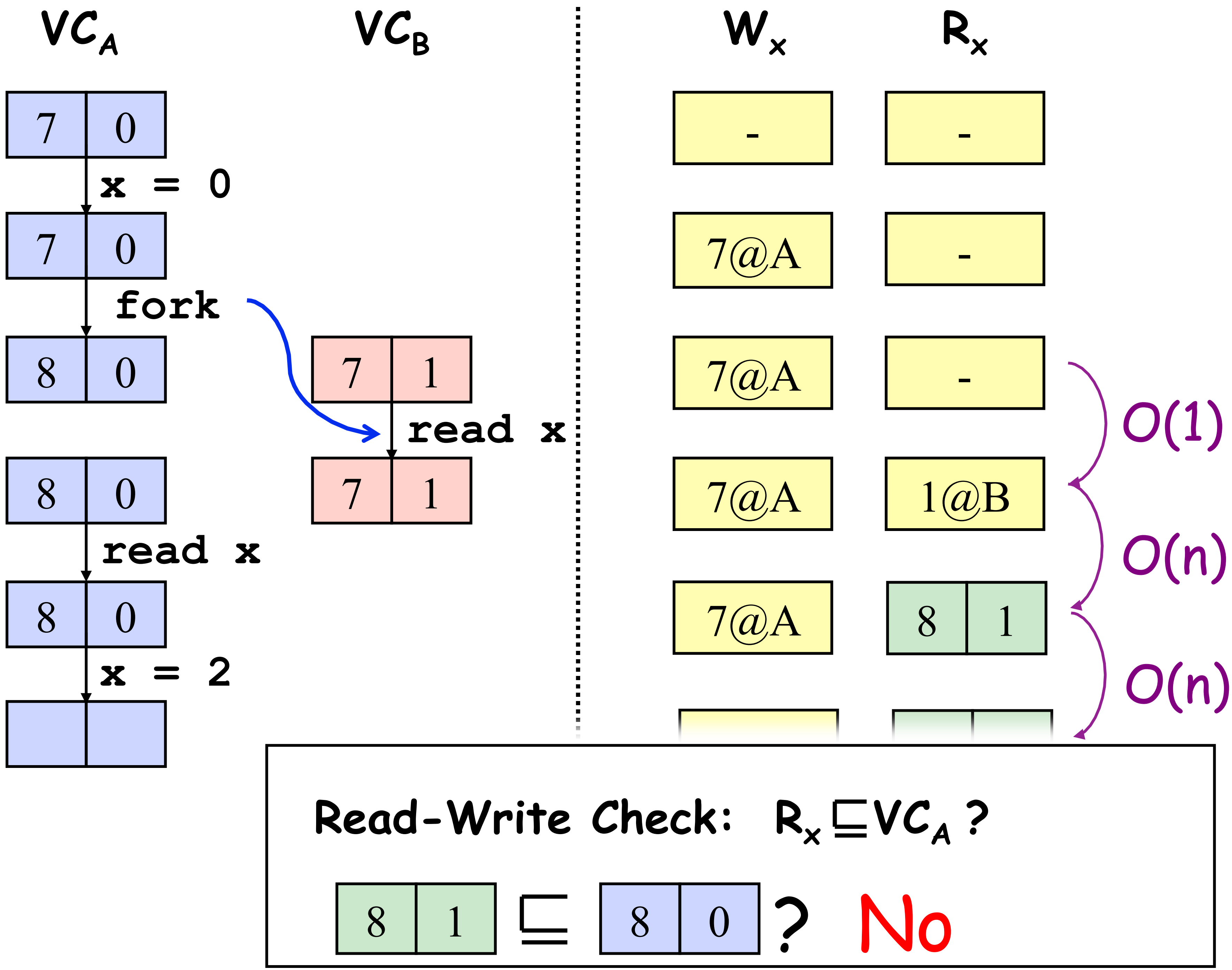
Read-Write Data Races -- Ordered Reads



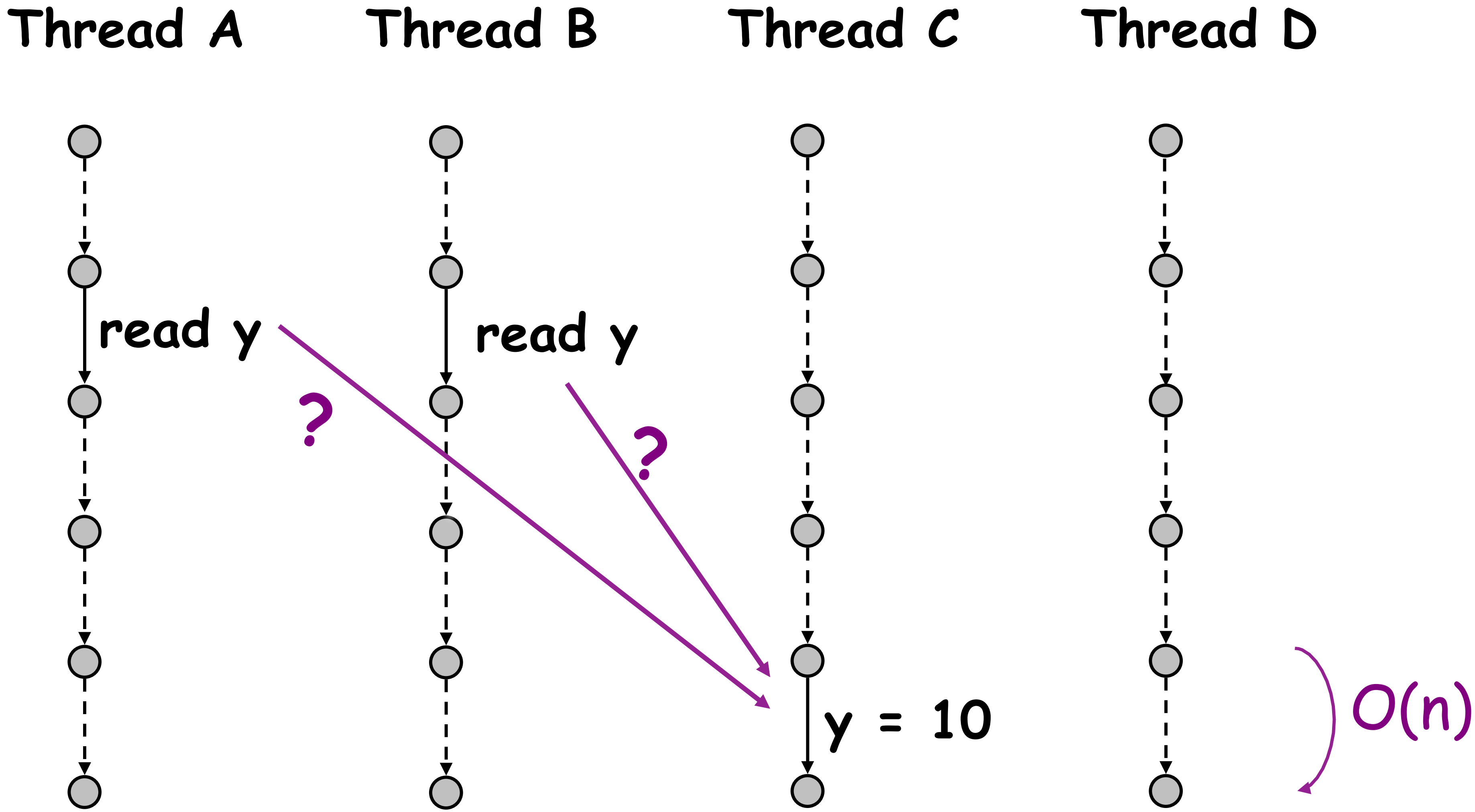
Most common case: thread-local, lock-protected, ...

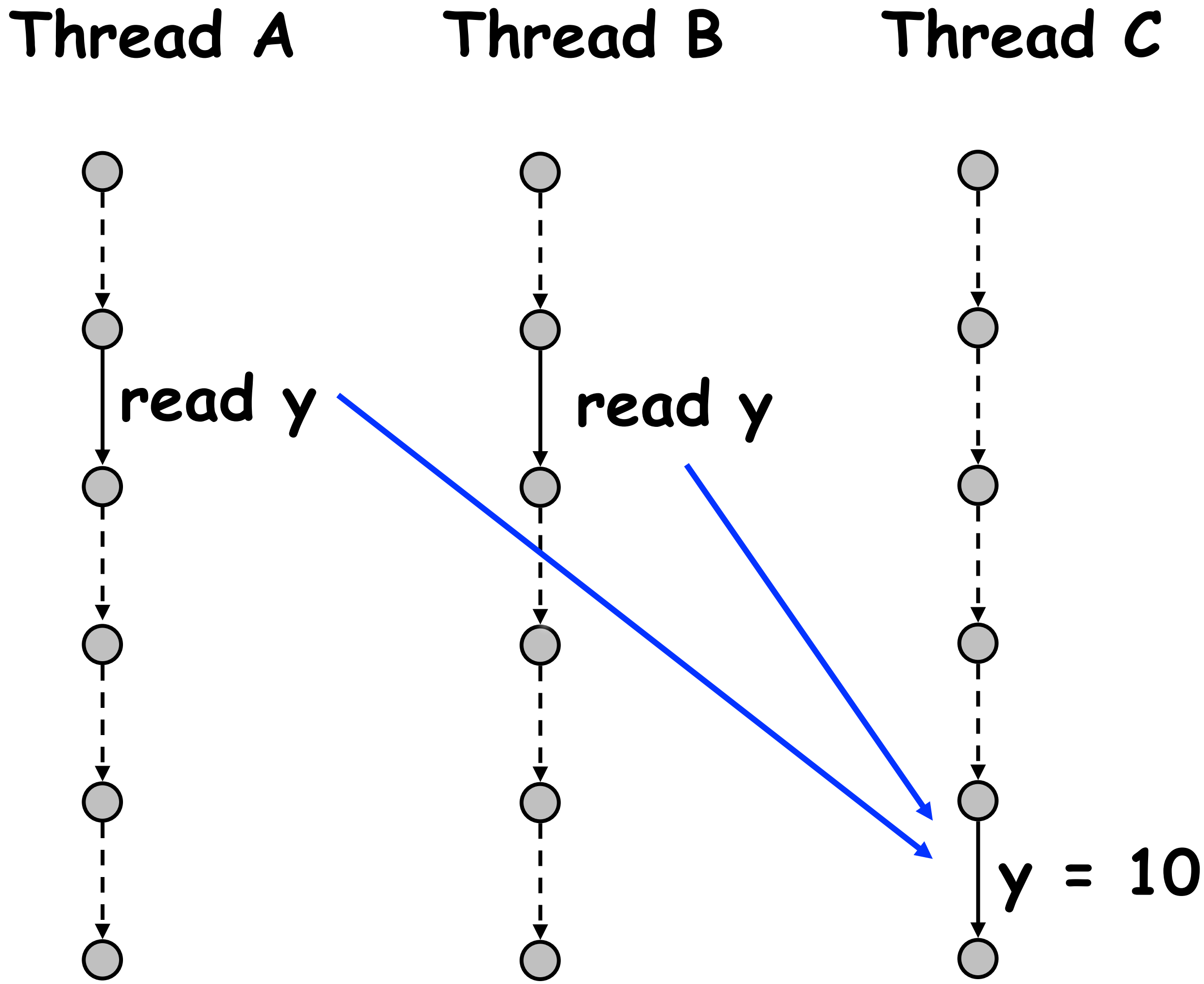
Read-Write Data Races -- Unordered Reads

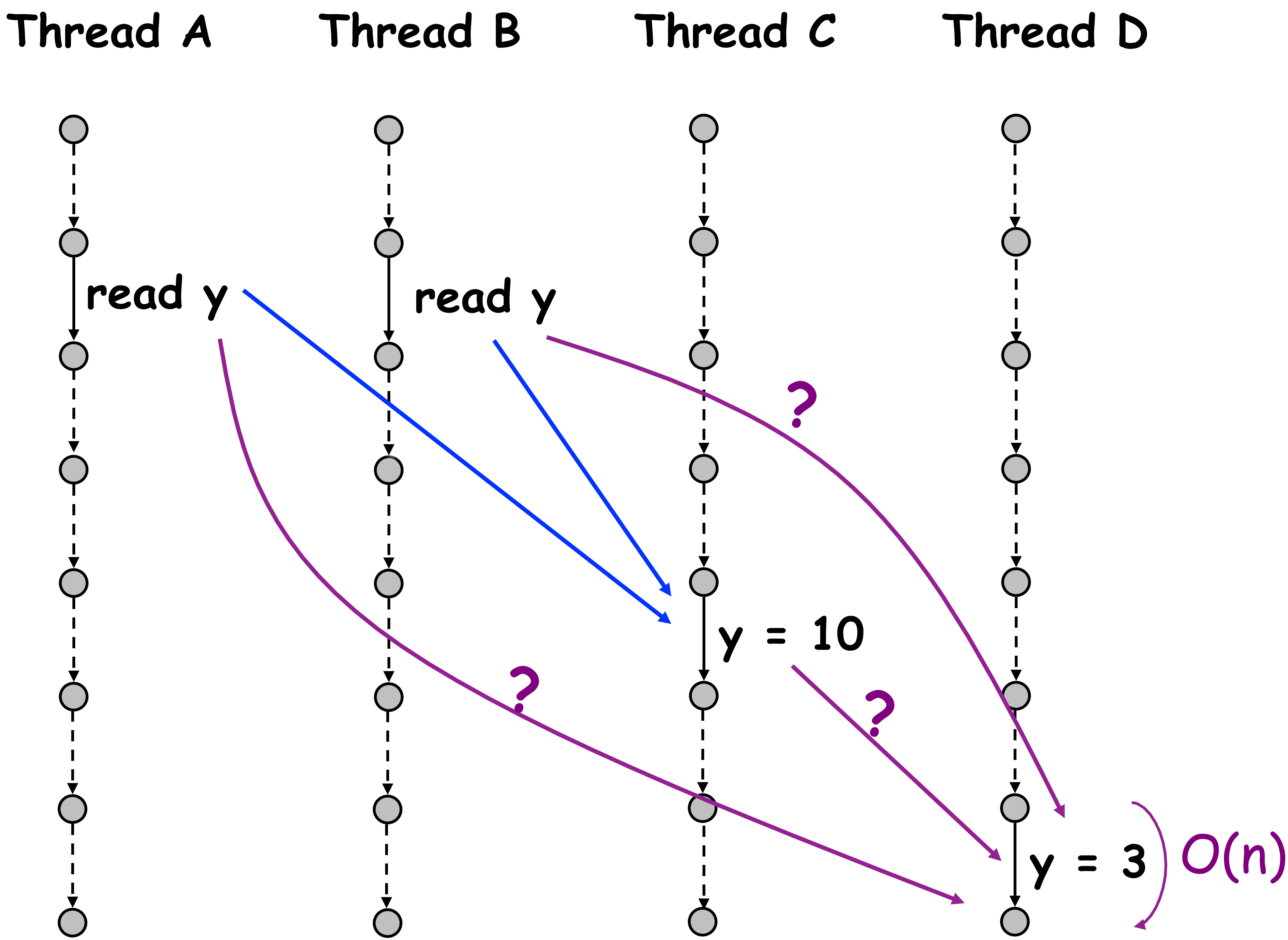


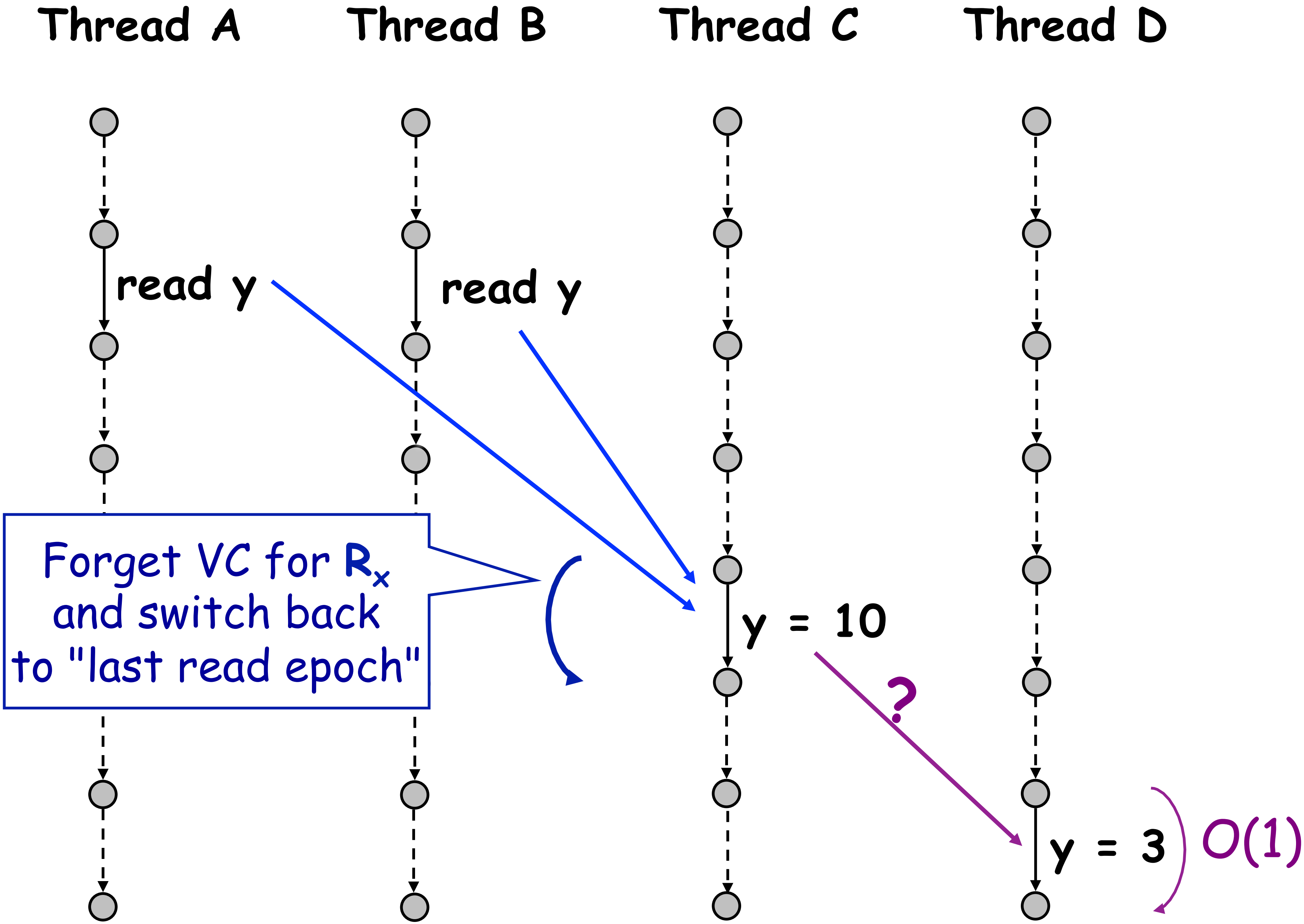


VC_A
VC_B
W_x
R_x
7 0
x = 0
7 0
fork
8 0
7 1
read x
8 0
7 1
read x
8 0
x = 2
-
-
7@A
-
7@A
-
7@A
1@B
7@A
8 1
O(1)
O(n)
O(n)
Read-Write Check: R_x ⊑ VC_A ?
8 1 ⊑ 8 0 ? No


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

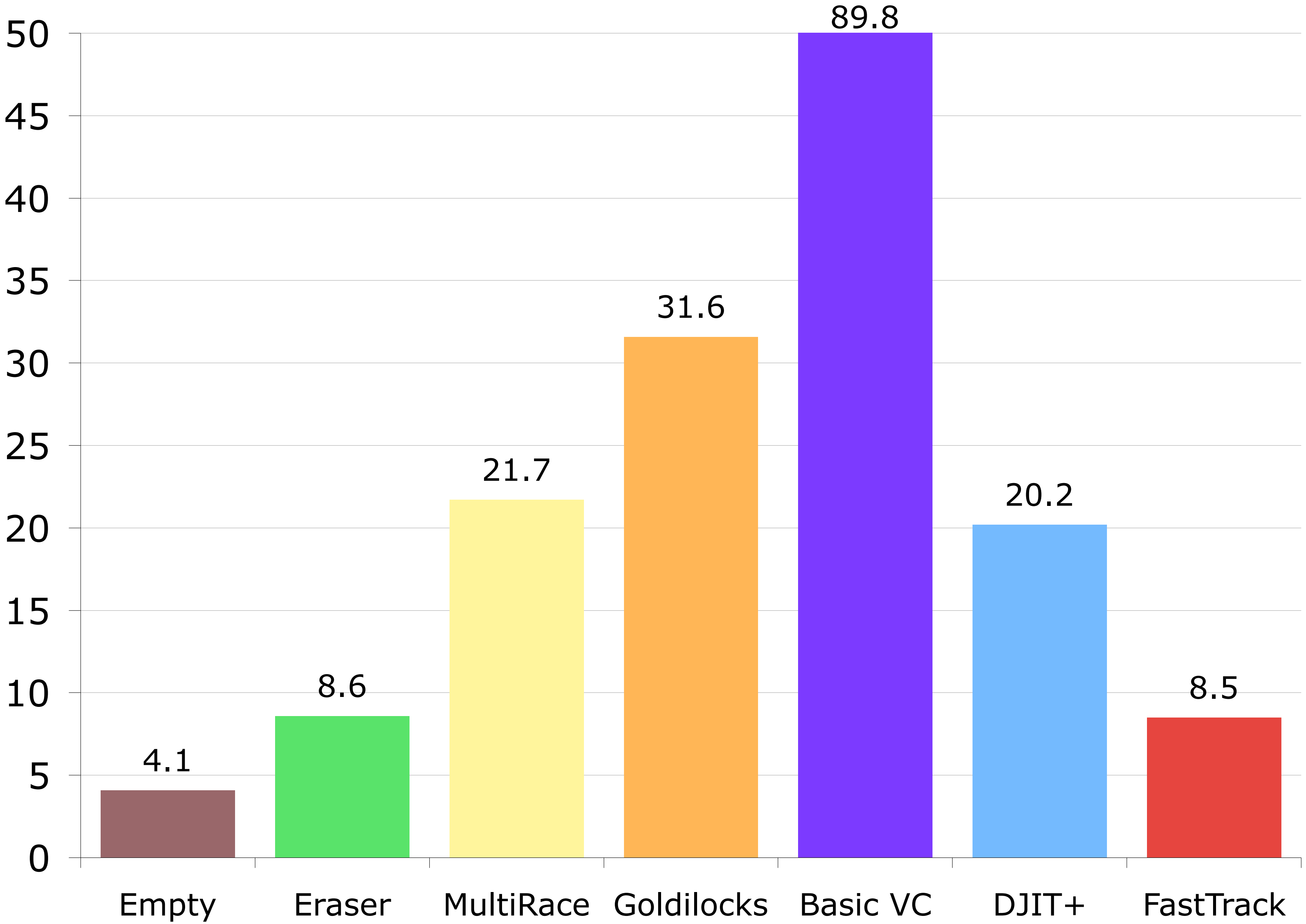

Thread A
Thread B
Thread C
Thread D
read y
read y
y = 10


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

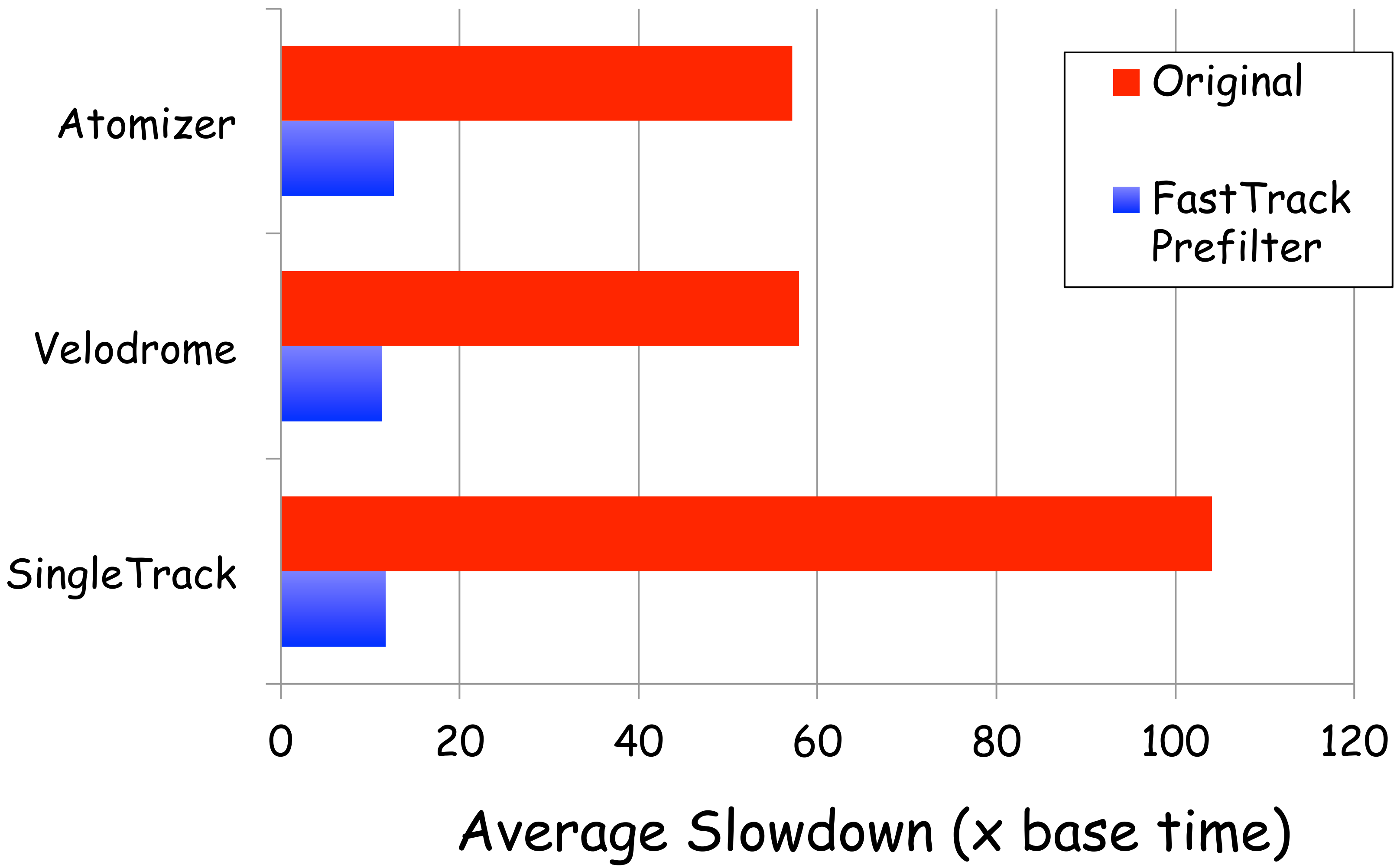
- FastTrack allocated ~200x fewer VCs

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

IMPLEMENTATION FRAMEWORKS

Building a Dynamic Data Race Detector

- Identify synchronization
- Instrument callbacks
  - At synchronization operations
  - At memory operations
- Implement a data race detection algorithm
- Report data races with debugging information

Design Considerations

- Performance overhead
- Tolerance to false positives
- Coverage
- Debuggability

Performance Overhead

- Why is overhead important?

Performance Overhead

- Why is overhead important?
- Tests take longer
- Interaction with timing behavior
  - Databases will trigger deadlock-recovery if transactions don't finish in X ms

Back of the Envelope Calculation

- ~ One in five instructions is a memory operation
- ~ One in two memory operation is to a non-stack location

- Data race detector is called every 10 instructions
- On every callback,
  - Need to perform at least one memory lookup to access the metadata
  - Synchronization to avoid data races (!) on metadata
  - And, we need some cycles to run the algorithm
  - ➔ OVERHEAD

False Positives

- False Data Races
  - A "bug" in the algorithm
    - Lockset will report a race if program does not follow a consistent lock discipline
  - A "bug" in the tool
    - don't cover all synchronizations
- Benign Data Races
  - Data Races that don't trigger assertion violations
  - Prone to memory model issues
    - Need to prove to the user that this data race can cause a problem under some memory model

Coverage

- Given an trace, does the tool find
  - All data races
  - The first data race
- Can the tool find data races in other “related” traces?
  - Happens-before algorithm finds all data races in traces with the same happens-before ordering of synchronization as the original
- Is it acceptable to miss data races?

Debuggability

- So, you have found a data race, now what?
- Need to collect stack trace information
  - For one thread?
  - For both threads?

- Tools usually find tons of data races instances
  - Need a good method to group data race reports

Building a Dynamic Data Race Detector

- Identify synchronization
- Instrument callbacks
  - At synchronization operations
  - At memory operations
- Implement a data race detection algorithm
- Report data races with debugging information

Identifying Synchronization

- Thread synchronization
  - Locks, semaphores, condition variables,…
- Volatile/Atomic accesses
  - Memory model specifies these as "synchronization"
  - Not recognizing them will report lots of benign data races
- Interprocess Communication

Example of IPC over threads

```
Thread A
x ++;
send_pipe();
```

Happens-before

```
Thread B
recv_pipe();
x ++;
```

Failure to handle → False Data Race

Thread A
```
x ++;
send_pipe();
```

Thread B
```
recv_pipe();
x ++;
```

False Data Race!

Data Race ?

```
Thread A
x ++;
malloc();
```

```
Thread B
malloc();
x ++;
```

Data Race ?

- Not, if you consider internal details of malloc()

```
Thread A
x ++;
malloc(){
   lock();
   …
   unlock();
}
```

Happens-before

```
Thread B
malloc(){
   lock();
   …
   unlock();
}
x ++;
```

Instrumenting Callbacks

- At Source
  - Compiler optimizes your instrumentation
  - Need good happens-before specification for third-party (library) binaries

- At Binary
  - More expensive instrumentation
  - Handle (and find data races in) libraries

Processing Callbacks

- Online
  - Run the data-race detection algorithm at runtime
  - Expensive processing
  - At least find one access in the action
- Offline
  - Log the events, and process them later
  - Lightweight processing
  - Log management is an issue

ROADRUNNER

Binary Instrumentation

- Atom, Vulcan, ASM, SOOT, Valgrind, PIN, ...
  (or modifying a VM)
- Can be difficult to build robust/efficient tools
  - Expose most features of actual hardware
  - Complex details of underlying machine
    - object layout, addressing modes, *thread impl.*
  - Hard to optimize instrumentation code
  - Large start-up cost
- Other issues
  - Portability
  - Comparisons between tools

RoadRunner [Flanagan-Freund 10]

1. A *general* framework to facilitate
   - *writing*
   - *composing*
   - *debugging*
   - *comparing*

   dynamic analyses for multithreaded code
2. Efficient for Java, without changing JVM

- Implemented >30 analyses in RoadRunner
  - Performance competitive with analysis-specific implementations built from scratch
  - And with implementations in Jikes RVM [Bond et al]

Using RoadRunner Checking Tools

- Single Checker:

```
rrrun -tool=LockSet Target
rrrun -tool=FastTrack Target
rrrun -tool=HappensBefore Target
```

- Composed Checkers:

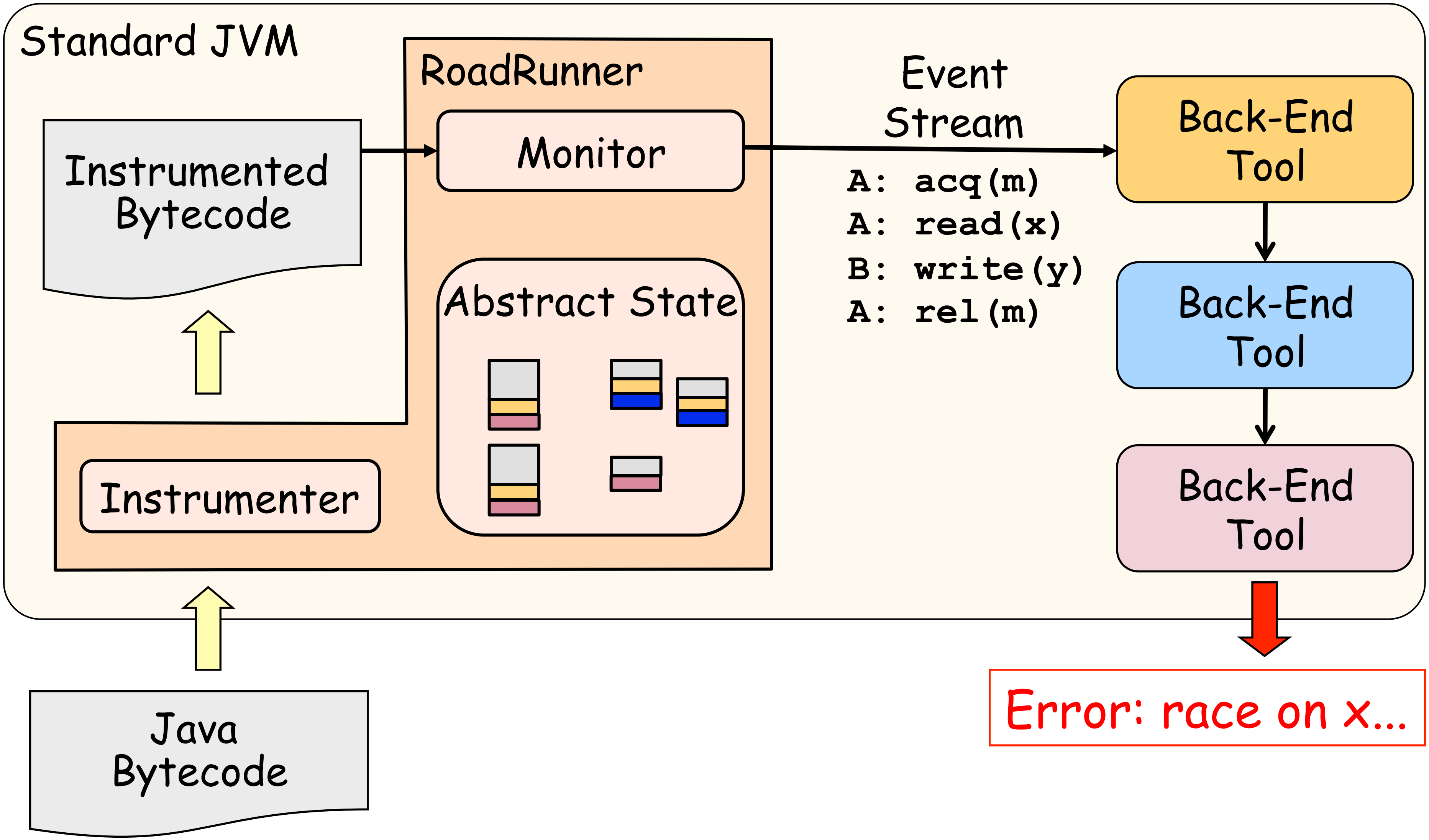
```
rrrun -tool=ThreadLocal:ReadOnly:LockSet Target
rrrun -tool=ThreadLocal:ReadOnly:LockSet:Atomizer Target
rrrun -tool=FastTrack:Atomizer Target
```

- Diagnostic Tools:

```
rrrun -tool=ThreadLocal:Print Target
rrrun -tool=FastTrack:Count:Atomizer Target
```

RoadRunner Tools	Size (lines)	Description
Empty	35	"No Op" Back End
Print	170	Print synch / memory ops
ThreadLocal	48	Local vs. Shared Data
LockSet [SBN97]	327	Race Conditions
DJIT+ [PS 07]	582	
MultiRace [PS 07]	923	
Goldilocks [EQT 07]	1,416	
FastTrack [FF 09]	758	
Atomizer [FF 04]	245	Serializability
Velodrome [FFY 08]	1,088	
SingleTrack [SFF 09]	1,655	Deterministic Parallelism
Jumble [FF 10]	1,326	Adversarial Memory
SideTrack [YSF 09]	500	Trace generalization

Architecture



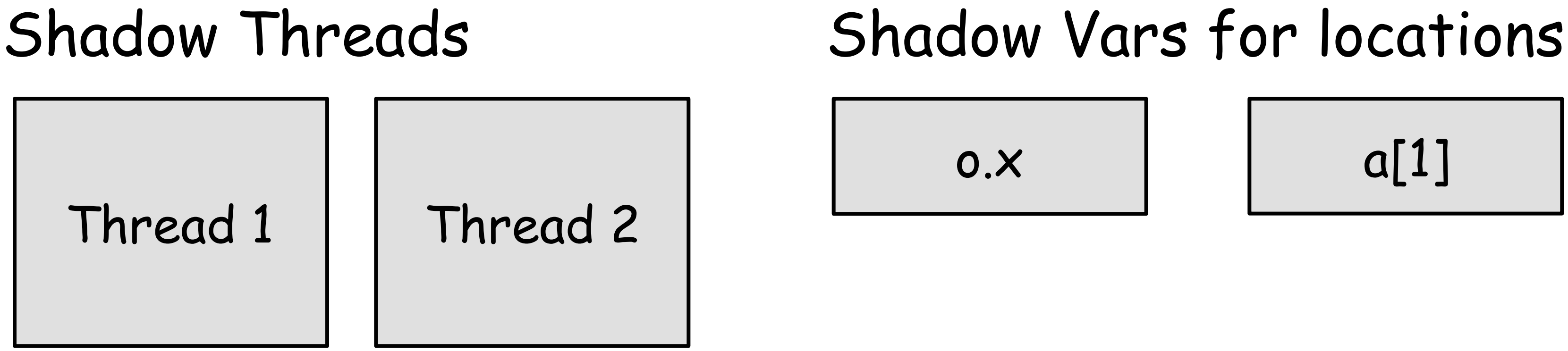
Others: Sofya [KDR 07], CalFuzzer JNPS 09]

Tool API (Without Composition)

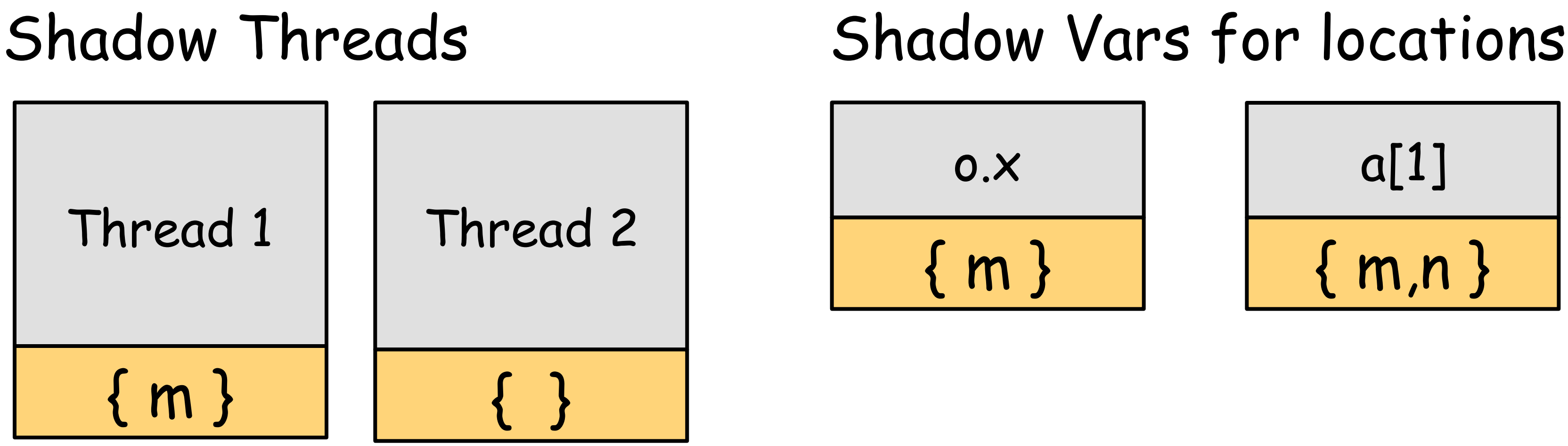
- Tool specifies:
  - handlers for synchronization / access events
  - data to store about abstract program state

```
abstract class Tool {
  void create(NewThreadEvent e)
  void acquire(AcquireEvent e)
  void release(ReleaseEvent e)
  void access(AccessEvent e)
  ...
}
```

RR Abstract State



RR Abstract State: LockSet



Decorations for Shadow Threads

- Maps with constant-time operations
  - Creation:

```
Decoration<ShadowThread,LSet> held =
  ShadowThread.makeDecoration(LSet.empty());
```

  - Usage:

```
LSet ls = held.get(thread);
held.set(thread, ls.add(lock));
```

- Values kept in small array stored in Key objects

Variable Shadows for Locations

- Different requirements
  - orders of magnitude more locations & uses
  - performance critical
  - decoration overhead too large

- RR stores single **`ShadowVar`** value for each loc.

- Tool specifies value for fresh location:

```
ShadowVar makeShadowVar(AccessEvent e) {
  return held.get(e.thread);
}
```

Event Stream

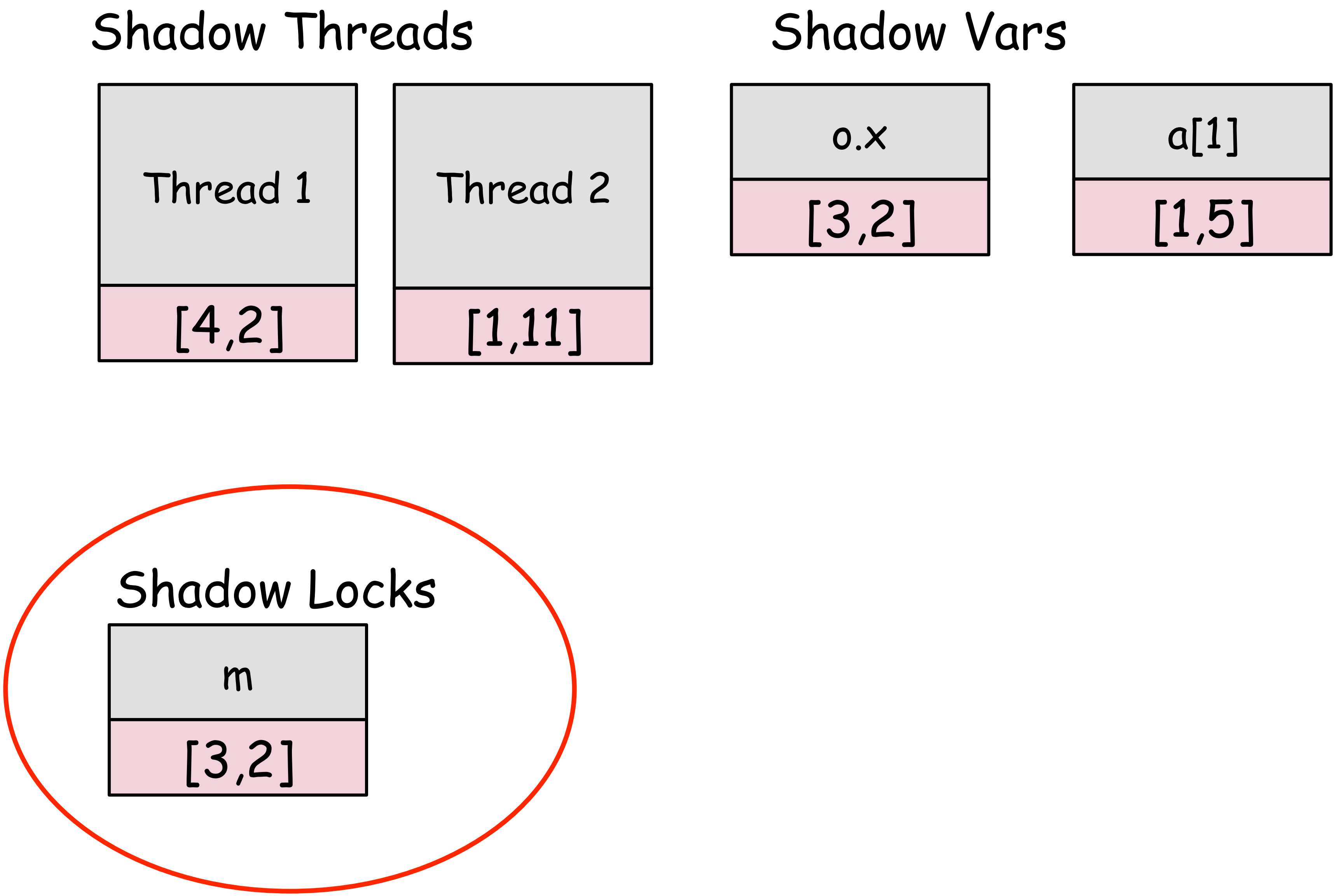
```
class AcquireEvent {
  AcquireInfo info;
  ShadowThread thread;
  ShadowLock lock;
}

class AccessEvent {
  AccessInfo info;
  ShadowThread thread;
  ShadowVar shadow;
  boolean putShadow(ShadowVar var)
}
```

update ShadowVar stored for location

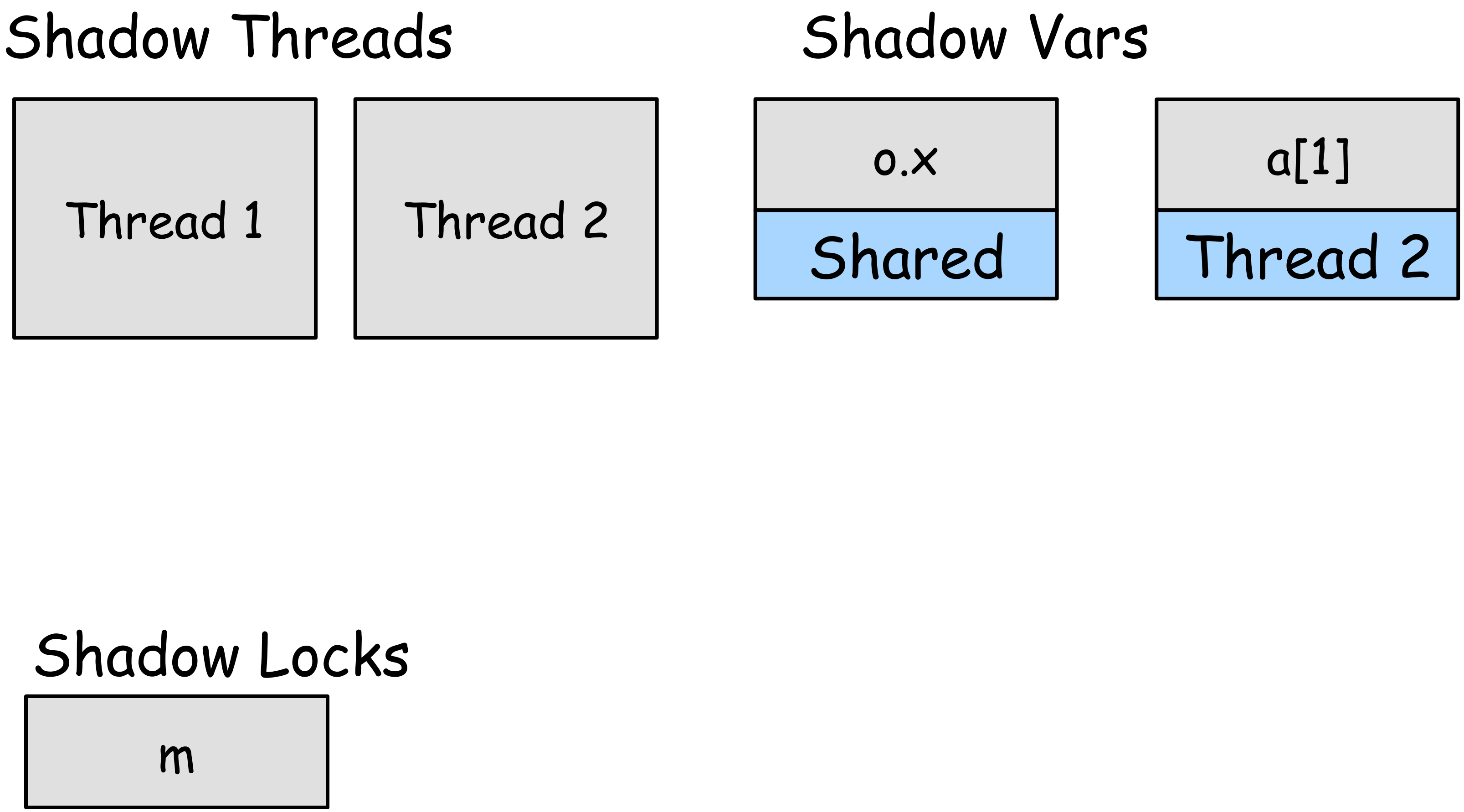
LockSet Handlers

```
public void acquire(AcquireEvent e) {
    LSet ls = held.get(e.thread);
    held.set(e.thread, ls.add(e.lock));
}

public void access(AccessEvent e) {
    LSet locks = (LSet)e.shadow;
    LSet held = held.get(e.thread);
    LSet newLocks = locks.intersect(held);
    e.putShadow(newLocks);
    if (newLocks.isEmpty()) {
      error(e.info);
    }
}
```

RR Abstract State: **HappensBefore**



RR Abstract State: ThreadLocal

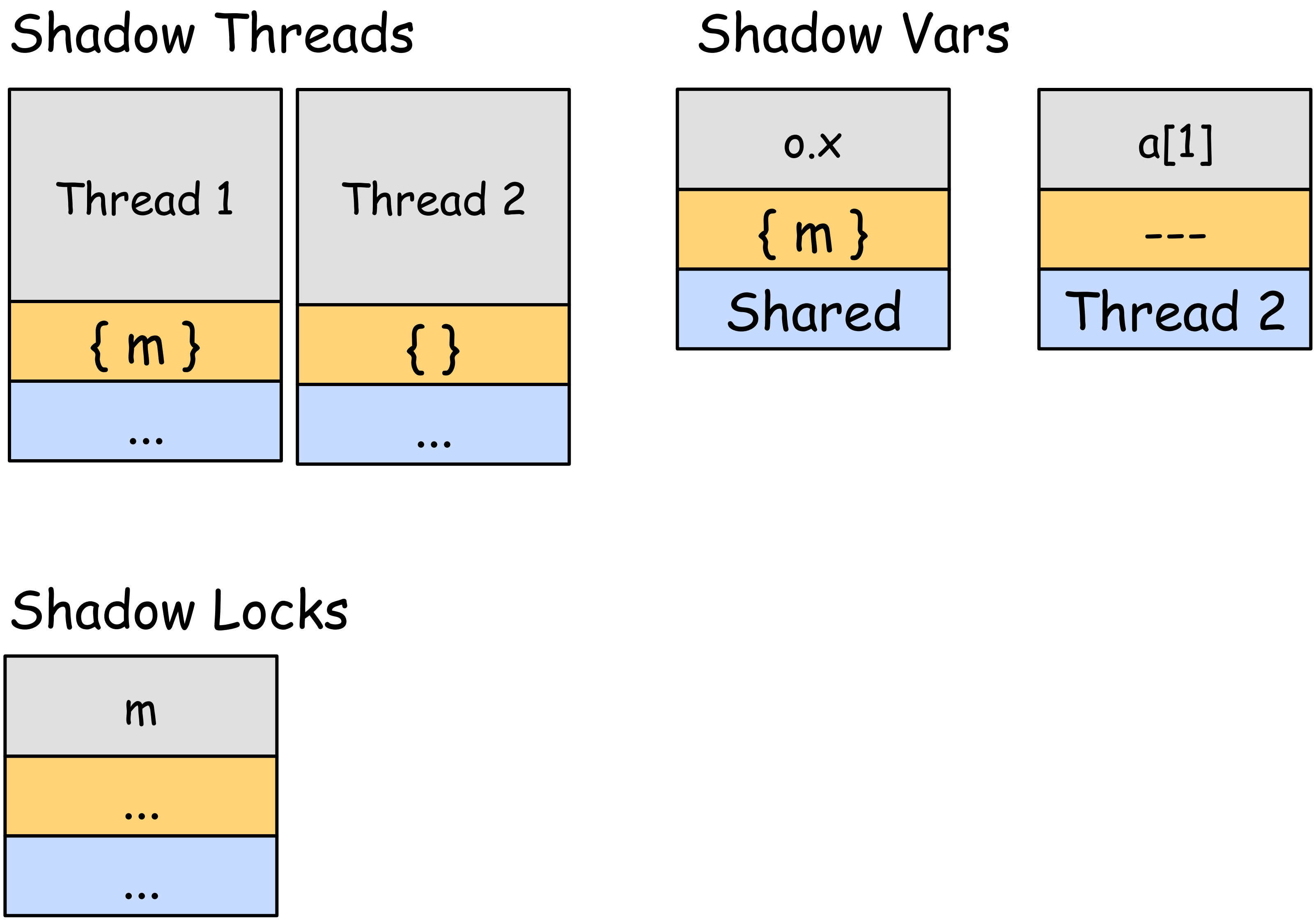


Tool API (With Composition)

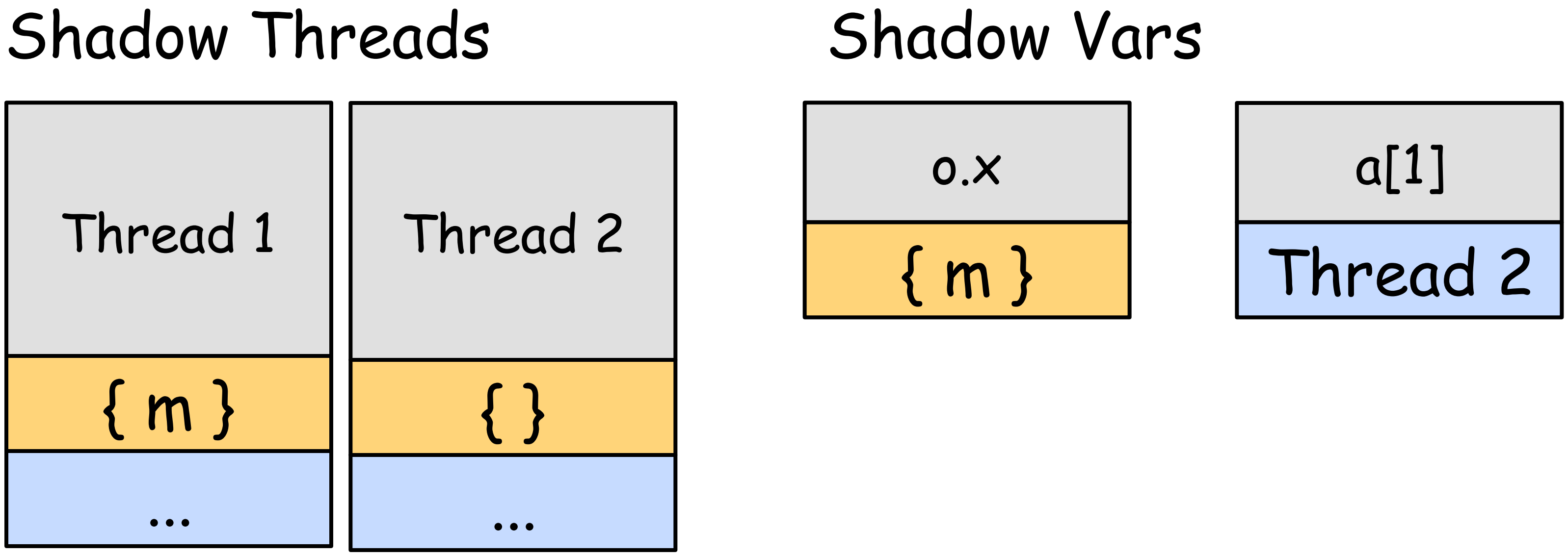
```
abstract class Tool {

  Tool next;

  void create(NewThreadEvent e) { next.create(e);  }
  void acquire(AcquireEvent e)  { next.acquire(e); }
  void release(ReleaseEvent e)  { next.release(e); }
  void access(AccessEvent e)    { next.access(e);  }
  ...
}
```

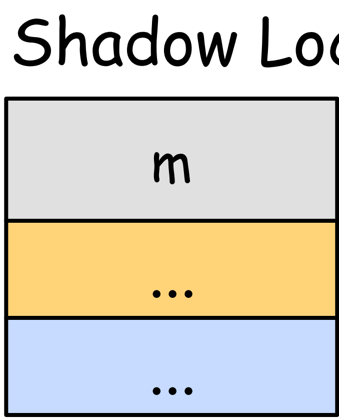
- Every Tool:
  - must pass all sync events to **next**
  - can filter out access events

Composed Tools: "ThreadLocal:LockSet"



ShadowVar Ownership





- Still keep a single `ShadowVar` for each location
- Type indicates current owner
- Tool explicitly passes ownership to next tool in chain

```
public void acquire(AcquireEvent e) {
   LSet ls = held.get(e.thread);
   held.set(e.thread, ls.add(e.lock));
}


public void access(AccessEvent e) {
     LSet locks = (LSet)e.shadow;
     LSet held = held.get(e.thread);
     LSet newLocks = locks.intersect(held);
     e.putShadow(newLocks);
     if (newLocks.isEmpty()) {
        error(e.info);
     }
}
```

```
public void acquire(AcquireEvent e) {
   LSet ls = held.get(e.thread);
   held.set(e.thread, ls.add(e.lock));
   next.acquire(e);
}

public void access(AccessEvent e) {
   if (!(e.shadow instanceof Set)) {
     next.access(e);          // Not owner
   } else {
     LSet locks = (LSet)e.shadow;
     LSet held = held.get(e.thread);
     LSet newLocks = locks.intersect(held);
     e.putShadow(newLocks);
     if (newLocks.isEmpty()) {
        error(e.info); this.advance(e);
     }
  }
```

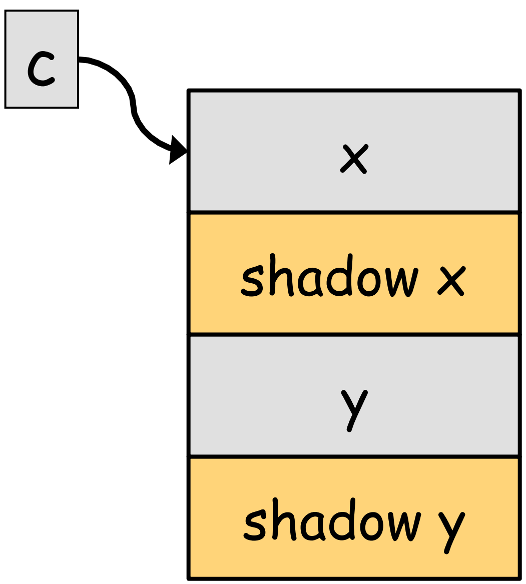
Performance

Tool	Slowdown (x base time)
Empty	5.6
Eraser (=ThreadLocal:ReadOnly:LockSet)	9.4
Eraser:RedundantSync:Atomizer	9.8
FastTrack	7.3
FastTrack:Velodrome	8.1

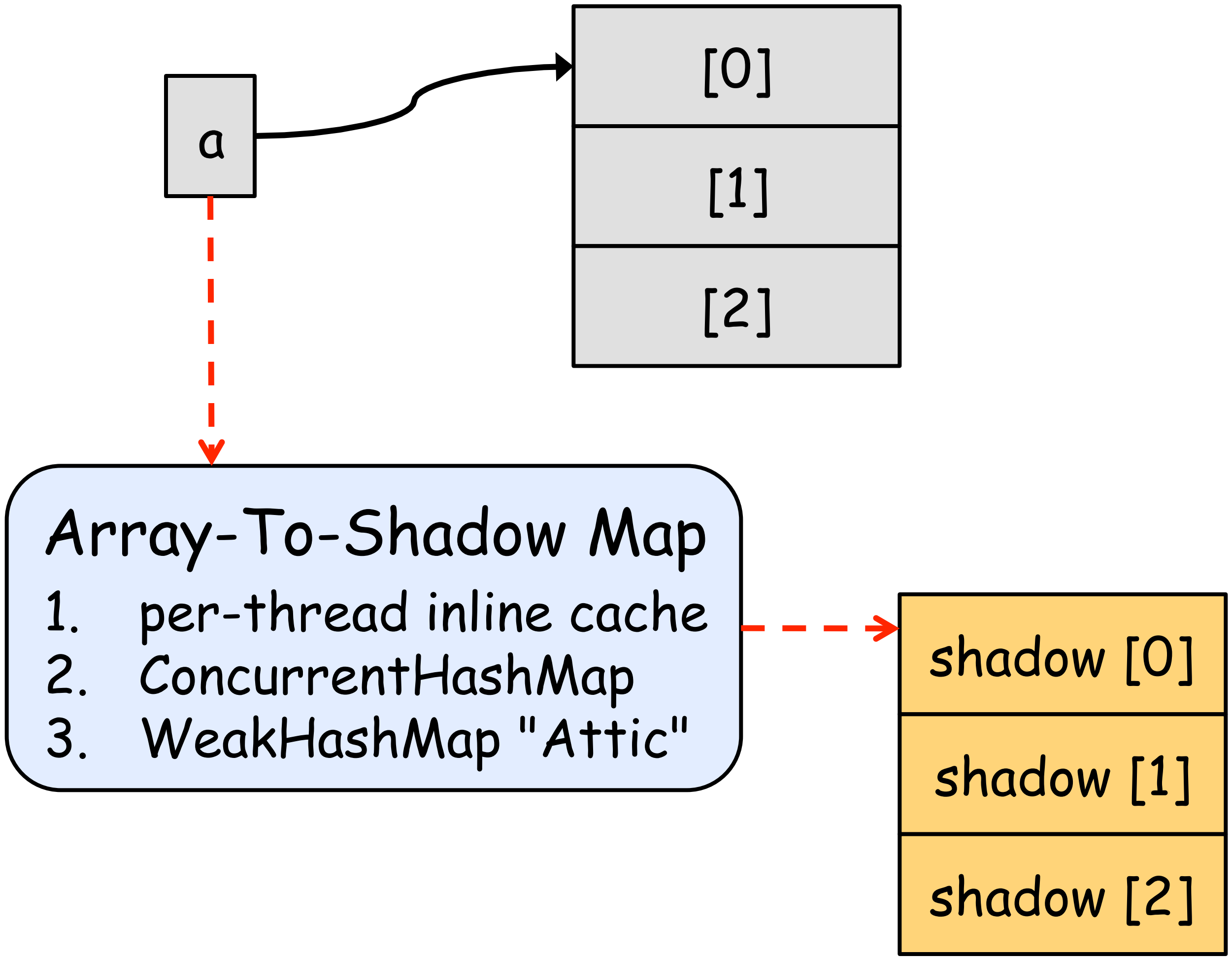
- Memory Overhead: at least 2x, due to **`ShadowVars`**
- Running times are competitive with analysis-specific checkers built from scratch

Implementation: ShadowVar State

```
class C {
   int x;
   int y;
}
...
C c = ...;
c.x = 3;
```




```
int a[] = ...;
a[2] = 3;
```



Implementation: Event Handling

- Thread performing operation executes handler
- Avoiding data races on ShadowVar for location:
  - serialize event stream
  - tool-provided synchronization
  - optimistic updates

```
public void access(AccessEvent e) {
  LSet held = held.get(e.thread);
  do {
    LSet locks = (LSet)e.shadow;
    LSet newLocks = locks.intersect(held);
  } while (!e.putShadow(newLocks));
  ...
}
```

Implementation: Optimizations

- Leverage JIT
- Event Object Reuse
- Array-To-Shadow Map
- Fast Path Inlining
  - most access events handled without modifying state or using full event info
  - RoadRunner inlines these "fast paths"

```
boolean readFP(ShadowVar v, ShadowThread cur) {
  return v == held.get(cur)
         && !((LSet)v).isEmpty();
}
```

Perspective

The Good

- JIT works great
- Efficient & Scalable
  - Eclipse, dacapo (mostly),...
- Event model matches analysis specification
- Uniform comparisons
- Tool composition
  - prototyping
  - debugging
  - profiling

Rough Edges

- JIT is moving target...
- Further scalability
  - mitigate memory overhead
  - offline instrumentation
- Hard JVM features
  - custom class loaders
  - native code
  - serialization
  - native libs
- Not C/C++

DATACOLLIDER: (NEAR) ZERO-OVERHEAD DATA-RACE DETECTION

A Data Race in Windows

```
RunContext(...)
{
  pctxt->dwfCtxt &=
    ~CTXTF_RUNNING;
}
```

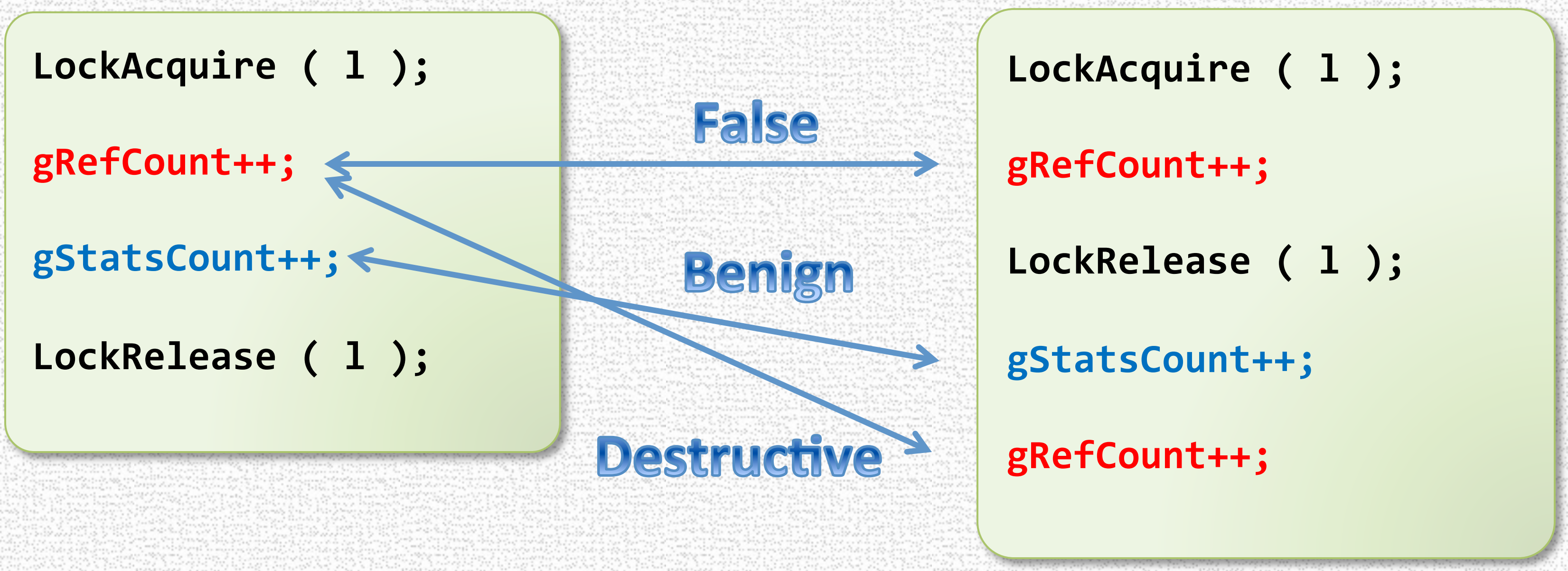
```
RestartCtxtCallback(...)
{
  pctxt->dwfCtxt |=
    CTXTF_NEED_CALLBACK;
}
```

- Clearing the RUNNING bit swallows the setting of the NEED_CALLBACK bit
- Resulted in a system hang during boot
  - Reproducible only on one hardware configuration
  - This bug caused release delays on said system
  - The hardware had to be shipped from Japan to Redmond for debugging

DataCollider

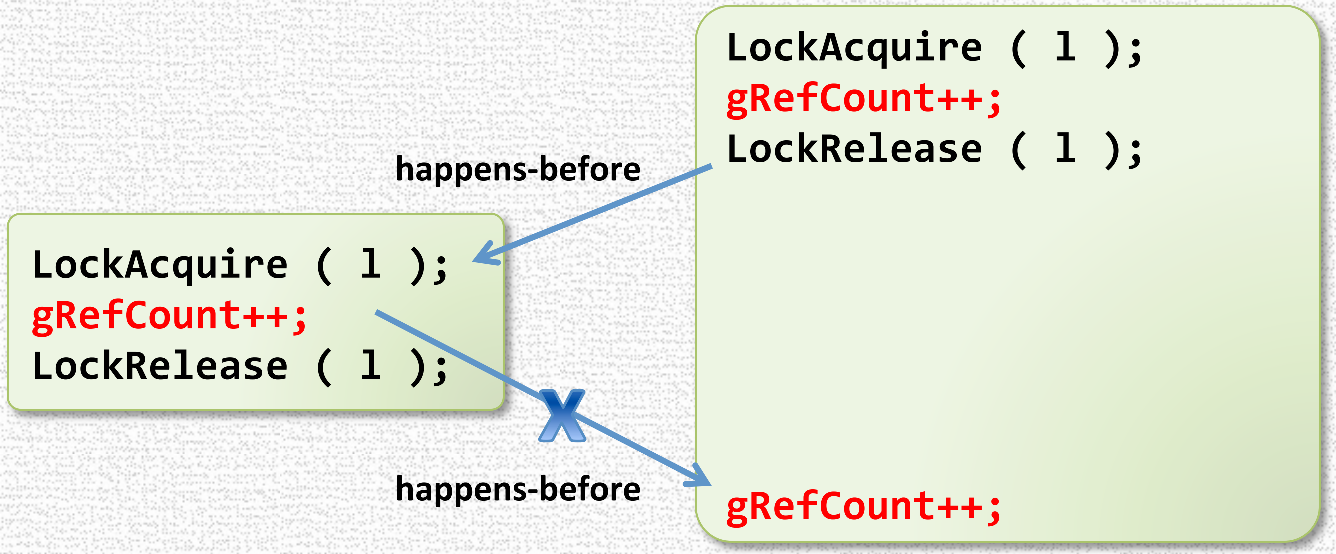
- A runtime tool for finding data races
- Low runtime overheads
- Readily implementable
  - Works for kernel-mode and user-mode Windows programs
- Successfully found many concurrency errors in
  - Windows kernel, Windows shell, Internet Explorer, SQL server, …

False vs. Benign Data Races



Existing Dynamic Approaches for Data-Race Detection

- Log data and synchronizations operations at runtime
- Infer conflicting data access that can happen concurrently
  - Using happens-before or lockset reasoning



Challenge 1: Large Runtime Overhead

- Classic example: Intel Thread Checker has 200x overhead
- BOE calculation for logging overheads
  - Logging sync. ops ~ 2% to 2x overhead
  - Logging data ops ~ 2x to 10x overhead
  - Logging debugging information (stack trace) ~ 10x to 100x overhead
- Large overheads skew execution timing
  - A kernel build is "broken" if it does not boot within 30 seconds
  - SQL server initiates deadlock recovery if a transaction takes more than 400 microseconds
  - Browser initiates recovery if a tab does not respond in 5 seconds
- New techniques (e.g. FastTrack) reduce overhead, but …

Challenge 2: Complex Synchronization Semantics

- Correctness depends on *exact* knowledge of synchronization
- Synchronizations can be homegrown and complex
  - (e.g. lock-free, events, processor affinities, IRQL manipulations,…)
- Missed synchronizations can result in false data races

```
AcquireMutex(gLock);
gRefCount++;
ReleaseMutex(gLock);
```

happens-before

```
AcquireMutex(gLock);
gRefCount++;
ReleaseMutex(gLock);
```

```
OpenFile(“foo”, EXCLUSIVE);
gRefCount++;
CloseFile();
```

happens-before ?

```
OpenFile(“foo”, EXCLUSIVE);
gRefCount++;
CloseFile();
```

Challenge 2: Complex Synchronization Semantics

- With multiple levels of interrupts, what is a thread?
  - In some ways each <thread, interrupt level> is its own execution entity
  - However, pre-thread data is shared across levels
  - Interrupt levels are their own form of synchronization

```
Device.Buffer = {0};
Device.SendWorkToHw();
```

```
Device::OnInterrupt()
{       Device.Buffer =
ReadHw();
}
```

```
RaiseIrql(INTERRUPT_LEVEL);
Device.Buffer = {0};
LowIrql();
```

?

```
Device::OnInterrupt()
{       Device.Buffer =
ReadHw();
}
```

```
SetAffinity/RaiseIrql();
inc RefCount[CurrentProc()]
ClearAffinity/LowerIrql();
```

```
SetAffinity/RaiseIrql();
inc RefCount[CurrentProc()]
ClearAffinity/LowerIrql();
```

Challenge 3: Actionable Data

- Information about data races help only insofar as it identifies the root cause

- Recording the state of the program is expensive for methods that use logging
  - Any data needed for debugging must be recorded for *every* memory access that could *potentially* be part of a data race.
  - *E.g*. If a stack trace is desired, then every memory access that *might* be part of a data race must have the stack trace stored.

DataCollider Key Ideas

- Cause a data-race to happen, rather than infer its occurrence
  - No inference => oblivious to synchronization protocols
  - Catching threads “in the act” => actionable error reports
- Use hardware breakpoints for hooks and conflict detection
  - Hardware does all the work => low runtime overhead
- Use sampling
  - Randomly sample accesses as candidates for data-race detection *at a user-controlled overhead*

Algorithm

- Randomly sprinkle code breakpoints on memory accesses
- When a code breakpoint fires at an access to x
  - Set a data breakpoint on x
  - Delay for a small time window
- Read x before and after the time window
  - Detects conflicts with non-CPU writes
  - Or writes through a different virtual address
- Ensure a user-defined number of code-breakpoint firings per second

```
PeridoicallyInsertRandomBreakpoints();

OnCodeBreakpoint( pc ) {

  // disassemble the instruction at pc
  (loc, size, isWrite) = disasm( pc );

  temp = read( loc, size );
  if ( isWrite )
    SetDataBreakpointRW( loc, size );
  else
    SetDataBreakpointW( loc, size );

  delay();

  ClearDataBreakpoint( loc, size );

  temp’ = read( loc, size );
  if(temp != temp’ || data breakpt hit)
     ReportDataRace( );
}
```

Sampling: What's the tradeoff?

- Short answer: It's up to the user!
- Long answer
  - Tradeoff: overhead vs. likelihood of finding a data race
  - User controls breakpoints/second & delay length
- Optimal usage?
  - # of threads >= (# of HW watchpoints (4 on x86) + # of processors), lots of both!
  - Processors are always busy

Sampling w.r.t. Software Projects

- Bug bar: how likely would it be that a customer would hit this data race?
- Lower overhead better approximates actual usage
- A data race found only at high overheads should be rarely encountered by end users
- Controlling the overhead can be a way of prioritizing data races

Sampling Instructions

- Challenge: sample hot and cold instructions equally

```
if (rand() % 1000 == 0)
{
    cold ();
}
else
{
    hot ();
}
```

Sampling Using Code Breakpoints

- Over time, code breakpoints aggregate towards cold-instructions
  - Cold instructions have a high sampling probability when they execute
- Samples instructions independent of their execution frequency
  - Hot and code instructions are sampled uniformly
- Cold-instruction sampling is well-suited for data-race detection
  - Buggy data races tend to occur on cold-paths
  - Data races on hot paths are likely to be benign

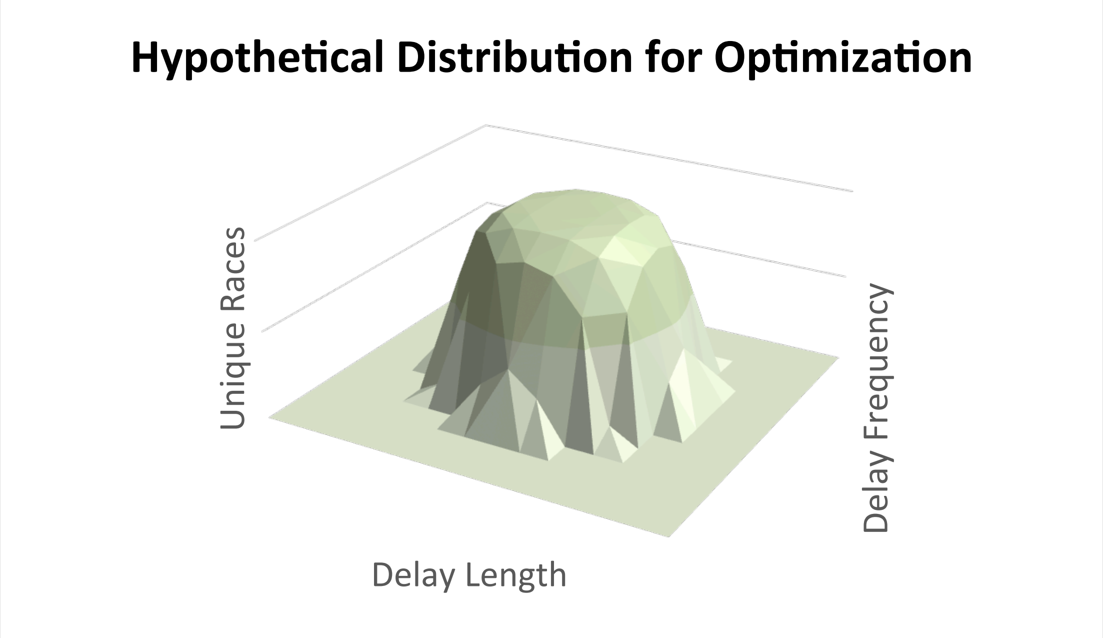
Experience from DataCollider

- All nontrivial programs have data races
- Most (>90%) of the dynamic occurrences are benign
  - Benign data race = The developer will not fix the race even when given infinite resources
- Many of the benign data races can be heuristically pruned
  - Races on variables with names containing "debug", "stats"
  - Races on variables tagged as volatile
  - Races that occur often
- Further research required to address the benign data-race problem – e.g. Adversarial memory & PortEnd

Data Race Category		Count
Benign – Heuristically Pruned	Statistic Counter Safe Flag Update Special Variable	52 29 5
	Subtotal	86
Benign – Manually Pruned	Double-check locking Volatile Write Same Value Other	8 8 1 1
	Subtotal	18
Real	Confirmed Investigating	5 4
	Subtotal	9
Total		113

Future Work

- Different sampling distributions
  - Placing statistical preference on “interesting” instructions per static analysis
- Different sampling rates
  - Breakpoints per second is abstract
  - Automated optimization



DataCollider Conclusion

- Puts the user in control of the overhead
- Fundamentally incapable of false data races
- Trivial to implement - requires no knowledge of synchronization methods
- Sampling is biased toward user-scenarios, but converges to a uniform distribution of static instructions
- Provides full debugging information (e.g. full memory dump)

CUZZ: CONCURRENCY FUZZING
FIND RACE CONDITIONS WITH PROBABILISTIC GUARANTEES

Cuzz: Concurrency Fuzzing

- Disciplined randomization of thread schedules
- Finds all concurrency bugs in every run of the program
  - With reasonably-large probability
- Scalable
  - In the no. of threads and program size
- Effective
  - Bugs in IE, Firefox, Office Communicator, Outlook, …
  - Bugs found in the first few runs

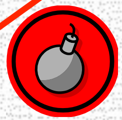
Concurrency Fuzzing in Three Steps

Parent

```
void* p = malloc;
CallCuzz();
CreateThd(child);
RandDelay();
p->f ++;
```

Child

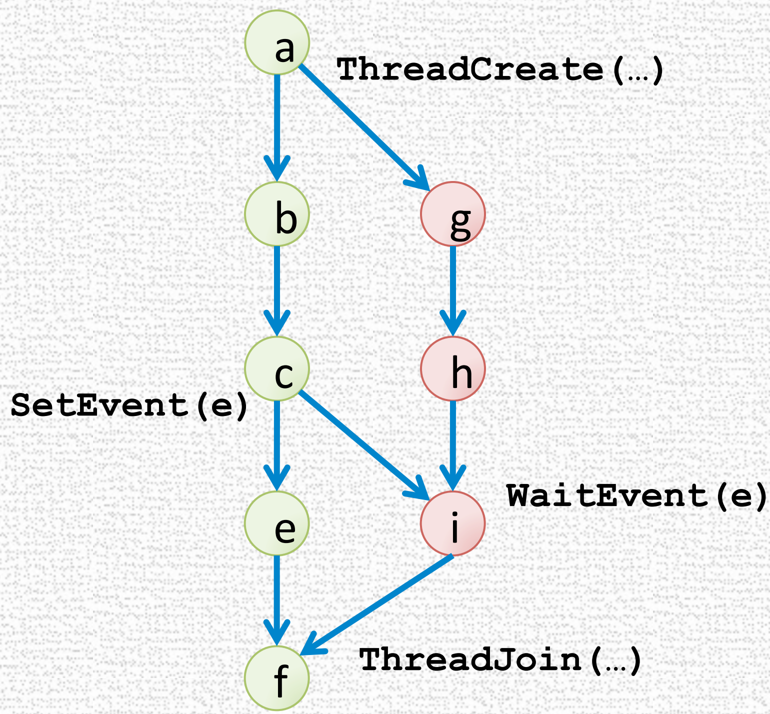
```
Init();
DoMoreWork();
free(p);
free(p);
```



1. Instrument calls to Cuzz
2. Insert random delays
3. Use the Cuzz algorithm to determine when and by how much to delay

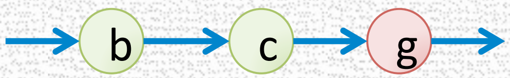
This is where all the magic is

Find all "use-after-free" bugs

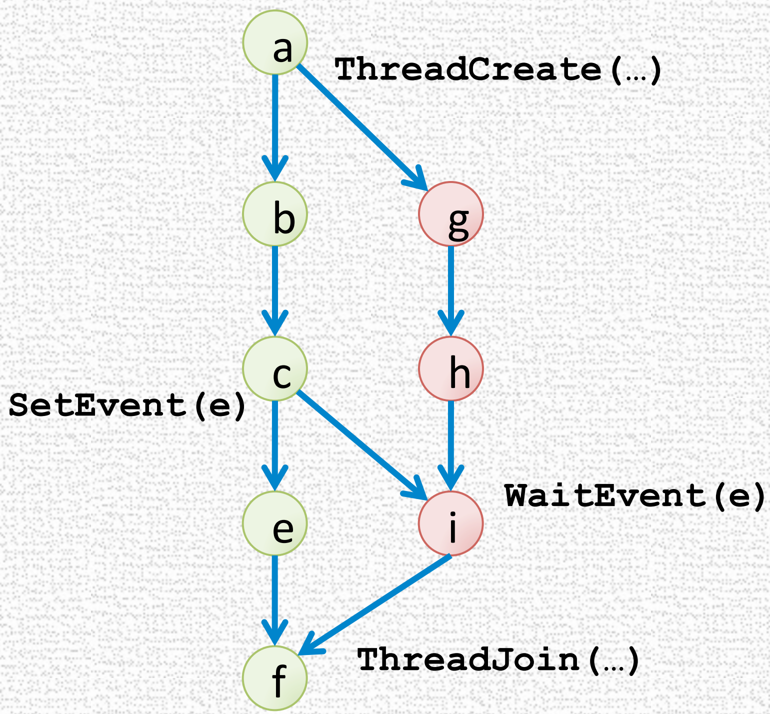


All nodes involve the use and free of some pointer

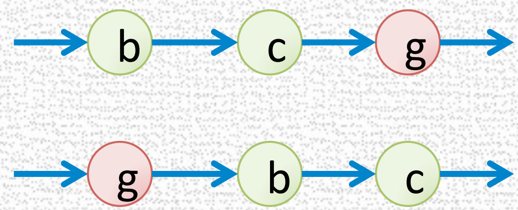
if b frees a pointer used by g, the following execution triggers the error



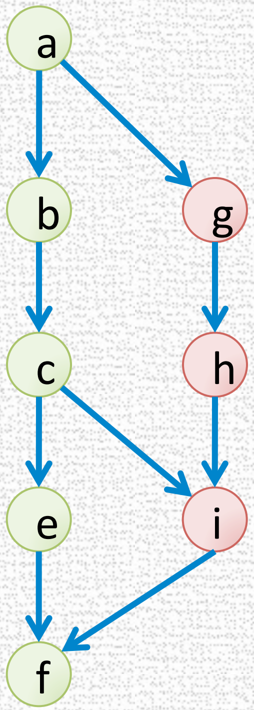
Find all "use-after-free" bugs



Problem:
For every unordered pair, say (b,g), cover both orderings:
e.g.

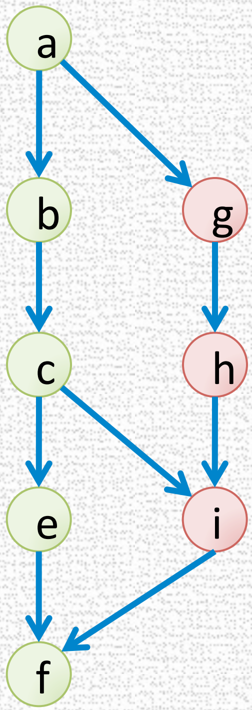


Find all “use-after-free” bugs



Approach 1: enumerate all interleavings

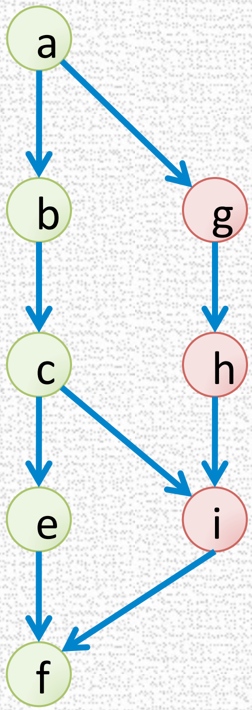
Find all "use-after-free" bugs



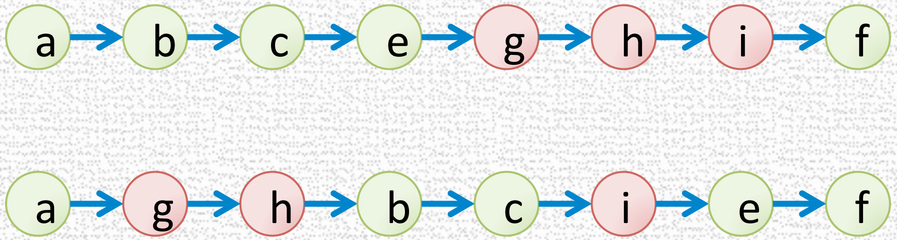
Approach 2: enumerate all unordered pairs

- b -> g
- g -> b
- b -> h
- …

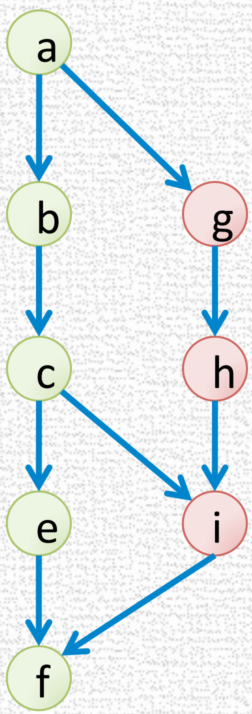
Find all "use-after-free" bugs



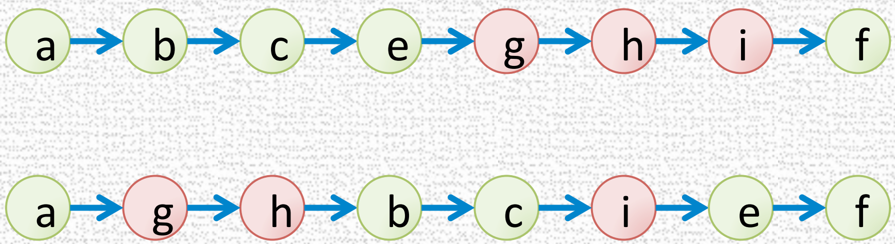
Two interleavings find all use-after-free bugs



Find all "use-after-free" bugs



Two interleavings find all use-after-free bugs



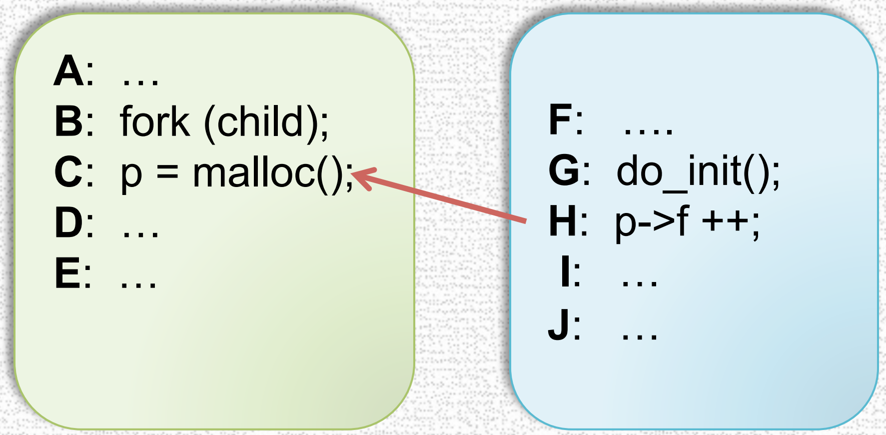
Cuzz picks each with 0.5 probability

Find all "use-after-free" bugs

- For a concurrent program with n threads
- There exists n interleavings that find **all** use-after-free bugs
- Cuzz explores each with probability 1/n

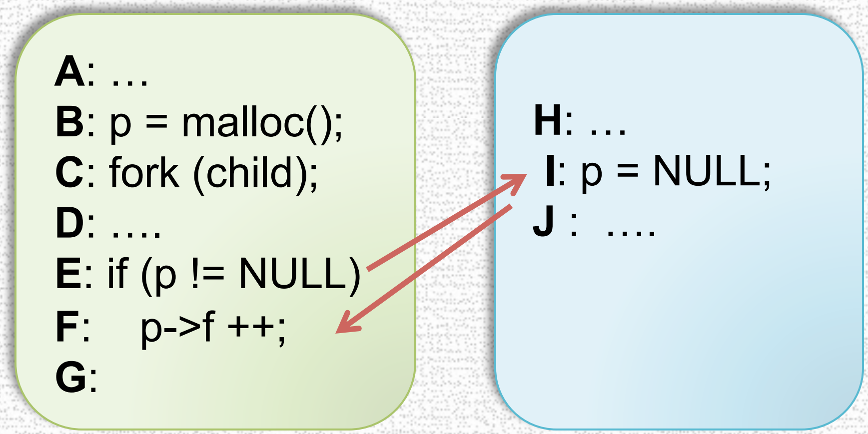
Concurrency Bug Depth

- Number of ordering constraints sufficient to find the bug
- Bugs of depth 1
  - Use after free
  - Use before initialization



Concurrency Bug Depth

- Number of ordering constraints sufficient to find the bug
- Bugs of depth 2
  - Pointer set to null between a null check and its use



Cuzz Guarantee

- n: max no. of concurrent threads (~tens)
- k: max no. of operations (~millions)

- There exists $n.k\uparrow d-1$ interleavings that find all bugs of depth d
- Cuzz picks each with a uniform probability

- Probability of finding a bug of depth d $\geq 1/n.k\uparrow d-1$

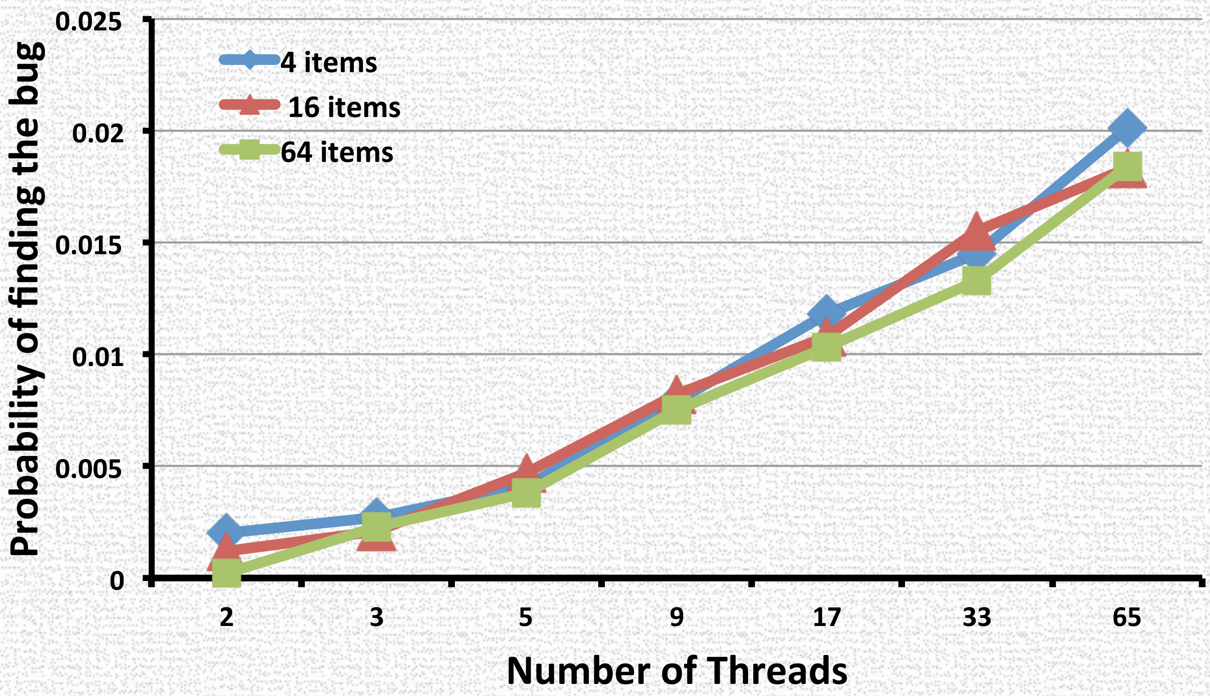
Cuzz Algorithm

Inputs: n: estimated bound on the number of threads
k: estimated bound on the number of steps
d: target bug depth

```
// 1. assign random priorities >= d to threads
for t in [1…n]  do  priority[t] = rand() + d;

// 2. chose d-1 lowering points at random
for i in [1...d)  do  lowering[i] = rand() % k;

steps = 0;
while (some thread enabled) {
    // 3. Honor thread priorities
    Let t be the highest-priority enabled thread;
    schedule t for one step;
    steps ++;

    // 4. At the ith lowering point, set the priority to i
     if steps == lowering[i] for some i
         priority[t] = i;
}
```

Empirical bug probability w.r.t worst-case bound

- Probability increases with n, stays the same with k
  - In contrast, worst-case bound = $1/nk^{d-1}$



Why Cuzz is very effective

- Cuzz (probabilistically) finds all bugs in a single run
- Programs have *lots* of bugs
  - Cuzz is looking for all of them simultaneously
  - Probability of finding any of them is more than the probability of finding one
- Buggy code is executed many times
  - Each dynamic occurrence provides a new opportunity for Cuzz

Conclusions

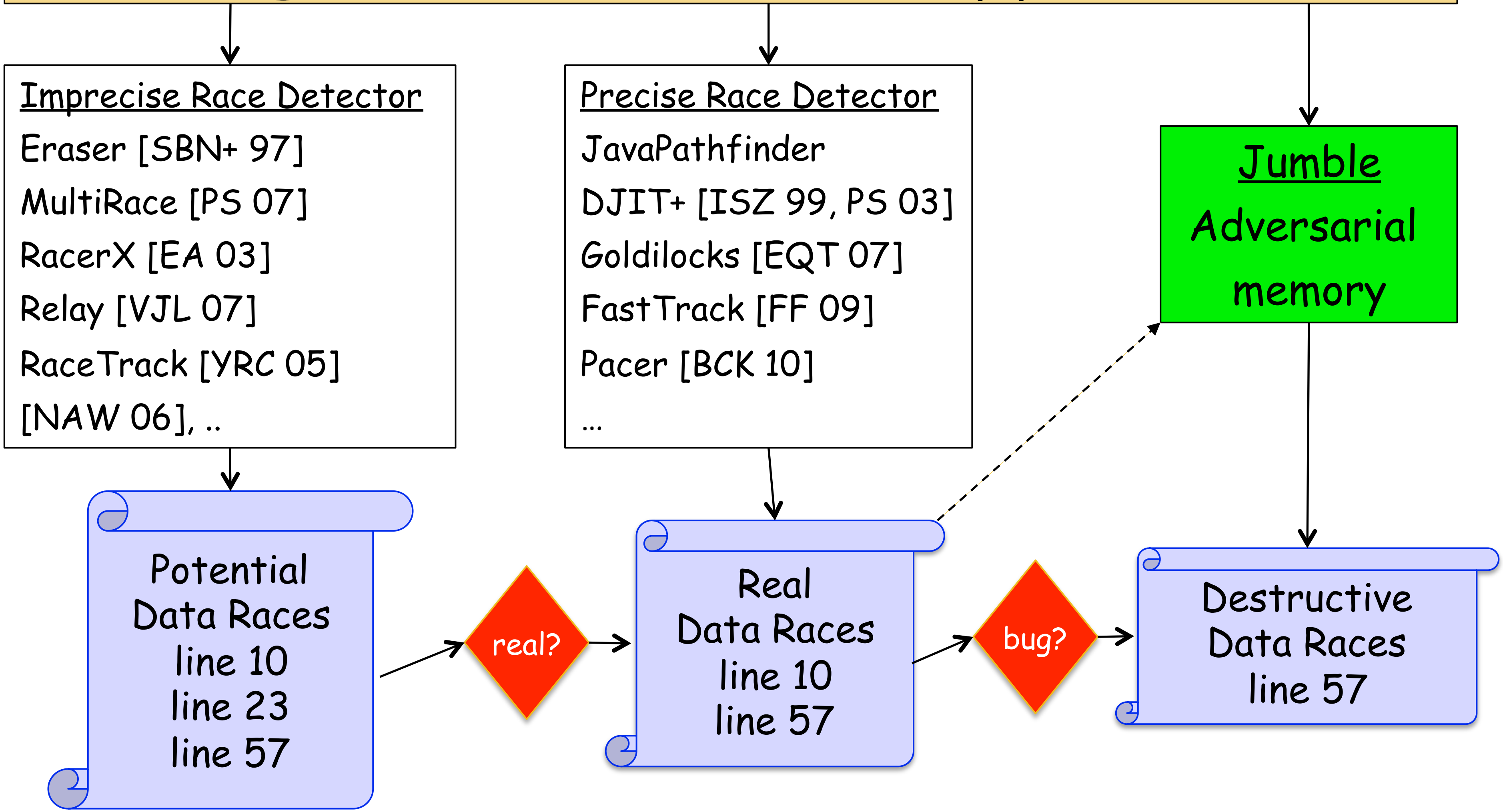
- Two tools for finding concurrency errors
  - DataCollider: Uses code/data breakpoints for finding data races efficiently
  - Cuzz: Inserts randomized delays to find race conditions
- Both are easily implementable
- Email: madanm@microsoft.com for questions/availability

ADVERSARIAL MEMORY FOR DESTRUCTIVE RACES

Beyond Detecting Data Race Conditions

- Checkers can find real **race conditions**
- But which race conditions are **real bugs**?
  - that cause erroneous behaviors (crashes, etc)
  - and are not "benign race conditions"

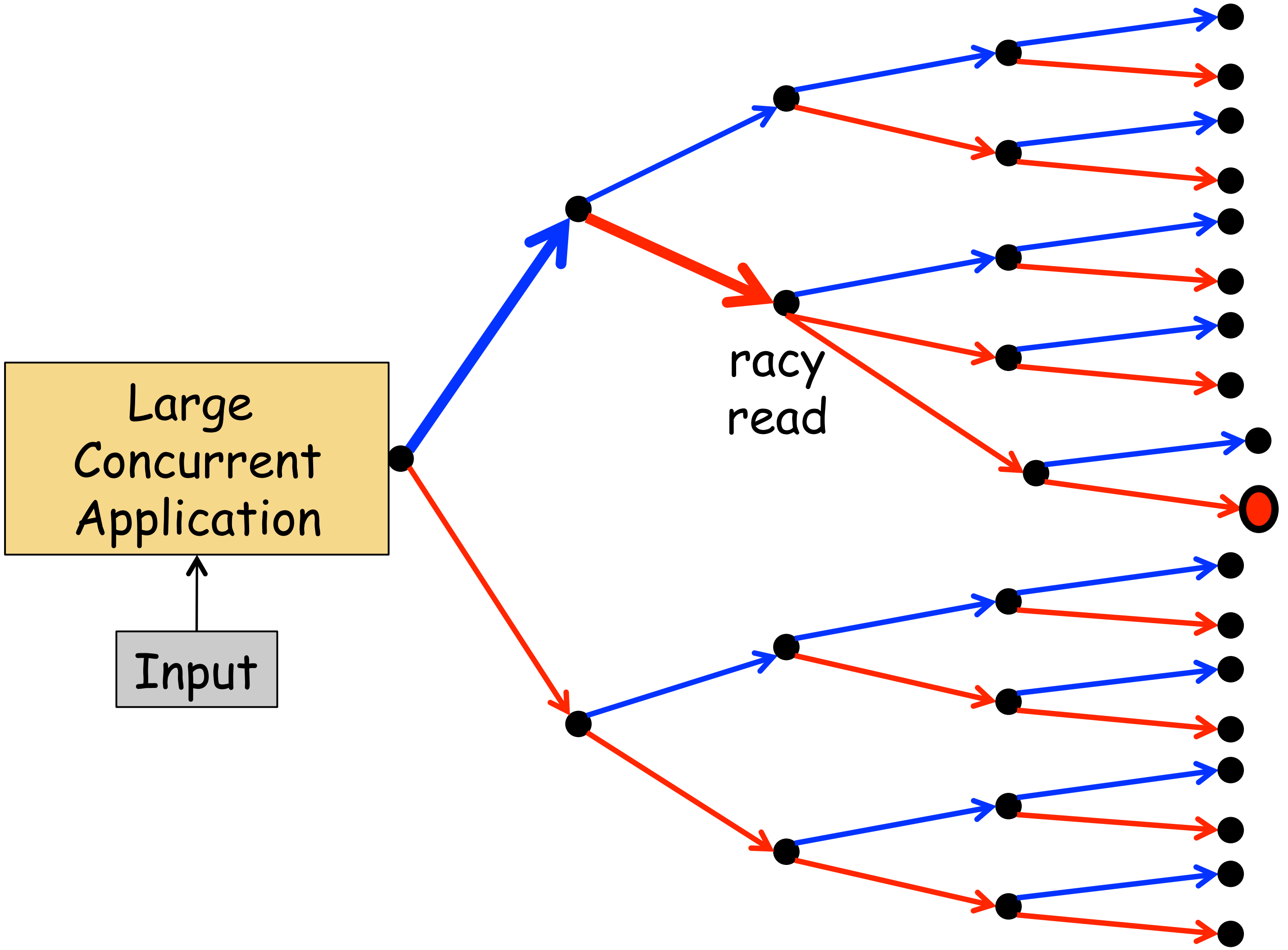
Large Multithreaded Application



Destructive data race: erroneous observable behavior

Benign data race: not a bug

Controlling Scheduling Non-Determinism



Thread A

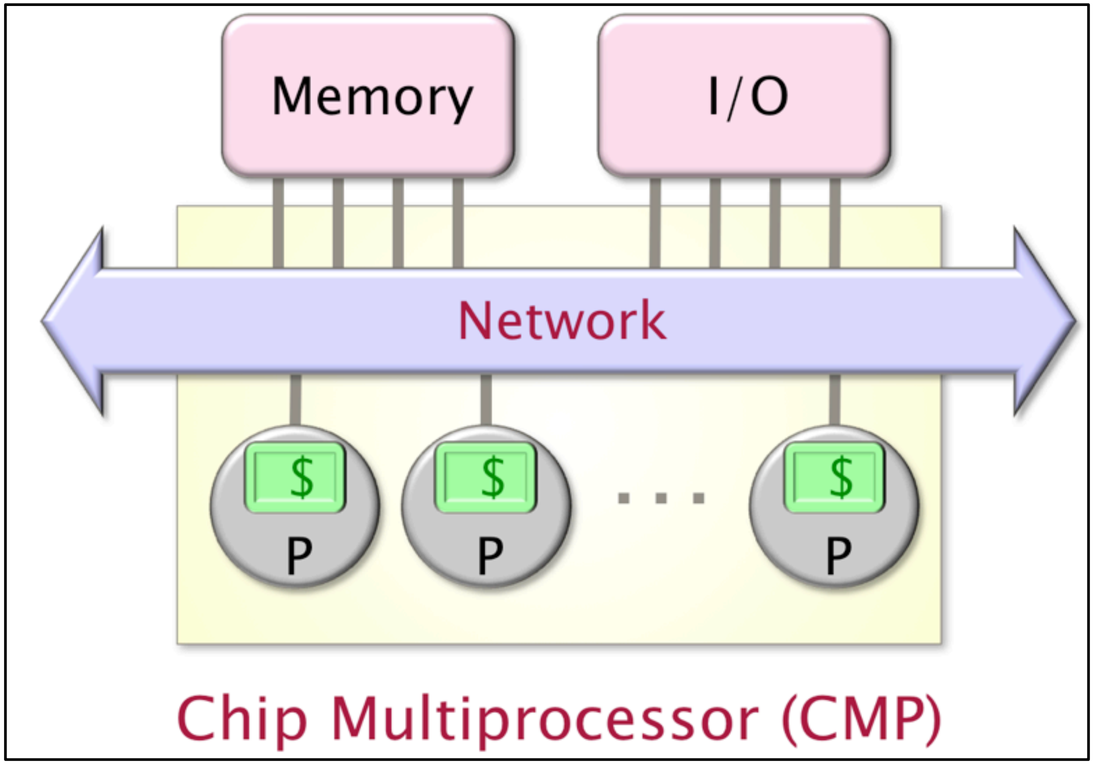
```
p = new Pt();
...
p = null;
```

Thread B

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

(eg: CalFuzzer, DataCollider, etc.)

Memory Models



- Each processor/core has a cache
- When do writes to *x* become visible to other processors?
  - Sequentially Consistent MM
  - Relaxed MM (JMM, x86-TSO, etc.)
    - more than one value written to *x* may be visible

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?

Example

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

Thread A

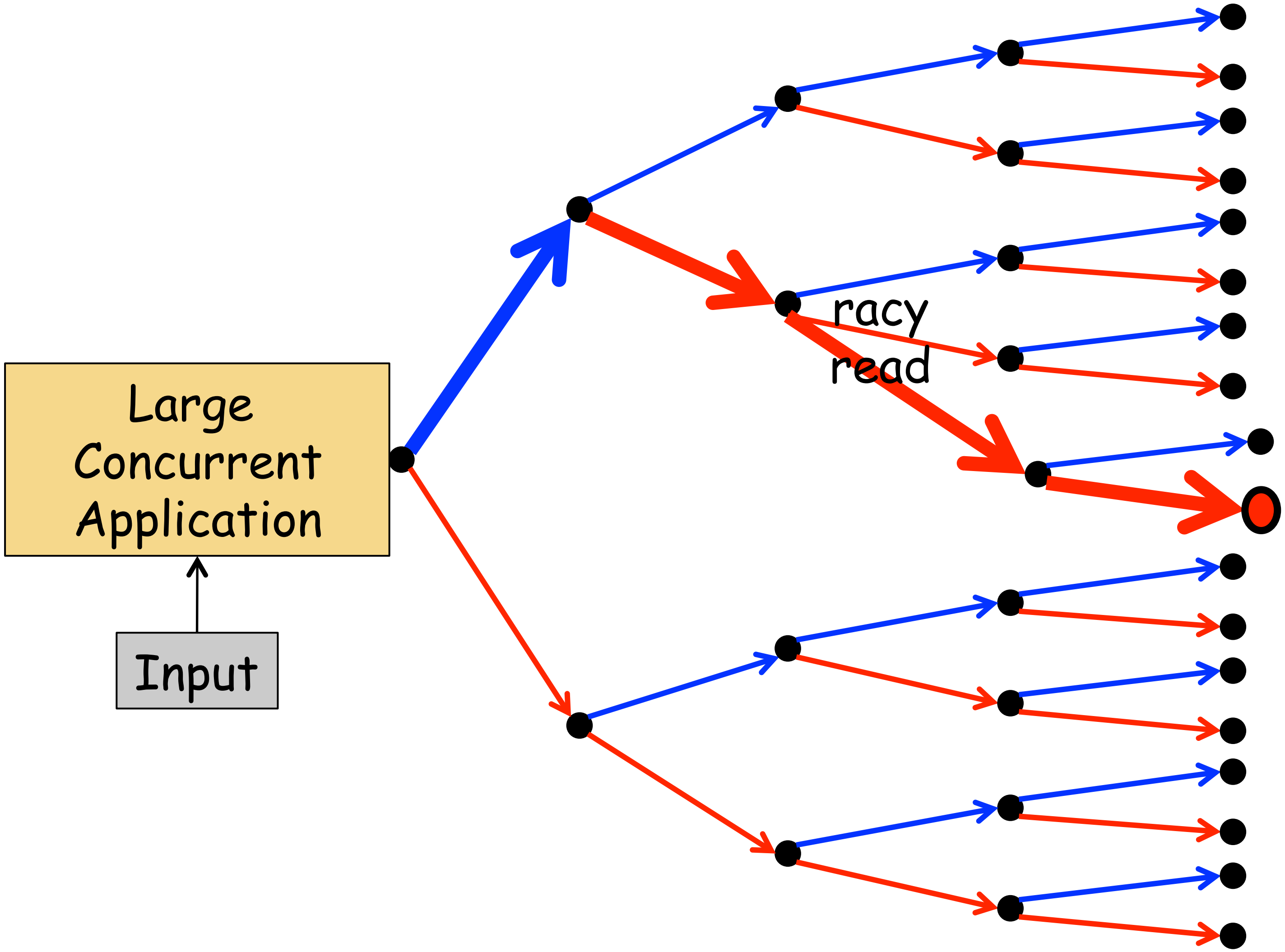
```
x = 10;
y = 20;
```

Thread B

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

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

Adversarial Memory [Flanagan-Freund 10]

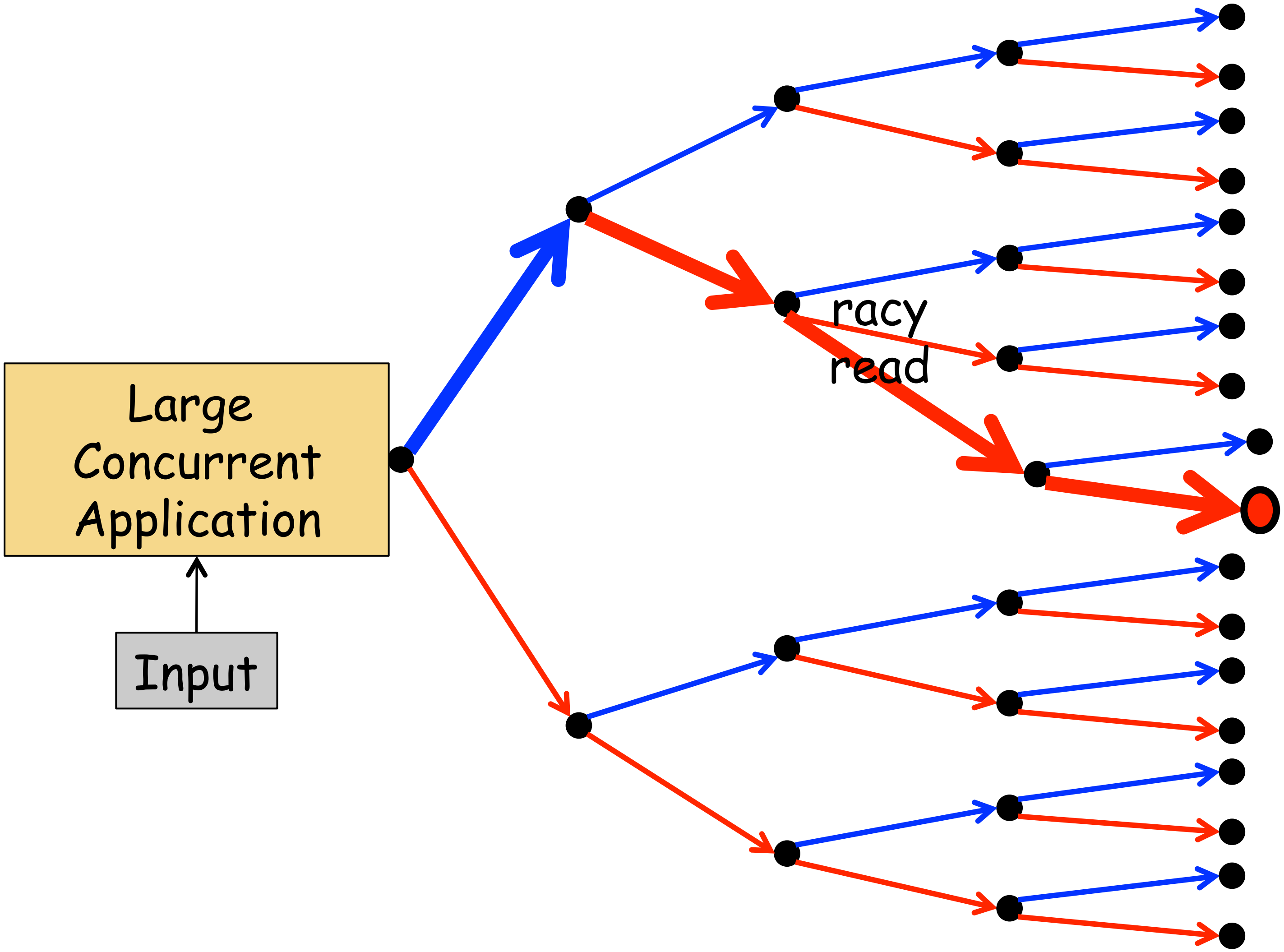


Adversarial memory exploits memory nondeterminism.

Racy read sees old value likely to crash application.

Complements schedule-based approaches, quite effective.

Adversarial Memory [Flanagan-Freund 10]



Thread A

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

Thread B

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

Example

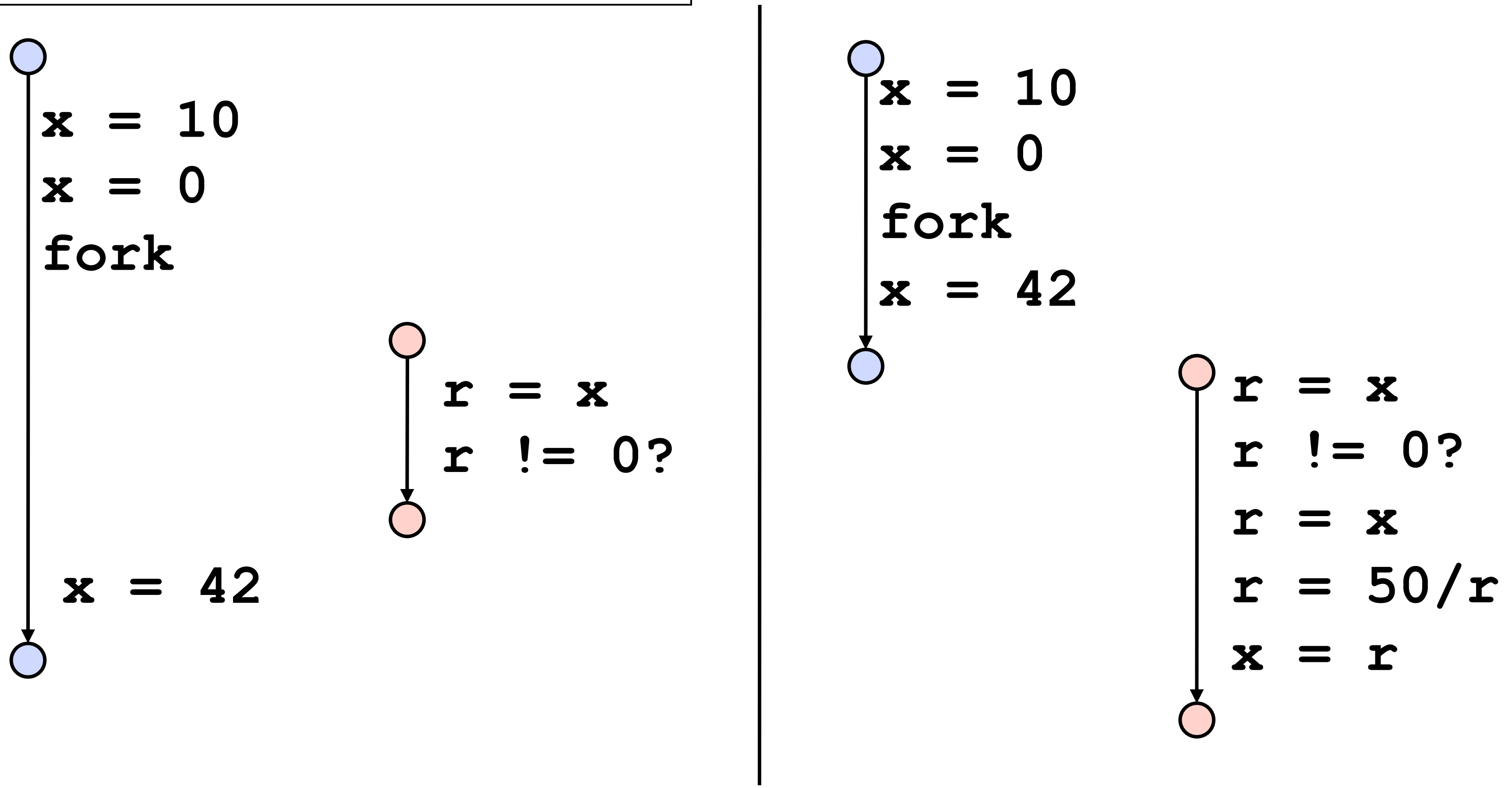
```
int x = 10;
x = 0;
fork{ if (x != 0) x = 50/x; }
x = 42;
```

- Data race on x
- Is this data race destructive?
- Can program divide by zero?

Sequentially Consistent Memory Model

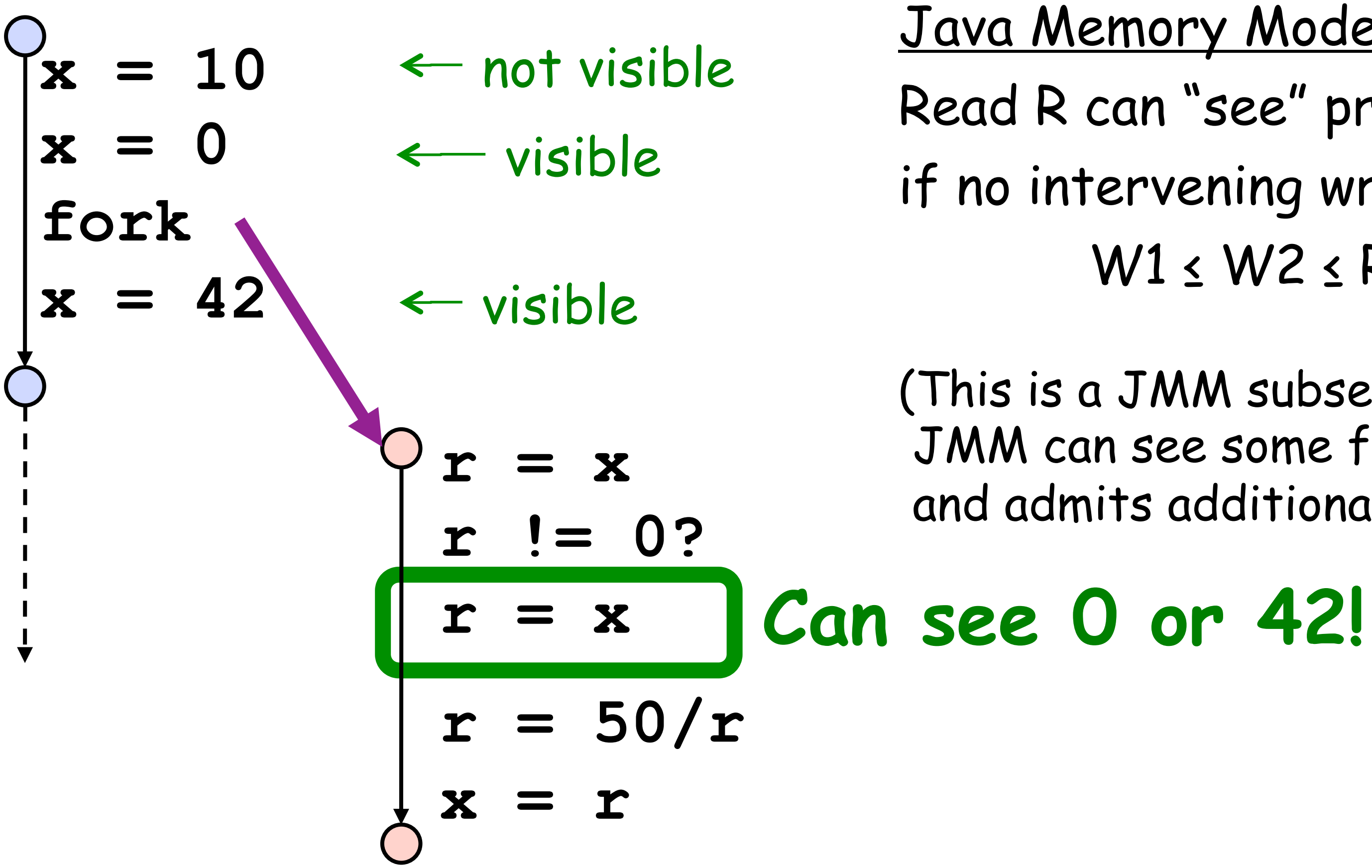
```
int x = 10;
x = 0;
fork{ if (x != 0) x = 50/x; }
x = 42;
```

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



Java Memory Model

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



Happens-Before Partial Order

- Program order edges
- Fork edges
- Release-acquire edges, …

Java Memory Model

Read R can "see" previous write W1 if no intervening write W2 with

$$W1 \leq W2 \leq R$$

(This is a JMM subset;
JMM can see some future writes and admits additional behaviors)

Jumble

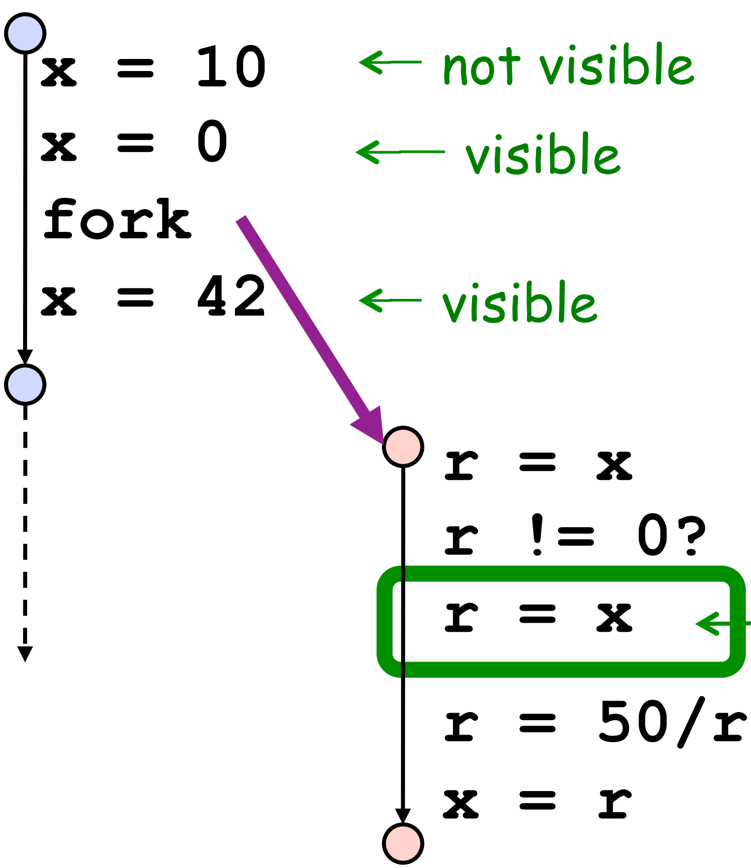
```
int x = 10;
x = 0;
fork{ if (x != 0) x = 50/x; }
x = 42;
```

Record:

- write buffer for racy vars
- happens-before relation

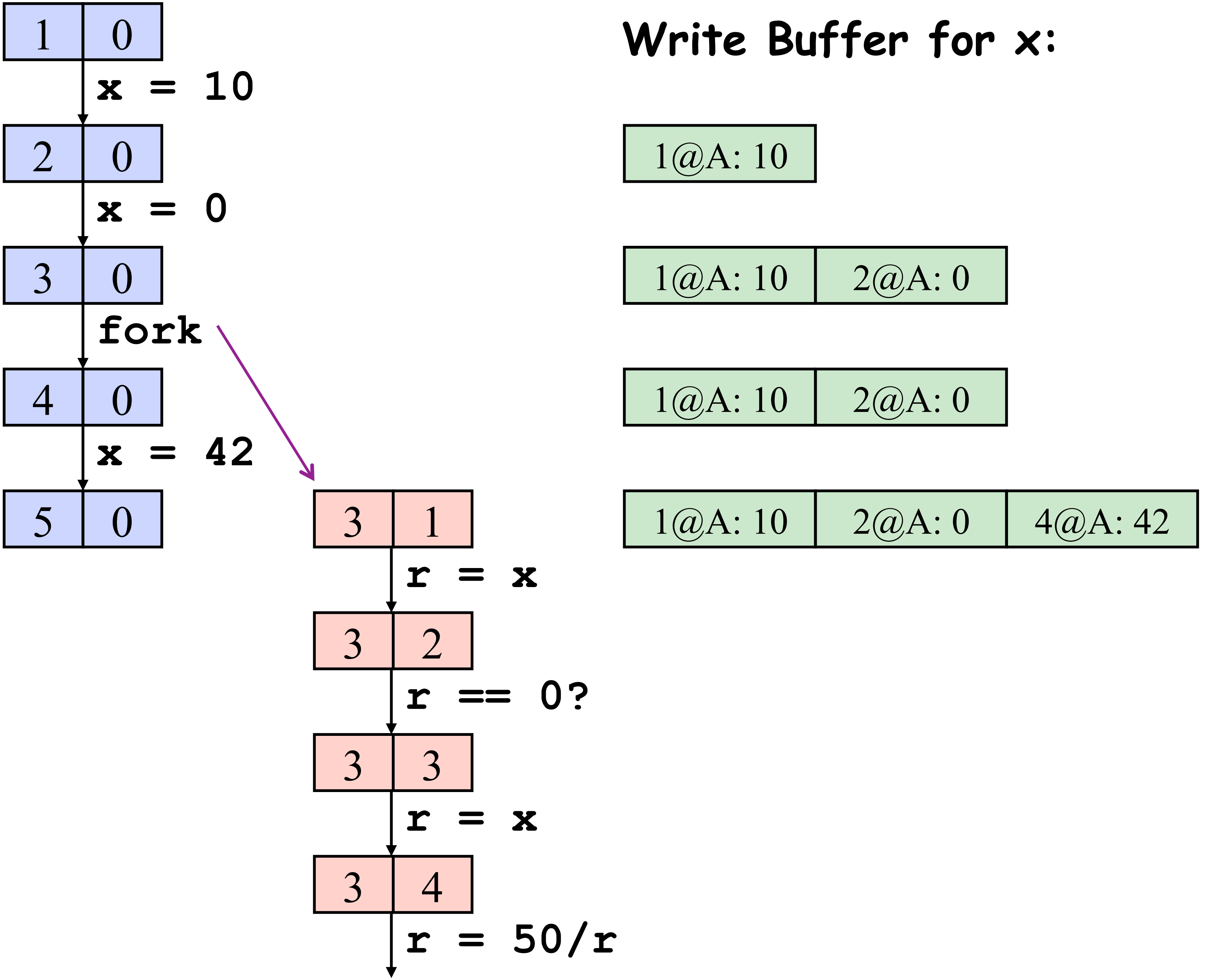
At each read:

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

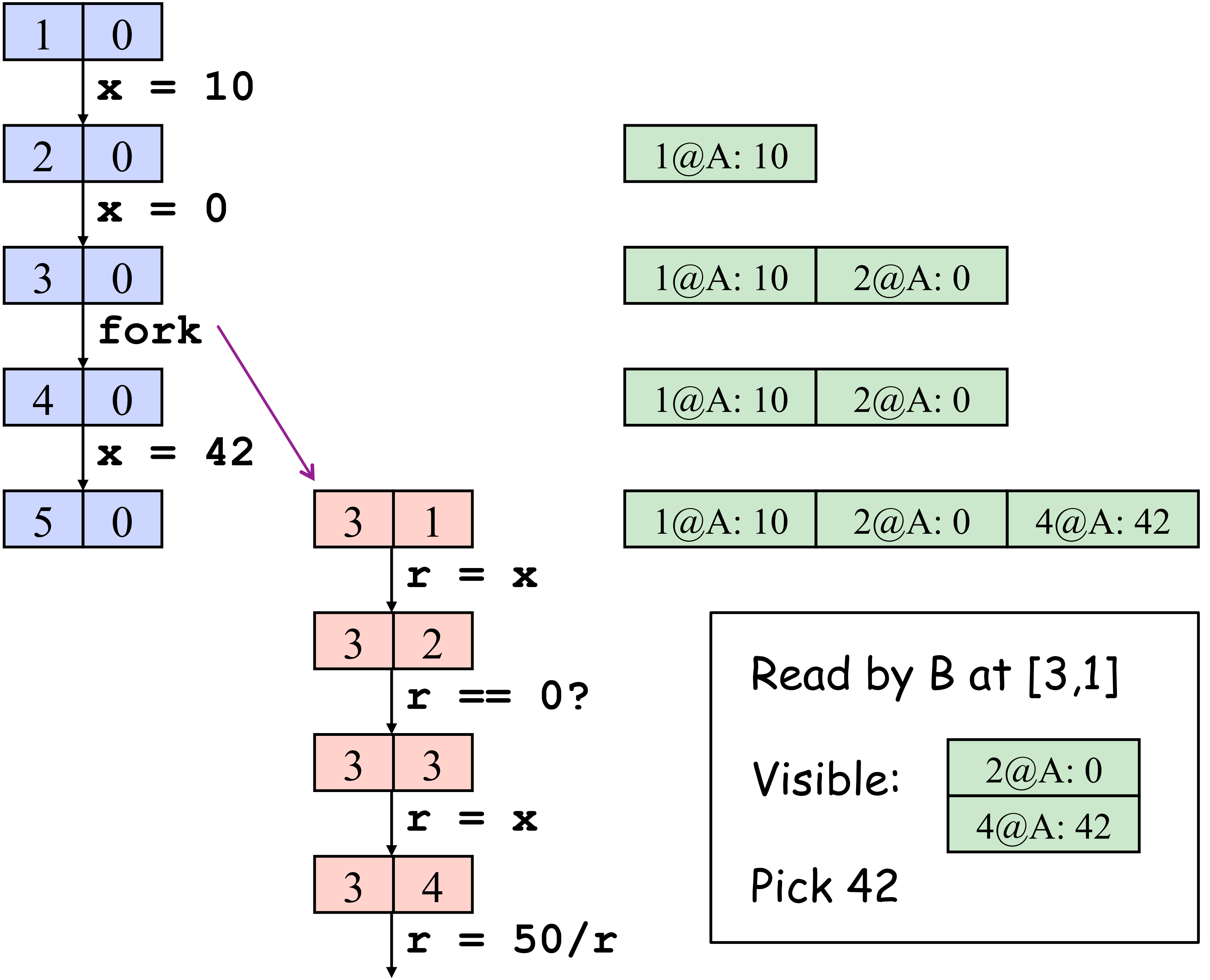


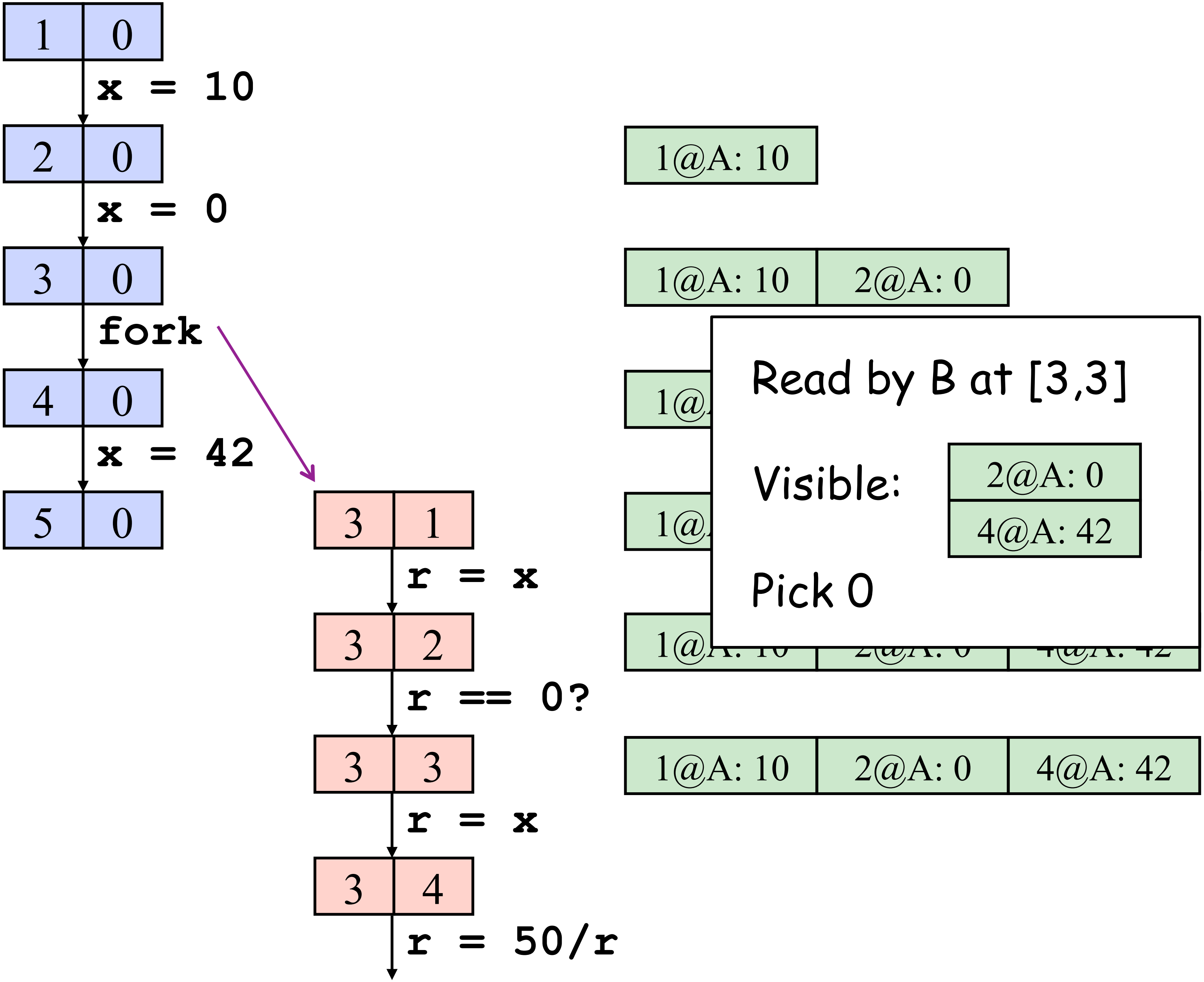
heuristically pick 0

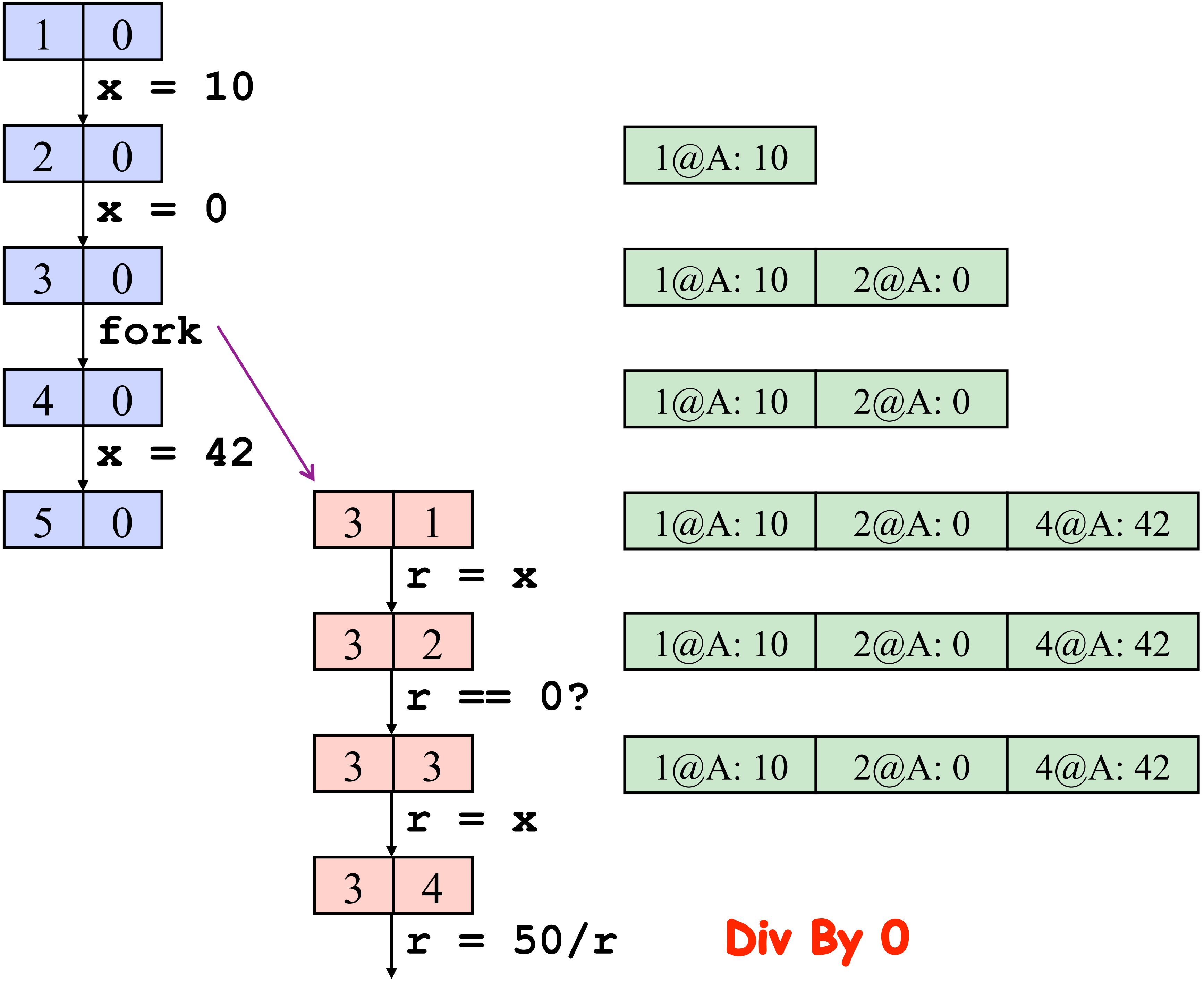
division by zero



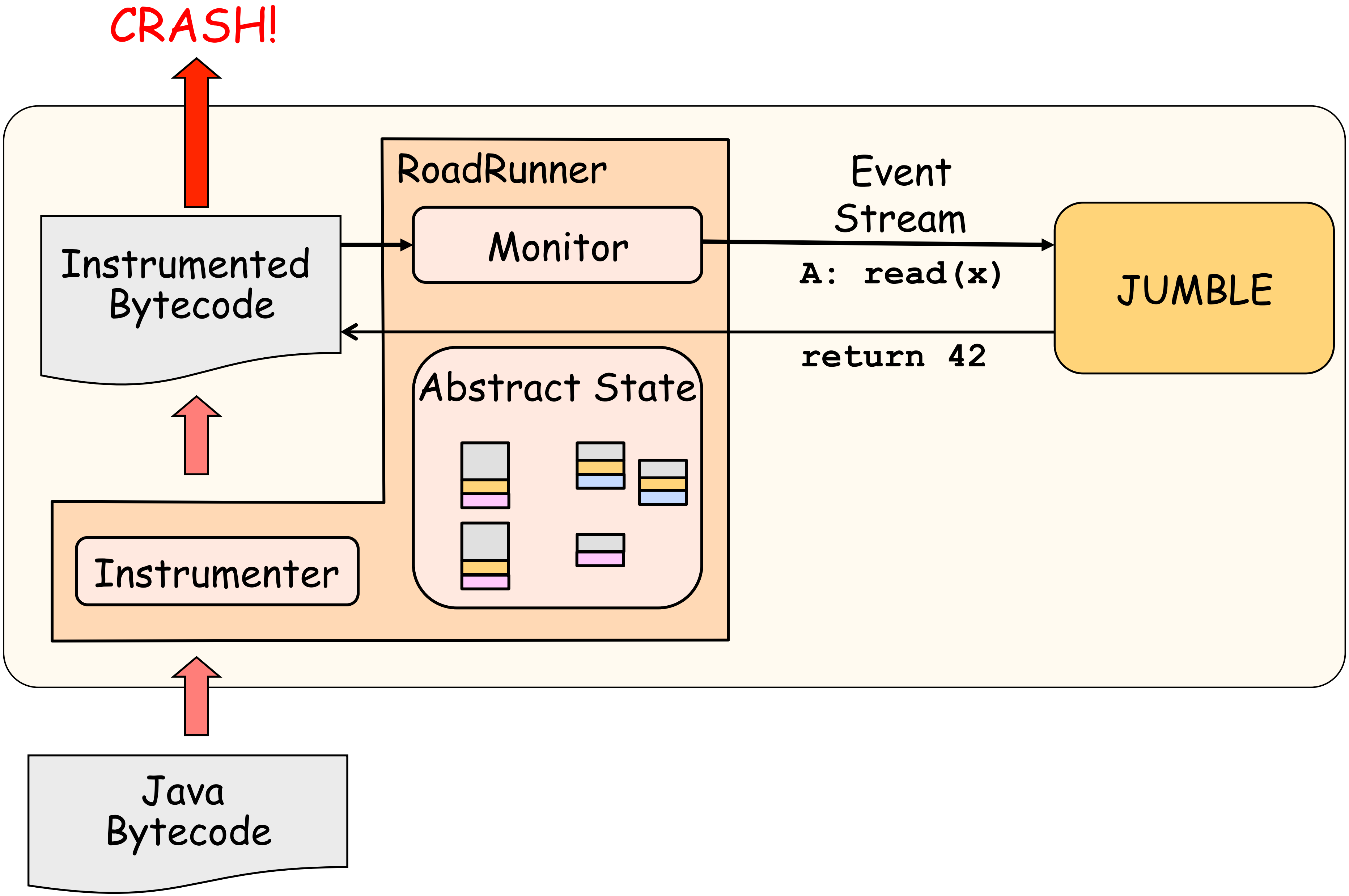
Write Buffer for x:
1 | 0
x = 10
2 | 0
x = 0
3 | 0
fork
4 | 0
x = 42
5 | 0
3 | 1
r = x
3 | 2
r == 0?
3 | 3
r = x
3 | 4
r = 50/r
1@A: 10
1@A: 10 | 2@A: 0
1@A: 10 | 2@A: 0
1@A: 10 | 2@A: 0 | 4@A: 42


1 0
x = 10
2 0
x = 0
3 0
fork
4 0
x = 42
5 0
3 1
r = x
3 2
r == 0?
3 3
r = x
3 4
r = 50/r
1@A: 10
1@A: 10 2@A: 0
1@A: 10 2@A: 0
1@A: 10 2@A: 0 4@A: 42
Read by B at [3,1]
Visible:
2@A: 0
4@A: 42
Pick 42


1 0
x = 10
2 0
x = 0
3 0
fork
4 0
x = 42
5 0
3 1
r = x
3 2
r == 0?
3 3
r = x
3 4
r = 50/r
1@A: 10
1@A: 10
2@A: 0
1@A: 10
2@A: 0
4@A: 42
Read by B at [3,3]
Visible:
2@A: 0
4@A: 42
Pick 0



1 0
x = 10
2 0
x = 0
3 0
fork
4 0
x = 42
5 0
3 1
r = x
3 2
r == 0?
3 3
r = x
3 4
r = 50/r
1@A: 10
1@A: 10 2@A: 0
1@A: 10 2@A: 0
1@A: 10 2@A: 0 4@A: 42
1@A: 10 2@A: 0 4@A: 42
1@A: 10 2@A: 0 4@A: 42
Div By 0


CRASH!
Instrumented Bytecode
RoadRunner
Monitor
Event Stream
A: read(x)
JUMBLE
return 42
Abstract State
Instrumenter
Java Bytecode

Jumble Performance

- Keep write buffers only for small # of locations
  - all instances of a particular field declaration
  - array sampled at indexes 0 and 1 (configurable)

- Slowdown of 1.2x to 5x

- Write buffers limited to 32 entries
  - eject writes when no longer visible or redundant
  - some capacity ejects

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 fields with data races
- ran Jumble manually once for each field
- found 4 destructive data races

Jumble Summary

- Identifying destructive data races
  - very difficult, time consuming, error prone

- Adversarial memory automates identification
  - reveals destructive data races with high confidence
  - helps focus effort on fixing real bugs

Where To Go From Here?

- **[Much work on all of these problems, some by the audience, by us, ...]**
- Performance, performance, performance ...
  - always-on detection, HW support,
  - static-dynamic hybrid analyses, language support
- Is sampling the way to go for debugging?
  - Does it miss rare data races?
- Prioritize and deal with benign data races
  - which data races are most critical?
- How to respond to data races?
  - warning / fail-fast / recovery
- Reproducing traces exhibiting rare data races
  - record and replay
- Generalization
  - reason about traces beyond the observed trace
- Finding memory model problems

Acknowledgments

- Some of our presentation on background material is based on slides from Dan Grossman
  - http://homes.cs.washington.edu/~djg/slides/grossman_russia_dataraces.pptx
- Thanks to Shaz Qadeer, Tom Ball, and Cormac Flanagan for valuable feedback on this presentation

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.
- Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs", OSDI 2008.
- 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

- John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk, "Effective Data-Race Detection for the Kernel", OSDI 2010.
- Madanlal Musuvathi, Sebastian Burckhardt, Pravesh Kothari, and Santosh Nagarakatte, "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs", ASPLOS 2010.
- Michael D. Bond, Katherine E. Coons, Kathryn S. McKinley, "PACER: proportional detection of data races", PLDI 2010.
- Cormac Flanagan and Stephen N. Freund, "Adversarial memory for detecting destructive races", PLDI 2010.