CS 636: Concurrency Bugs

Swarnendu Biswas

Semester 2020-2021-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

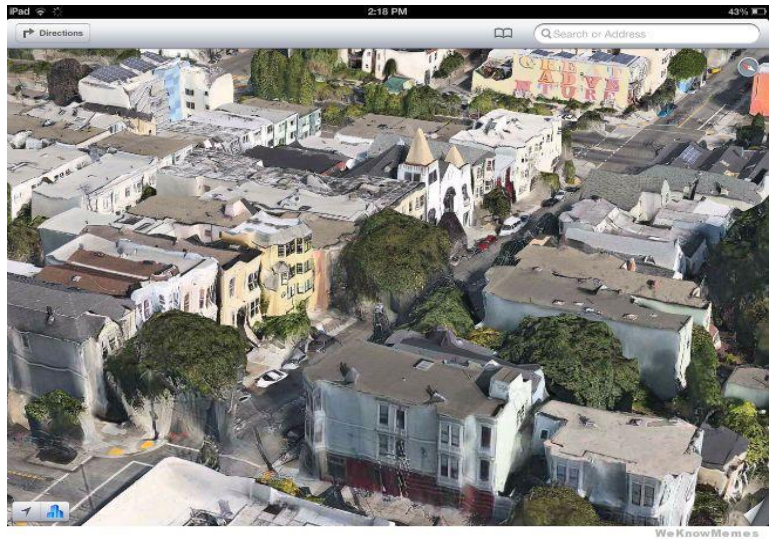
Production Software Contains Bugs!

- **AT&T hangs up its long-distance service (1990)**
  - For nine hours in January 1990 no AT&T customer could make a long-distance call. The problem was the software that controlled the company's long-distance relay switches—software that had just been updated. AT&T wound up losing $60 million in charges that day—a very expensive bug.

- **The Pentium chip’s FDIV math error (1993)**

- **The Mars Climate Orbiter disintegrates in space (1998)**
  - NASA's $655-million robotic space probe plowed into Mars's upper atmosphere at the wrong angle, burning up in the process. The problem? In the software that ran the ground computers the thrusters' output was calculated in the wrong units (pound–seconds instead of newton–seconds).

Other Examples of Real-World Concurrency Bugs



NASDAQ

business

READY

Nasdaq's Facebook Glitch Came From Race Conditions

Joab Jackson
@Joab_Jackson

May 21, 2012 12:30 PM

The Nasdaq computer system that delayed trade notices of the Facebook IPO on Friday was plagued by race conditions, the stock exchange announced Monday. As a result of this technical glitch in its Nasdaq OMX system, the market expects to pay out US$13 million or even more to traders.

A number of trading firms lost money due to mismatched Facebook share prices. About 30 million shares' worth of trading were affected, the exchange estimated.

NASDAQ's Glitch Cost Facebook Investors ~$500M. It Will Pay Out Just $62M. IPO Elsewhere.

Josh Constine @joshconstine / 6 years ago

Comment

KILLED BY A MACHINE: THE THERAC-25

by: Adam Fabio

139 Comments

October 26, 2015

The Therac-25 was not a device anyone was happy to see. It was a radiation therapy machine. In layman's terms it was a "cancer zapper"; a linear accelerator with a human as its target. Using X-rays or a beam of electrons,

SEARCH

Search ... SEARCH

NEVER MISS A HACK

SUBSCRIBE

Enter Email Address SUBSCRIBE

Therac-25 Accident

- Therac-25 was a computer-controlled radiation therapy machine
- It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation. Because of concurrent programming errors, it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.

https://en.wikipedia.org/wiki/Therac-25

Challenges in Concurrent Programming

Develop Parallel Programs

From my perspective, parallelism is the biggest challenge since high-level programming languages. It's the biggest thing in 50 years because industry is betting its future that parallel programming will be useful.
…
Industry is building parallel hardware, assuming people can use it. And I think there's a chance they'll fail since the software is not necessarily in place. So this is a gigantic challenge facing the computer science community.

– David Patterson, ACM Queue, 2006.

Develop Parallel Programs

To save the IT industry, researchers must demonstrate greater end-user value of from an increasing number of cores

–

A View of Parallel Computing Landscape, CACM 2009

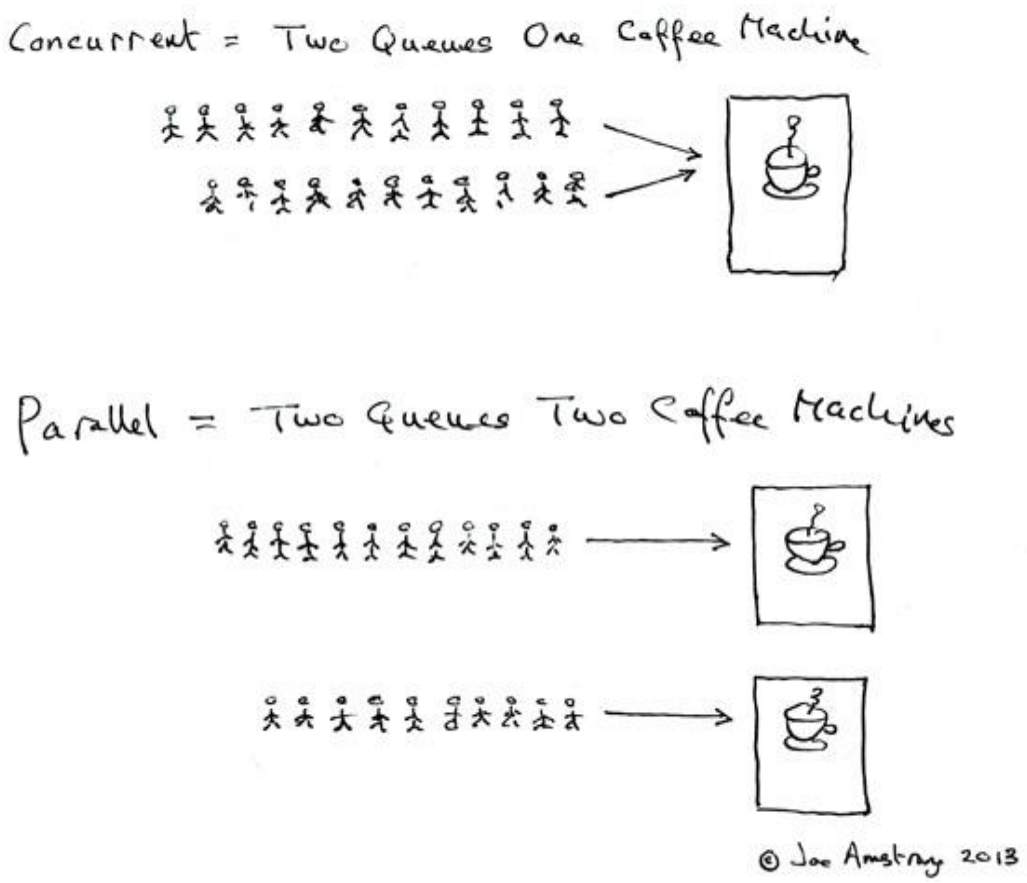
Programmer's tend to think sequentially

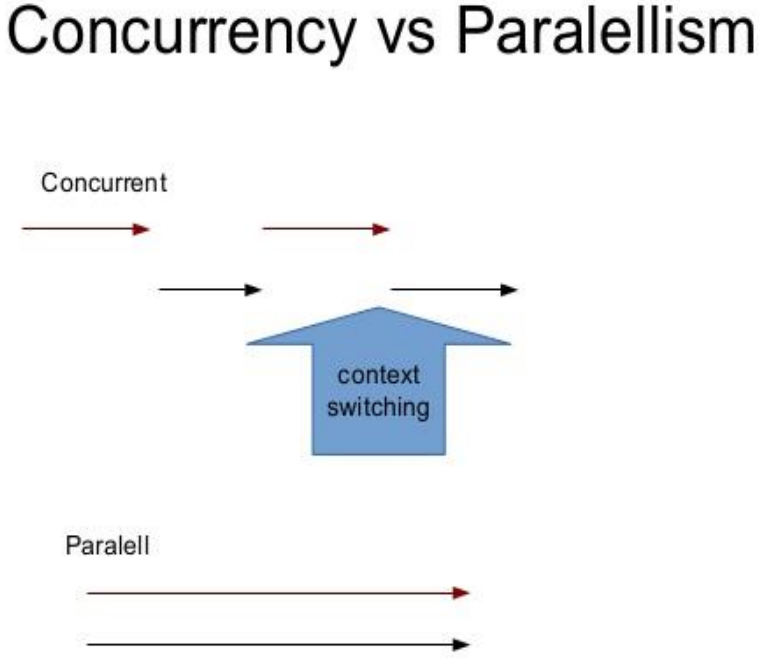
Challenges in Developing Parallel Programs

- Programmers tend to **think sequentially**
  - Correctness issues – concurrency bugs like data races and deadlocks
  - Performance issues – minimize communication across cores

- Other challenges
  - Amdahl's law, overheads of parallel execution, load balancing

Parallelism vs Concurrency





Parallelism vs Concurrency

Parallel programming

- Use additional resources to speed up computation
- Performance perspective

Concurrent programming

- Correct and efficient control of access to shared resources
- Correctness perspective

Distinction is not absolute

Types of Concurrency Bugs

Order Violation

Thread 1

```
void init(…) {
  …
  mThread=
       PR_CreateThread(mMain, …);
  …
}
```

Thread 2

```
void mMain() {
  mState=mThread->State;
}
```

Mozilla
nsthread.cpp

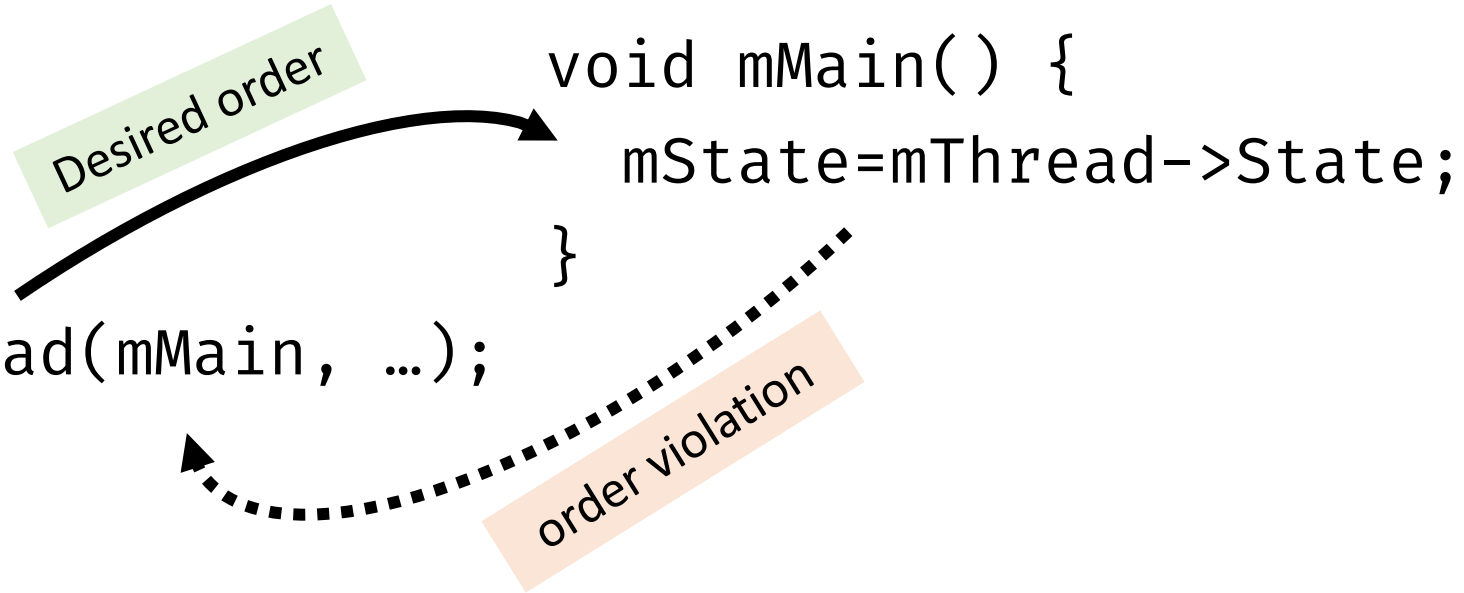
Order Violation

Thread 1

```
void init(…) {
  …
  mThread=
      PR_CreateThread(mMain, …);
  …
}
```

Thread 2

```
void mMain() {
  mState=mThread->State;
}
```



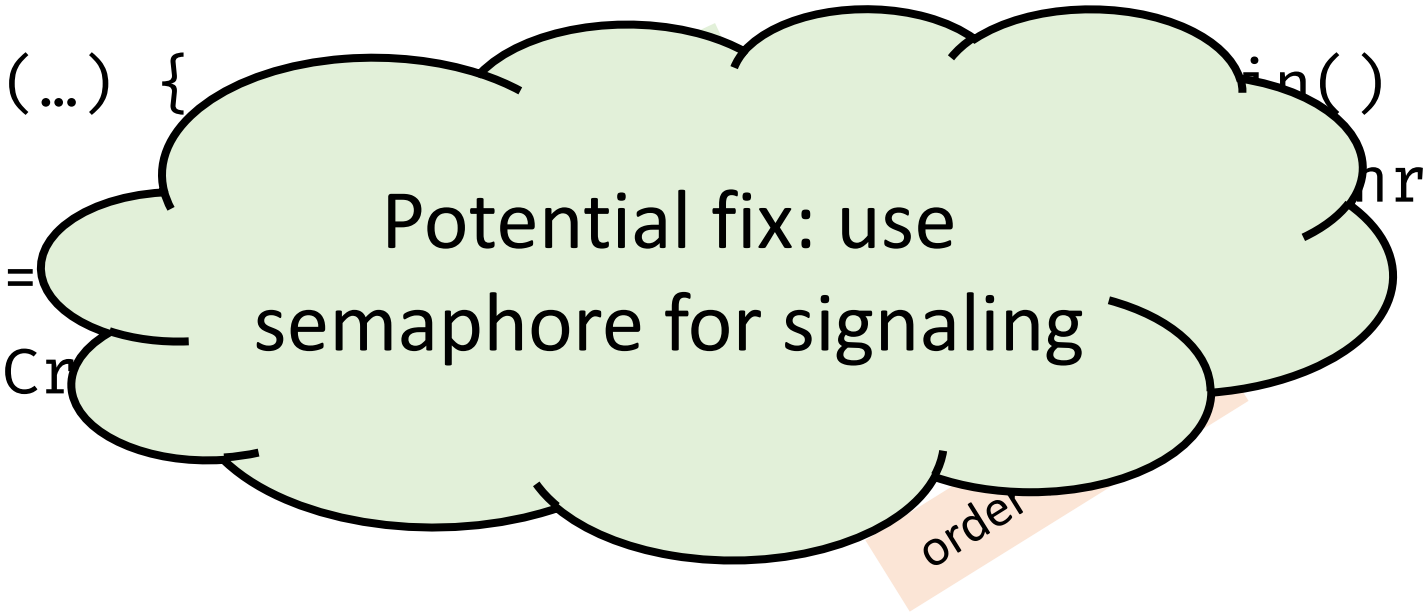
Order Violation

Thread 1

```
void init(…) {
  …
  mThread=
       PR_Cr
  …
}
```

Thread 2

```
in() {
hread->State;
```



Atomicity Violation

Thread 1

```
if (thd->proc_info)

  fputs(thd->proc_info, …)
```

Thread 2

```
thd->proc_info = NULL;
```

MySQL
ha_innodb.cc

Atomicity Violation

Thread 1 **Thread 2**

```
if (thd->proc_i
                                    fo = NULL;
  fputs(th
```

Potential fix: use locks for mutual exclusion

MySQL
ha_innodb.cc

Sequential Consistency Violation

```
Object X = null;
boolean done= false;
```

Thread T1

```
X = new Object();
done = true;
```

Thread T2

```
while (!done) {}
X.compute();
```

Sequential Consistency Violation

```
Object X = null;
boolean done= false;
```

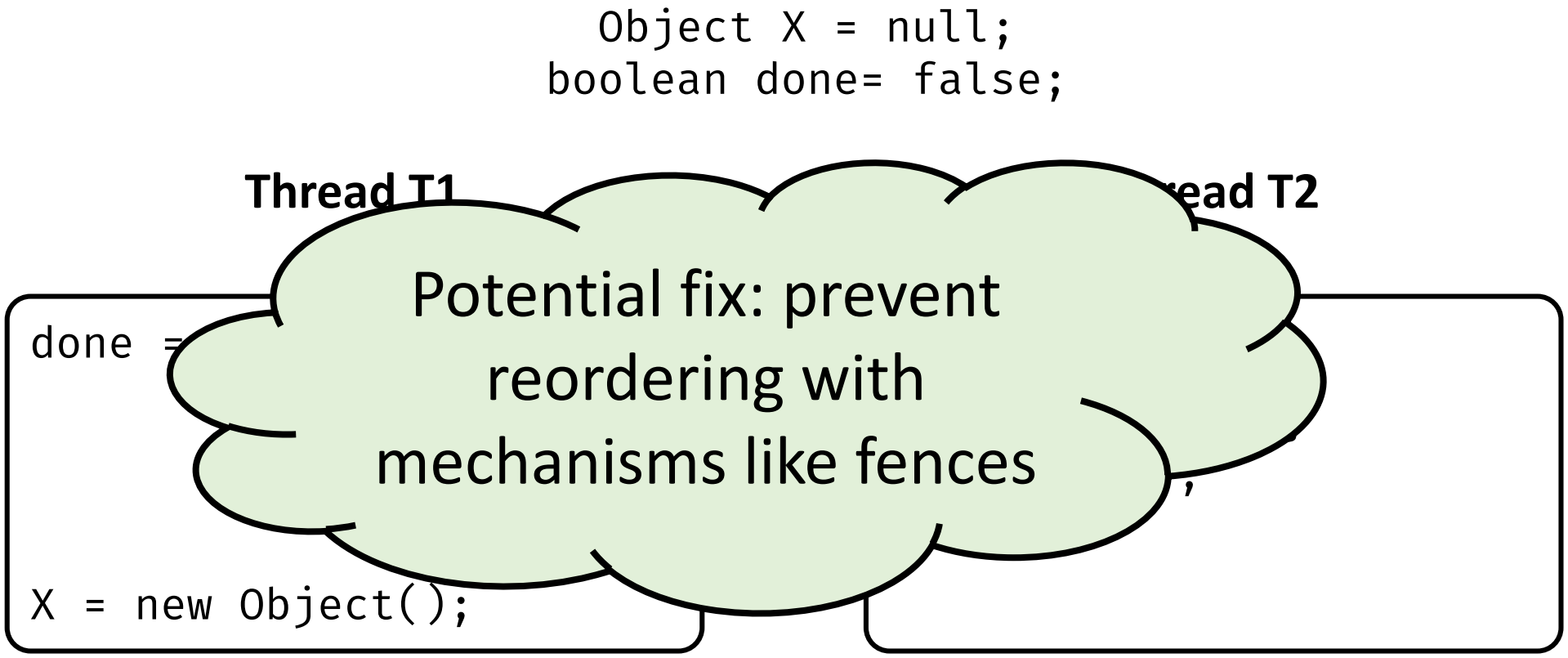
Thread T1

```
done = true;



X = new Object();
```

Thread T2

```
while (!done) {}
X.compute();
```

Sequential Consistency Violation

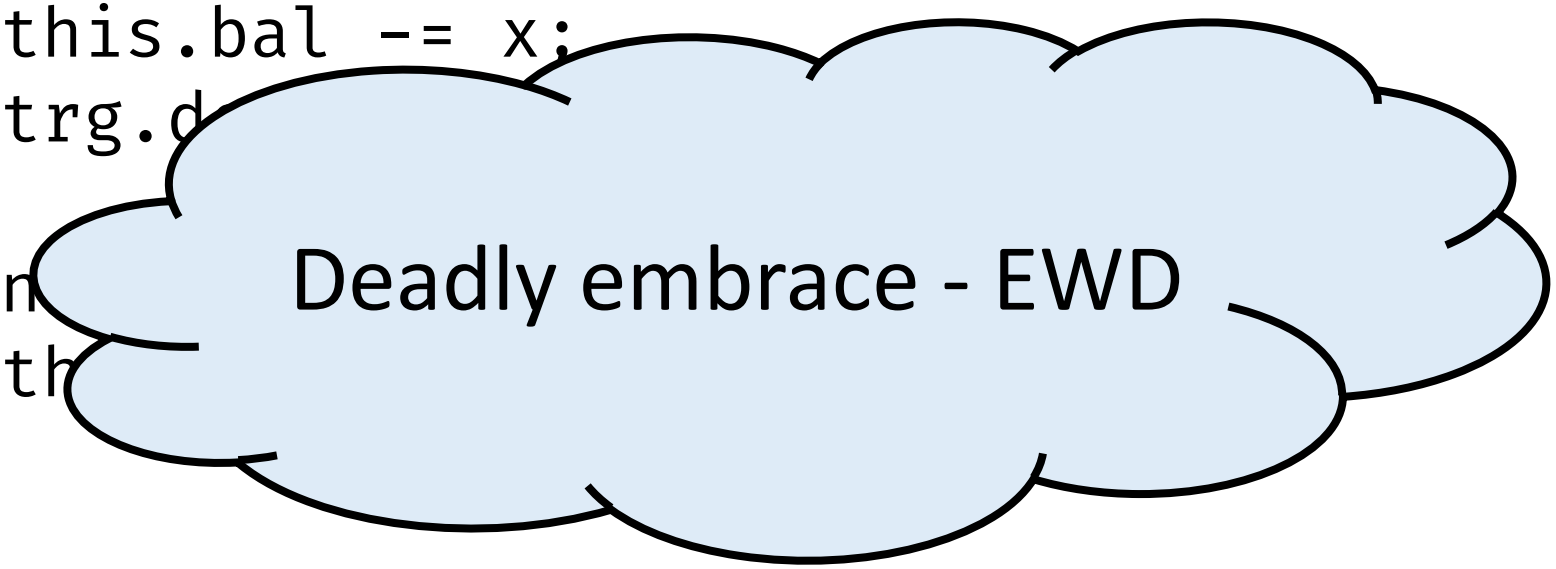


Deadlock

```
public class Account {
  int bal = 0;
  synchronized void transfer(int x, Account trg) {
    this.bal -= x;
    trg.deposit(x);
  }
  synchronized void deposit(int x) {
    this.bal += x;
  }
}
```

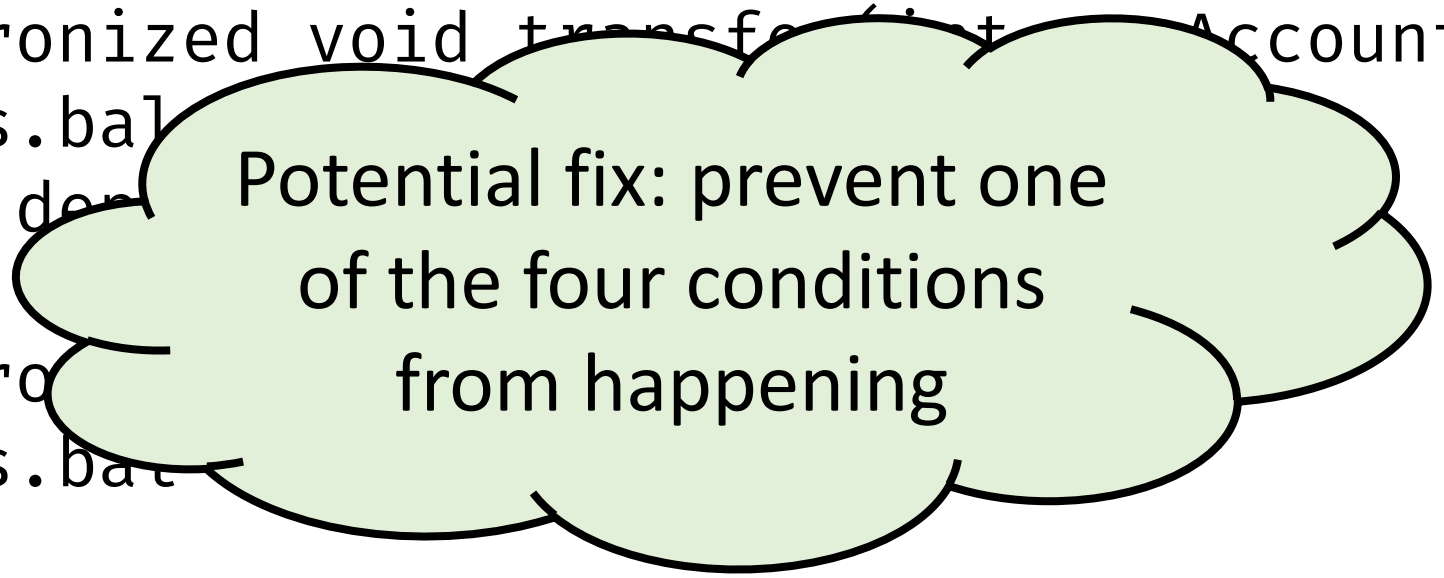
Deadlock

```
public class Account {
  int bal = 0;
  synchronized void transfer(int x, Account trg) {
    this.bal -= x;
    trg.d
  }
  syn
    th
  }
}
```



Deadlock

```
public class Account {
  int bal = 0;
  synchronized void transfe(int      Account trg) {
    this.bal
    trg.dep
  }
  synchro
    this.bal
  }
}
```



Starvation and Livelock

- **Starvation**
  - A thread is unable to get regular access to shared resources and so is unable to make progress

- **Livelock**
  - Threads are not blocked, their states change, but they are unable to make progress

Non-Deadlock Concurrency Bugs

97% of non-deadlock concurrency bugs are due to **atomicity and order violations**

Two-thirds of non-deadlock concurrency bugs are due to **atomicity violations**

Two-thirds of non-deadlock concurrency bugs are due to concurrent accesses to **one variable**

S. Lu et al. Learning from Mistakes -- A Comprehensive Study on Real World Concurrency Bug Characteristics. ASPLOS'08.

Deadlock Bugs

30% of concurrency bugs are due to deadlocks

97% of deadlocks are due to two threads circularly waiting for at most two resources

Considerations with Concurrency Bugs

- Bugs can be **non-deterministic**
  - No assumptions can be made on the order of execution between threads
  - Makes it super-hard to debug and analyze

Challenging to Find the Sweet Spot!

Order, atomicity & sequential consistency violations

Concurrent and correct

Deadlock

Poor performance: lock contention, serialization

Detecting Data Races

An Example of a Data Race

```
Object X = null;
boolean done= false;
```

Thread T1

```
X = new Object();
done = true;
```

Thread T2

```
while (!done) {}
X.compute();
```

Data Race

Conflicting and concurrent accesses

- **Conflicting**
  - Two threads access the same shared variable where at least one access is a write
- **Concurrent**
  - Accesses are not ordered by synchronization operations

Data Race

Concurrent

- "Two threads simultaneously access the same memory location, with at least one access being a write"

Conflicting

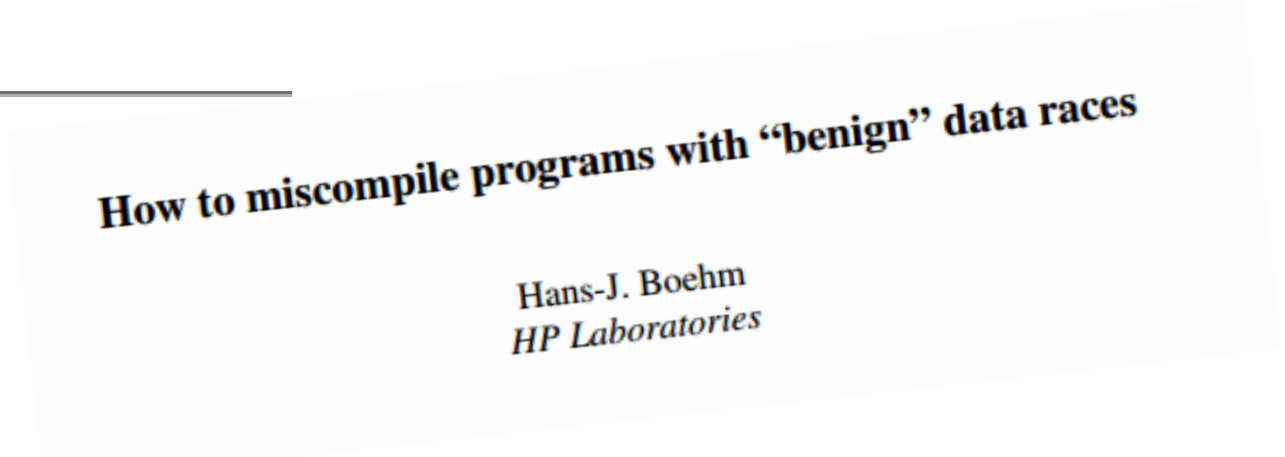
Data Races are Evil!

research highlights

Technical Perspective
Data Races are Evil with No Exceptions

By Sarita Adve

EXPLOITING PARALLELISM HAS become the primary means to higher performance. | racy code. Java's safety requiremer preclude the use of "undefined" beha



How to miscompile programs with "benign" data races

Hans-J. Boehm
HP Laboratories

Data Races are Evil!

- Often indicate the presence of other types of concurrency errors

- Data races ≠ Race conditions
  - Race conditions are timing errors on thread interleavings, lock operations
  - Data races are explicitly on “data variables”

Get Rid of Data Races

Avoiding and/or eliminating data races efficiently is a challenging and unsolved problem

Detecting Data Races

- Notoriously difficult to detect
  - May be induced only by specific thread interleavings
  - Impact on output may not be easily observable unlike deadlocks
  - There are potentially many shared memory locations to monitor

Data Race Detection Techniques

- Happens-before-based algorithms
- Lockset algorithms
- Hybrid analysis
- Other partial order relation-based algorithms (predictive analysis)
- Other techniques

Some terminologies

Sound analysis

- Analysis does not miss any occurrence of bugs
- False negatives imply analysis is unsound

Precise analysis

- Analysis does not report false occurrence of bugs
- False positives imply imprecise analysis

These are not standard terms across all domains, architects might refer to these properties as complete and sound

Static Data Race Detection

- Compile-time analysis of the code
- **Advantages**
  - Can reason about all inputs/interleavings
  - No run-time overhead
- Type-based analysis
  - Augmented language type system to encode synchronization relations
  - Correctly typed program → no data race
  - **Restrictive** and **tedious**

```
class Account {
  int balance guarded by this;

  int deposit(int x) requires this {
    this.balance = this.balance + x
  }
}
```

Challenges with Static Data Race Detection

- Static analysis does **NOT** scale well
  - E.g.: may/must-happen-in-parallel
- Language features like dynamic class loading and reflection in Java make static analysis difficult
  - Too conservative leading to many false positives

Dynamic Data Race Detection

- Monitor program operations **during** execution
- Program may be "**instrumented**" with additional instructions
- Instrumentation should **NOT** change program functionality

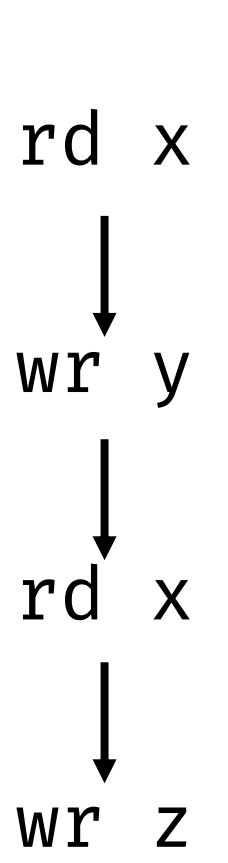
Dynamic Data Race Detection

- Monitor program operations **during** execution
- Program may be "**instrumented**" with additional instructions
- Instrumentation should **NOT** change program functionality

  - Post-mortem analysis
  - On-the-fly methods

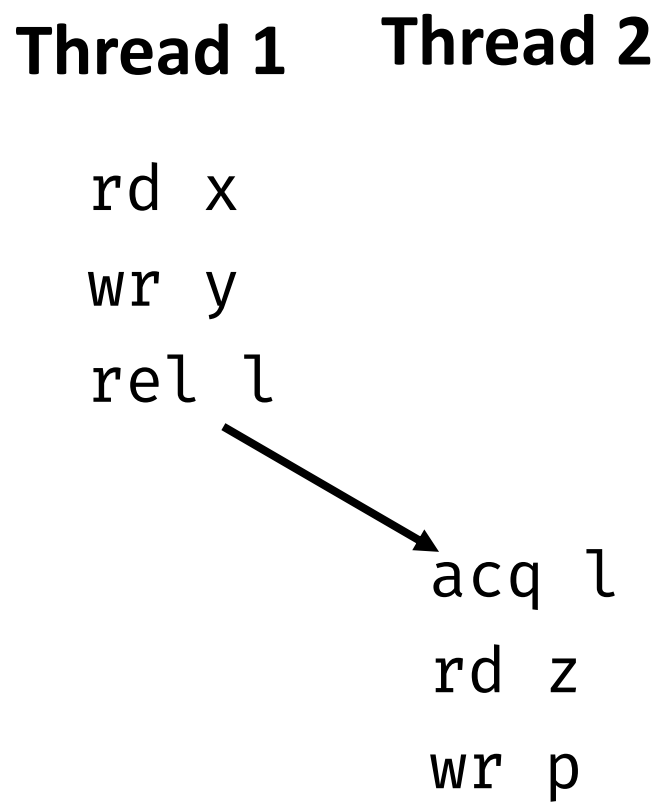
Happens-before Relation

- Smallest transitively-closed relation $\prec_{HB}$ over operations
- Given two operations a and b, a $\prec_{HB}$ b if one of the following conditions hold
  - **Program order**
    - Operation a is performed by the same thread before operation b



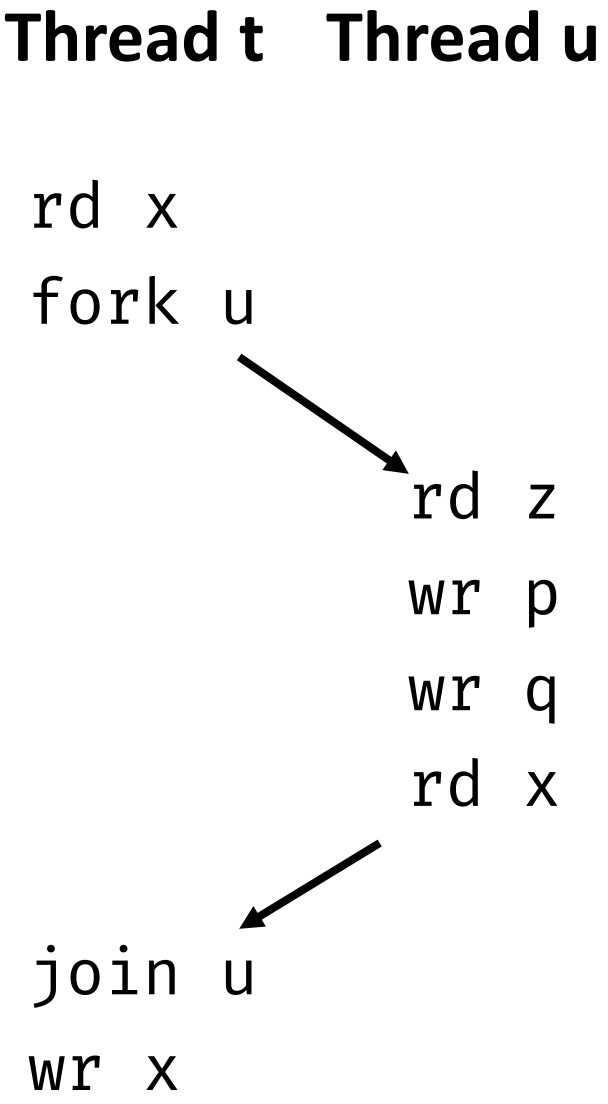
Happens-before Relation

- Smallest transitively-closed relation $\prec_{HB}$ over operations
- Given two operations a and b, a $\prec_{HB}$ b if one of the following conditions hold
  - Program order
    - Operation a is performed by the same thread before operation b
  - **Synchronization order**
    - a is a lock release and b is an acquire of the **same** lock



Happens-before Relation

- Smallest transitively-closed relation $\prec_{HB}$ over operations
- Given two operations a and b, a $\prec_{HB}$ b if one of the following conditions hold
  - Program order
    - Operation a is performed by the same thread before operation b
  - Synchronization order
    - a is a lock release and b is an acquire of the **same** lock
  - **Fork-join order**
    - a is a fork operation (e.g., fork(t, u)) and b is by thread u
    - a is by thread u and b is a join operation (e.g., join(t, u))



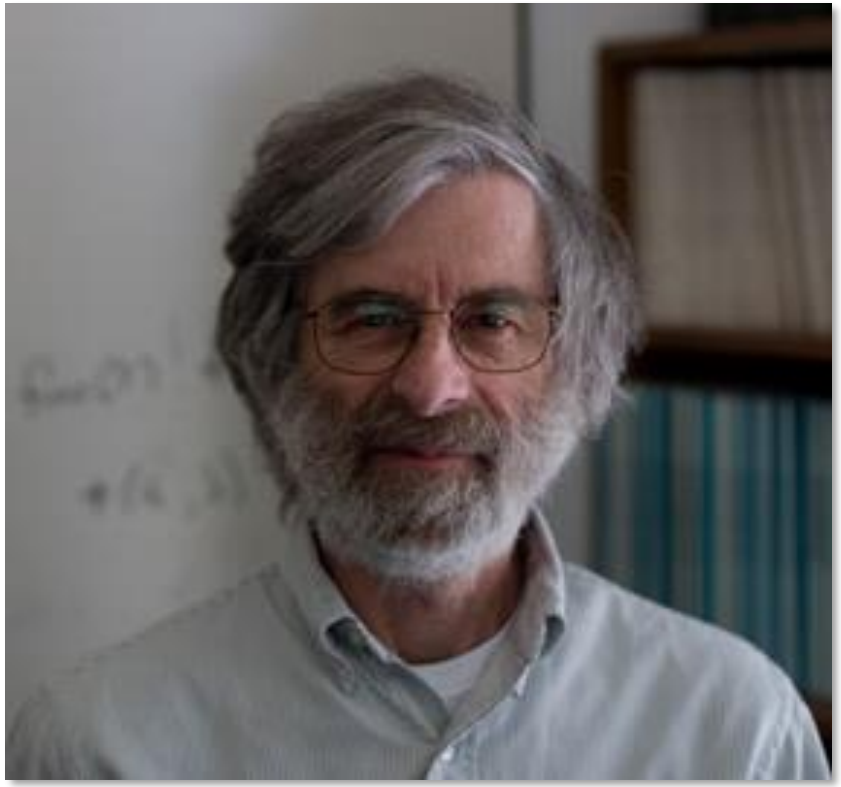
Happens-before Relation

If $a \prec_{HB} b$ and $b \prec_{HB} c$, then $a \prec_{HB} c$

If $a \not\prec_{HB} b$ and $b \not\prec_{HB} a$, then $a \parallel_{HB} b$

Happens-before Relation

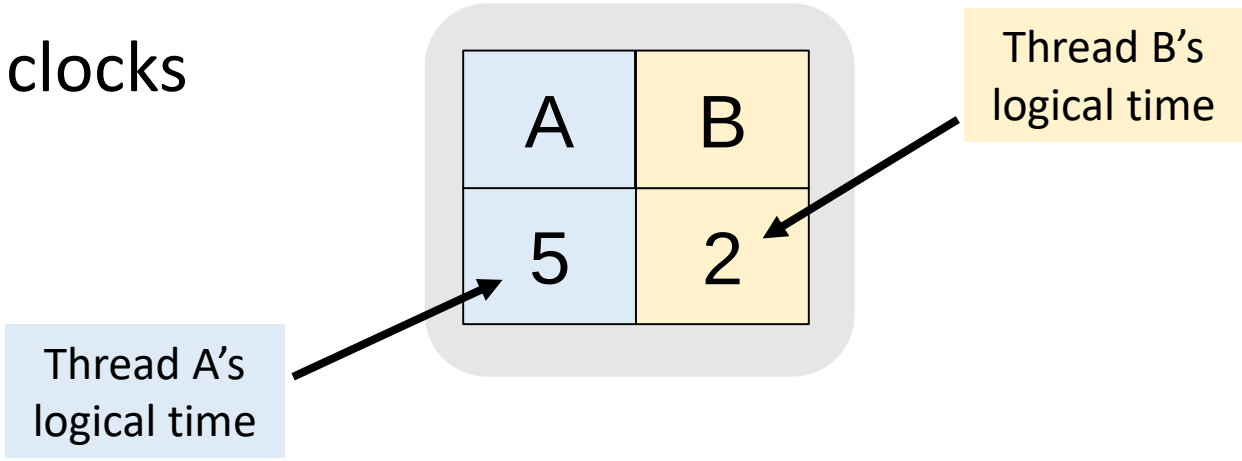
- **Leslie Lamport**
  - Winner of the 2013 Turing award for advances in reliability of distributed/concurrent systems
  - Lamport clocks, Happens-before relation, sequential consistency, Bakery algorithm, LaTeX, …



Vector Clock

- Each thread T maintains its own logical clock ‘c’
  - Initially c=0 when T starts
  - Clock is **incremented** at synchronization release operations
    - For example, release(m), volatile write

- Vector clock is a vector of logical clocks
  - For all the threads in the process



Vector Clock and Happens-before

$$VC_1 \sqsubseteq VC_2 \quad iff \quad \forall t \; VC_1(t) \leq VC_2(t)$$

A	B	C
4	2	3

$\sqsubseteq$

A	B	C
4	2	3

A	B	C
4	5	3

$\sqsubseteq$

A	B	C
4	2	3

Vector Clocks

Join

$$VC_1 \sqcup VC_2 = \lambda t.\max(VC_1(t), VC_2(t))$$

Initialization

$$\perp_V = \lambda t.0$$

Increment

$$inc_t(V) = \lambda u.\, if\ u = t\ then\ VC(u) + 1\ else\ V(u)$$

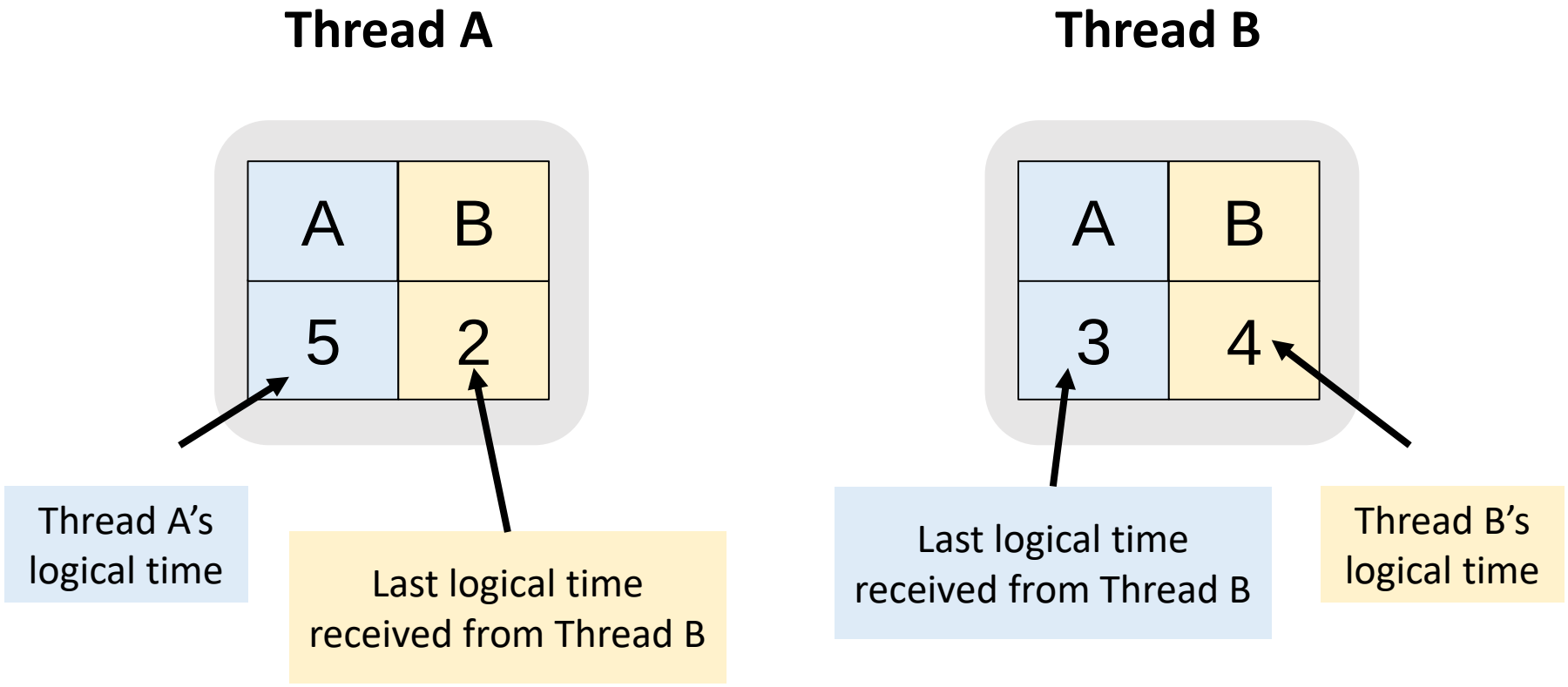
Properties of Vector Clocks

$$if\ VC_a \sqsubset VC_b, then\ a \prec b$$

$$if\ VC_a \sqsubset VC_b, then\ \neg(VC_b \sqsubset VC_a)$$

$$if\ (VC_a \sqsubset VC_b)\ \wedge\ (VC_b\ \sqsubset VC_c), then\ VC_a \prec VC_c$$

Vector Clock-based Race Detection



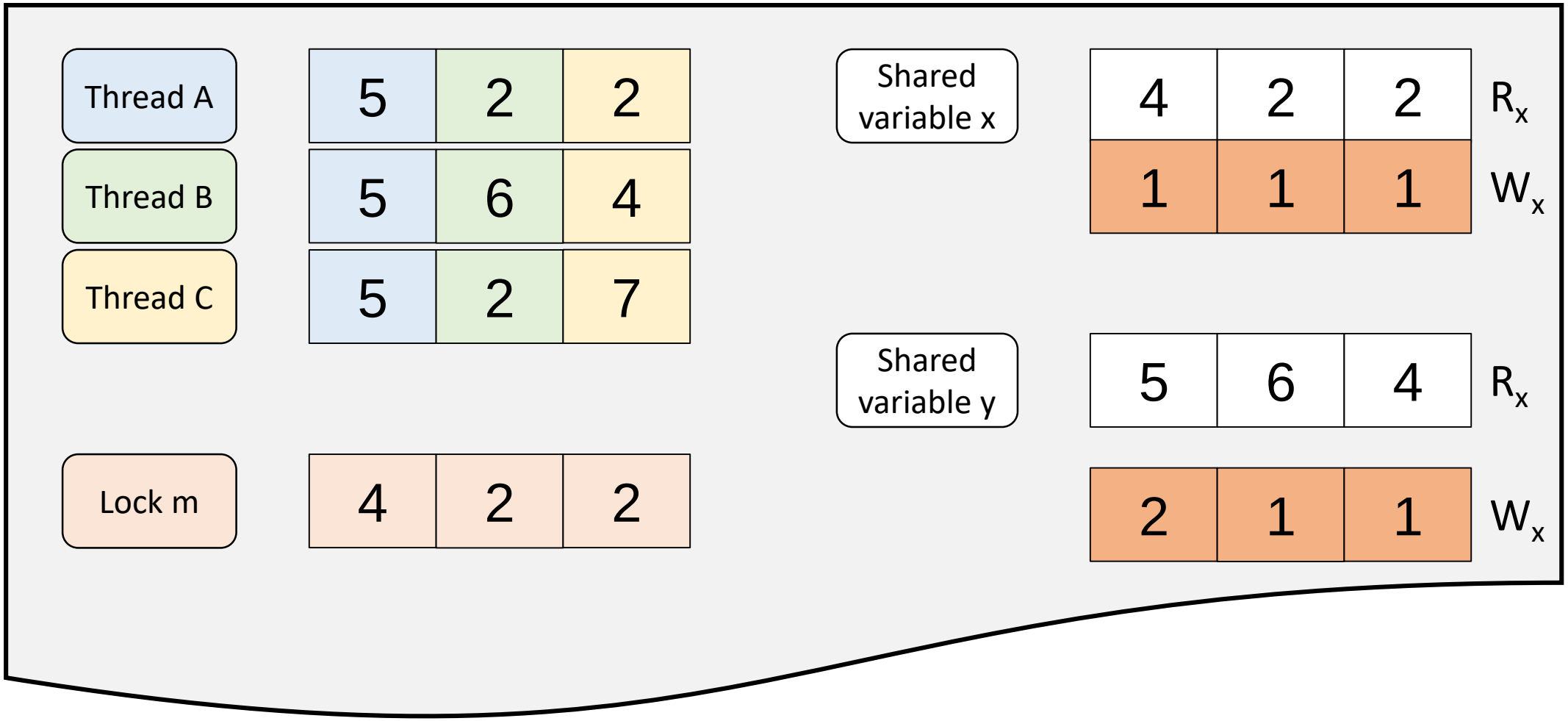
DJIT$^+$ Algorithm

- Each thread has its own clock that is incremented at lock synchronization operations with release semantics
- Each thread also keeps **a vector clock** C_t
  - For a thread u, $C_t(u)$ gives the clock for the last operation of u that happened before the current operation of t
- Each lock has **a vector clock**
- Each shared variable x has **two vector clocks** R_x and W_x

---

E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. CCPE, 2007

Snapshot of Process Memory



Thread A

A	B
5	2

Thread B

A	B
3	4

Thread A

A	B
5	2

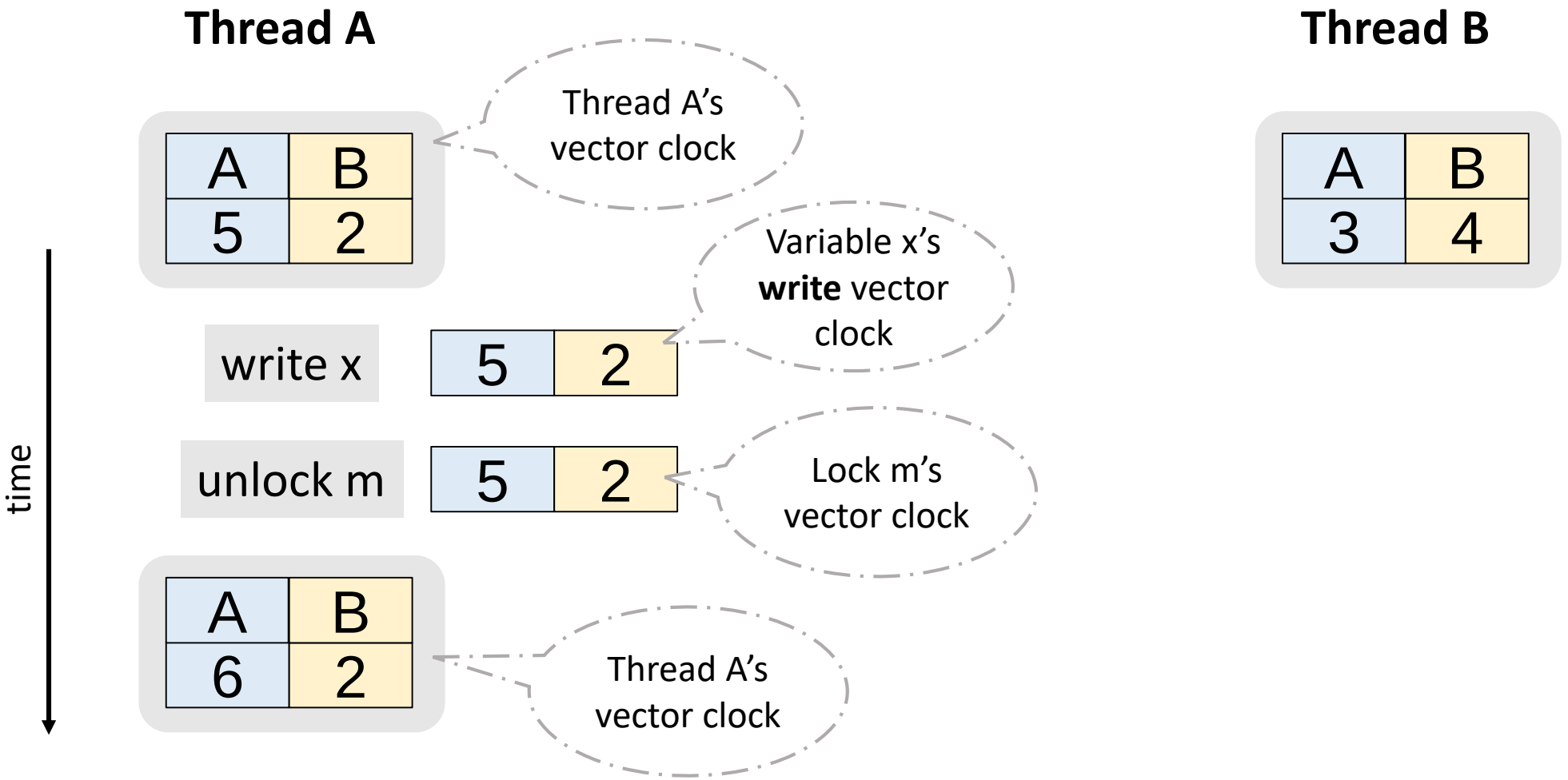
write x

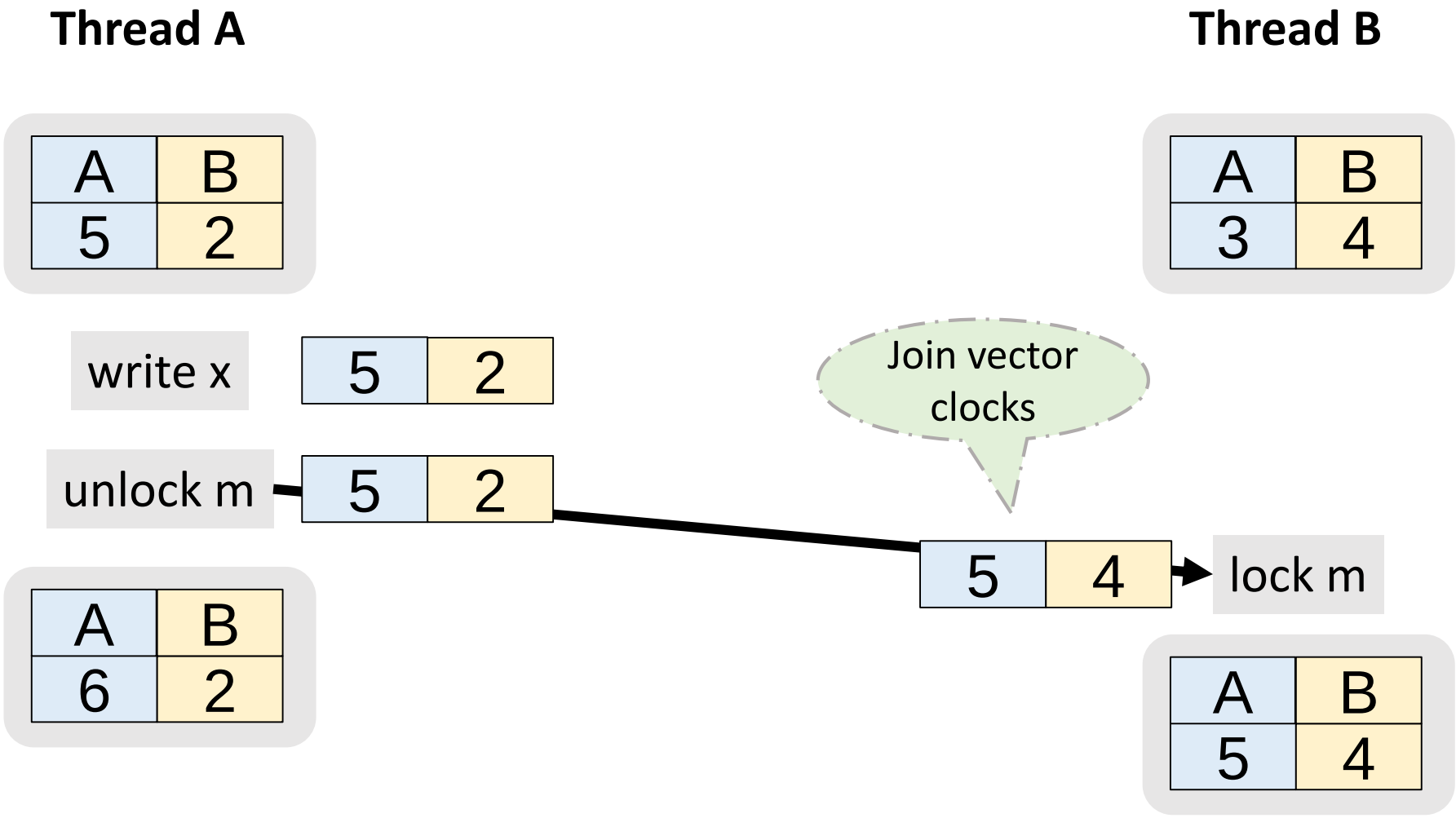
5	2

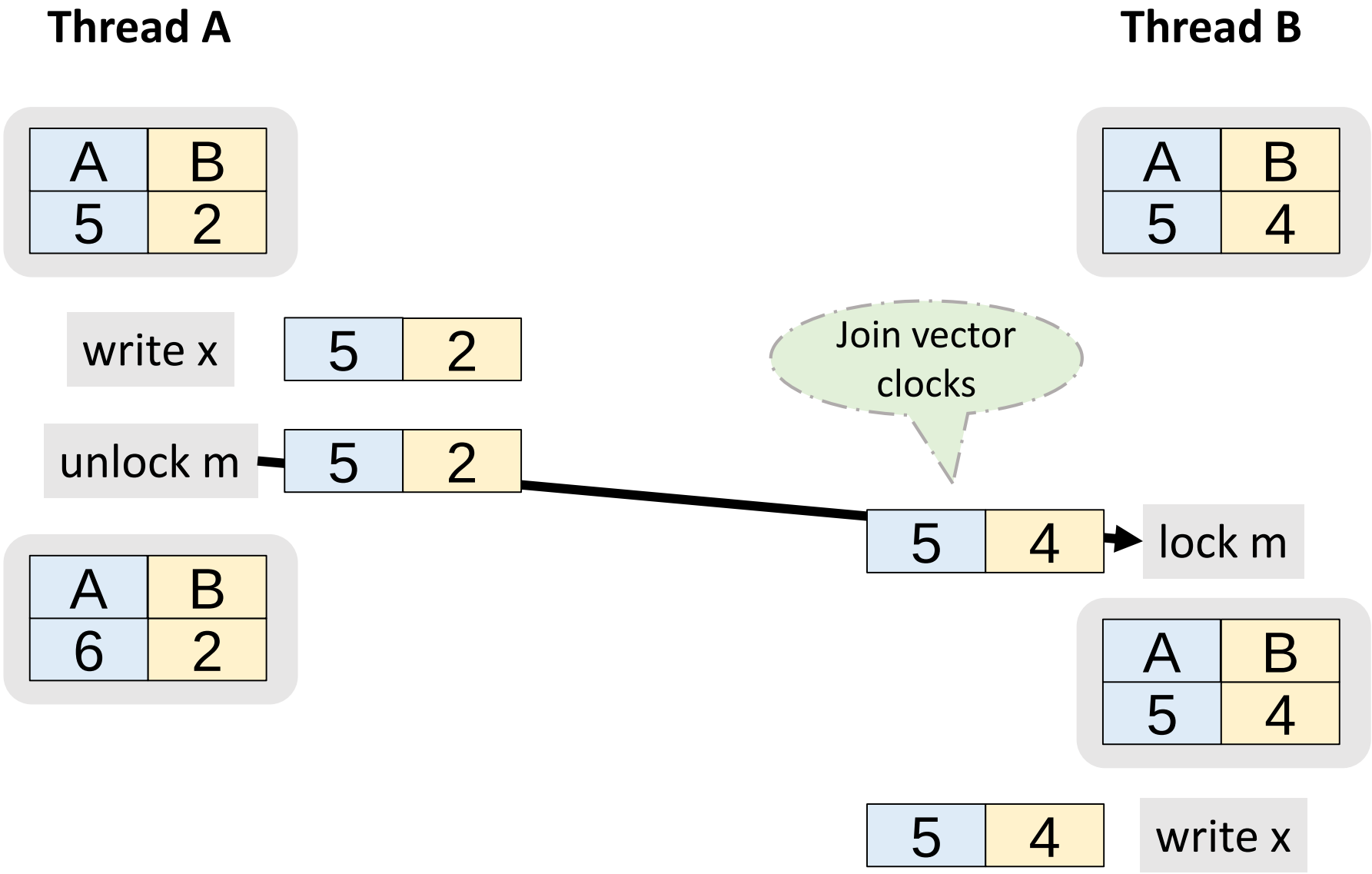
time

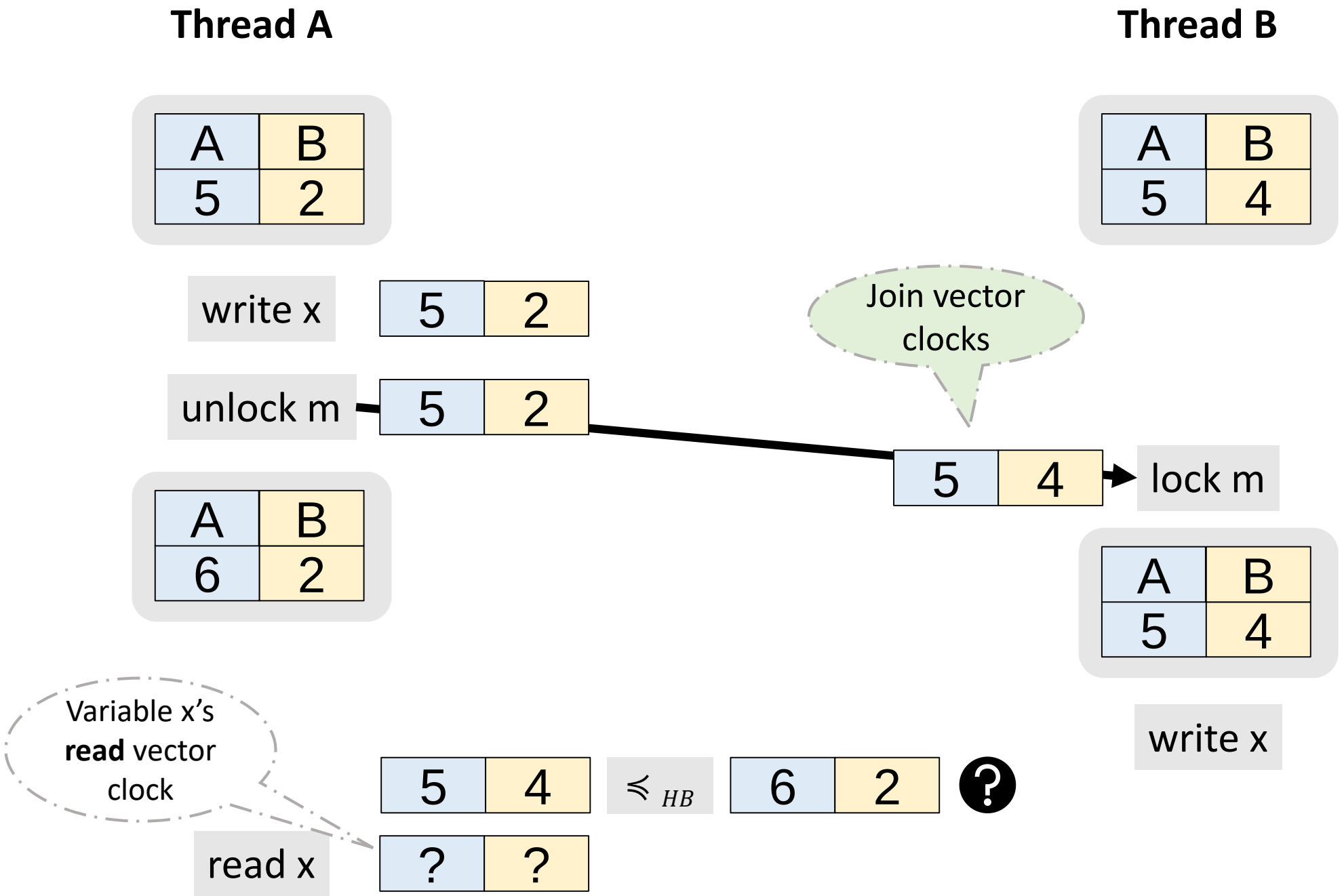
Thread B

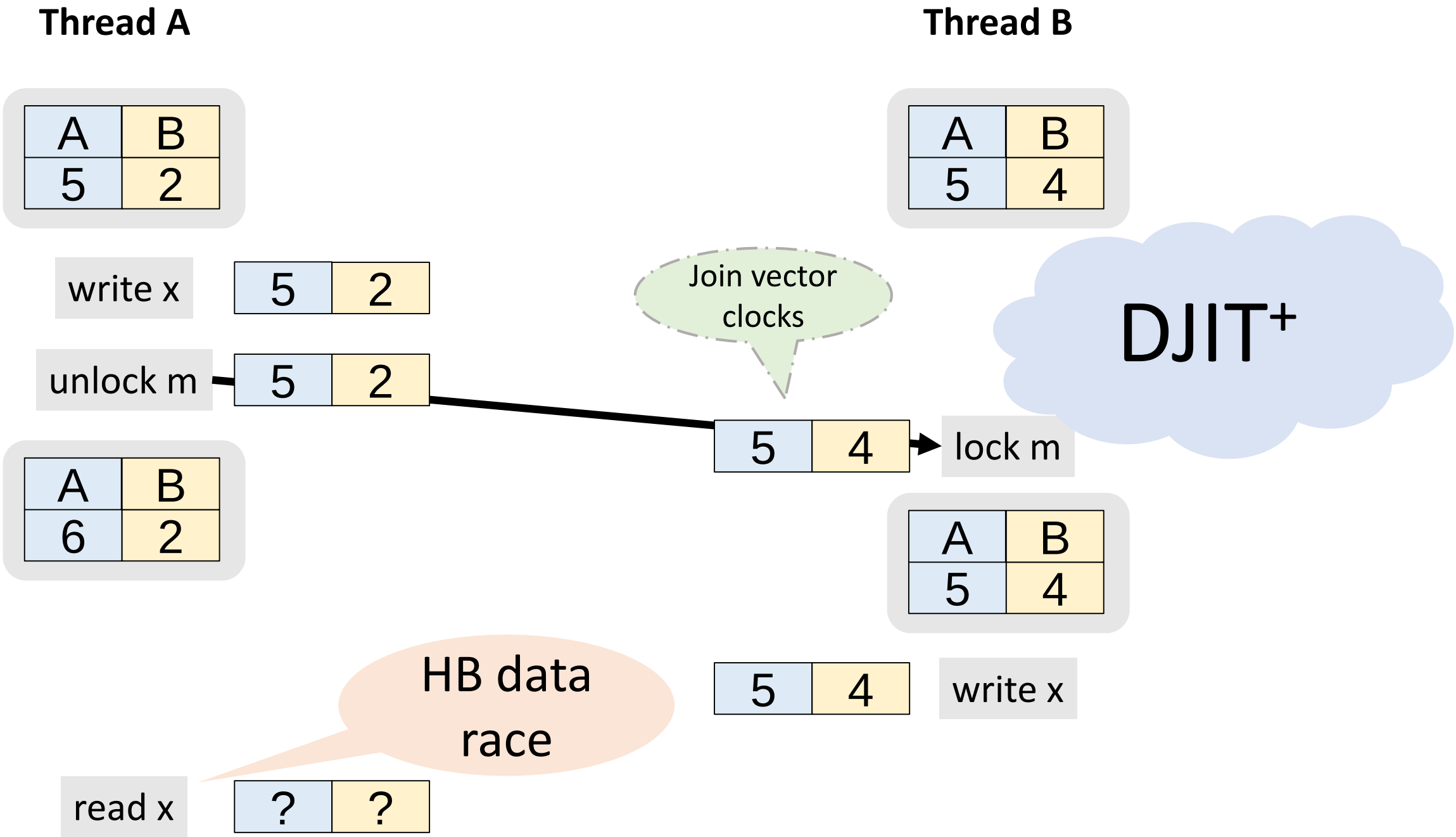
A	B
3	4


Thread A
Thread B
A
B
5
2
Thread A's vector clock
A
B
3
4
Variable x's write vector clock
write x
5
2
unlock m
5
2
Lock m's vector clock
time
A
B
6
2
Thread A's vector clock


Thread A
Thread B
A
B
5
2
A
B
3
4
write x
5
2
Join vector clocks
unlock m
5
2
5
4
lock m
A
B
6
2
A
B
5
4


Thread A
Thread B
A
B
5
2
A
B
5
4
write x
5
2
Join vector clocks
unlock m
5
2
5
4
lock m
A
B
6
2
A
B
5
4
5
4
write x


Thread A
Thread B
A B
5 2
A B
5 4
write x
5 2
Join vector clocks
unlock m
5 2
5 4
lock m
A B
6 2
A B
5 4
write x
Variable x's **read** vector clock
5 4
≼ HB
6 2
?
read x
? ?


Thread A
Thread B
A B
5 2
A B
5 4
write x
5 2
Join vector clocks
DJIT+
unlock m
5 2
5 4
lock m
A B
6 2
A B
5 4
HB data race
5 4
write x
read x
? ?

Analysis of HB Tracking

- HB analysis are
  - **precise**, i.e., no false positives
  - **dynamically sound**, i.e., no false negatives given the observed run

HB analysis can however **MISS** data races that did not manifest in observed run, but may happen in **ANOTHER** interleaving

Question: Is there a HB data race on variable y?

Thread A

```
y = y + 1

lock m
v = v + 1
unlock m
```

Thread B

```
lock m
v = v + 1
unlock m

y = y + 1
```

time

Track HB edges with vector clocks!

Lockset Algorithms

- **Assumption**: all shared-memory accesses follow a consistent locking discipline

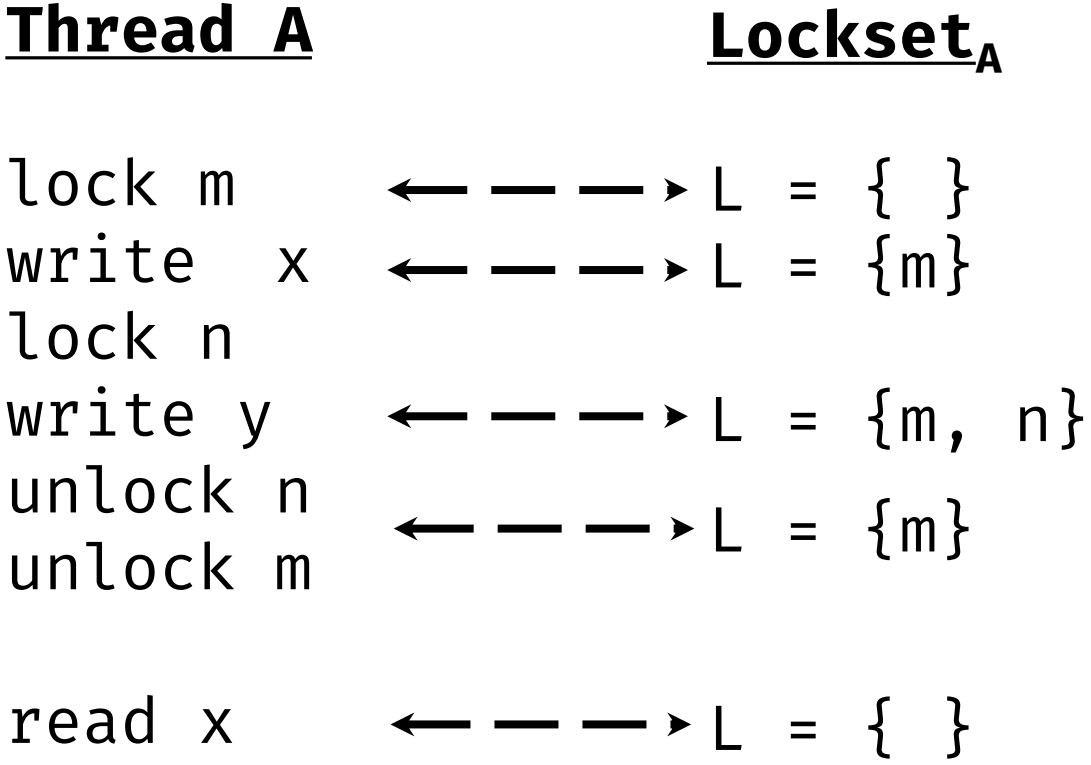
- Keeps track of the locks associated with each thread and program variable

---

S. Savage et al. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. TOCS, 1997.

Lockset Algorithms

- **Assumption**: all shared-memory accesses follow a consistent locking discipline
- Keeps track of the locks associated with each thread and program variable

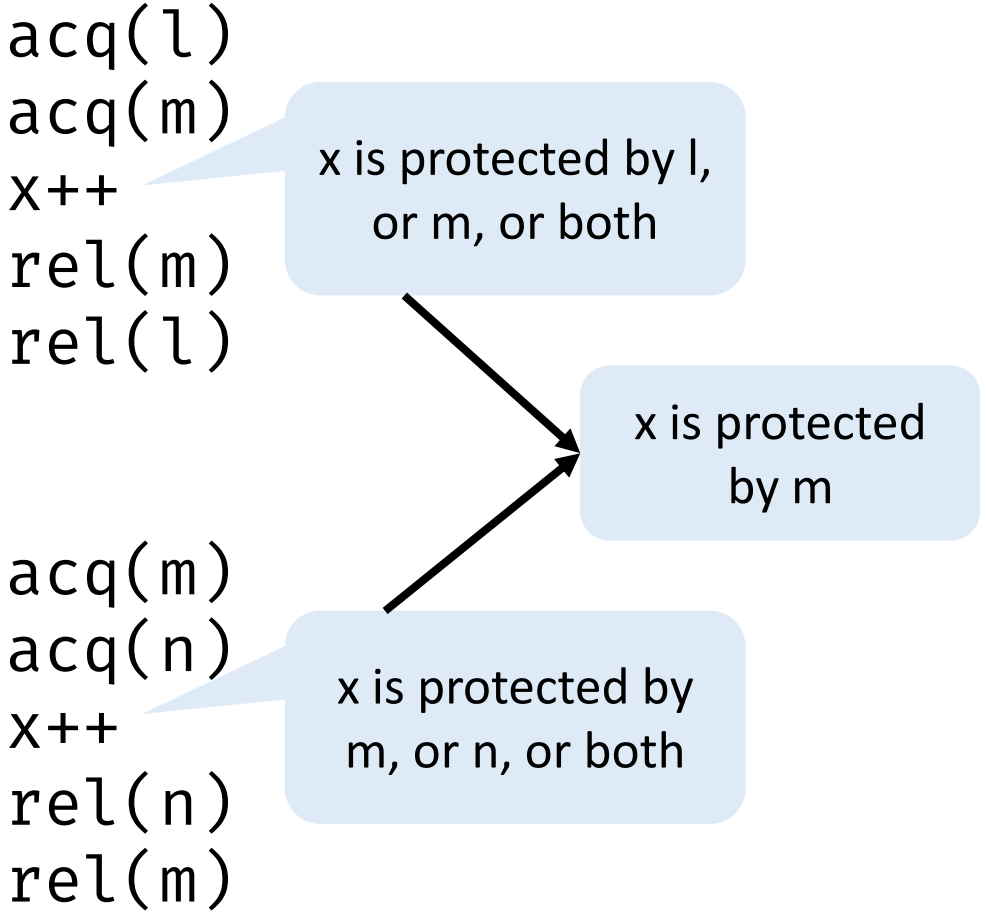


Lockset Algorithms

- Two accesses from different threads with **non-intersecting locksets** form a data race

Inferring the Locking Discipline

- How do we know which lock protects which variable?
  - Programmer annotations is cumbersome
- Infer from the program



Eraser Algorithm

- Eraser monitors every read/write and lock/unlock operation in an execution

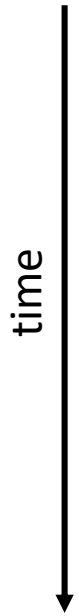
- Eraser assumes that it knows the full set of locks in advance

Eraser Algorithm

- For each variable v, Eraser maintains the lockset C(v), candidate locks for the lock discipline
  - For each variable v, initialize C(v) to the set of all locks
- For each read/write on variable v by thread t
  - Let L(t) be the set of locks held by thread t
  - C(v) := C(v) ∩ L(t)
  - If C(v) = ∅, report that there is a data race for v

Lockset refinement

Question: Is there a data race on variable y?



Thread A

```
y = y + 1

lock m
v = v + 1
unlock m
```

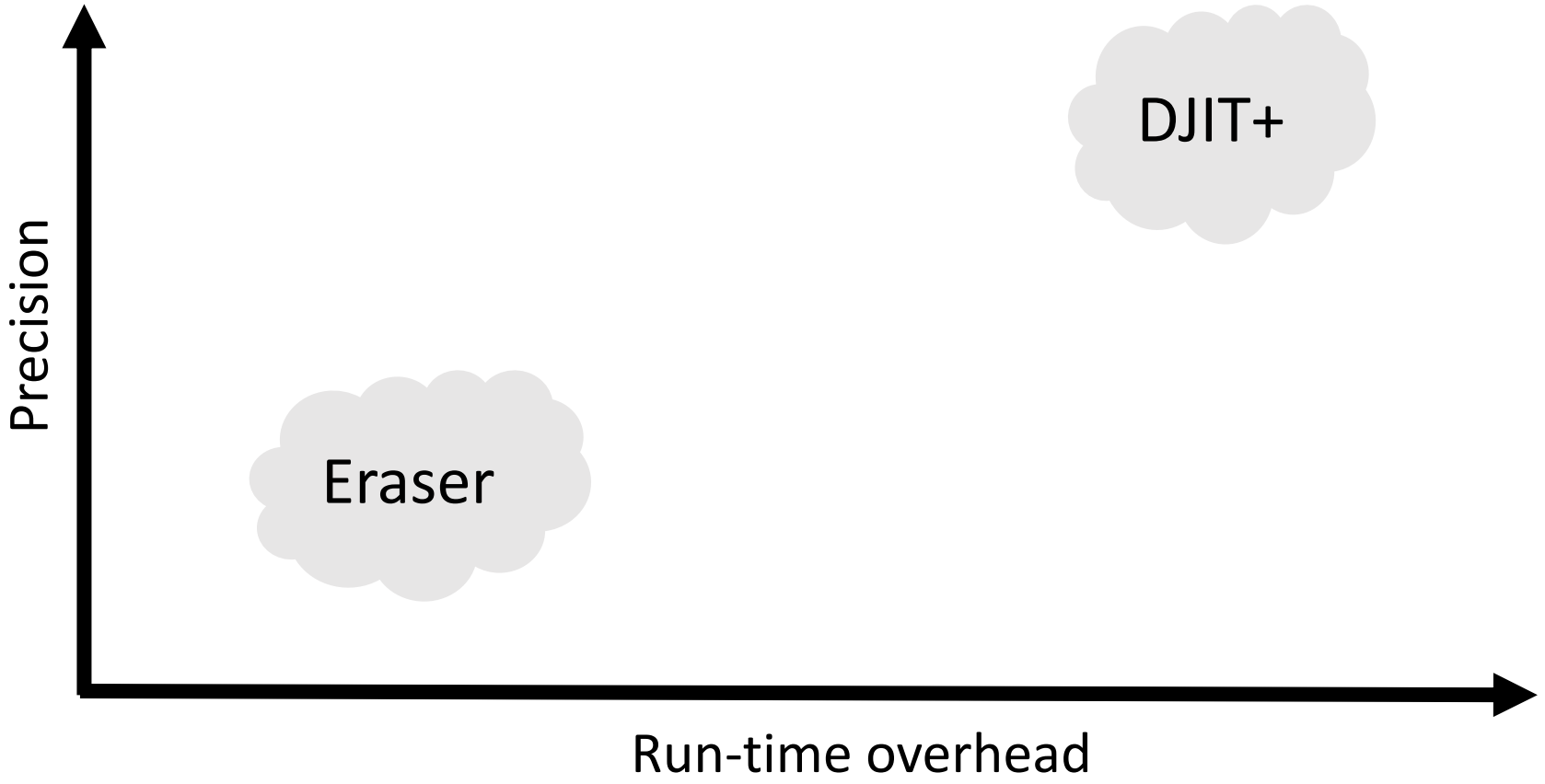
Thread B

```
lock m
v = v + 1
unlock m

y = y + 1
```

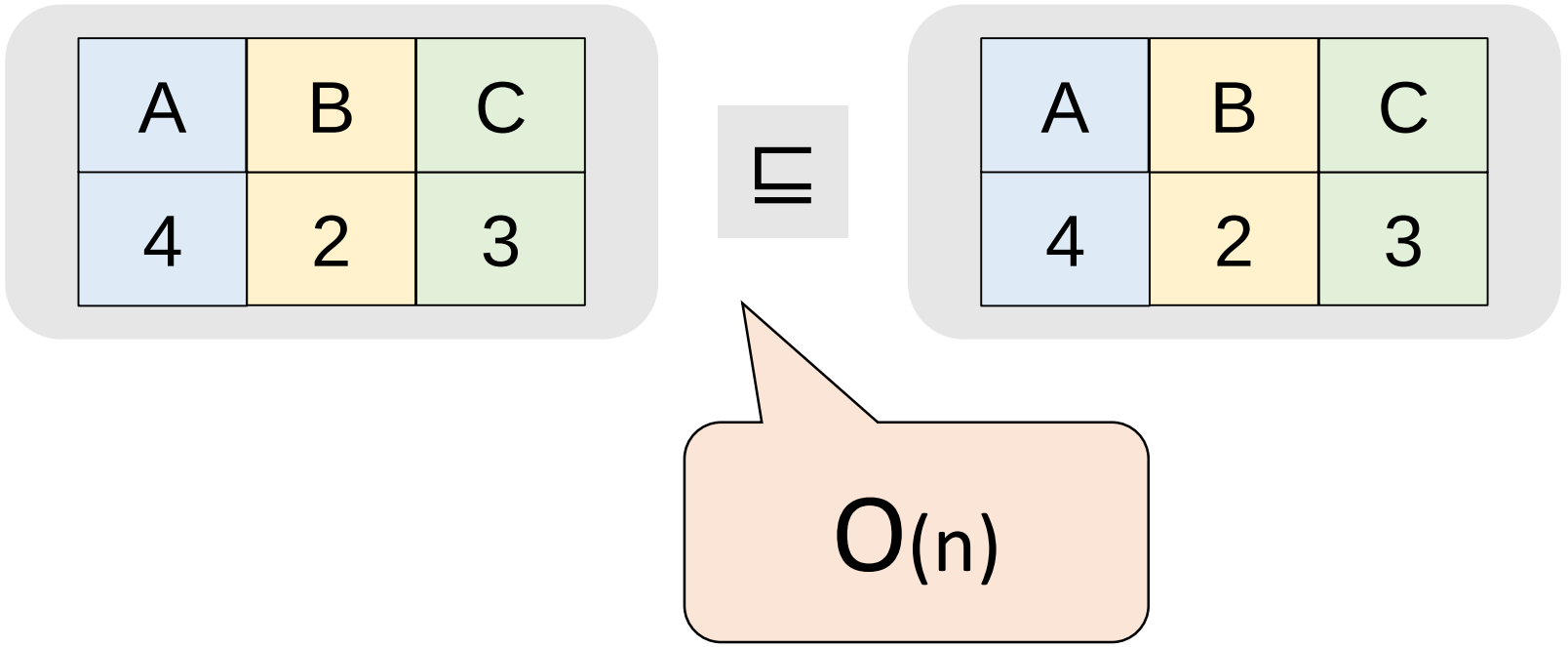
Properties of Lockset Algorithms

- **Question**
  - Argue whether lockset algorithms are precise or imprecise?

DJIT$^+$ vs Eraser



Why is DJIT$^+$ expensive?

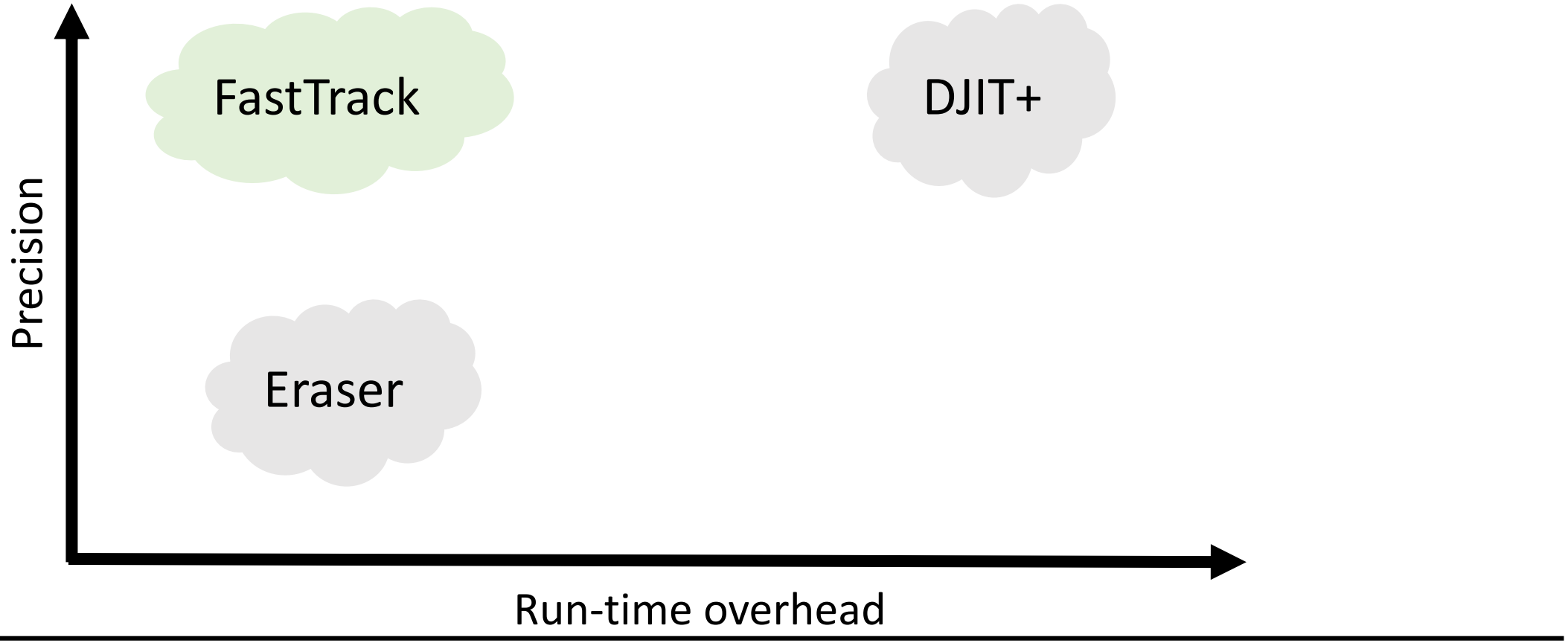


Why is DJIT$^+$ expensive?

Reads and writes to shared-memory locations (i.e., scalar fields and array elements) constitute >= 90% of all monitored operations

$O(n)$

FastTrack: Efficient HB Tracking



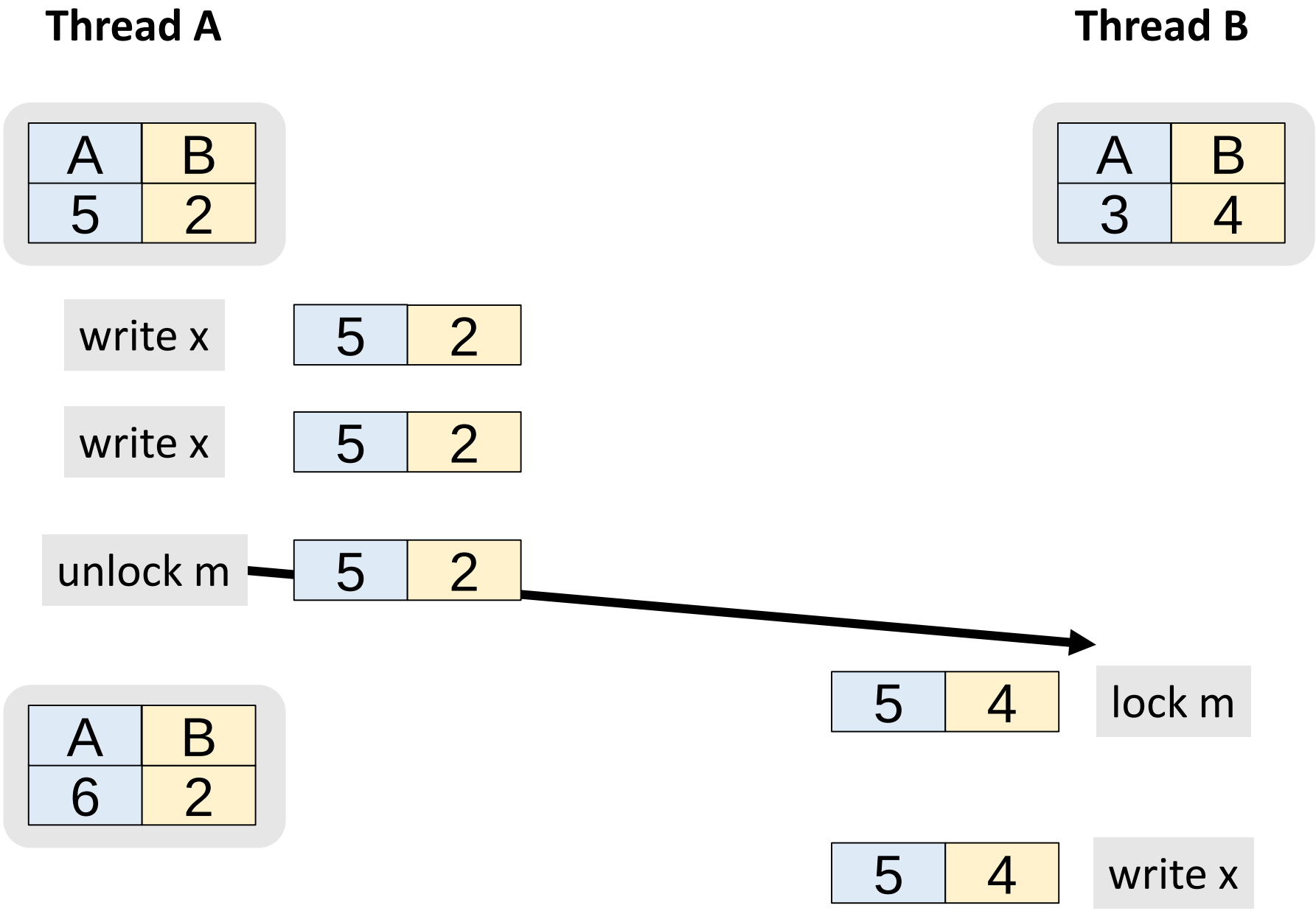
C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Data Race Detection. PLDI, 2009.

FastTrack: Efficient HB Tracking

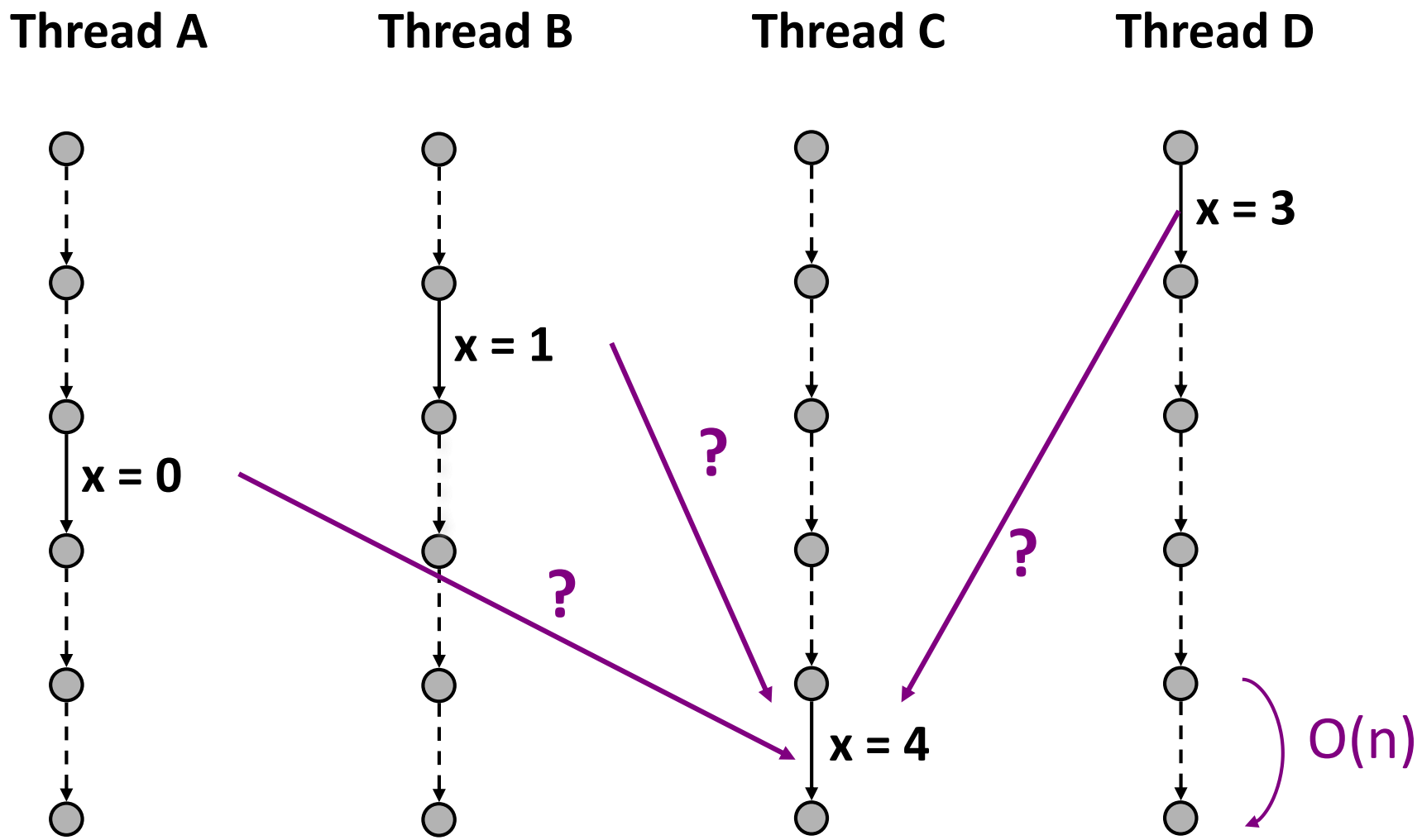
- **Insight:** HB relation is a partial order

All writes to a shared variable **till the first race** is totally ordered

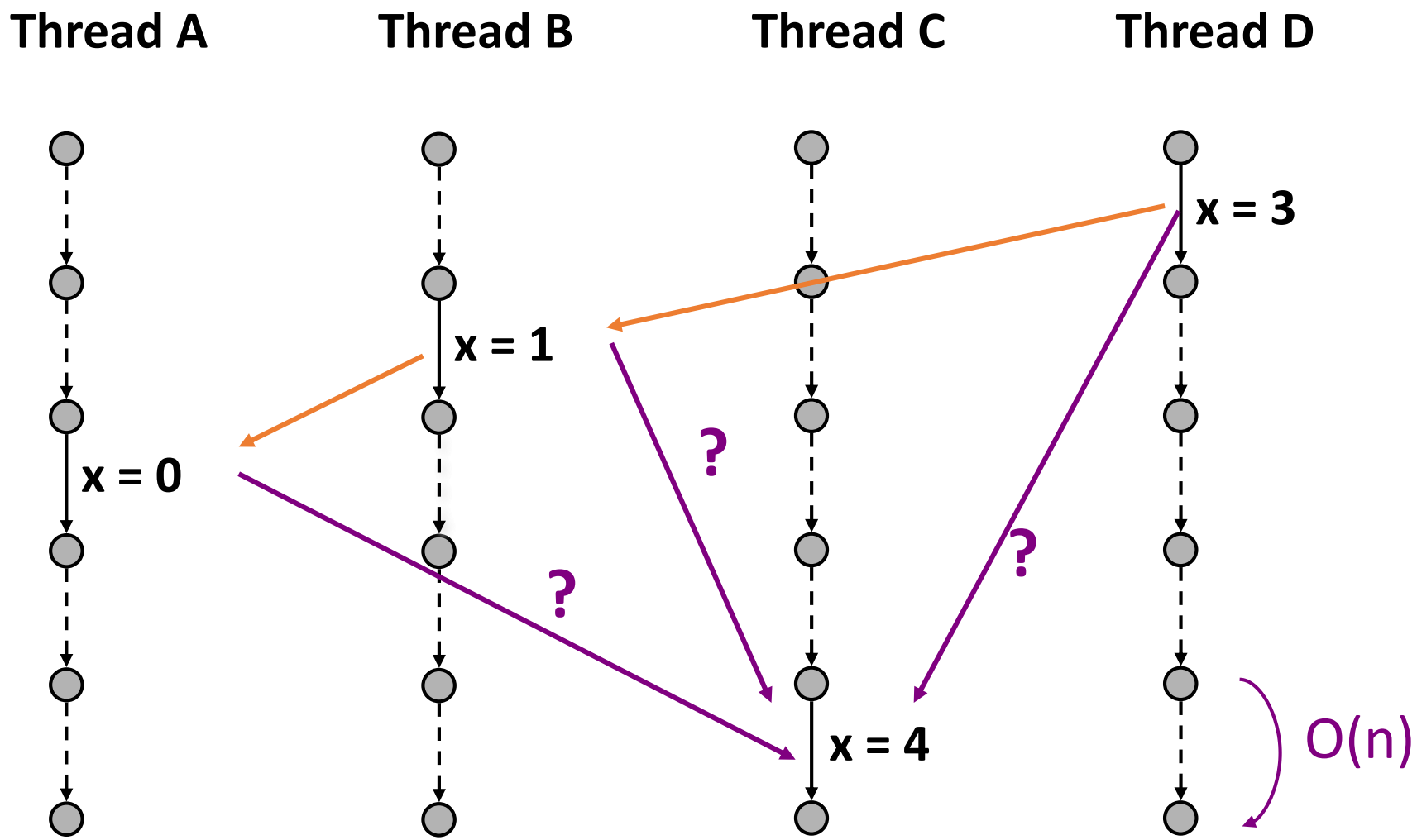
- Remember: Reads are **NOT** totally-ordered even in data-race-free programs
  - E.g.: Read-shared data


Thread A
Thread B
A
B
5
2
A
B
3
4
write x
5
2
write x
5
2
unlock m
5
2
5
4
lock m
A
B
6
2
5
4
write x

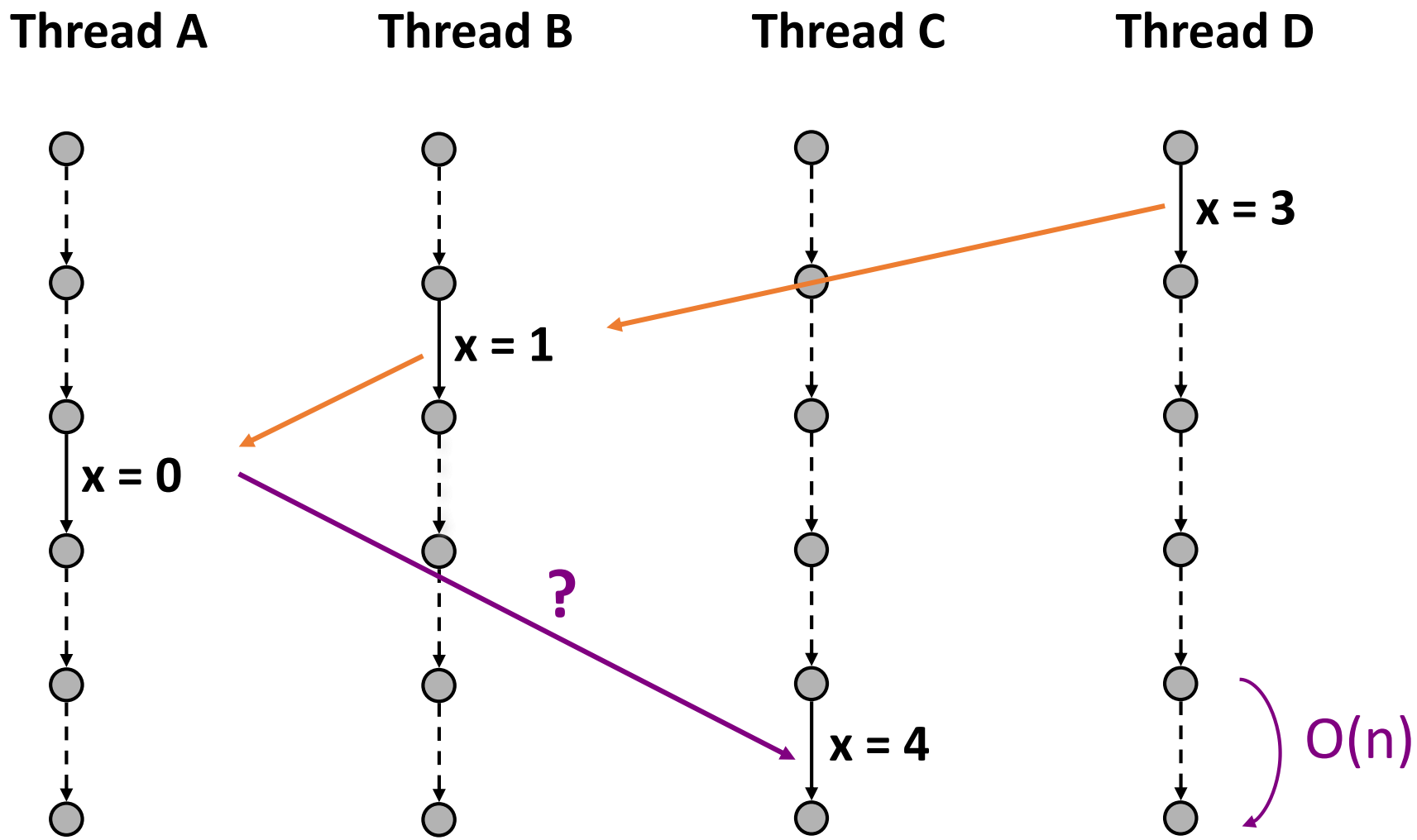
Write-Write and Write-Read Data Races



No Data Races Yet: Writes Totally Ordered

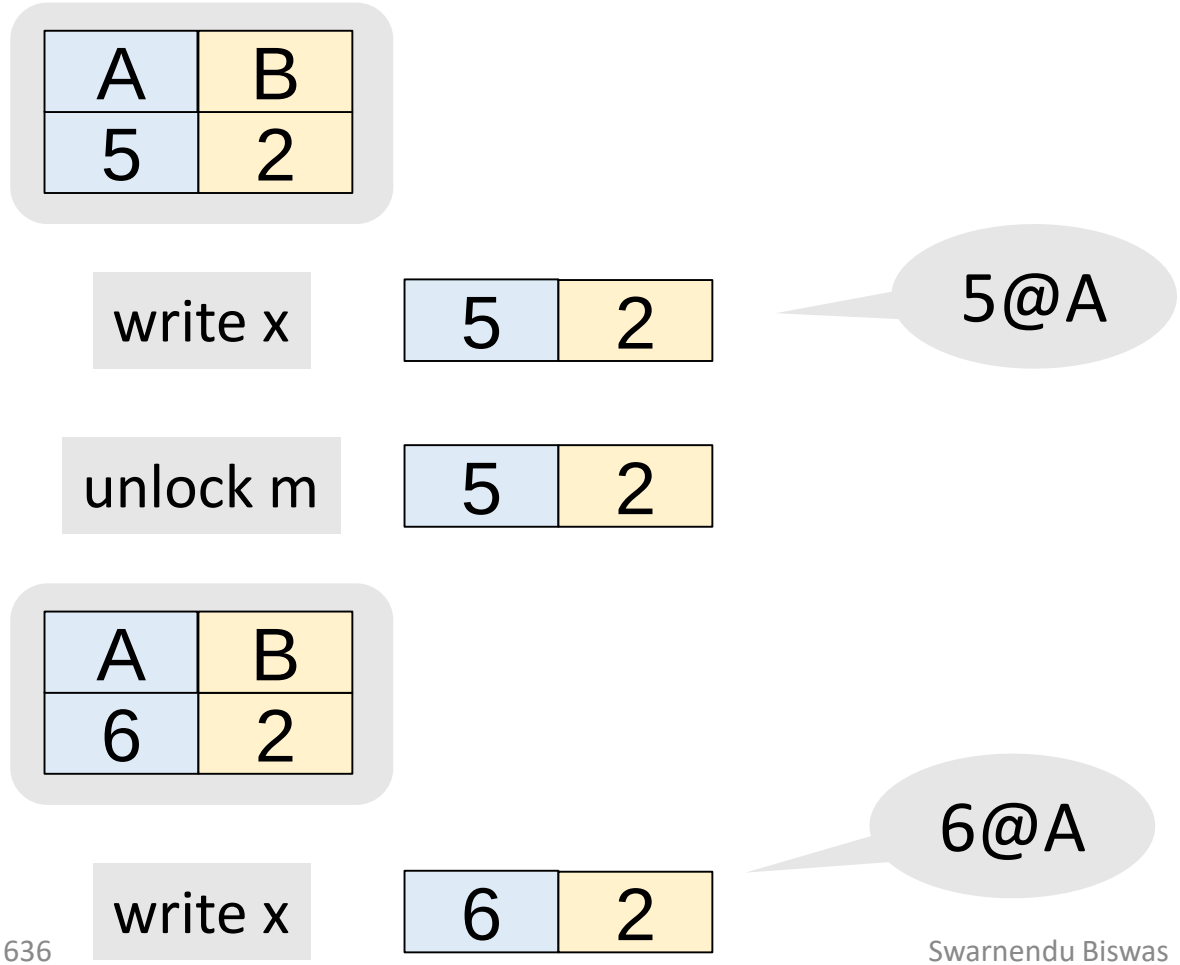


No Data Races Yet: Writes Totally Ordered



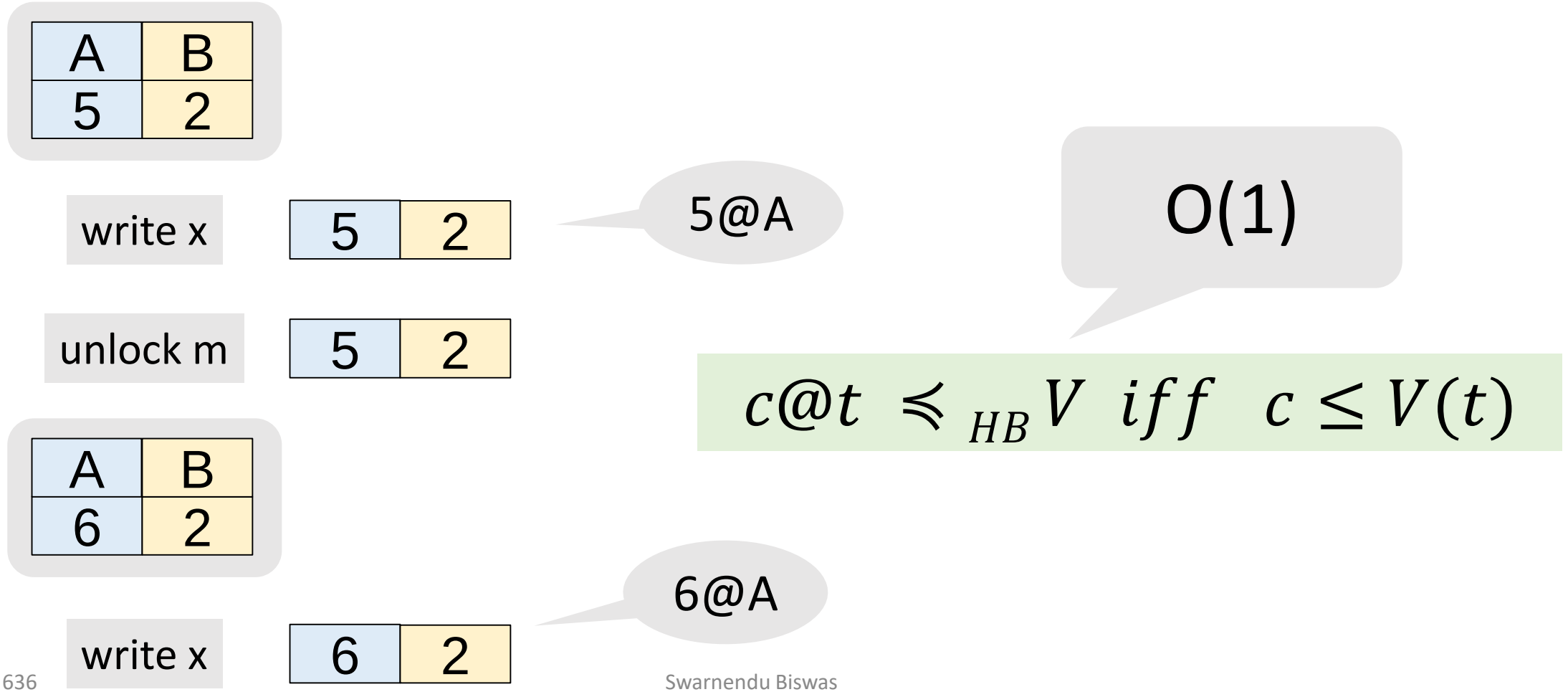
Last Writer Epoch

Thread A



Last Writer Epoch

Thread A



$$c@t \preccurlyeq_{HB} V \;\; iff \;\; c \leq V(t)$$

Thread A

A	B
5	2

Thread B

A	B
3	4

Thread A

A	B
5	2

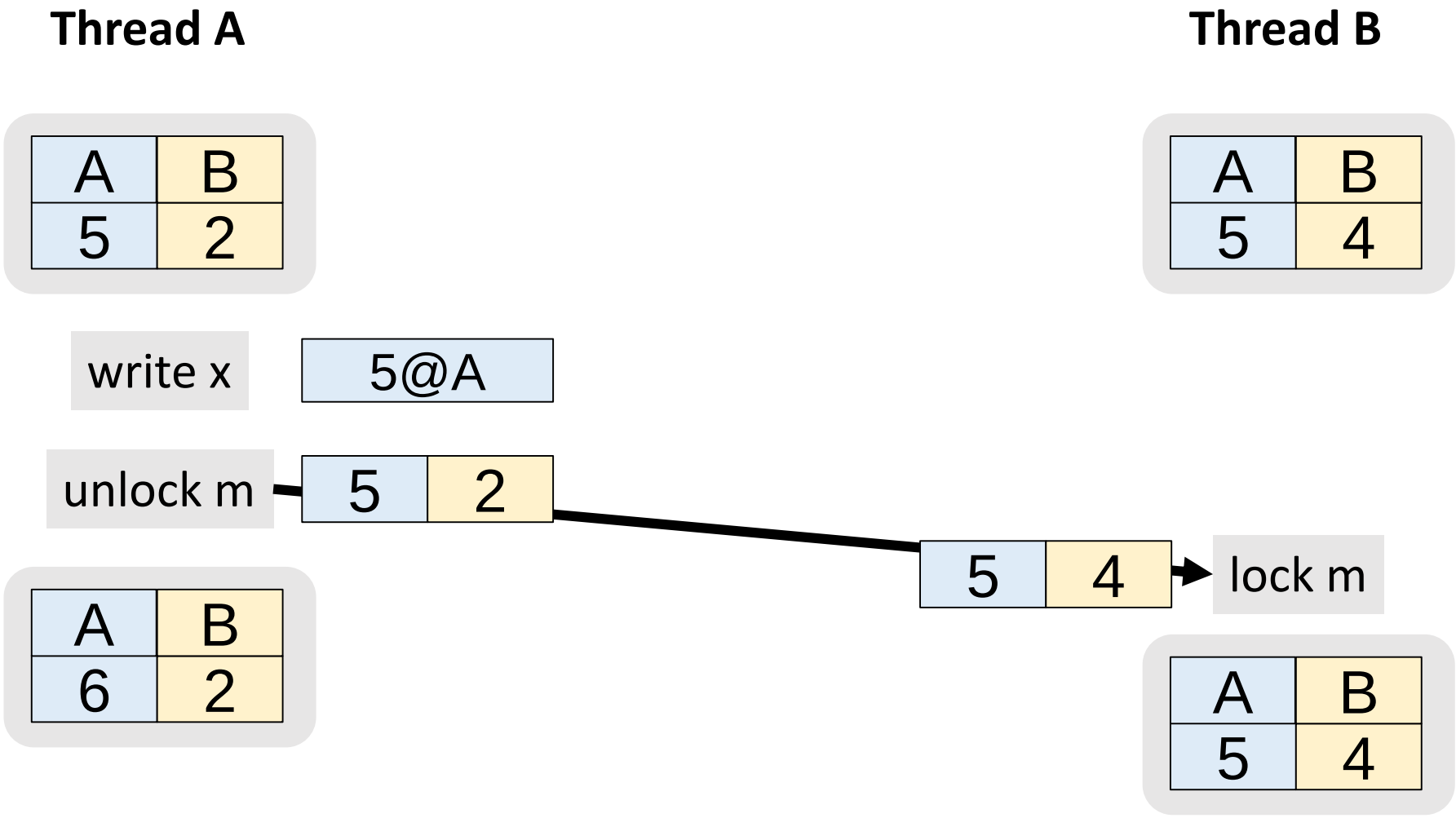
write x | 5@A

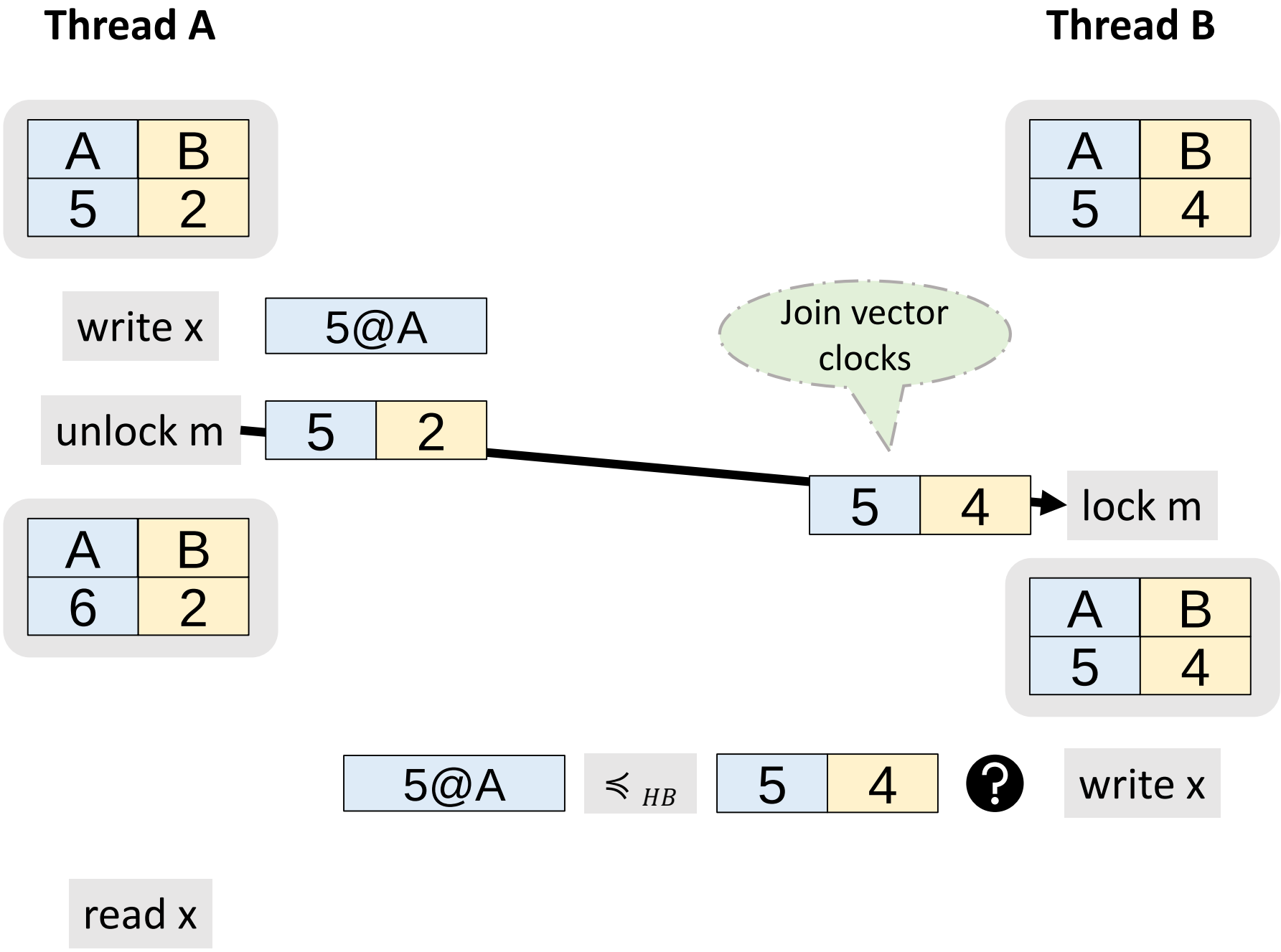
unlock m | 5 | 2

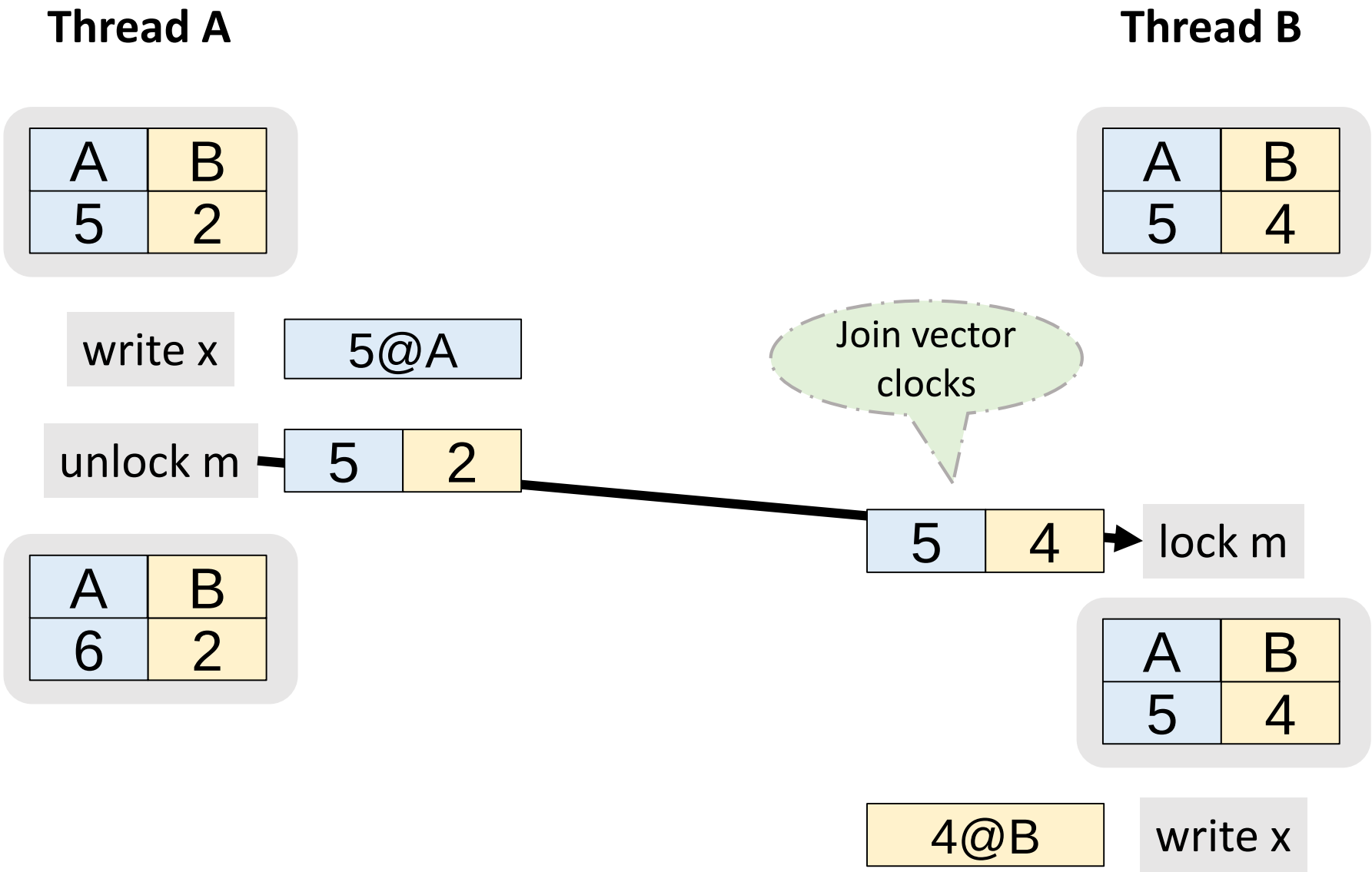
A	B
6	2

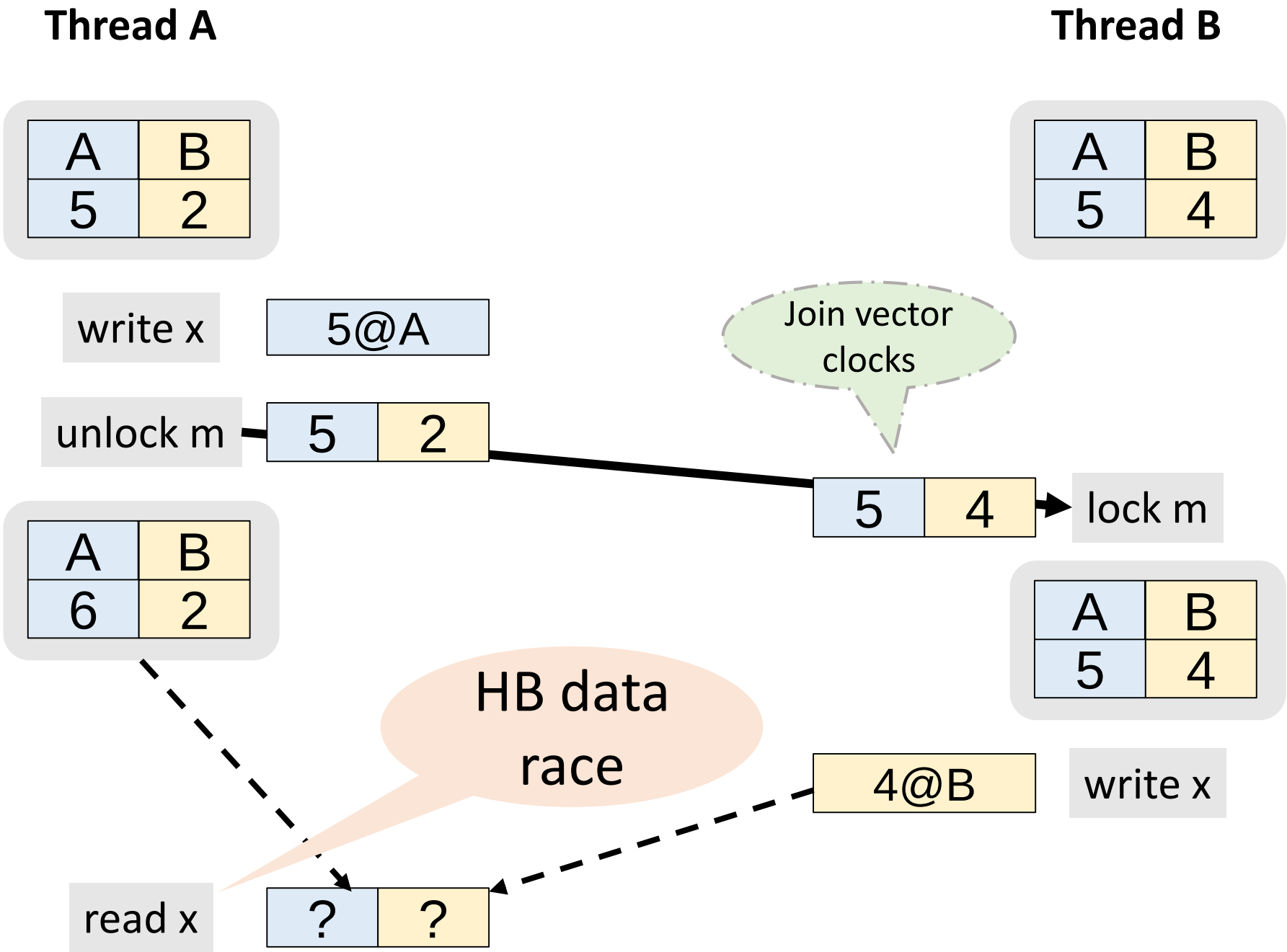
Thread B

A	B
3	4

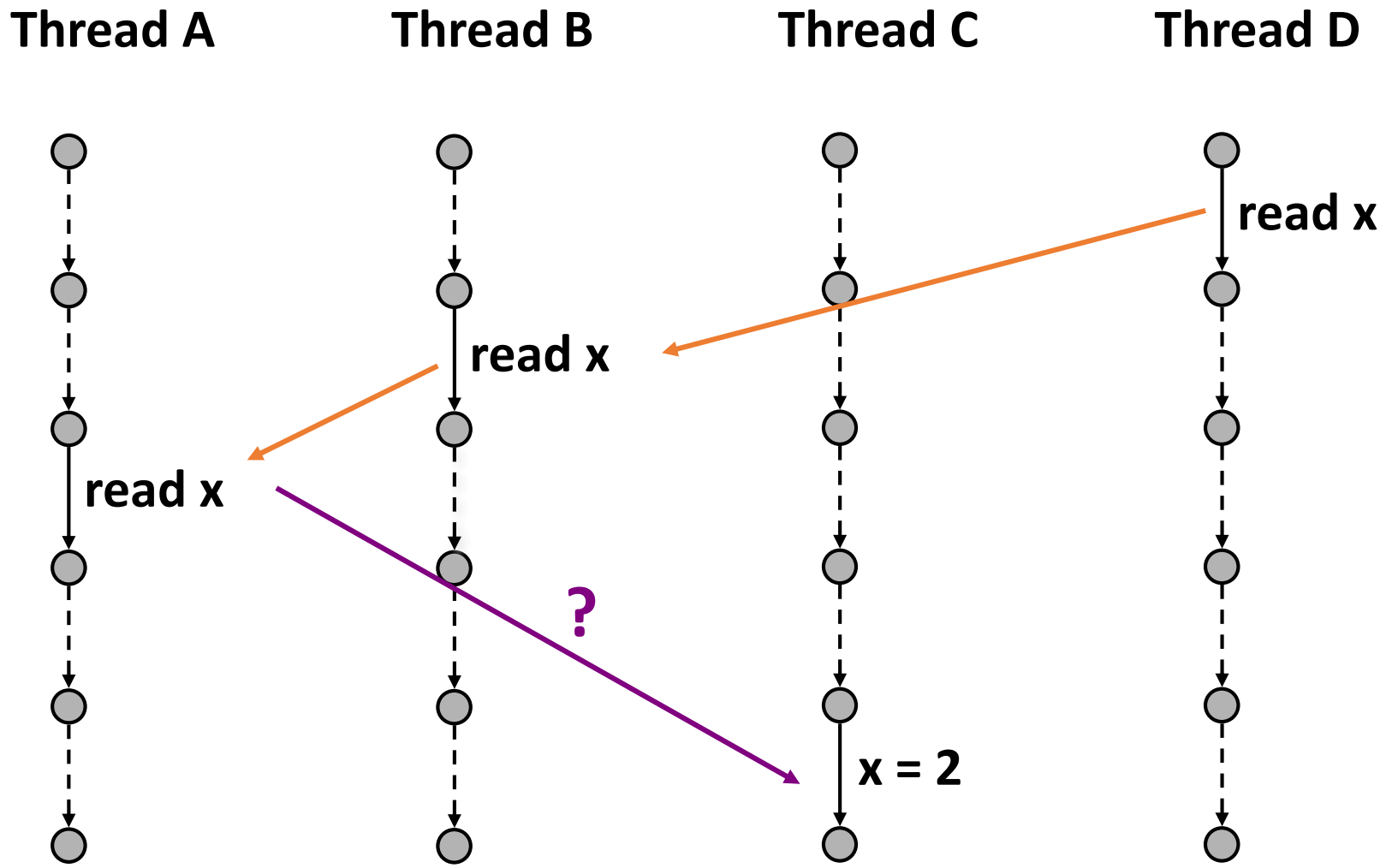

Thread A
Thread B
A
B
5
2
A
B
5
4
write x
5@A
unlock m
5
2
5
4
lock m
A
B
6
2
A
B
5
4


Thread A
Thread B
A
B
5
2
A
B
5
4
write x
5@A
Join vector clocks
unlock m
5
2
5
4
lock m
A
B
6
2
A
B
5
4
5@A
$\preccurlyeq_{HB}$
5
4
write x
read x


Thread A
Thread B
A
B
5
2
A
B
5
4
write x
5@A
Join vector clocks
unlock m
5
2
5
4
lock m
A
B
6
2
A
B
5
4
4@B
write x

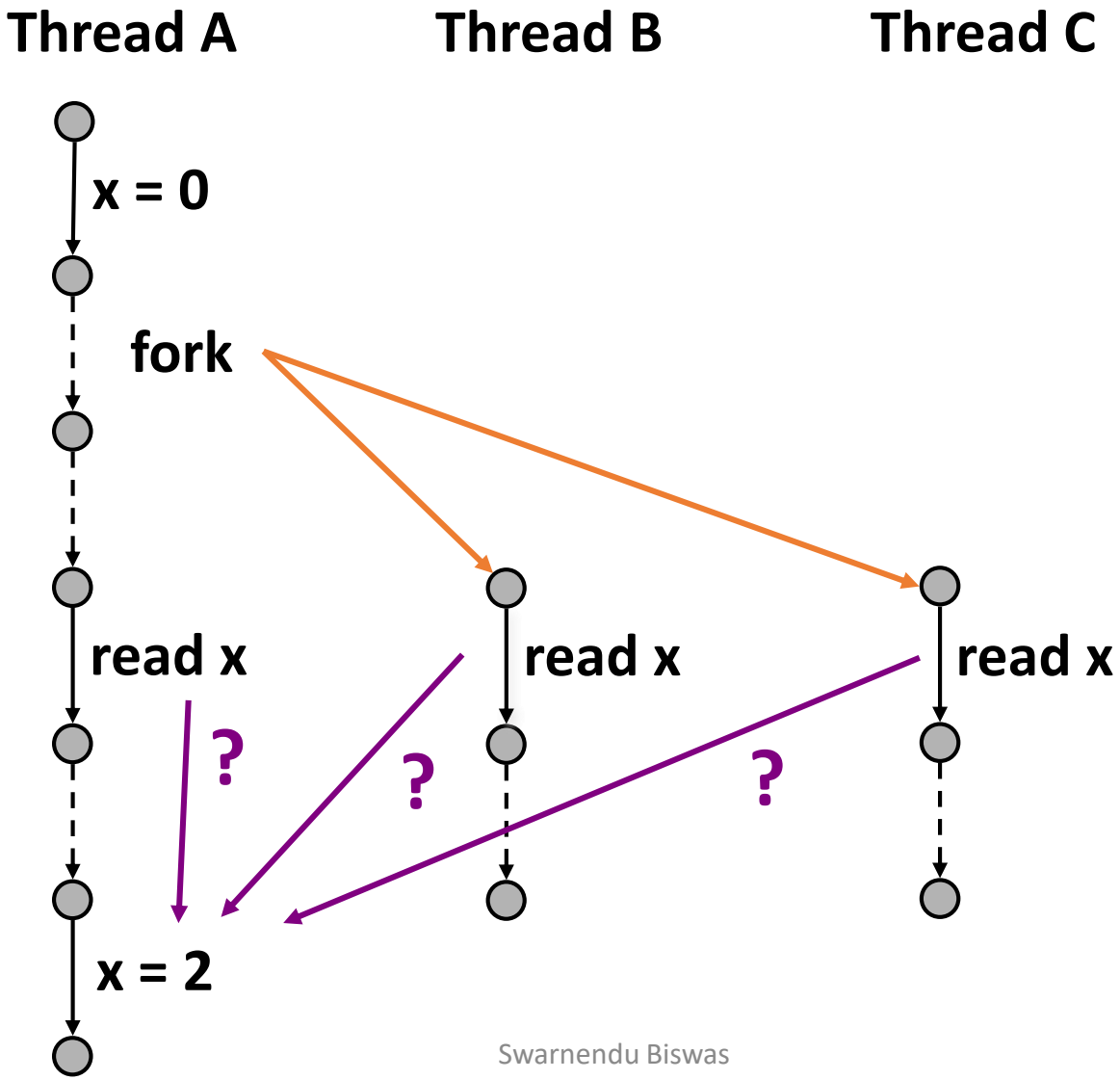

Thread A
Thread B
A
B
5
2
A
B
5
4
write x
5@A
Join vector clocks
unlock m
5
2
5
4
lock m
A
B
6
2
A
B
5
4
HB data race
4@B
write x
read x
?
?

Read-Write Data Races -- Ordered Reads



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

Read-Write Data Races -- Unordered Reads



States in FastTrack

$$C : Tid \rightarrow VC$$

$$L : Lock \rightarrow VC$$

$$W : Var \rightarrow Epoch$$

$$R : Var \rightarrow Epoch \cup VC$$

Read Share

$$\frac{R_x = c@u, \quad W_x \preccurlyeq C_t \quad V = \bot_V[t := C_t(t), u := c] \quad R' = R[x := V]}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

Read Same Epoch

$$\frac{R_x = E(t)}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R, W)}$$

Read Shared

$$\frac{R_x \in VC, \quad W_x \preccurlyeq C_t \quad R' = R[x := R_x[t := C_t(t)]}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

Read Exclusive

$$\frac{R_x \in Epoch, R_x \preccurlyeq C_t \quad W_x \preccurlyeq C_t, \quad R' = R[x := E(t)]}{(C, L, R, W) \Rightarrow^{rd(t,x)} (C, L, R', W)}$$

Write Same Epoch

$$\frac{W_x = E(t)}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W)}$$

Write Exclusive

$$\frac{\begin{array}{c} R_x \in Epoch,\ R_x \preccurlyeq C_t \\ W_x \preccurlyeq C_t,\ \ W' = W[x := E(t)] \end{array}}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R, W')}$$

Write Shared

$$\frac{\begin{array}{c} R_x \in VC,\ R_x \sqsubseteq C_t \\ W_x \preccurlyeq C_t,\ \ W' = W[x := E(t)] \\ R' = R[x := \bot_e] \end{array}}{(C, L, R, W) \Rightarrow^{wr(t,x)} (C, L, R', W')}$$

Acquire

$$\frac{C' = C[t \coloneqq C_t \sqcup L_m]}{(C, L, R, W) \Rightarrow {}^{acq(t,\,m)}(C', L, R, W)}$$

Fork

$$\frac{C' = C[u \coloneqq C_u \sqcup C_t], t = inc_t(C_t)}{(C, L, R, W) \Rightarrow {}^{fork(t,\,u)}(C', L, R, W)}$$

Release

$$\frac{\begin{array}{c} L' = L[m \coloneqq C_t] \\ C' = C[t \coloneqq inc_t(C_t)] \end{array}}{(C, L, R, W) \Rightarrow {}^{rel(t,\,m)}(C', L', R, W)}$$

Join

$$\frac{C' = C[t \coloneqq C_t \sqcup C_u], u = inc_u(C_u)}{(C, L, R, W) \Rightarrow {}^{join(t,\,u)}(C', L, R, W)}$$

Data Race Detection Techniques

Lockset analysis	Imprecise, reports many false positives
	Assumes consistent locking discipline
Happens-before analysis	Dynamically sound and precise
	Not scalable, incurs space overhead
	Coverage limited to observed executions
	Correctness depends on exact knowledge of synchronization

sound – no missed races
precise – no false races

Performance of Lockset and HB Algorithms

- FastTrack's slowdowns are still ~4-8X
- Intel Thread Checker has 200X overhead
- Google's ThreadSanitizer (now part of LLVM) incurs around ~5-15X overhead

- Large overheads impact the thread interleaving pattern

Looking Forward!

Can we run data race detectors in **production environments**?

Can we **catch** data races **as it is about to happen**?

Existing Approaches for Data Race Detection on Production Runs

- Happens-before-based sampling approaches
  - E.g., LiteRace[1], Pacer[2]
  - Overheads are **still too high** for a reasonable sampling rate
    - Pacer with 3% sampling rate incurs 86% overhead!!!

1. D. Marino et al. LiteRace: Effective Sampling for Lightweight Data-Race Detection. PLDI 2009.
2. M. Bond et al. Pacer: Proportional Detection of Data Races. PLDI 2010.

Question: Is there a Data Race?

```
Object X = null;
volatile boolean done= false;
```

Thread T1

```
X = new Object();
done = true;
```

Thread T2

```
while (!done) {}
X.compute();
```

Question: Is there a Data Race?

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

Thread T1

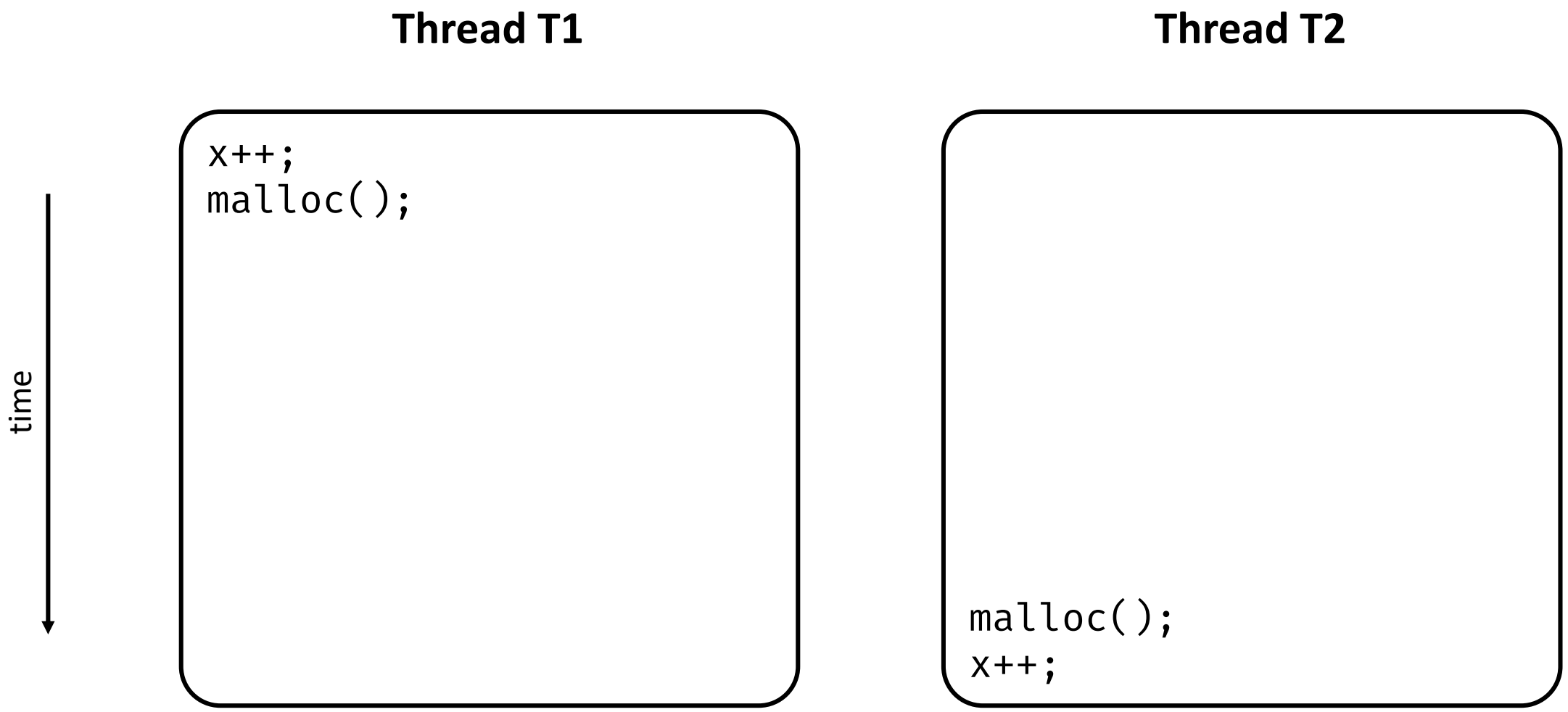
```
data = …;
synchronized(m) {
  flag = true;
}
```

Thread T2

```
boolean f;
synchronized(m) {
  f = flag;
}
if (f) {
  … = data;
}
```

time

Question: Is there a Data Race?



Question: Is there a Data Race?

Thread T1

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

Thread T2

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

time

Collision Analysis

Basic idea: Make two conflicting accesses happen at the **same time**

(1) Pause one thread just before accessing a memory location x
(2) Catch other threads that make conflicting accesses to x in the meantime

Implementation: Either software or hardware (more efficient but has other limitations)

Instrument Racy Accesses

- The figure shows one potential race pair

```
avrora.sim.radio.Medium:
access$302() byte offset 0
```

```
avrora.sim.radio.Medium:
access$402() byte offset 2
```

Try to Collide Racy Accesses

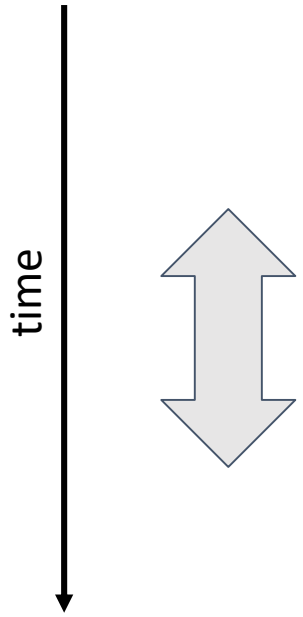
- Block thread for some time

`avrora.sim.radio.Medium: access$302() byte offset 0`

`avrora.sim.radio.Medium: access$402() byte offset 2`

Dynamic instance 992

Dynamic instance 993



Collision is Successful

avrora.sim.radio.Medium: access$302() byte offset 0

avrora.sim.radio.Medium: access$402() byte offset 2

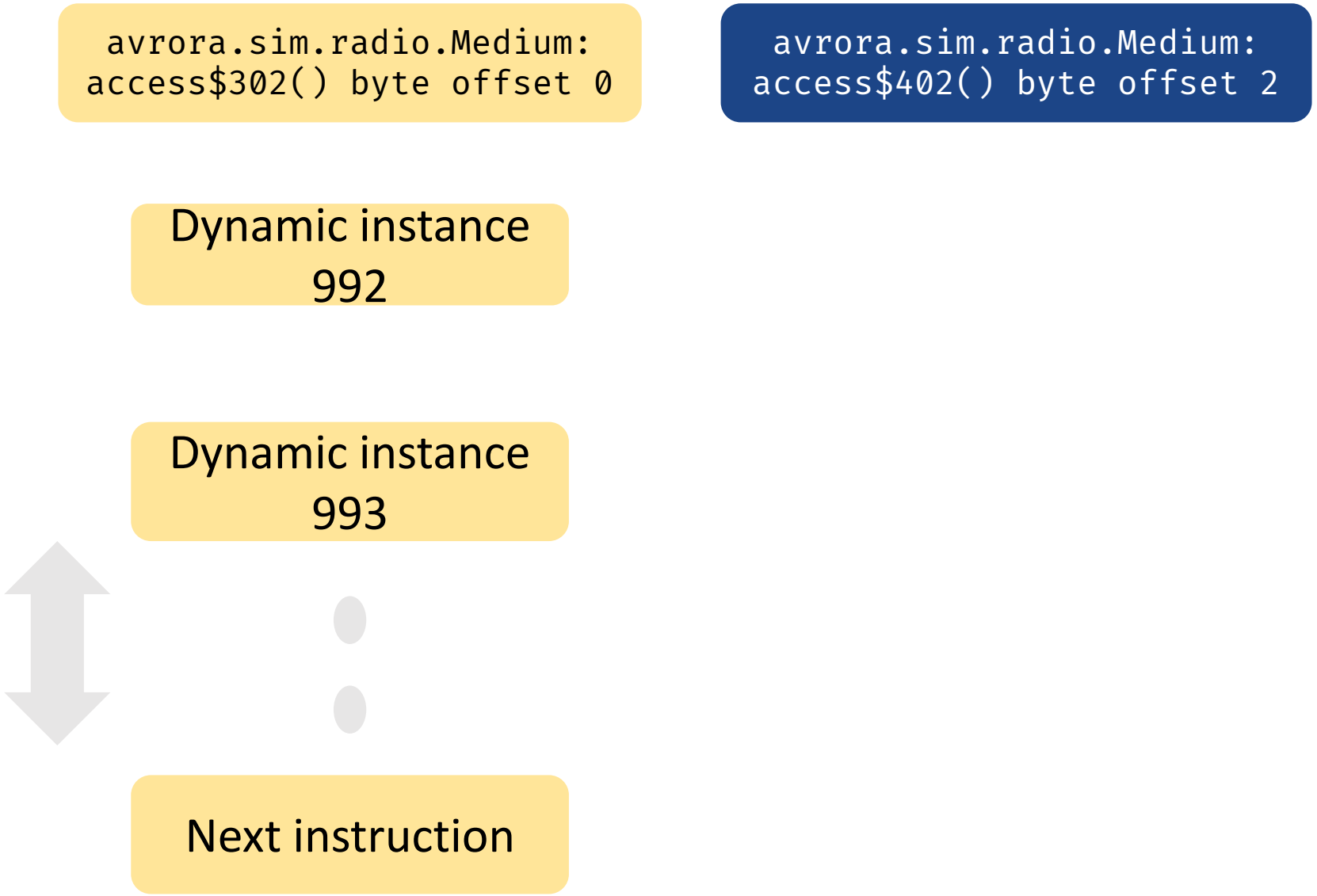
Dynamic instance 992

Dynamic instance 993

Dynamic instance 215

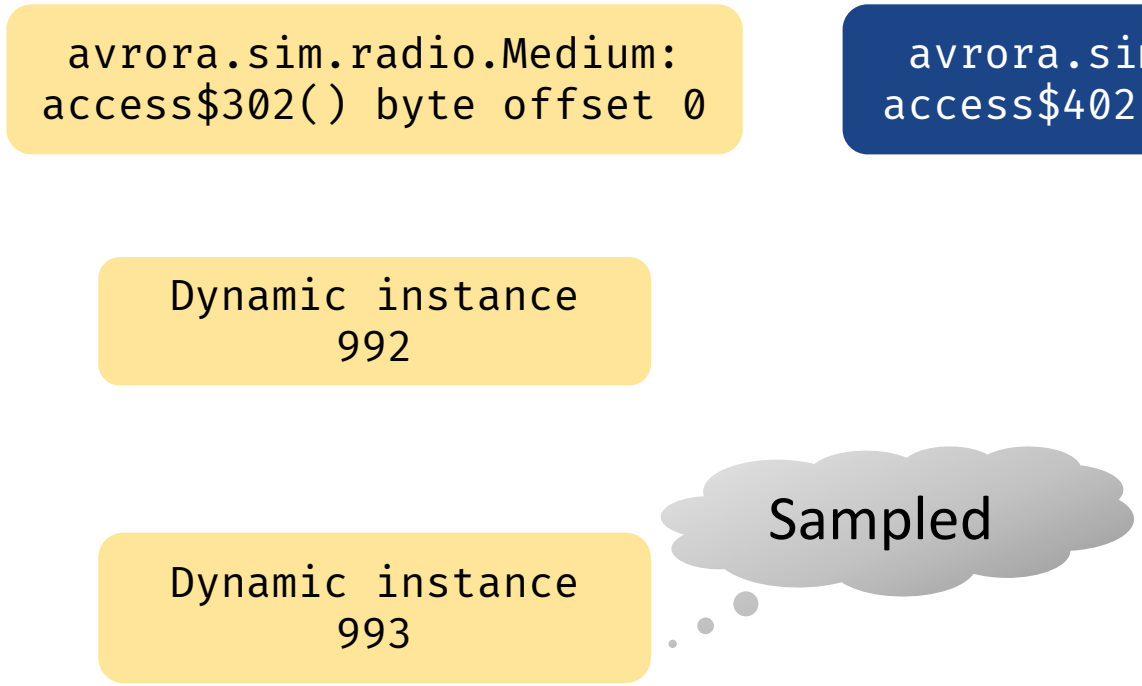
Collision is Unsuccessful

- Thread unblocks, resets the analysis state, and continues execution



Randomly Sample Racy Accesses

- Use frequency of samples taken
  and
  Compute overhead introduced by waiting



Advantages of Collision Analysis

- No inference ➔ oblivious to synchronization patterns
- Can potentially detect data races that are hidden by spurious HB relations
- Race coverage is sensitive to perturbation and delay
  - Prior studies indicate that data races often happen close in time
- Low memory overhead compared to maintaining vector clocks

DataCollider: Hardware Implementation of Collision Analysis

- Uses hardware debug registers to monitor access locations
- x86 has four usable debug registers (DR0…DR7)
  - Two are aliases are two are for control

- Write an address to a debug register, set the control flags
- Generates a trap when some other thread tries to access the address
  - Good performance, hardware does all the work

J. Erickson et al. Effective Data-Race Detection for the Kernel. OSDI 2009.

Challenges with DataCollider

- Delays at several shared-memory accesses would still introduce large overheads

- **Sampling**: Only execute slow path when certain conditions are met
  - Prioritize **cold** code regions
  - Sample based on **allowed tolerable overhead**

```
runtime_instrumentation() {
  numCounter++;
  if (numCounter % 10 == 0) {
    do_analysis();
  } else {
    // Do nothing
  }
}
```

Challenges with DataCollider

- # of threads ≫ 4 (i.e., # debug registers)
  - Not very effective analysis

- Cost of setting/clearing debug registers may increase with increase in core count

Model Checking for Race Conditions

- Develop a system model
- Explore the model to check for reachable error states
  - Detailed model – more compute-intensive
  - Simpler model – needs to contain enough information of interest
- Model checking of concurrent programs is a challenge
  - Very large state space given all possible thread interleavings
  - Sound as long as the analysis terminates

J. Huang et al. Maximal Sound Predictive Race Detection with Control Flow Abstraction. PLDI 2014.

Current Research on Data Race Detection

Last five years

Not a lot of new ideas in trying to improve performance targeted to production environments

Existing tools usually combine several ideas like static race detection, lockset analysis and HB analysis

More focus on trying to **improve race detection coverage**

Many relationships weaker than HB (like CP, WCP, and DC have been proposed)

Still remains one of the **most actively-researched** topics in PL

java.lang.StringBuffer

Is it thread-safe?

```
public final class StringBuffer {
  public synchronized StringBuffer append(StringBuffer sb) {
    int len = sb.length();
    sb.getChars(0, len, value, count);
    ...
  }
  public synchronized int length() { ... }
  public synchronized void getChars(...) { ... }
  ...
}
```

Are there Data Races?

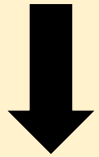
Is it thread-safe?

```
class Set {
  final Vector elems = new Vector();
  void add(Object x) {
    if (!elems.contains(x)) {
      elems.add(x);
    }
  }
}

class Vector {
  synchronized void add(Object o) { ... }
  synchronized boolean contains(Object o) { ... }
}
```

Are there Data Races?

```
class Set {
  final Vector elems = new Vector();
  void add(Object x) {
    if (!elems.contains(x)) {
      elems.add(x);
    }
  }
}

class Vector {
  synchronized void add(Object o) { ... }
  synchronized boolean contains(Object o) { ... }
}
```

Free of data races



But can still violate atomicity!

Data Race Freedom (DRF)

Data race freedom is **neither necessary nor sufficient** to ensure absence of concurrency bugs

Atomicity is a more fundamental non-interference property

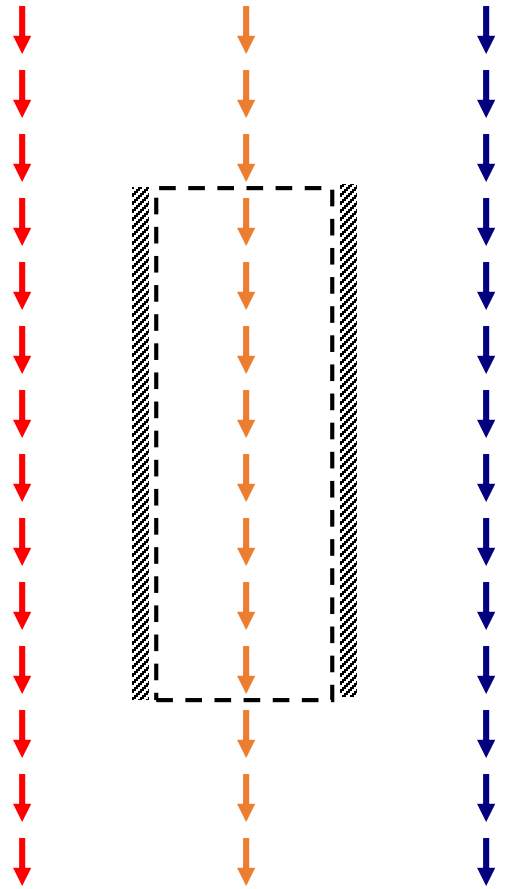
Detecting Atomicity Violations

Atomicity Property

Atomicity	Synonymous with **serializability** for programming language semantics
	Program execution must be equivalent to a serial execution of atomic regions
	Atomic region's execution appears not to be interleaved with other concurrent threads

Multithreaded Program Execution

- Maximal non-interference property
- Enables sequential reasoning



Why Study Atomicity Violation Detection?

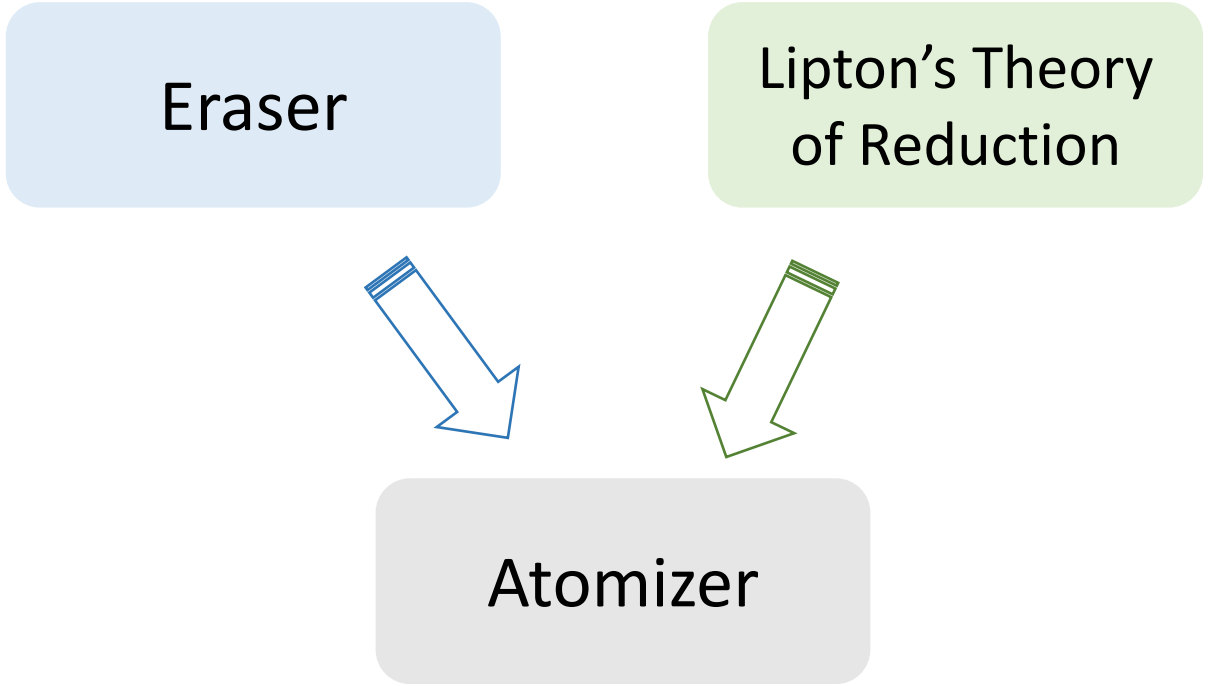
Violation of atomicity is the **most common** (almost two-thirds) type of all non-deadlock concurrency bugs

Atomizer

- **Idea**: Given operations from a region marked “atomic”, check whether we can always guarantee that the instructions can be shuffled into an uninterrupted sequence by local, pairwise swaps
- Warn if the reordering attempts fail with the given set of operations

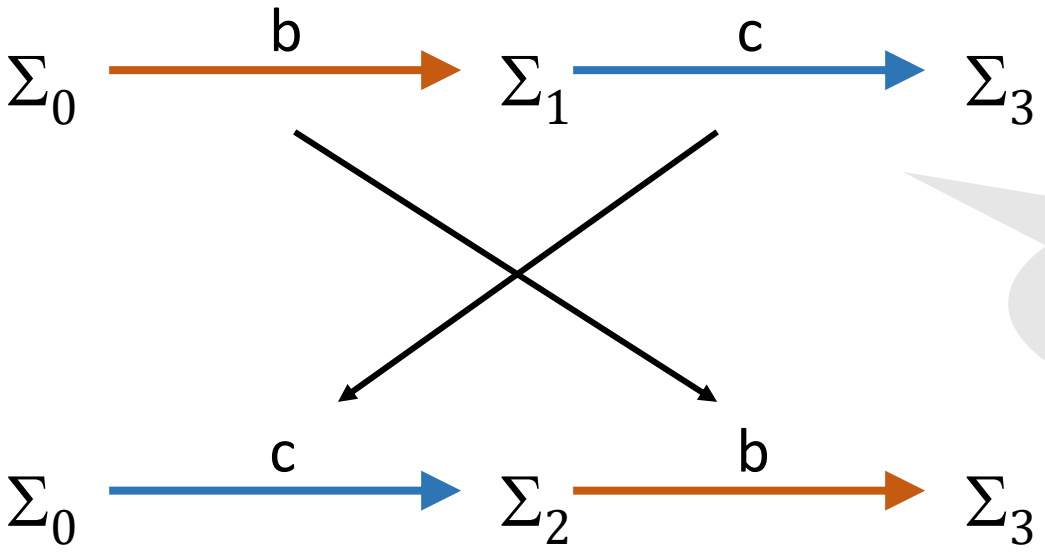
C. Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. POPL, 2004.

Atomizer



C. Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. POPL, 2004.

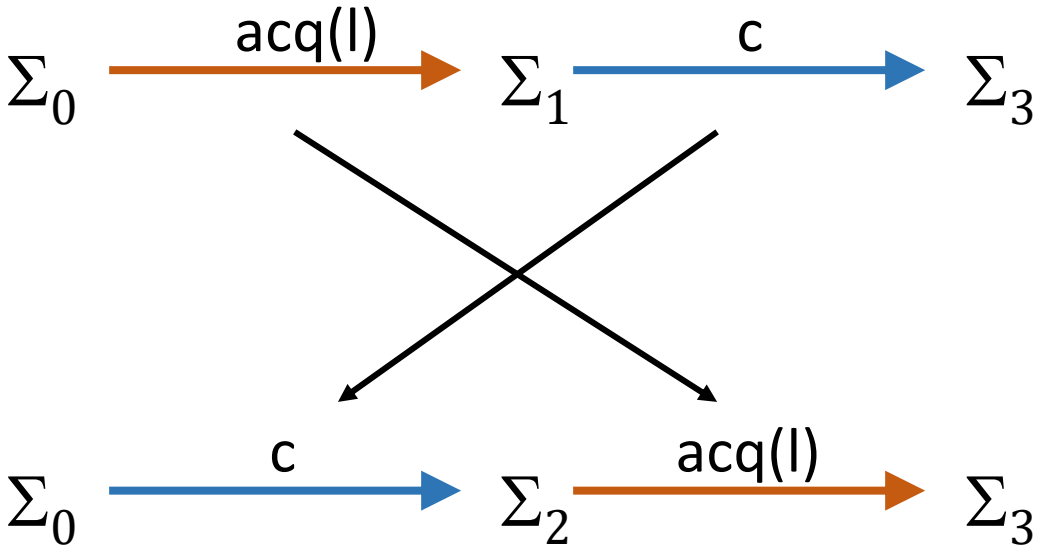
Commuting Actions: Right Mover



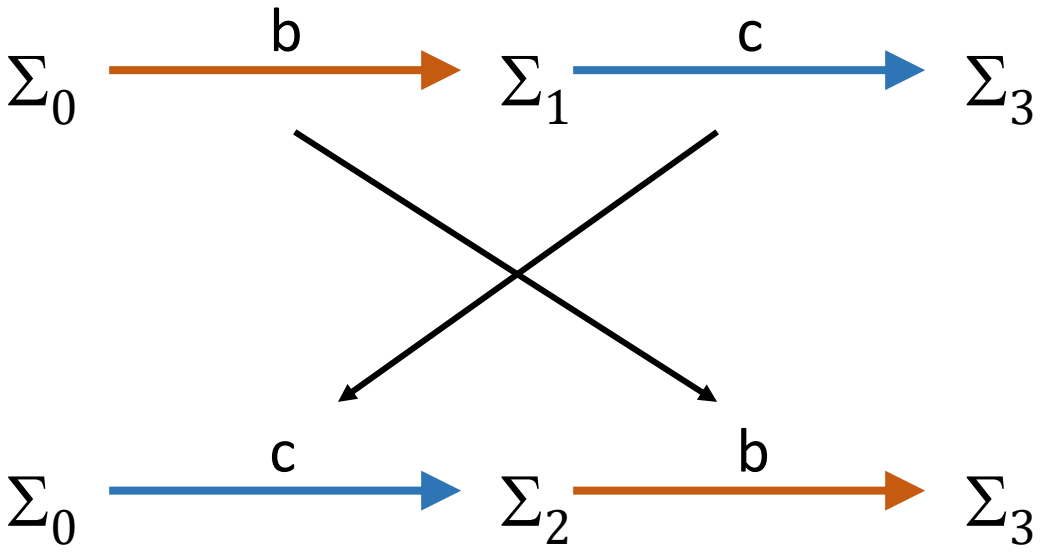
b and c are operations from concurrent threads

b is right mover if swapping the operations do not change the resulting state

Commuting Actions: Right Mover

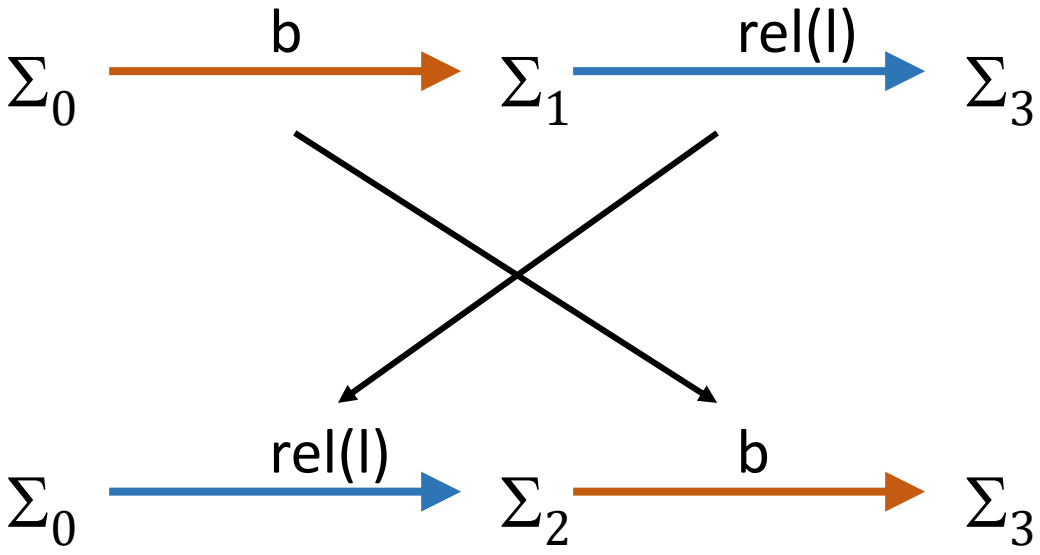


Commuting Actions: Left Mover



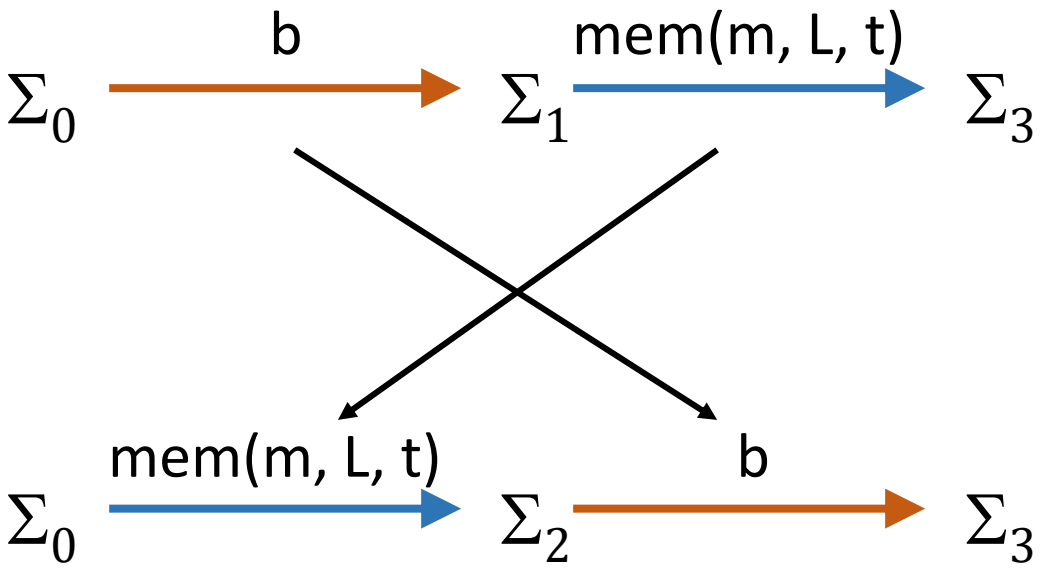
c is left mover if swapping the operations do not change the resulting state

Commuting Actions: Left Mover



lock release

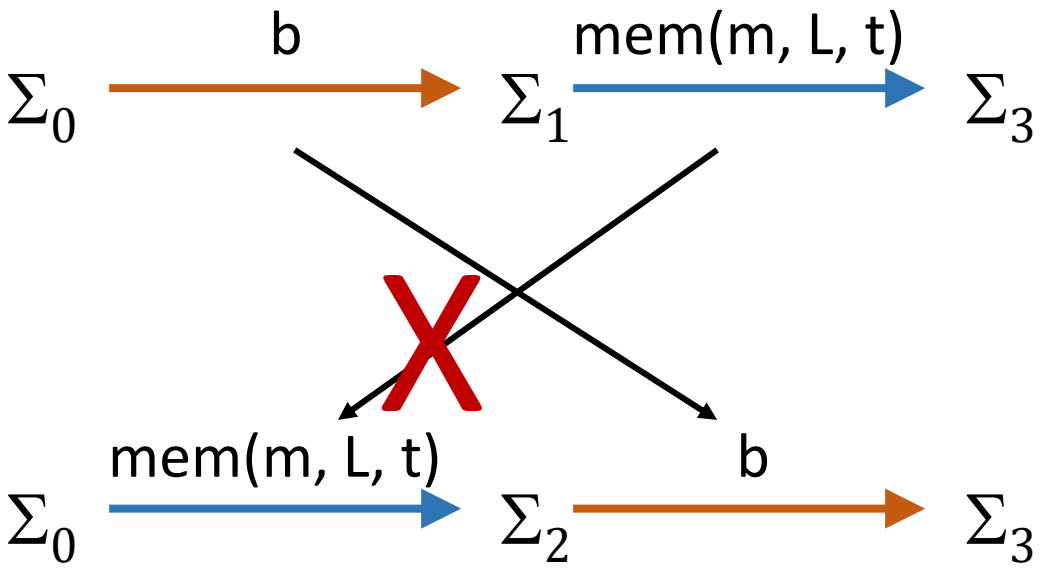
Commuting Actions: Both Mover



race-free field access

Memory access to m is always protected by lockset L, and thread t holds at least one lock during the access

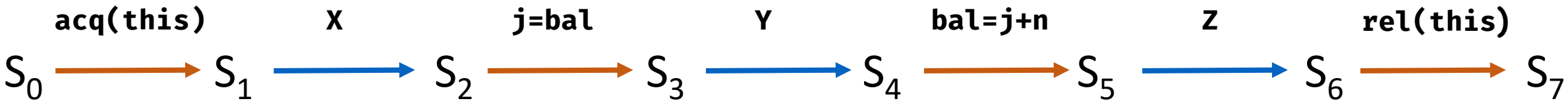
Commuting Actions: Non-Mover



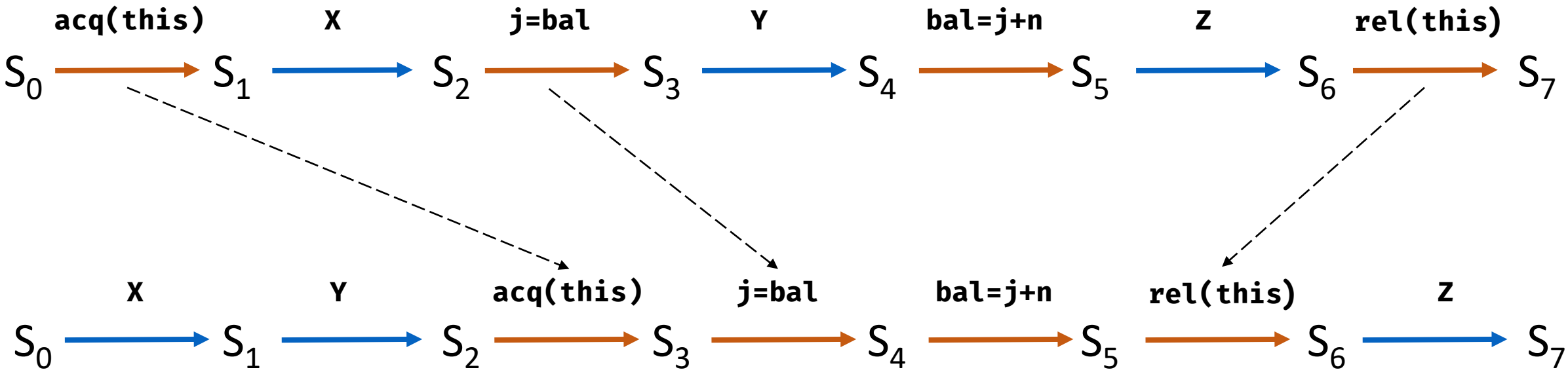
racy field access

Memory access to m is always protected by lockset L, but none of the locks in L is held by thread t during the access

Theory of Reduction [R. Lipton ‘75]

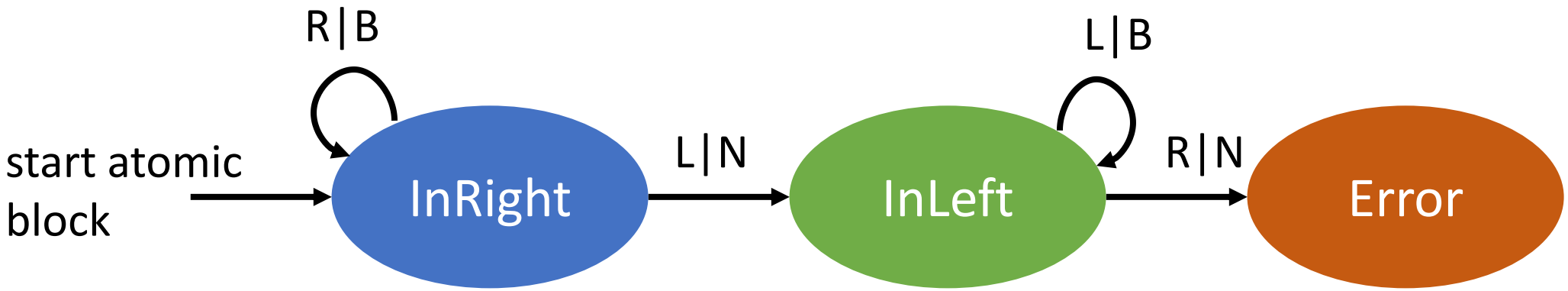


Theory of Reduction [R. Lipton ‘75]

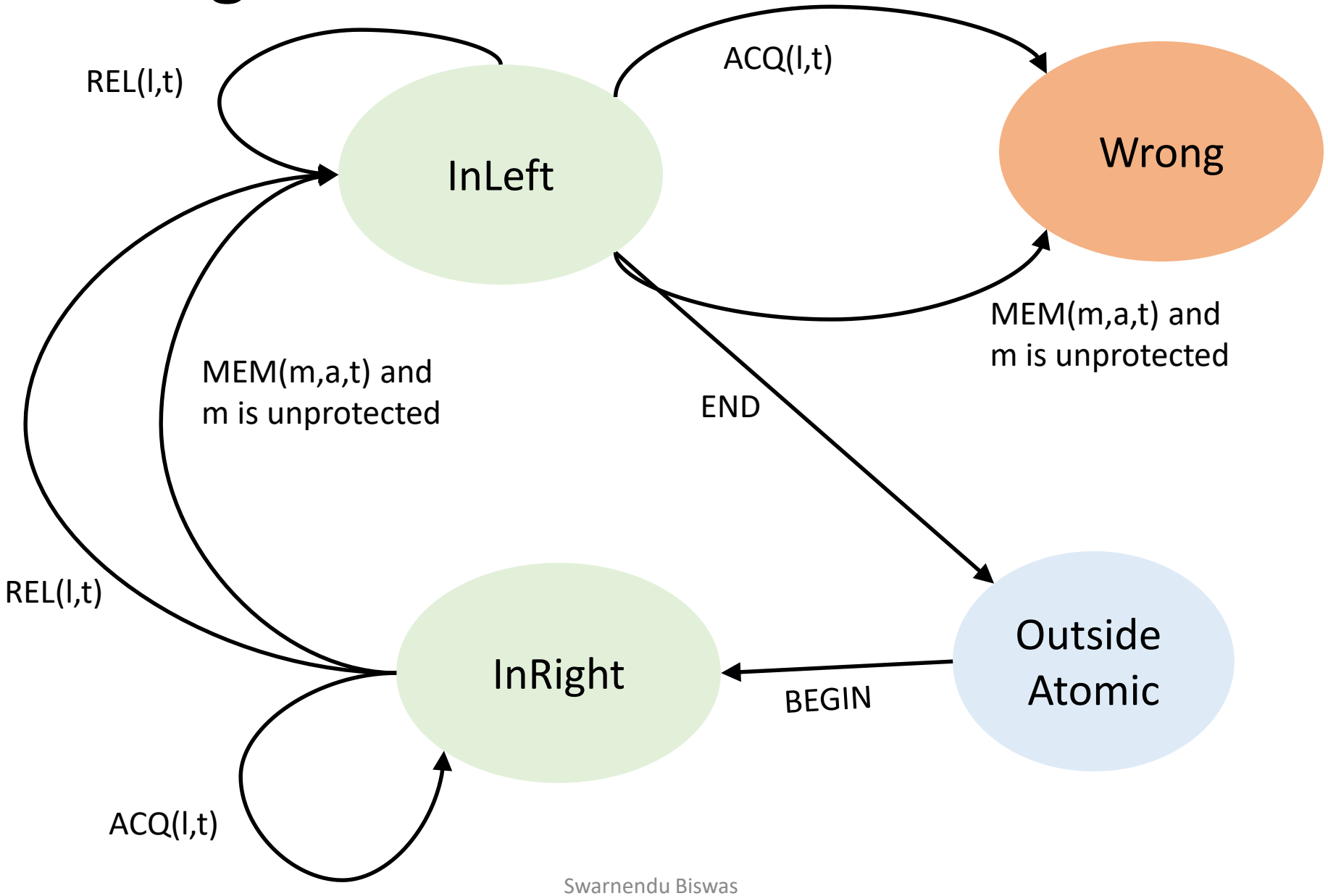


Performing Reduction Dynamically

- Reducible methods `(R|B)*[N](L|B)*`



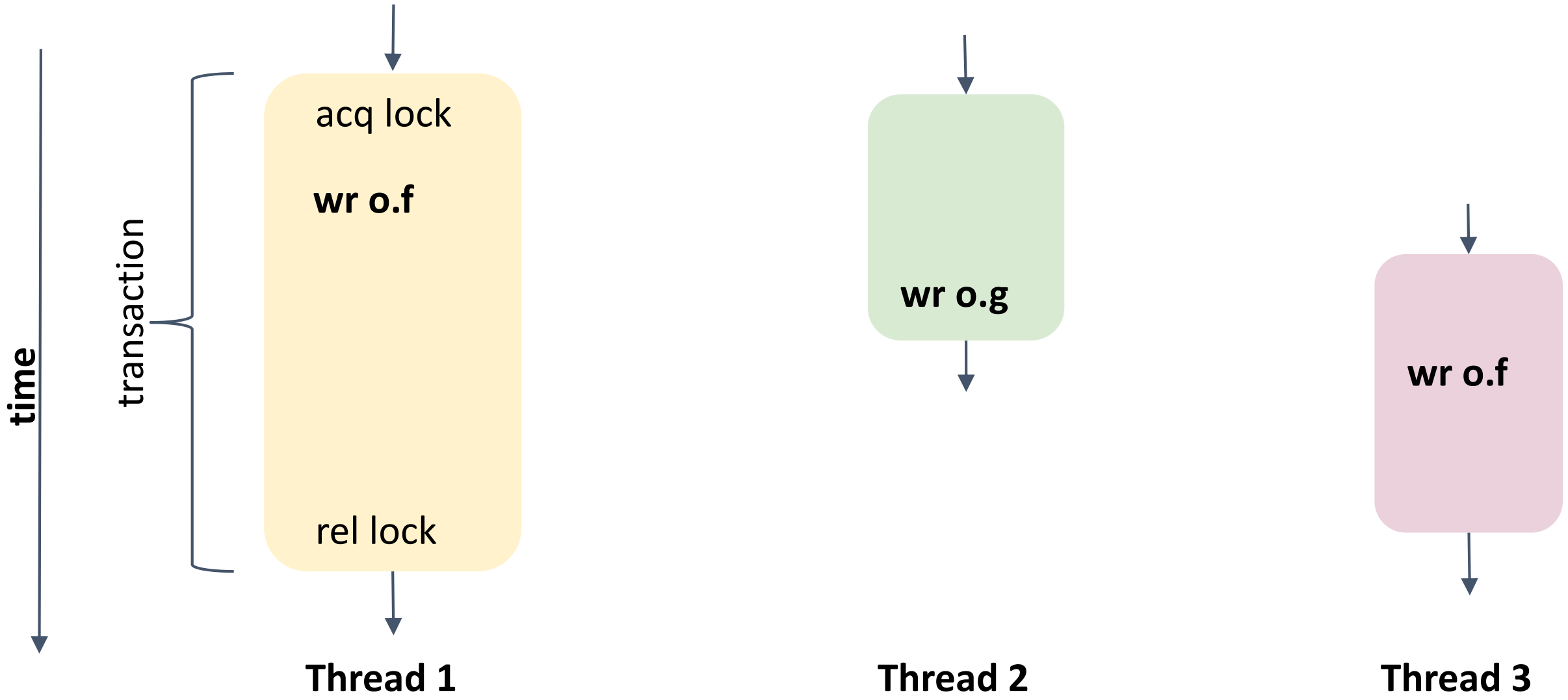
Atomizer Algorithm



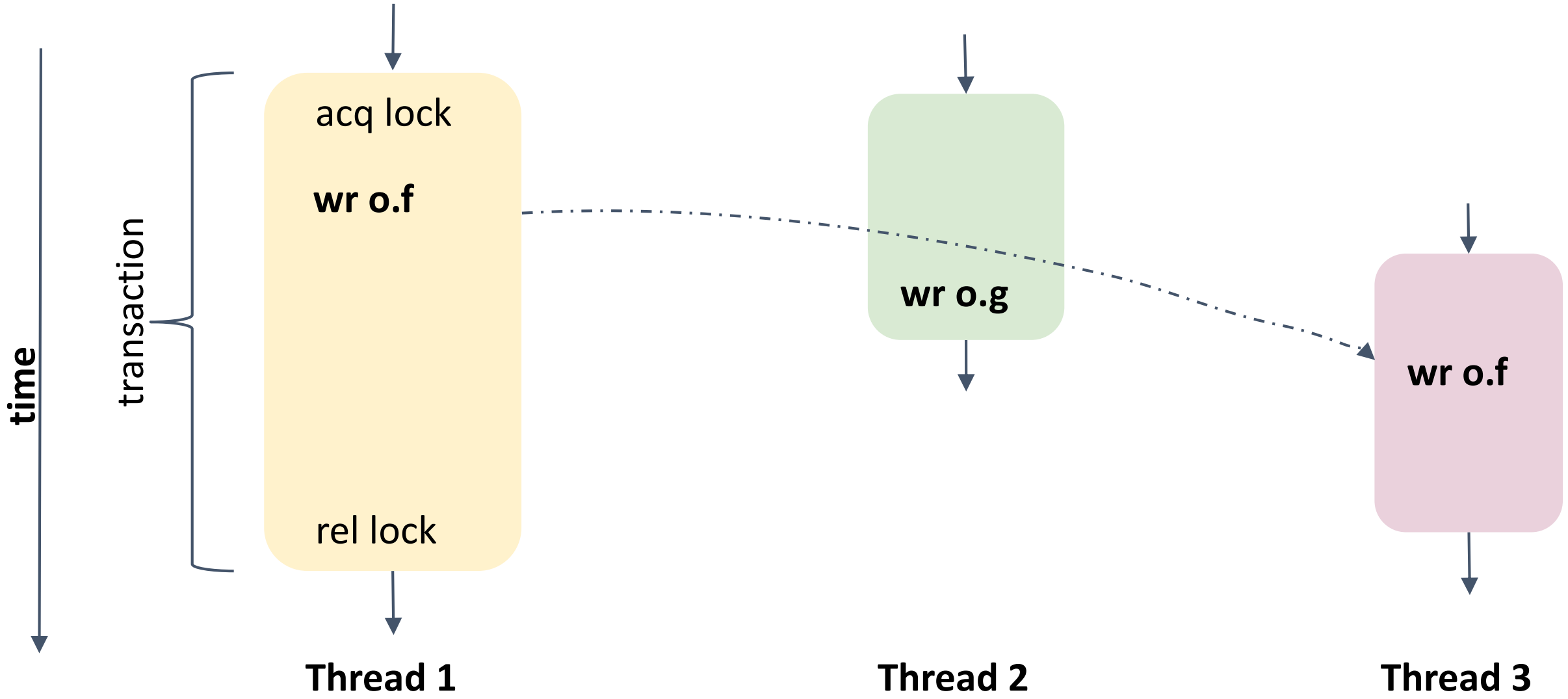
Velodrome: Dynamically Sound and Precise Atomicity Checking

- Tracks HB relations **between** transactions (i.e., atomic regions)
  - A transaction is a dynamic execution of an atomic block
  - **Lifts** HB relations from operations to transactions
- Builds a transactional dependence graph
- Checks for presence of cycles in the graph
  - Depicts violations of conflict serializability

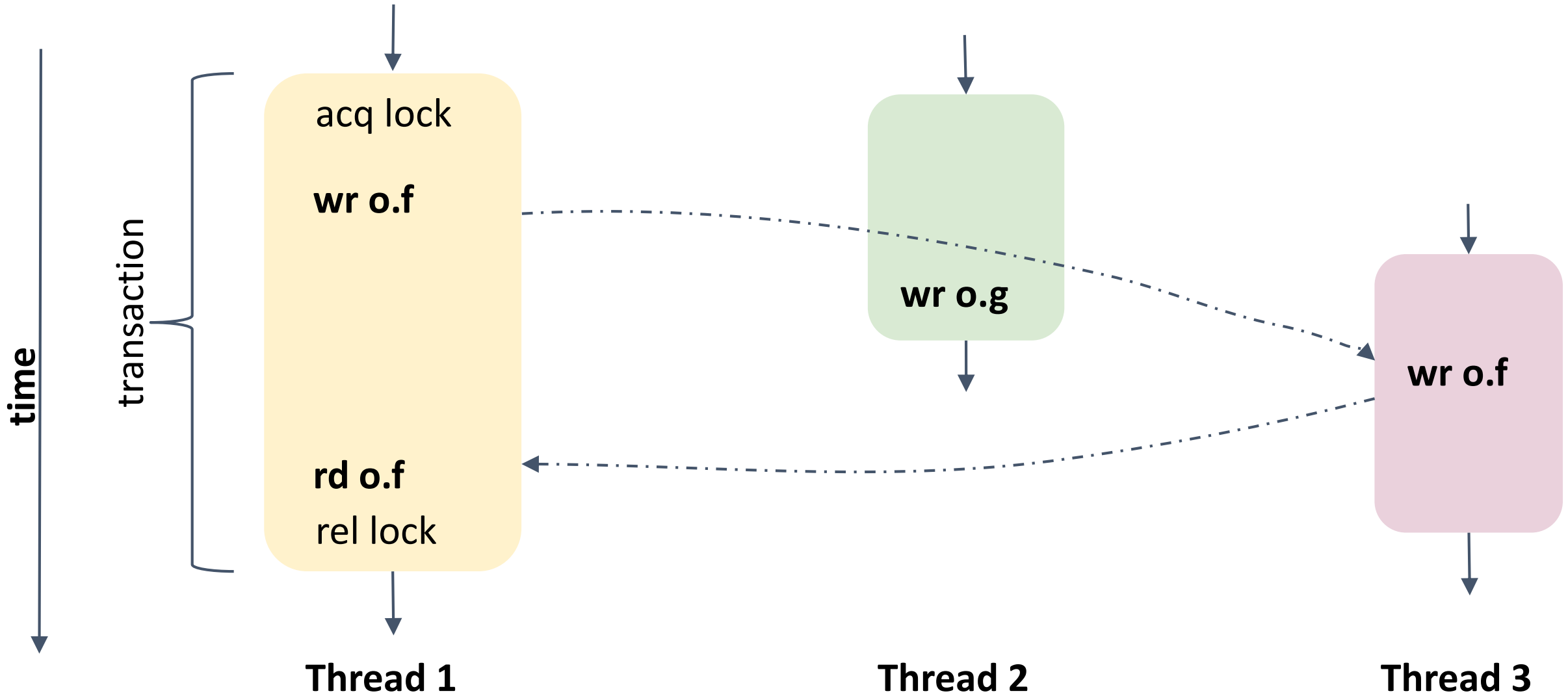
C. Flanagan and S. Freund. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. PLDI, 2008.



Transactional Dependence Graph



Transactional Dependence Graph



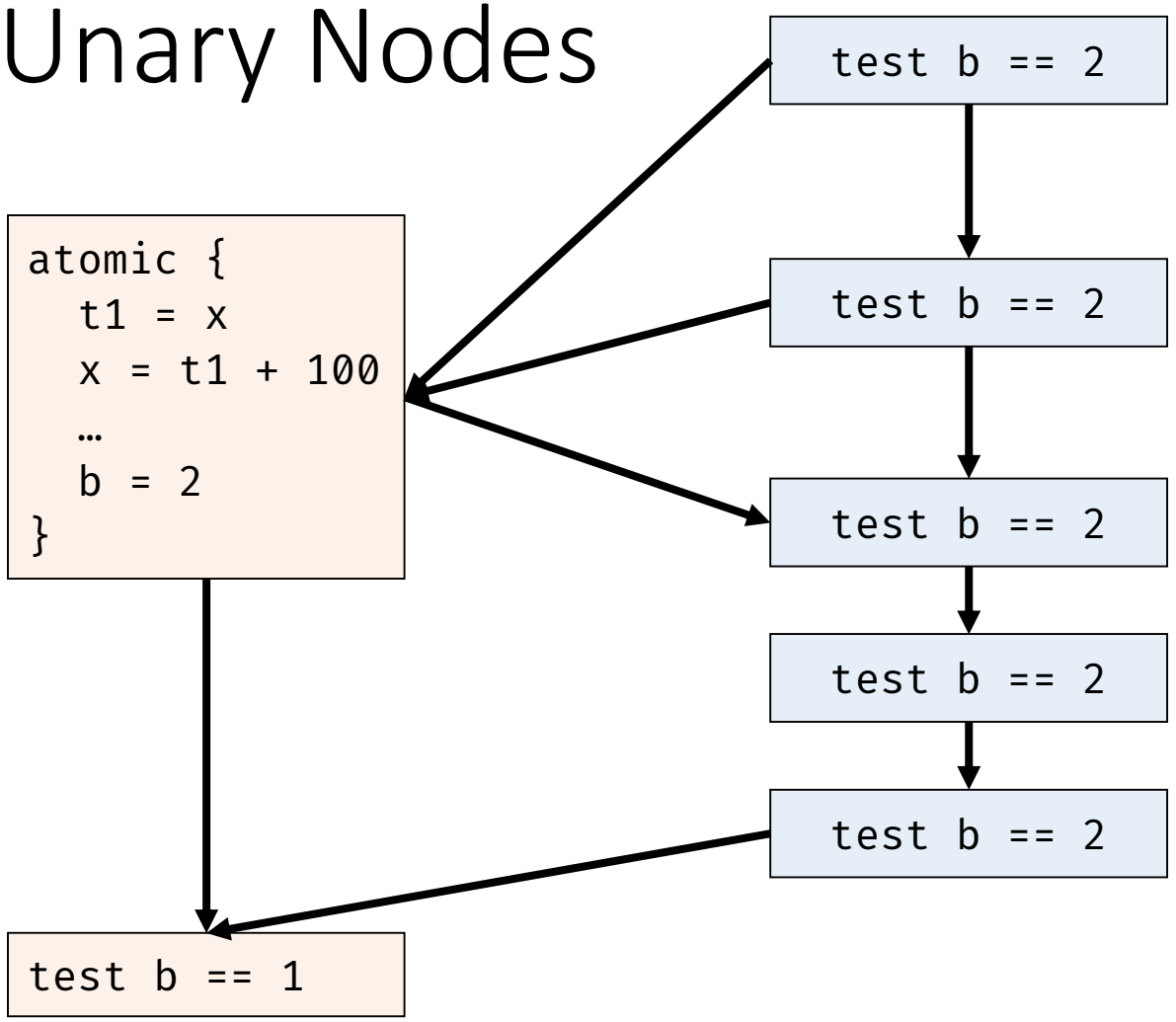
Cycle means Atomicity Violation

Other Challenges in Velodrome

- Transactional HB graph can become **HUGE…**
  - Notion of unary transactions
- Garbage collect completed transactions if they have no IN edges
  - Only the current transaction can create in edges
  - Will never be in a cycle
- Optimize allocation of unary nodes
  - Avoid allocation if they do not have in edges
  - If there is a single in edge, then reuse predecessor node

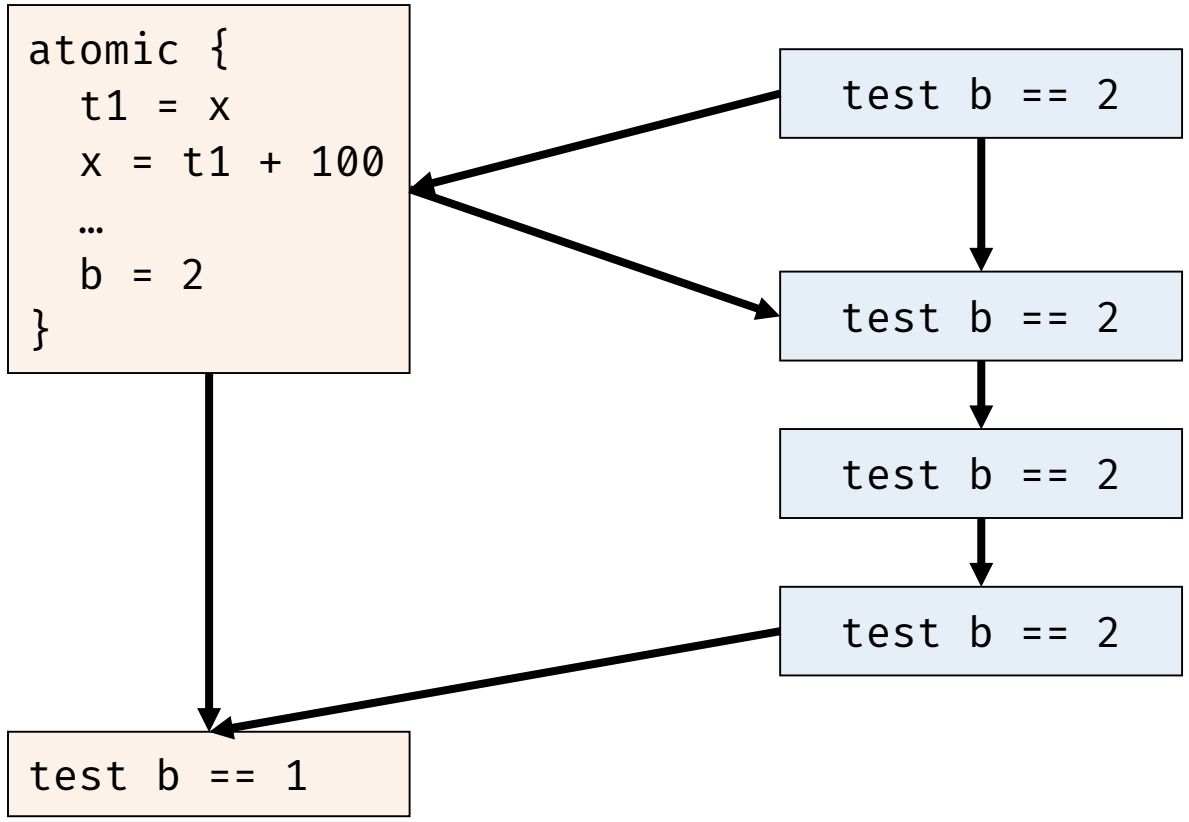
Optimize allocation of Unary Nodes

- Avoid allocation if there are no in edges
  - Will never have in edges
  - Can never participate in a cycle
  - Not even allocated



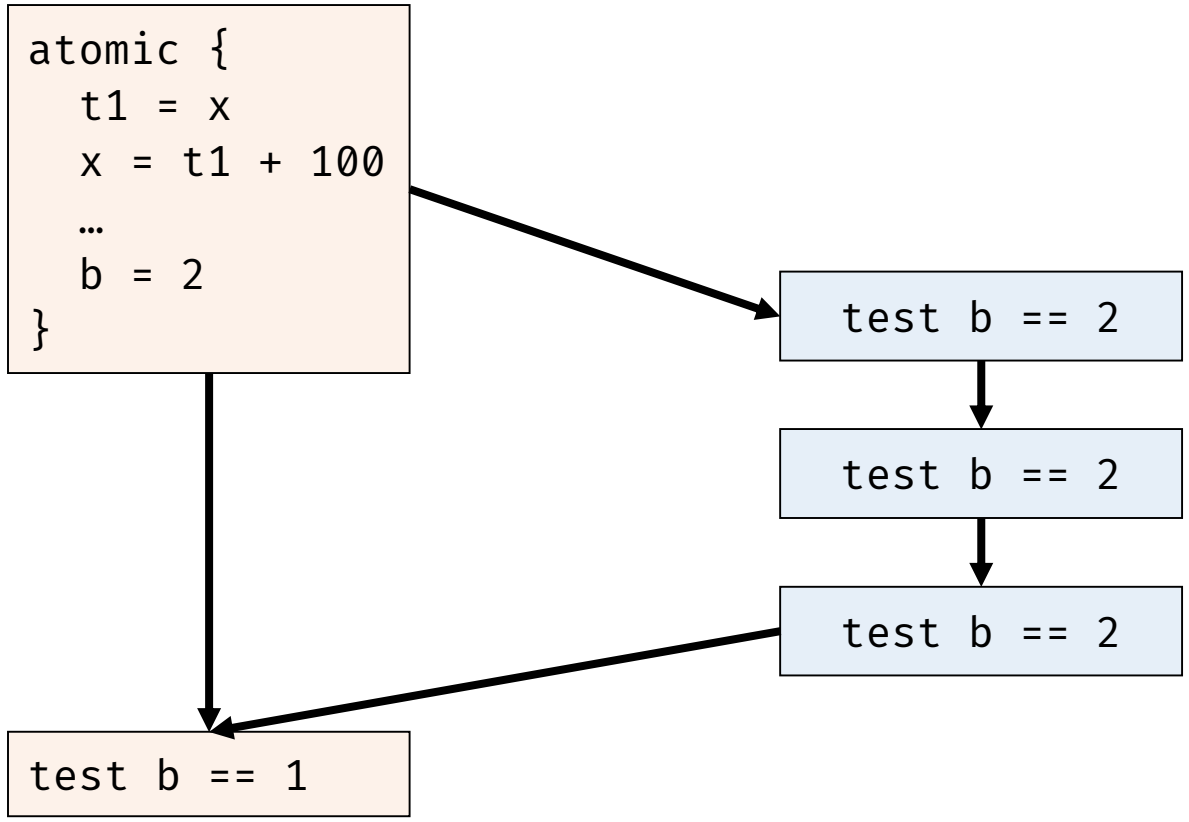
Optimize allocation of Unary Nodes

- Avoid allocation if there are no in edges
  - Will never have in edges
  - Can never participate in a cycle
  - Not even allocated



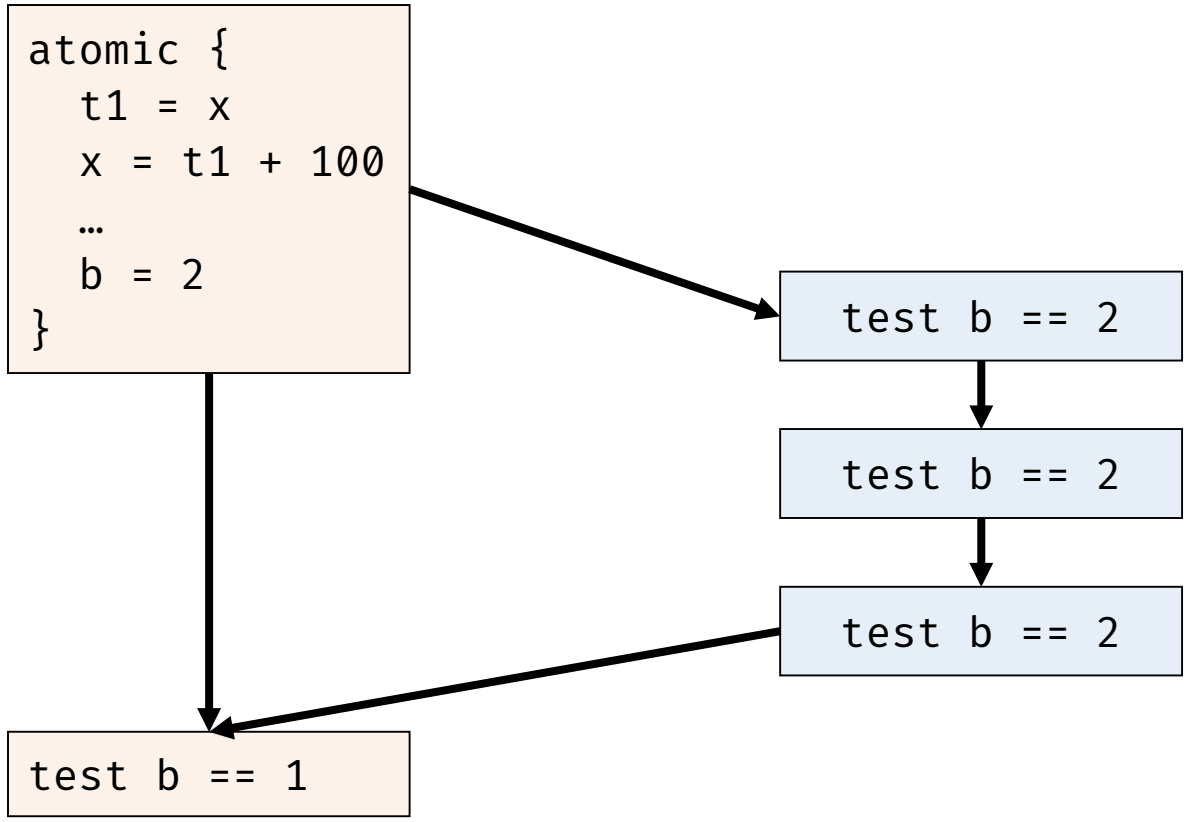
Optimize allocation of Unary Nodes

- Avoid allocation if there are no in edges
  - Will never have in edges
  - Can never participate in a cycle
  - Not even allocated



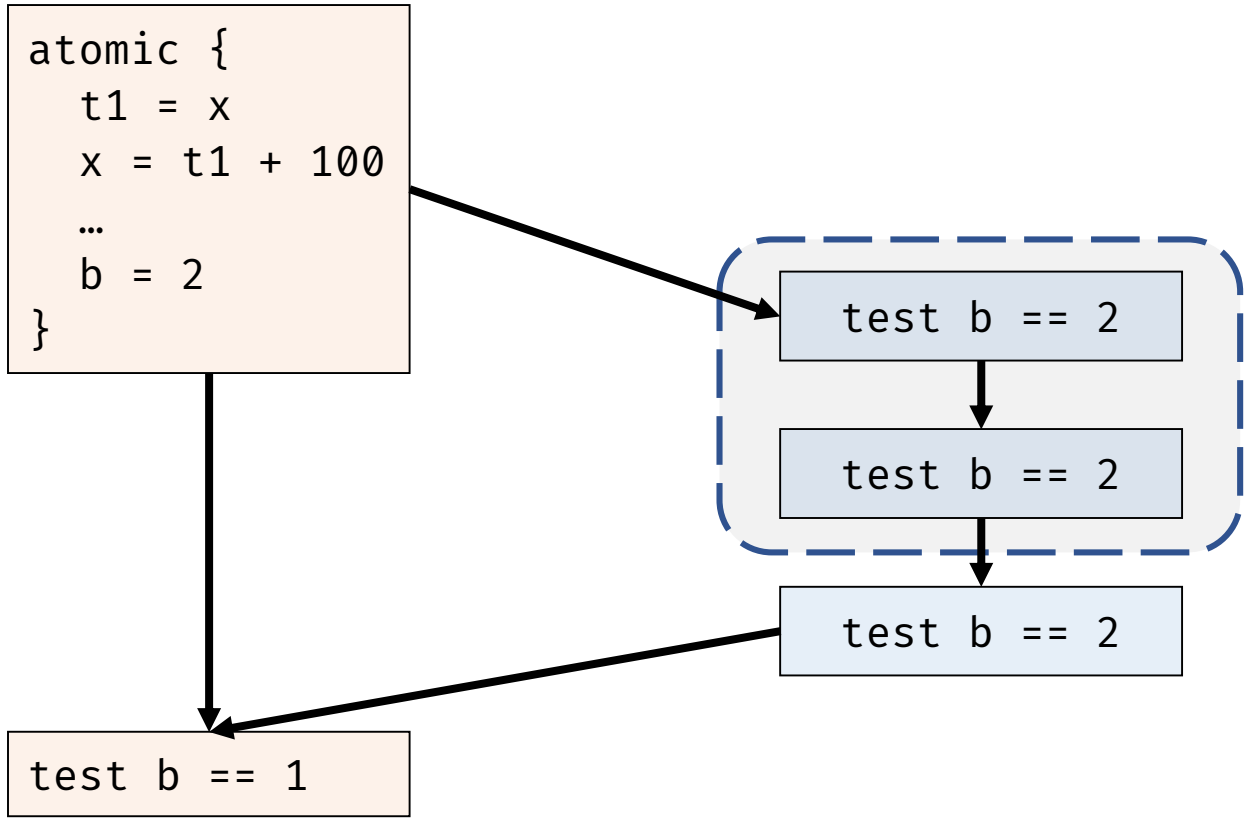
Optimize allocation of Unary Nodes

- Avoid allocation if there are no in edges
  - Will never have in edges
  - Can never participate in a cycle
  - Not even allocated
- If there is a single in edge, then reuse predecessor node



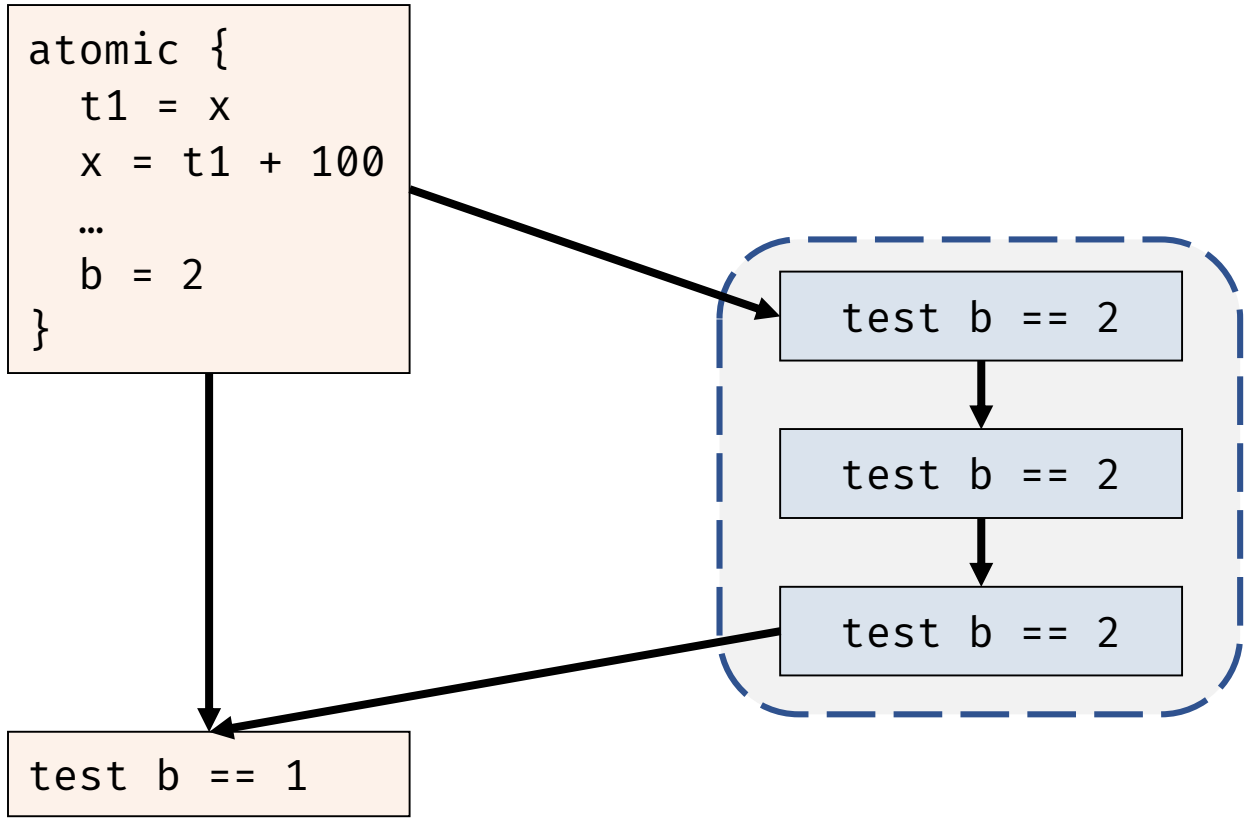
Optimize allocation of Unary Nodes

- Avoid allocation if there are no in edges
  - Will never have in edges
  - Can never participate in a cycle
  - Not even allocated
- If there is a single in edge, then reuse predecessor node



Optimize allocation of Unary Nodes

- Avoid allocation if there are no in edges
  - Will never have in edges
  - Can never participate in a cycle
  - Not even allocated
- If there is a single in edge, then reuse predecessor node

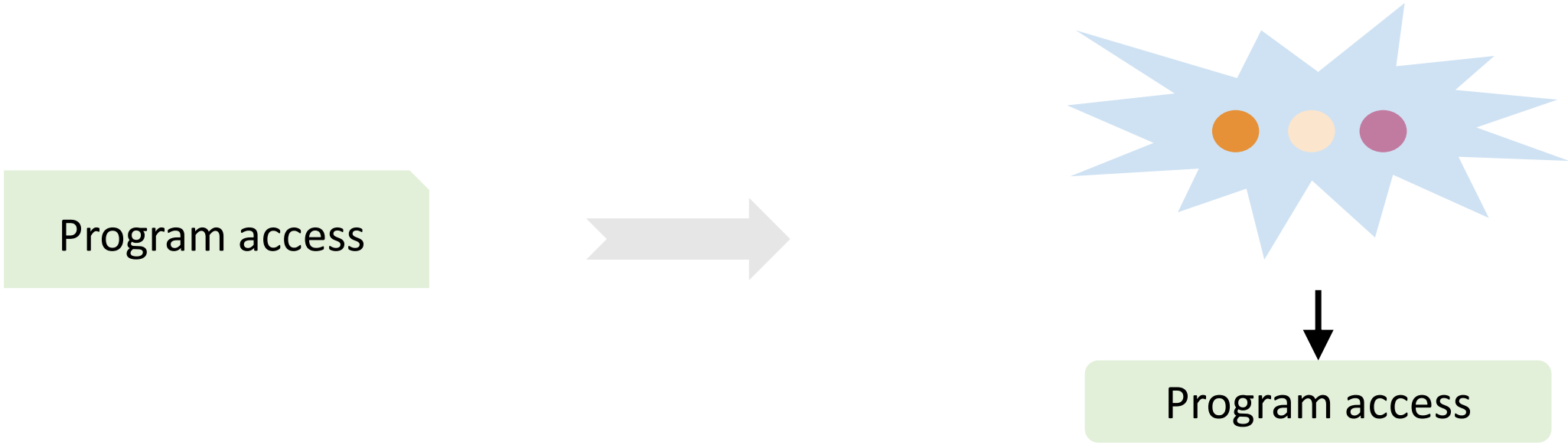


Performance Challenges with Velodrome

- Precise tracking is expensive
  - "**last transaction(s) to read/write**" every field or array element
  - Need **atomic updates** in the instrumentation

- ~6X overhead reported by implementations

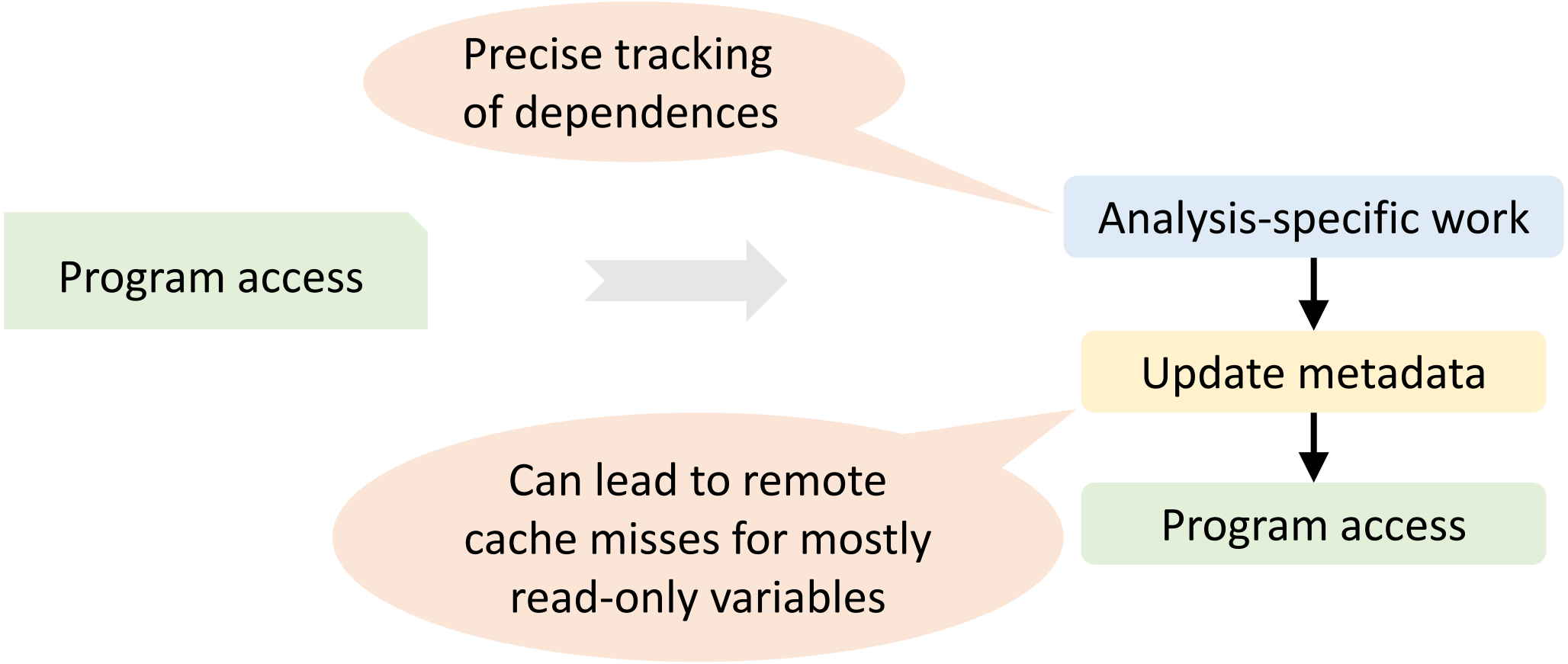
Instrumentation Approach



Uninstrumented program

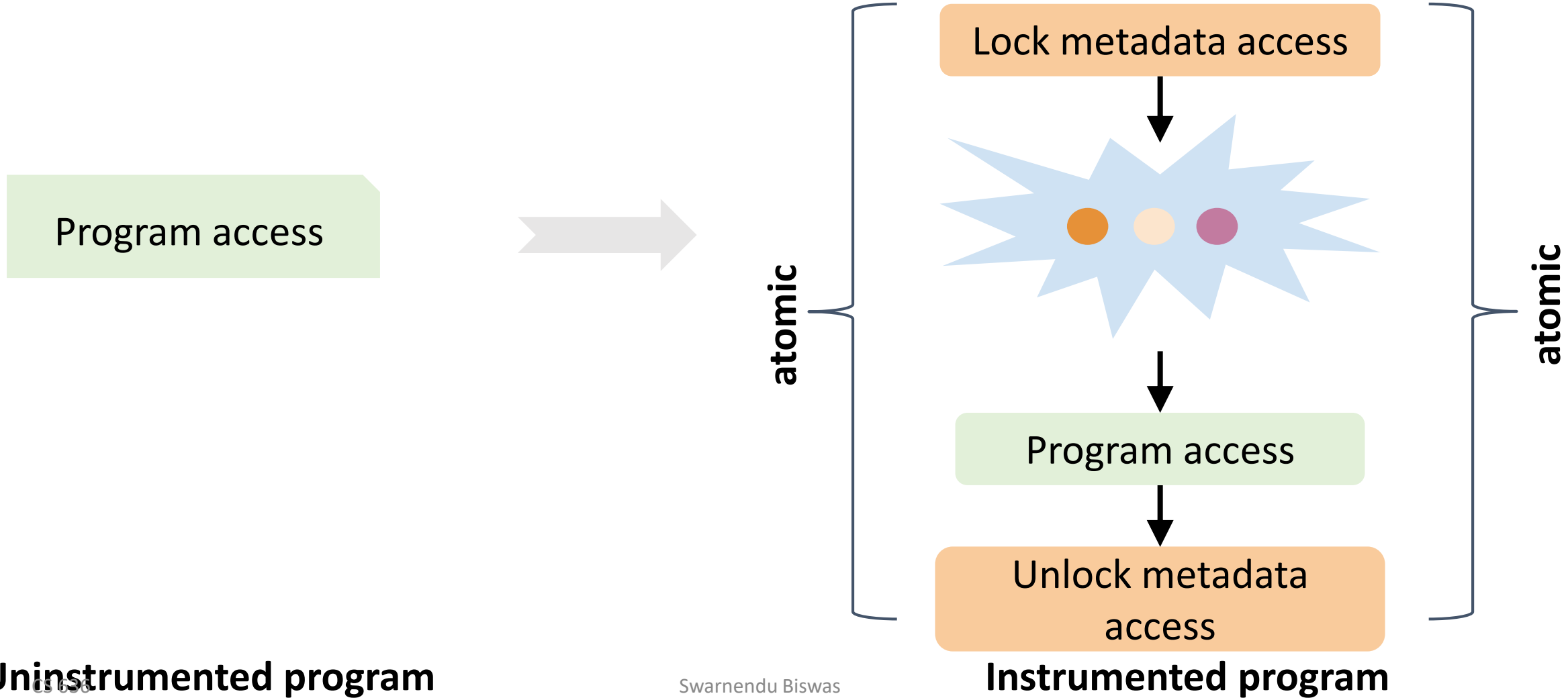
Instrumented program

Precise Tracking is Expensive!

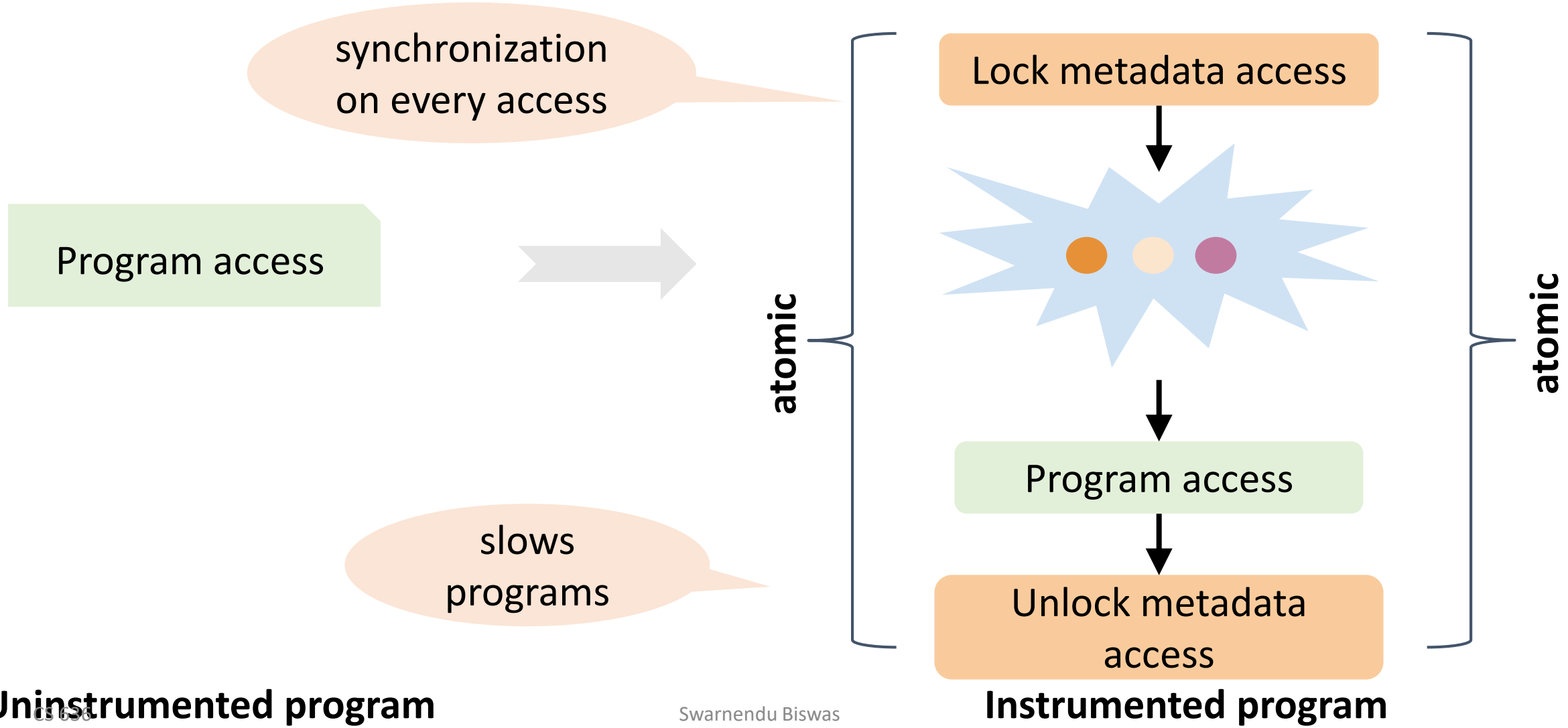


Uninstrumented program **Instrumented program**

Synchronized Updates are Expensive!



Synchronized Updates are Expensive!



Related Work on Atomicity Checking

- Dynamic analysis
  - Conflict-serializability-based approaches
    - Flanagan et al., PLDI 2008; Farzan and Madhusudan, CAV 2008; AeroDrome, ASPLOS 2020
  - Inferring atomicity
    - Lu et al., ASPLOS 2006; Xu et al., PLDI 2005; Hammer et al., ICSE 2008
  - Predictive approaches
    - Sinha et al., MEMOCODE 2011; Sorrentino et al., FSE 2010
  - Other approaches
    - Wang and Stoller, PPoPP 2006; Wang and Stoller, TSE 2006

References

- Mike Bond. CS 6341: Concurrency and Parallelism. Ohio State University.
- Arjun Radhakrishna. CIS 673: Computer Aided Verification. University of Pennsylvania.
- Shan Lu et al. Learning from Mistakes -- A Comprehensive Study on Real World Concurrency Bug Characteristics. ASPLOS 2008.
- S. Savage et al. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, 1997.
- J Erickson et al. Dynamic Analyses for Data Race Detection. RV 2012.
- C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Data Race Detection. PLDI 2009.
- C Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. POPL 2004.