CS 636: Concurrency Bugs

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2024-25-II

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 with the software that controlled the company's long-distance relay switches—software that had just been updated. AT&T lost $60 million that day.
- The Pentium chip's FDIV math error (1993)
  - An hardware bug that affected the floating point unit of early Pentium processors. Because of the bug, the processor would return incorrect binary floating point results when dividing certain pairs of high-precision numbers. Intel incurred $475 million to recall the defective processors in December 1994 and issue replacements.
- 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 on the ground computers, the thrusters' output was calculated in the wrong units (pound–seconds instead of newton–seconds).

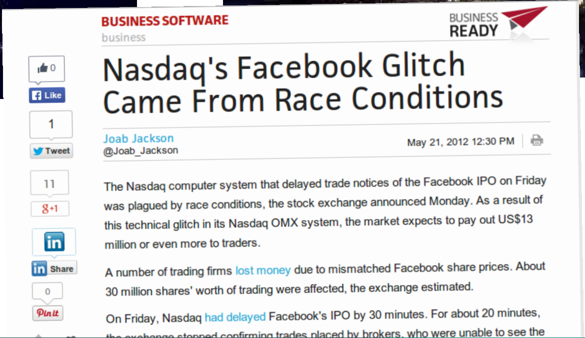
D. Burke. All Circuits are Busy Now: The 1990 AT&T Long Distance Network Collapse. Nov 1995.

Pentium FDIV bug

Mars Climate Orbiter

Examples of Real-World Concurrency Bugs

Examples of Real-World Concurrency Bugs



BUSINESS SOFTWARE

business

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.

On Friday, Nasdaq had delayed Facebook's IPO by 30 minutes. For about 20 minutes, the exchange stopped confirming trades placed by brokers, who were unable to see the

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 from an increasing number of cores.
>
> – A View of Parallel Computing Landscape, CACM 2009.

Challenges in Developing Parallel Programs

Programmers tend to think sequentially

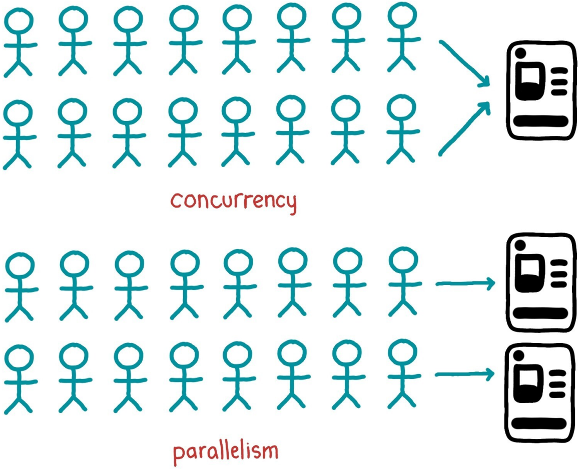
Correctness issues concurrency bugs like data races and deadlocks

Performance issues minimize communication across cores

Other challenges

Amdahl’s law, overheads of parallel execution, load balancing

Concurrency vs Parallelism



Confusing terms in system design: concurrency vs parallelism, performance vs scalability, proxy vs reverse proxy

Concurrency vs Parallelism

Concurrency

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

Parallelism

- Use additional resources to speed up computation
- Performance perspective

What is the difference between concurrency and parallelism?

Types of Concurrent Bugs

Order Violation

Thread 1

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

Thread 2

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

Mozilla: `nsthread.cpp`

Atomicity Violation

Thread 1

```
...
if (thd->proc_info) {



  puts(thd->proc_info, ...)
}
...
```

Thread 2

```

...

thd->proc_info = NULL;

...


```

MySQL: ha_innodb.cc

Sequential Consistency Violation

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

Thread 1

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

Thread 2

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

Deadlock

```java
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;
  }
}
```

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

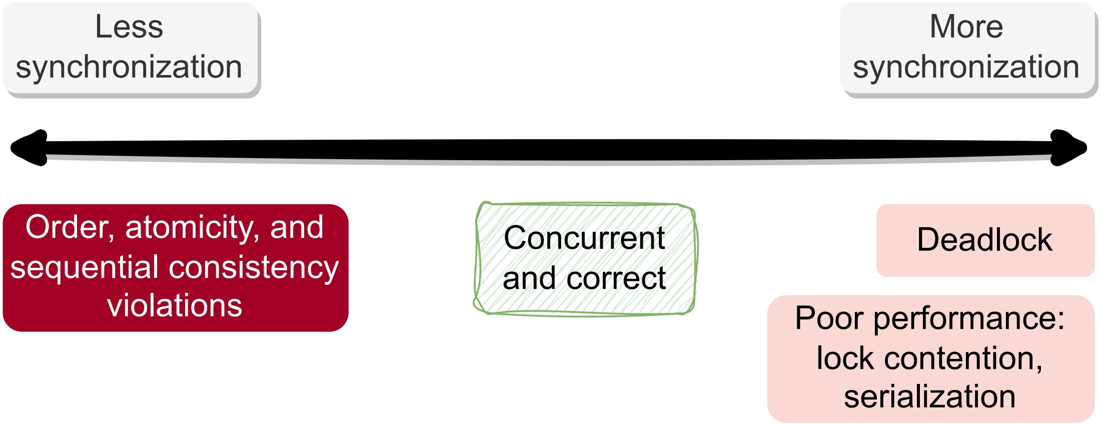
S. Lu et al. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. ASPLOS'08.

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

Challenges with Concurrent Programming



Less synchronization

More synchronization

Order, atomicity, and 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 1

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

Thread 2

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

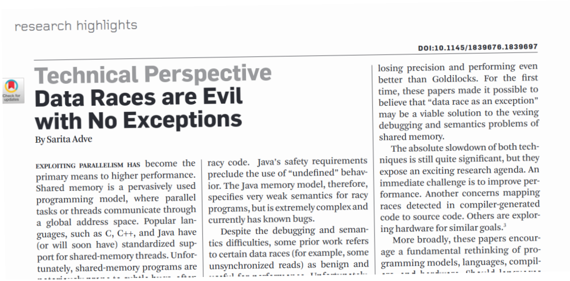
Data Race

- Two accesses from two different threads **conflict** when they access the same shared variable where at least one access is a write
- Accesses are not ordered by synchronization operations are **concurrent** (i.e., can happen at the same time)

Concurrent and conflicting accesses form a data race

Data Races are Evil!

- Often indicate the presence of other types of concurrency errors
- Data races $\neq$ race conditions
  - Race conditions are timing errors on thread interleavings, lock operations
  - Data races are on "data variables"



research highlights

DOI:10.1145/1839676.1839697

Technical Perspective
Data Races are Evil with No Exceptions

By Sarita Adve

EXPLOITING PARALLELISM HAS become the primary means to higher performance. Shared memory is a pervasively used programming model, where parallel tasks or threads communicate through a global address space. Popular languages, such as C, C++, and Java have (or will soon have) standardized support for shared-memory threads. Unfortunately, shared-memory programs are

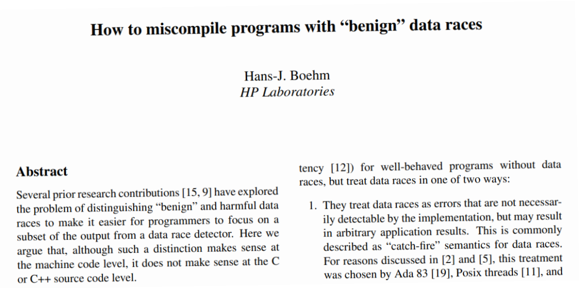
racy code. Java's safety requirements preclude the use of "undefined" behavior. The Java memory model, therefore, specifies very weak semantics for racy programs, but is extremely complex and currently has known bugs.

Despite the debugging and semantics difficulties, some prior work refers to certain data races (for example, some unsynchronized reads) as benign and

losing precision and performing even better than Goldilocks. For the first time, these papers made it possible to believe that "data race as an exception" may be a viable solution to the vexing debugging and semantics problems of shared memory.

The absolute slowdown of both techniques is still quite significant, but they expose an exciting research agenda. An immediate challenge is to improve performance. Another concerns mapping races detected in compiler-generated code to source code. Others are exploring hardware for similar goals.[3]

More broadly, these papers encourage a fundamental rethinking of programming models, languages, compil-



How to miscompile programs with "benign" data races

Hans-J. Boehm
HP Laboratories

Abstract

Several prior research contributions [15, 9] have explored the problem of distinguishing "benign" and harmful data races to make it easier for programmers to focus on a subset of the output from a data race detector. Here we argue that, although such a distinction makes sense at the machine code level, it does not make sense at the C or C++ source code level.

tency [12]) for well-behaved programs without data races, but treat data races in one of two ways:

1. They treat data races as errors that are not necessarily detectable by the implementation, but may result in arbitrary application results. This is commonly described as "catch-fire" semantics for data races. For reasons discussed in [2] and [5], this treatment was chosen by Ada 83 [19], Posix threads [11], and

Get Rid of Data Races

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

Notoriously difficult to detect data races

- 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

Soundness and Precision

- 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, others refer to these properties as complete and sound.

What is soundness (in static analysis)?

Soundness and Completeness: Defined With Precision

B. Livshits et al. In Defense of Soundness: A Manifesto. CACM, vol. 58, no. 2, pp. 44–46, Jan. 2015.

Static Data Race Detection

- Compile-time analysis of the code
- Advantages
  - + Can potentially reason about all inputs and interleavings
  - + No run-time overhead

```
class Account {
  int balance guarded by this;
  int deposit(int x) requires this {
    this.balance = this.balance + x
  }
}
```

- Type-based analysis
  - ▶ Augmented language type system to encode synchronization relations
  - ▶ A correctly typed program implies there is no data race
  - — Restrictive and tedious

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

M. Naik et al. Effective Static Race Detection for Java. PLDI, 2006.

Dynamic Data Race Detection

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

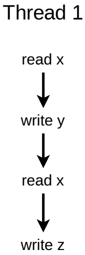
- Post-mortem analyses
- On-the-fly analyses

Dynamic Data Race Detection Techniques

- Happens-before-based algorithms [e.g., DJIT$^+$, FastTrack, Pacer]
- Lockset algorithms [e.g., Eraser]
- Hybrid analysis [e.g., Goldilocks]
- Other partial order relation-based algorithms [e.g., CP, RVPredict, WCP]
- Other techniques [e.g., DataCollider, RaceChaser]

Happens-before Relation

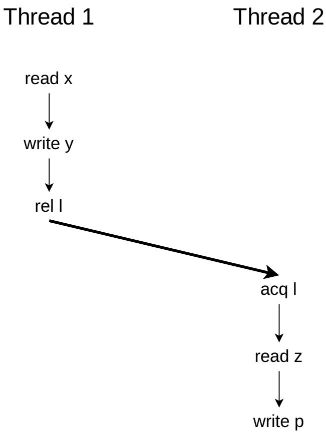
- Smallest transitively-closed relation $\prec_{HB}$ over operations
- Given two operations a and b, a $\prec_{HB}$ b if one of the following conditions hold
    - Program order
        - Operation a is performed by the same thread before operation b



L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. CACM, vol. 21, no. 7, pp. 558–565, Jul. 1978.

Happens-before Relation

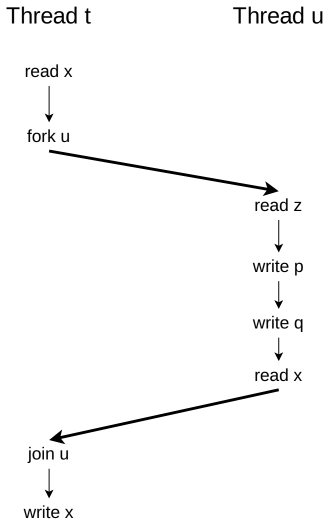
- Smallest transitively-closed relation $\prec_{HB}$ over operations
- Given two operations a and b, a $\prec_{HB}$ b if one of the following conditions hold
  - Program order
    - Operation a is performed by the same thread before operation b
  - Synchronization order
    - a is a lock release and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



L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. CACM, vol. 21, no. 7, pp. 558–565, Jul. 1978.

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))



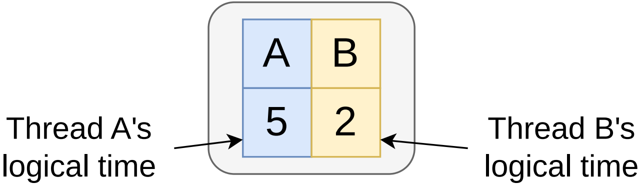
Happens-before (HB) Relation

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

Tracking Happens-before with Vector Clocks

- Each thread T maintains its own logical clock “c”
  - Initially c=0 when T starts
  - Clock is incremented at synchronization release operations (e.g., release(m) and volatile write)
- Vector clock is a vector of logical clocks for all the threads in the process



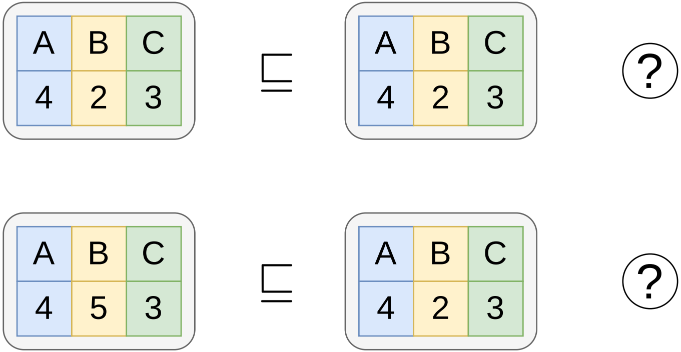
A	B
5	2

Thread A's logical time

Thread B's logical time

Understanding Vector Clocks

$$VC_1 \sqsubseteq VC_2 \text{ iff } \forall\, t\ VC_1(t) \leq VC_2(t)$$



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

?

Properties on Vector Clocks

$$VC_1 \sqsubseteq VC_2 \text{ iff } \forall\, t \; VC_1(t) \leq VC_2(t)$$

$$\text{if } VC_a \sqsubset VC_b, \text{ then } \neg(VC_b \sqsubset VC_a)$$

$$\text{if } VC_a \sqsubset VC_b, \text{ then } a \prec_{HB} b$$

$$\text{if } (VC_a \sqsubset VC_b) \wedge (VC_b \sqsubset VC_c), \text{ then } VC_a \prec_{HB} VC_b$$

Operations on Vector Clocks

Join

$$VC_1 \sqcup VC_2 \;= \lambda t.\text{max}(VC_1(t), VC_2(t))$$

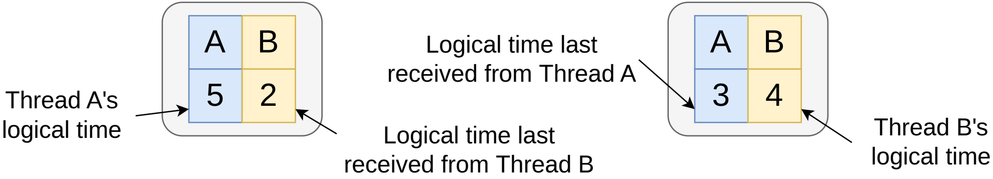
Initialization

$$\bot_V = \lambda t.0$$

Increment

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

Vector Clock-based Race Detection: DJIT$^+$ Algorithm



Thread A's logical time

A	B
5	2

Logical time last received from Thread B

Logical time last received from Thread A

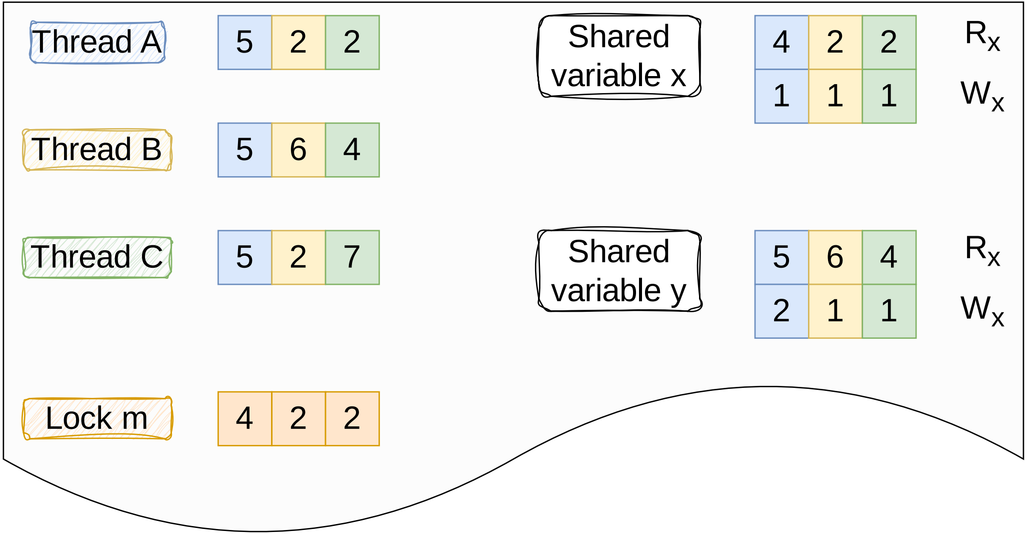
A	B
3	4

Thread B's logical time

- 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. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. PPoPP, 2003.

Snapshot of Process Memory



Thread A	5	2	2
Thread B	5	6	4
Thread C	5	2	7
Lock m	4	2	2

Shared variable x

4	2	2	R_x
1	1	1	W_x

Shared variable y

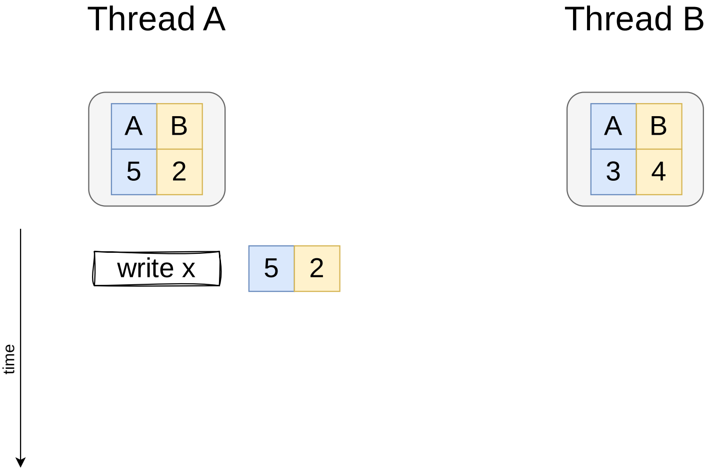
5	6	4	R_x
2	1	1	W_x

Thread A

A	B
5	2

Thread B

A	B
3	4



Thread A

A	B
5	2

write x

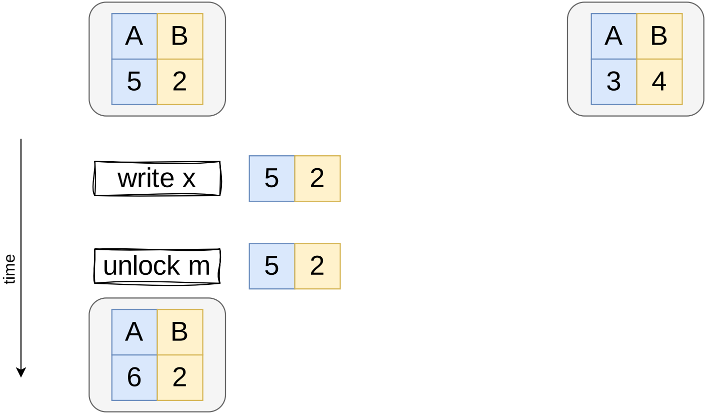
5	2

Thread B

A	B
3	4

Thread A

Thread B



Thread A:

A	B
5	2

Thread B:

A	B
3	4

write x

5	2

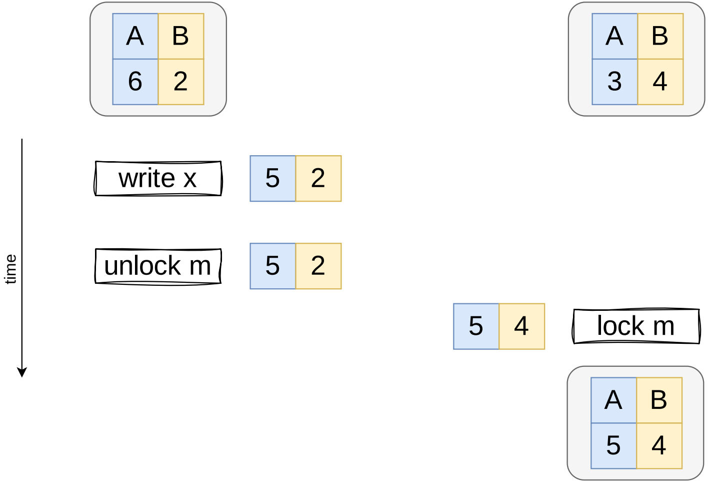
unlock m

5	2

A	B
6	2

Thread A

Thread B



A	B
6	2

A	B
3	4

time

write x

5	2

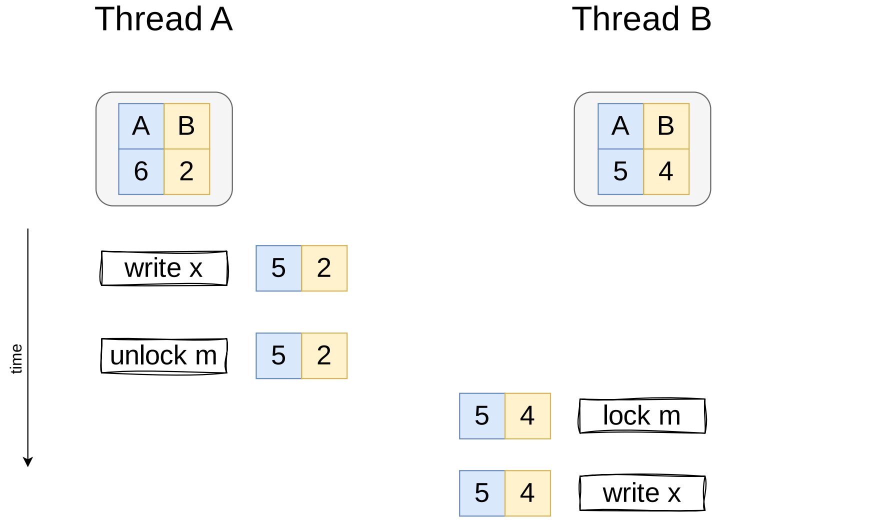
unlock m

5	2

5	4

lock m

A	B
5	4



Thread A

A	B
6	2

write x 5 2

unlock m 5 2

time

Thread B

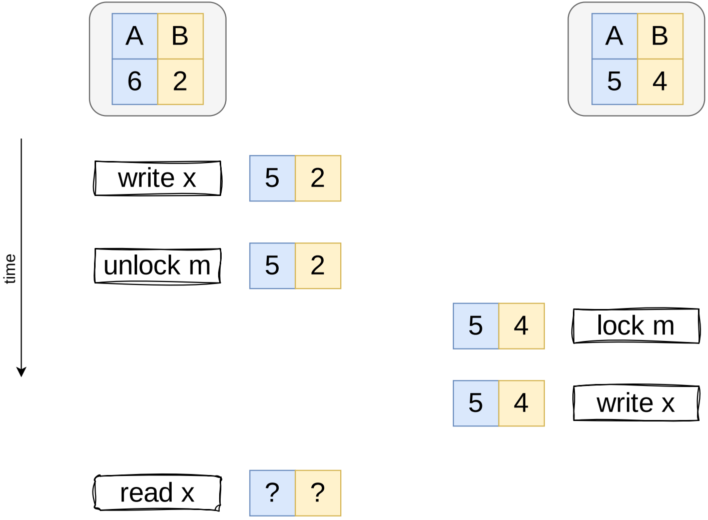
A	B
5	4

5 4 lock m

5 4 write x

Thread A

Thread B



A	B
6	2

A	B
5	4

time

write x 5 2

unlock m 5 2

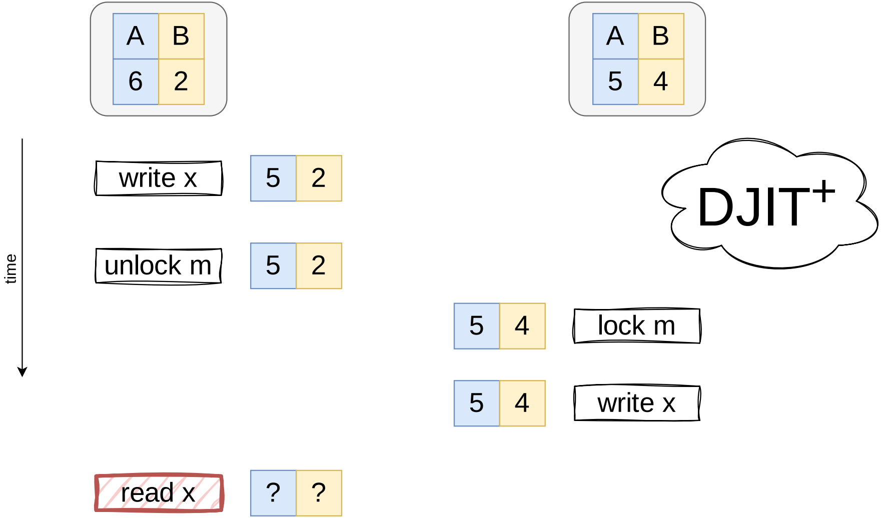
5 4 lock m

5 4 write x

read x ? ?

Thread A

Thread B



A	B
6	2

A	B
5	4

DJIT$^+$

time

write x 5 2

unlock m 5 2

5 4 lock m

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
  - can **miss** data races that did not manifest in observed run, but may happen in **another** interleaving

Thread A

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




```

Thread B

```




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

Lockset Algorithms

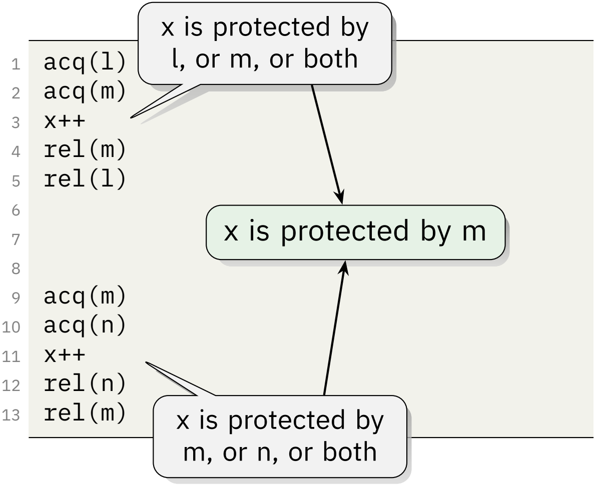
- Assumes that all shared-memory accesses follow a consistent locking discipline
- Keeps track of the locks associated with each thread and program variable

Thread A	Lockset$_A$
	L = {}
lock m	
	L = {m}
write x	
	L = {m}
lock n	
	L = {m, n}
write y	
	L = {m, n}
unlock n	
	L = {m}
unlock m	
	L = {}
read x	
	L = {}

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

Inferring the Locking Discipline

- Assumes that all shared-memory accesses follow a consistent locking discipline
- Keeps track of the locks associated with each thread and program variable
- Two accesses from different threads with non-intersecting locksets form a data race
- How do we know which lock protects which variable?
  - Programmer annotations are cumbersome



```
acq(l)
acq(m)
x++
rel(m)
rel(l)



acq(m)
acq(n)
x++
rel(n)
rel(m)
```

x is protected by l, or m, or both
x is protected by m
x is protected by m, or n, or both

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
- 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
  - $C(v) := C(v) \cap L(t)$
  - If $C(v) = \phi$, report that there is a data race for v

Lockset refinement

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

Properties of Lockset Algorithms

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

Thread A

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

Thread B

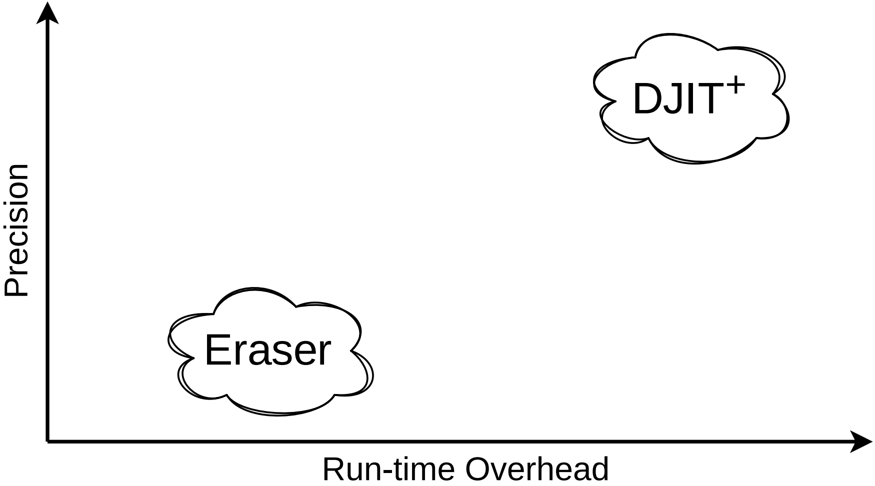
```




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

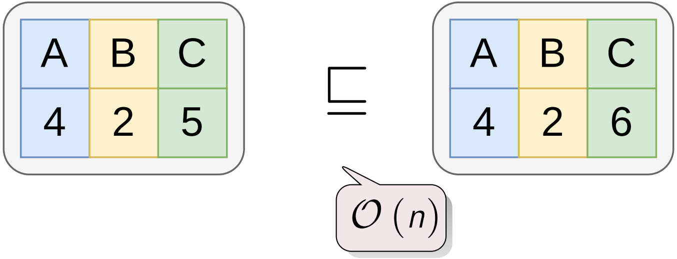
Are lockset algorithms sound and precise?

DJIT$^+$ vs Eraser



Why is DJIT$^+$ expensive?

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



A	B	C
4	2	5

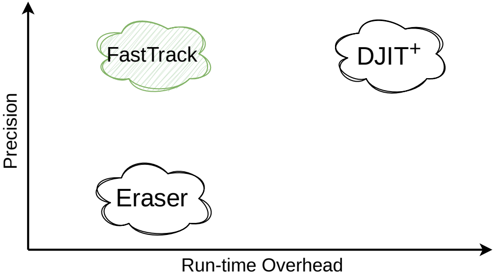
$\sqsubseteq$

A	B	C
4	2	6

$\mathcal{O}(n)$

U. Mathur et al. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. ASPLOS, 2022.

FastTrack: Efficient HB Tracking



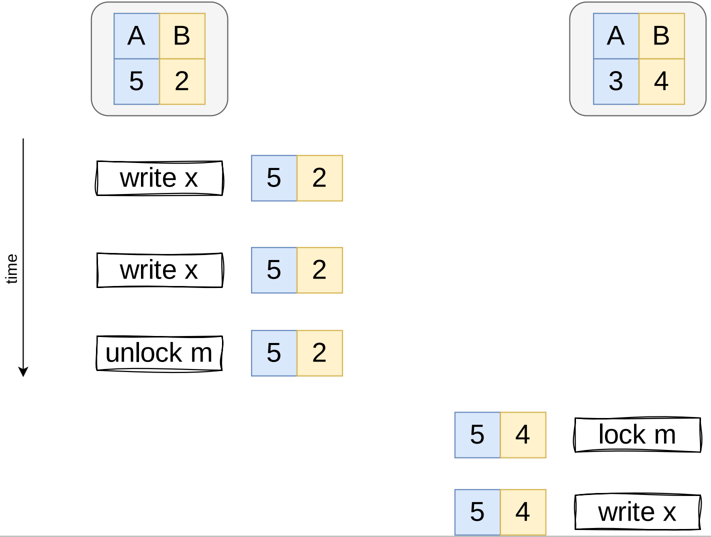
Insight

- All writes to a shared variable, till the first race, are totally ordered
- Reads are not totally-ordered even in data-race-free programs (e.g., read-shared data)

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

Thread A

Thread B



Thread A

A	B
5	2

Thread B

A	B
3	4

time

write x 5 2

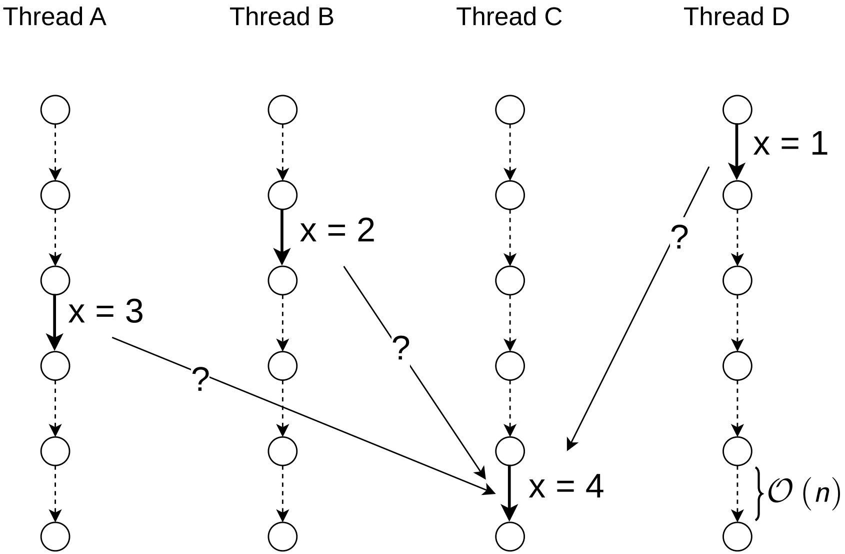
write x 5 2

unlock m 5 2

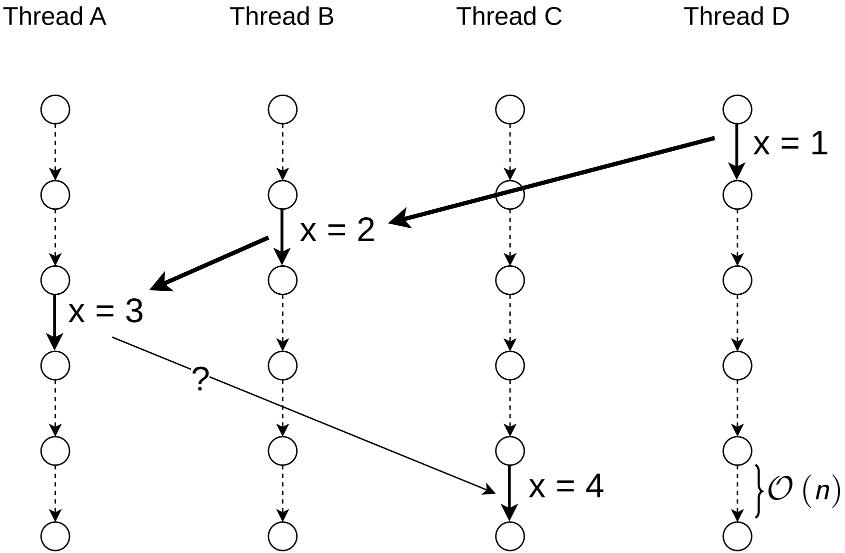
5 4 lock m

5 4 write x

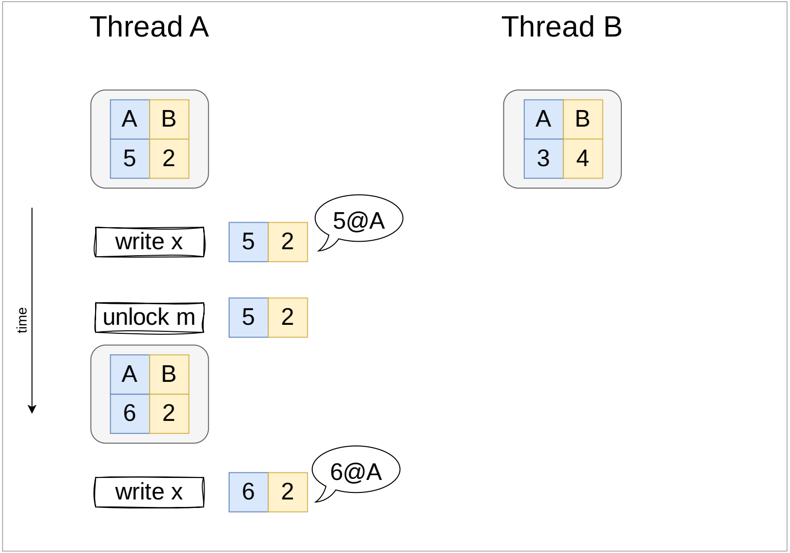
Write-Write and Write-Read Data Races



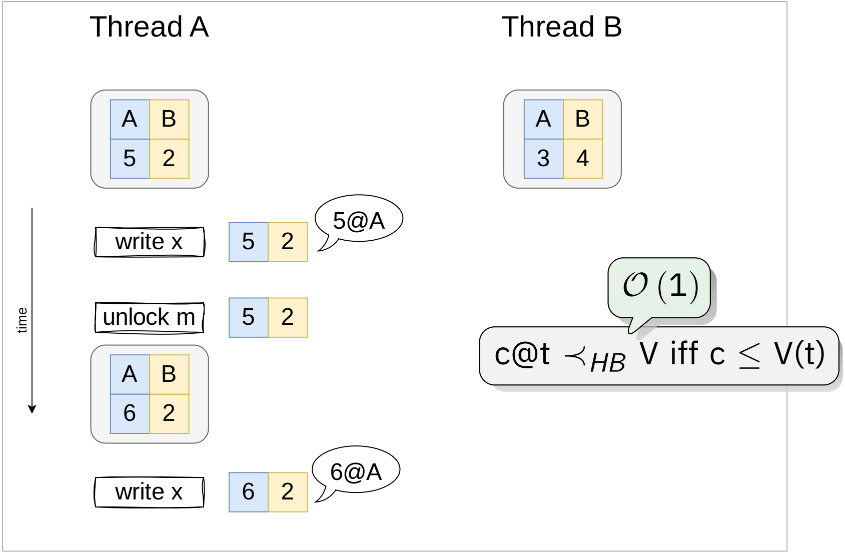
No Data Races Yet: Writes Totally Ordered



Last Writer Epoch



Last Writer Epoch



Thread A

A	B
5	2

write x

5	2

5@A

unlock m

5	2

A	B
6	2

write x

6	2

6@A

Thread B

A	B
3	4

$\mathcal{O}(1)$

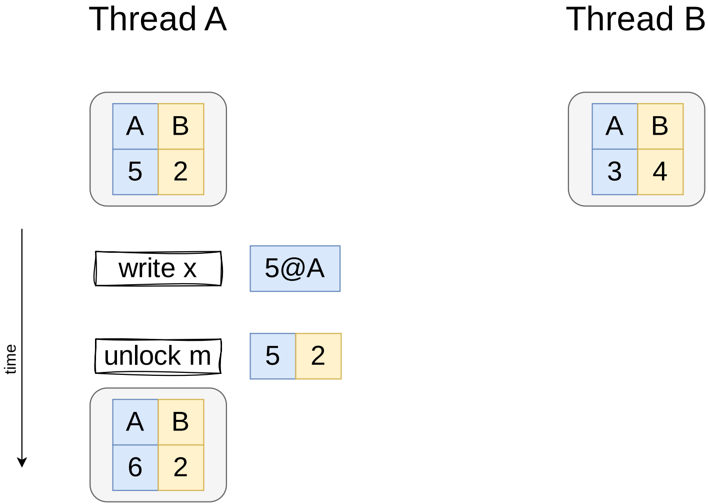
$c@t \prec_{HB} V$ iff $c \leq V(t)$

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

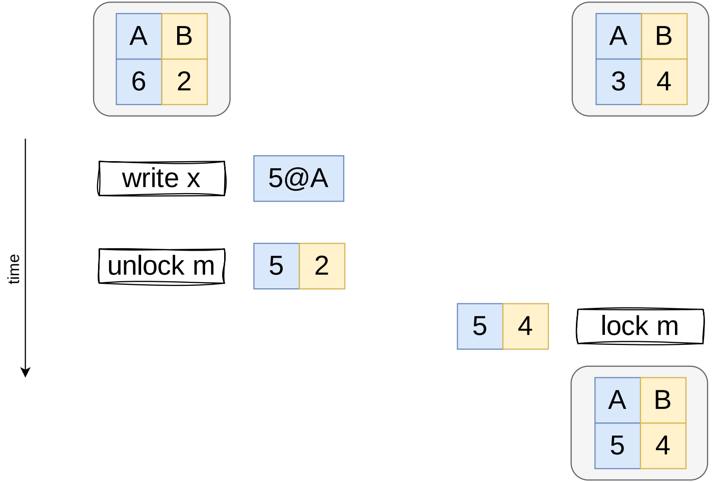
A	B
6	2

Thread B

A	B
3	4

Thread A

Thread B



A	B
6	2

A	B
3	4

write x

5@A

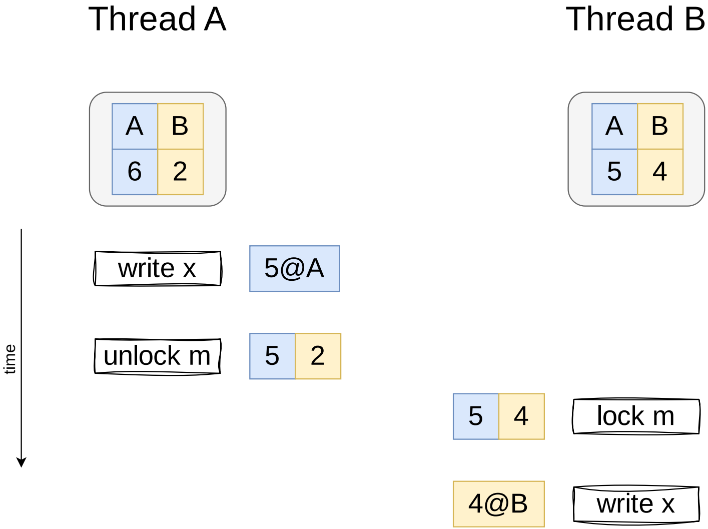
unlock m

5	2

5	4

lock m

A	B
5	4



Thread A

A	B
6	2

write x — 5@A

unlock m — 5 | 2

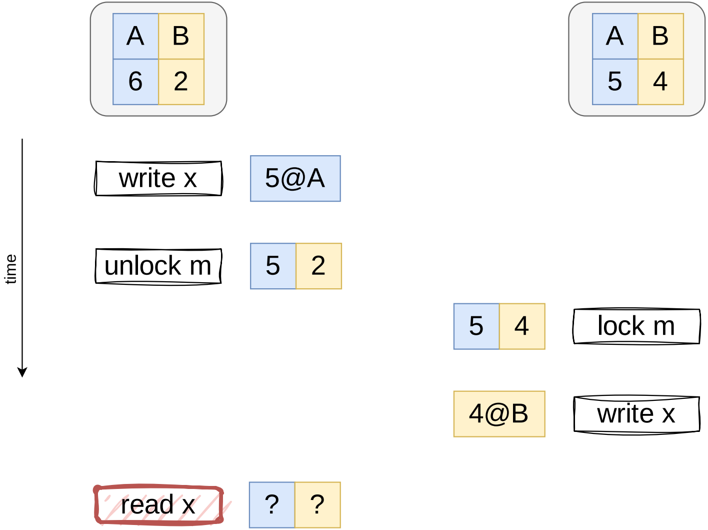
Thread B

A	B
5	4

5 | 4 — lock m

4@B — write x

Thread A Thread B



A B
6 2

A B
5 4

time

write x 5@A

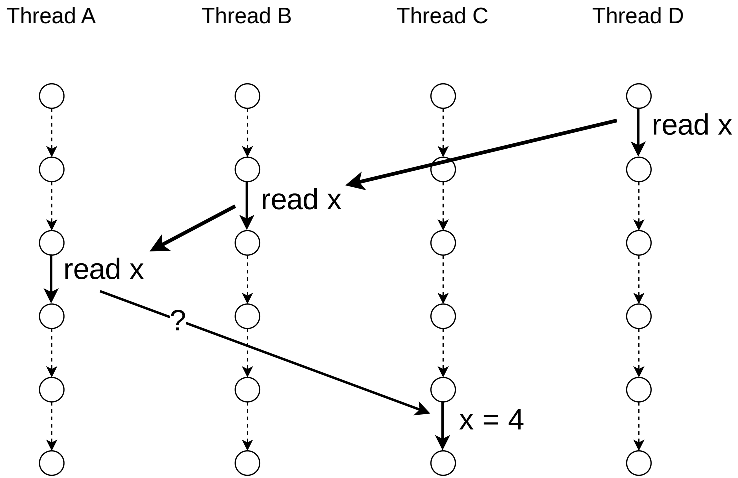
unlock m 5 2

5 4 lock m

4@B write x

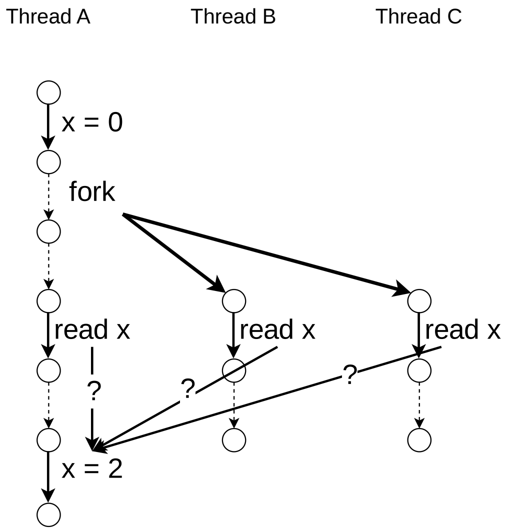
read x ? ?

Read-Write Data Races — Ordered Reads



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

Read-Write Data Races — Unordered Reads



Comparing Lockset and HB Analyses

Lockset analysis

- Assumes consistent locking discipline
- Imprecise, reports many false positives

Happens-before analysis

- Dynamically sound and precise
- Coverage limited to observed executions
- Not scalable, incurs space overhead
- Correctness depends on exact knowledge of synchronization

Performance of Lockset and HB Algorithms

- FastTrack's slowdowns are still ∼4–8X
- Intel Thread Checker has 200X overhead
- Google's ThreadSanitizer (part of LLVM) incurs around ∼5–15X overhead
- Happens-before-based sampling approaches (e.g., LiteRace[†] and Pacer[§])
  - Overheads are still too high for a reasonable sampling rate
  - Pacer with 3% sampling rate incurs 86% overhead!!!

Large overheads impact the thread interleaving pattern

[†] D. Marino et al. LiteRace: Effective Sampling for Lightweight Data-Race Detection. PLDI 2009.

[§] M. Bond et al. Pacer: Proportional Detection of Data Races. PLDI 2010

Is there a data race?

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

Thread 1

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

Thread 2

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

Is there a data race?

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

Thread 1

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

Thread 2

```java




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

Is there a data race?

Thread 1

```
x++;
malloc();










```

Thread 2

```










malloc();
x++;
```

Is there a data race?

Thread 1

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

Thread 2

```






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

Is there a data race?

Thread 1

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

Thread 2

```






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

Correctness depends on exact knowledge of synchronization

Collision Analysis

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

(i) Pause one thread just before accessing a memory location x

(ii) 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

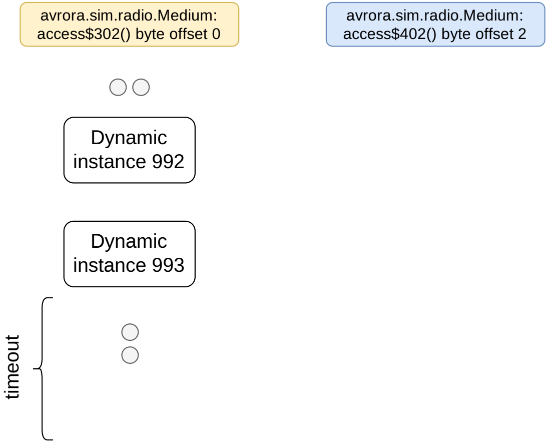
avrora.sim.radio.Medium: access$302() byte offset 0

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

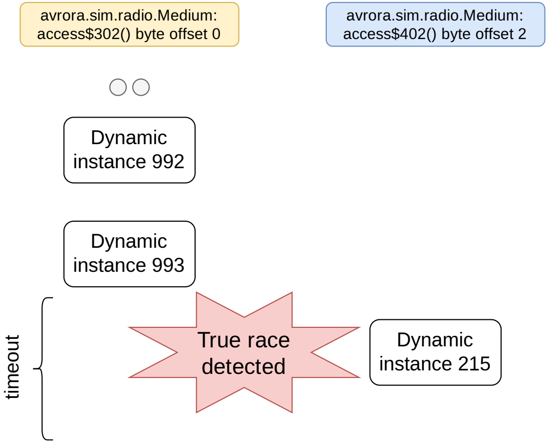
Randomly Sample Racy Accesses and Try to Collide Them

Block thread for some time

Benefits: Can track frequency of samples taken and estimate overhead introduced by waiting


avrora.sim.radio.Medium: access$302() byte offset 0
avrora.sim.radio.Medium: access$402() byte offset 2
Dynamic instance 992
Dynamic instance 993
timeout

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

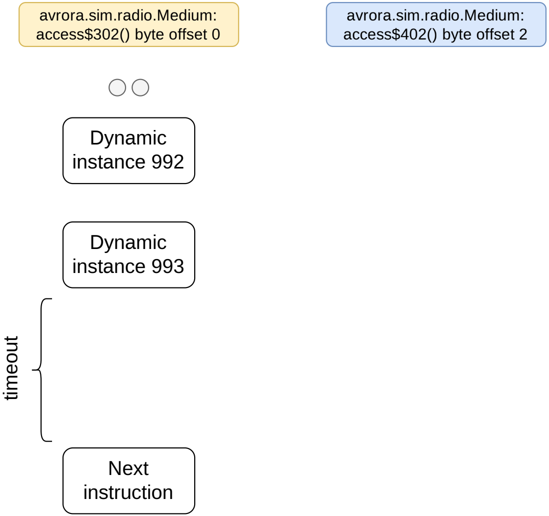
timeout

True race detected

Dynamic instance 215

Collision is Unsuccessful

Thread unblocks, resets the analysis state, and continues execution


avrora.sim.radio.Medium: access$302() byte offset 0
avrora.sim.radio.Medium: access$402() byte offset 2
Dynamic instance 992
Dynamic instance 993
timeout
Next instruction

Advantages of Collision Analysis

- + Oblivious to synchronization patterns (i.e., no inference is necessary)
- + Low memory overhead compared to maintaining vector clocks
- + 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

DataCollider: Hardware Implementation of Collision Analysis

- Uses hardware debug registers (DR0...DR7) to monitor access locations
  - x86 has four usable debug registers (DR0...DR3)
  - Two are aliases (DR4 and DR5) and two are for control (DR6 and DR7)
- Writes an address to a debug register, sets 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

- Small delays at several shared-memory accesses would still introduce large overheads
- Uses sampling, i.e., only execute slow path when certain conditions are met
  - Prioritize cold code regions or sample based on allowed tolerable overhead

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

— Usually, # of threads $\gg$ # debug registers (i.e., 4) which reduces the effectiveness of the analysis

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
- Any verification using model-based techniques is only as good as the model of the system
- 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.

Recent Research on Data Race Detection

- 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

[†] S. Blackshear et al. RacerD: Compositional Static Race Detection. OOPSLA, 2018.

[§] Y. Smaragdakis et al. Sound Predictive Race Detection in Polynomial Time. POPL 2012.

[¶] D. Kini et al. Dynamic Race Prediction in Linear Time. PLDI 2017.

Detecting Atomicity Violations

Are there Data Races?

Following snippet is from an old version of `java.lang.stringbuffer`

```java
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(...) { ... }
  ...
}
```

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

Data Race Freedom (DRF) vs Atomicity

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

Atomicity is a more fundamental non-interference property

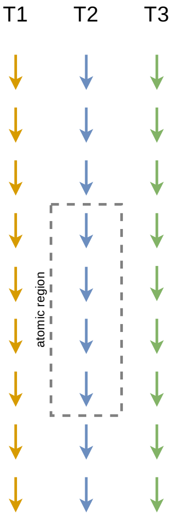
Why Study Atomicity Violation Detection?

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

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

Atomicity Property

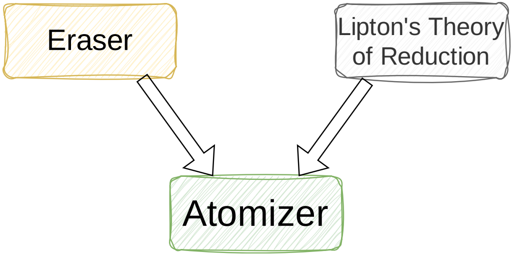
- Maximal non-interference property that enables sequential reasoning
- Atomic region's execution appears not to be interleaved with other concurrent threads
- Program execution must be equivalent to a serial execution of atomic regions
- Synonymous with serializability for programming language semantics



Atomizer

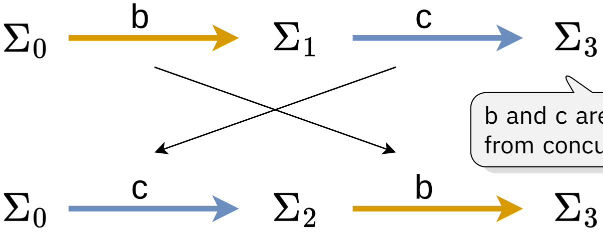
Idea

- 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.

Commuting Actions: Left and Right Movers

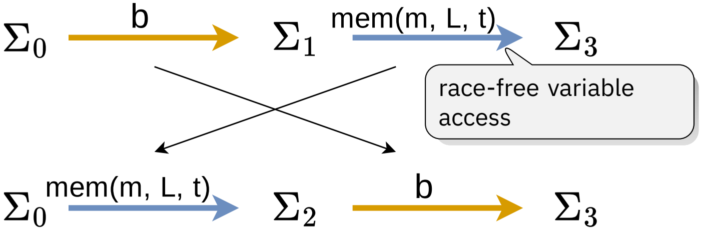


b and c are operations from concurrent threads

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

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

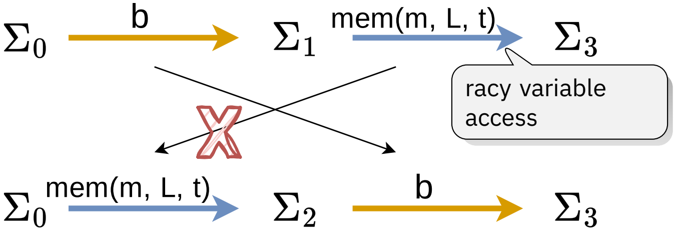
Commuting Actions: Both Mover



race-free variable 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



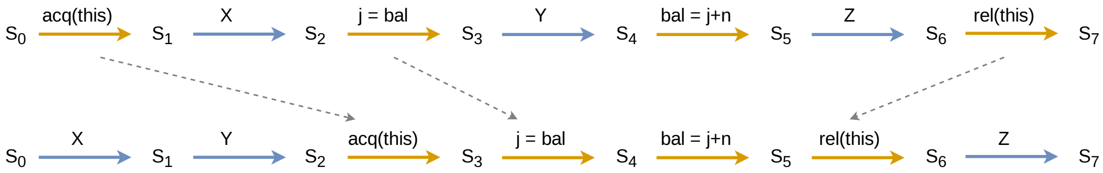
racy variable access

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]

$$S_0 \xrightarrow{\text{acq(this)}} S_1 \xrightarrow{X} S_2 \xrightarrow{\text{j = bal}} S_3 \xrightarrow{Y} S_4 \xrightarrow{\text{bal = j+n}} S_5 \xrightarrow{Z} S_6 \xrightarrow{\text{rel(this)}} S_7$$

Theory of Reduction [R. Lipton'75]



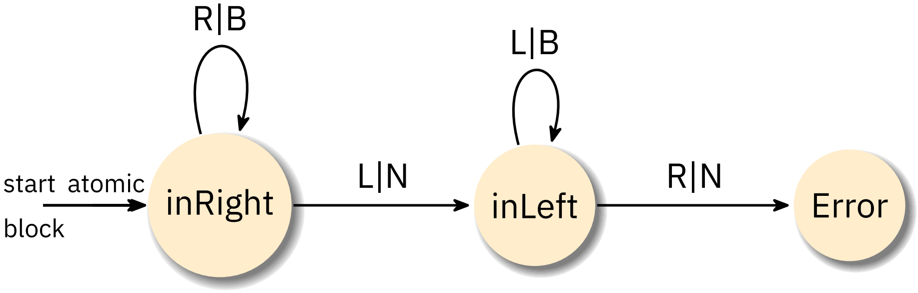
$S_0 \xrightarrow{\text{acq(this)}} S_1 \xrightarrow{\text{X}} S_2 \xrightarrow{\text{j = bal}} S_3 \xrightarrow{\text{Y}} S_4 \xrightarrow{\text{bal = j+n}} S_5 \xrightarrow{\text{Z}} S_6 \xrightarrow{\text{rel(this)}} S_7$

$S_0 \xrightarrow{\text{X}} S_1 \xrightarrow{\text{Y}} S_2 \xrightarrow{\text{acq(this)}} S_3 \xrightarrow{\text{j = bal}} S_4 \xrightarrow{\text{bal = j+n}} S_5 \xrightarrow{\text{rel(this)}} S_6 \xrightarrow{\text{Z}} S_7$

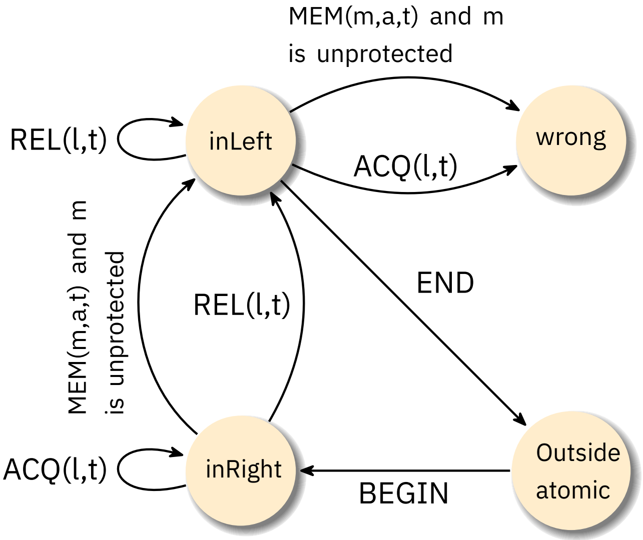
Suppose a path through a code block contains a sequence of right-movers, followed by at most one non-mover action and then a sequence of left-movers. Then this path can be reduced to an equivalent serial execution, with the same resulting state, where the path is executed without any interleaved actions by other threads.

Performing Reduction Dynamically

Reducible methods: $(R|B)^* \; [N] \; (L|B)^*$



Atomizer Algorithm

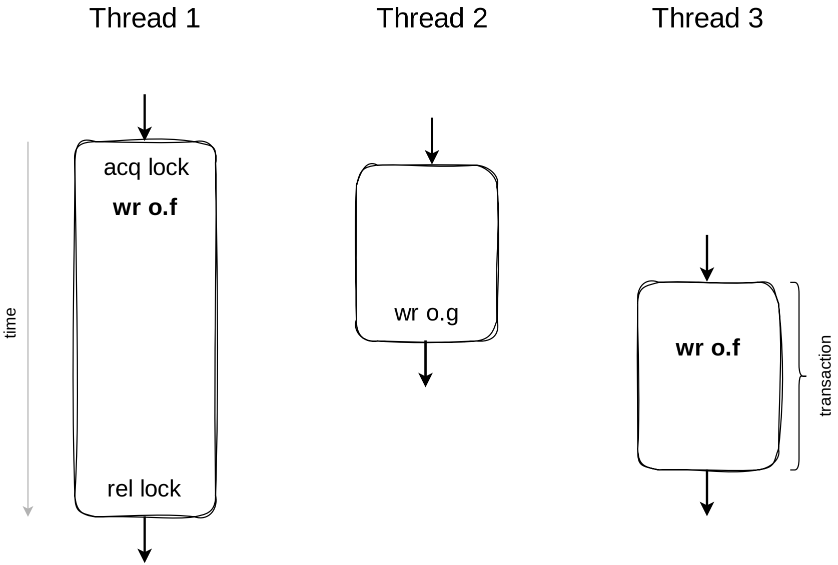


Velodrome: Dynamically Sound and Precise Atomicity Checking

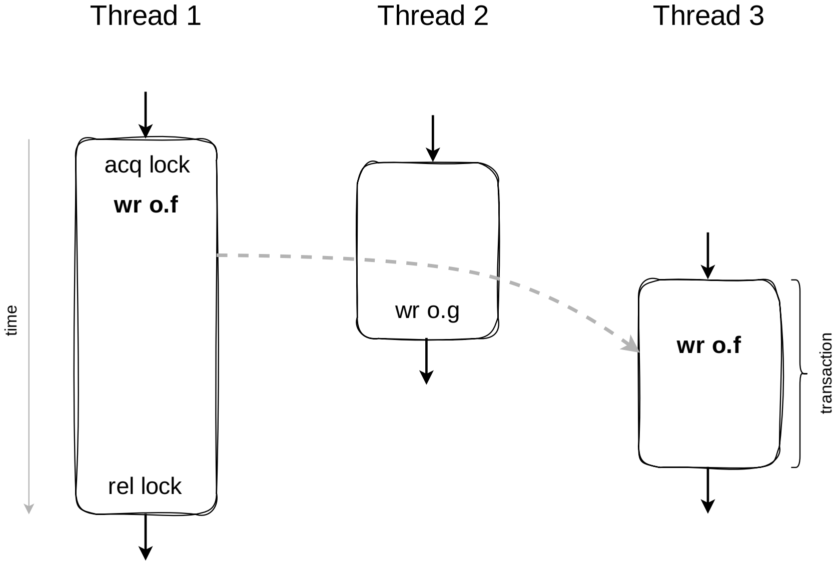
- Tracks HB relations between transactions (i.e., atomic regions)
  - A transaction is a dynamic execution of an atomic block
  - Lifts HB relations from operations to transactions
- Builds a transactional dependence graph
- Checks for presence of cycles in the graph
  - Depicts violations of conflict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



Thread 1

Thread 2

Thread 3

acq lock

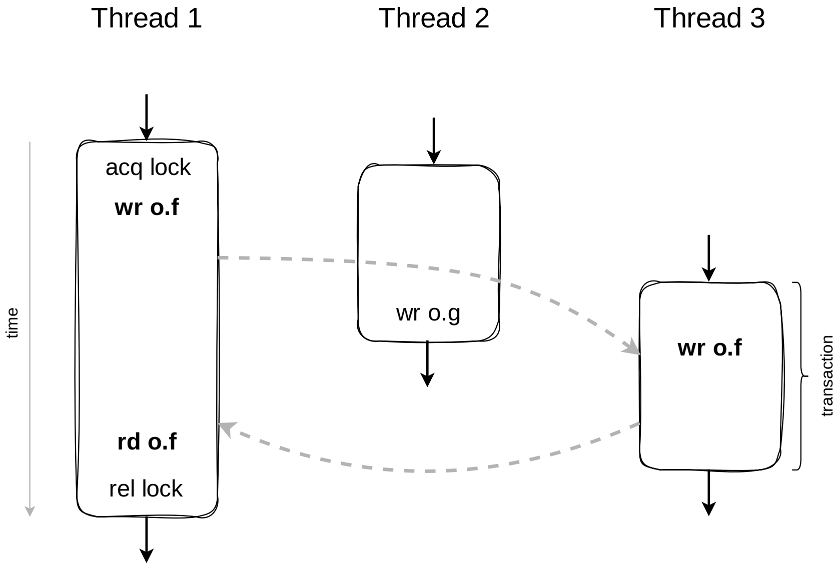
wr o.f

rel lock

wr o.g

wr o.f

Cycle means Atomicity Violation



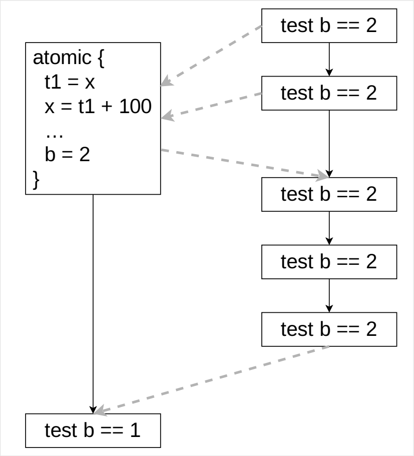
Challenges in Velodrome

- Transactional dependence graph can become huge
  - Each statement outside an atomic region requires a node (i.e., unary transactions)
  - Basic analysis is correct but will not scale for large programs

- 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 (e.g., last readers and writer nodes have already been collected)
  - If there is a single in edge, then reuse predecessor node

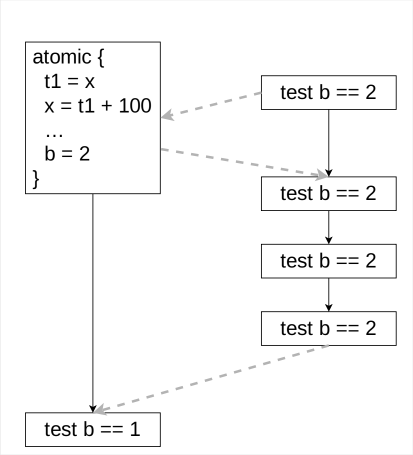
Optimize Allocation of Unary Nodes

- Avoid node allocation if there are no in edges
  - Can never participate in a cycle



Optimize Allocation of Unary Nodes

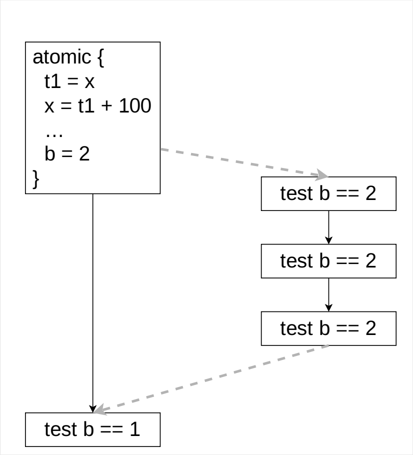
- Avoid node allocation if there are no in edges
  - Can never participate in a cycle



atomic {
  t1 = x
  x = t1 + 100
  …
  b = 2
}

Optimize Allocation of Unary Nodes

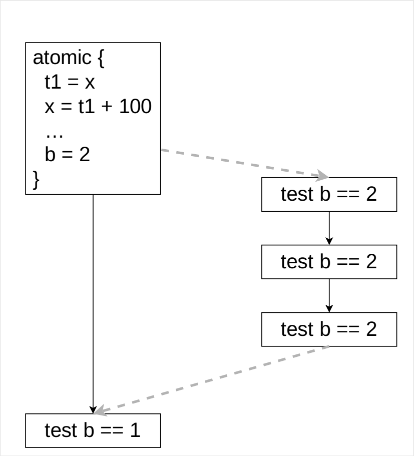
- Avoid node allocation if there are no in edges
  - Can never participate in a cycle



```
atomic {
  t1 = x
  x = t1 + 100
  …
  b = 2
}
```

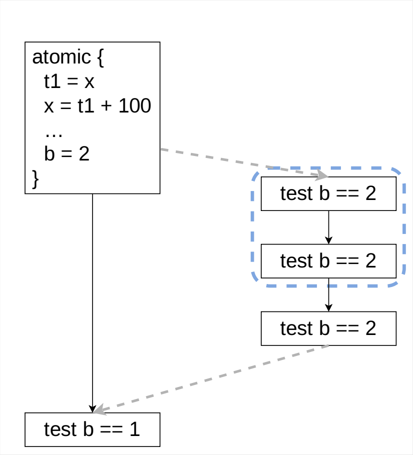
Optimize Allocation of Unary Nodes

- Avoid node allocation if there are no in edges
  - Can never participate in a cycle
- If there is a single in edge, then reuse predecessor node



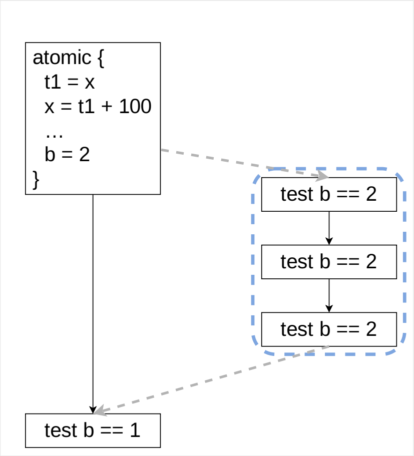
Optimize Allocation of Unary Nodes

- Avoid node allocation if there are no in edges
  - Can never participate in a cycle
- If there is a single in edge, then reuse predecessor node



Optimize Allocation of Unary Nodes

- Avoid node allocation if there are no in edges
    - Can never participate in a cycle
- If there is a single in edge, then reuse predecessor node



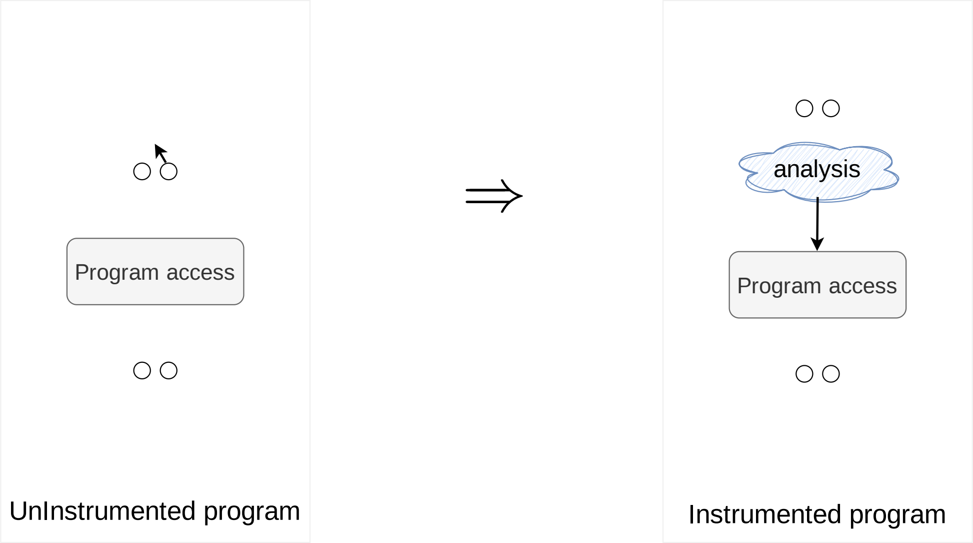
```
atomic {
  t1 = x
  x = t1 + 100
  ...
  b = 2
}
```

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



Program access

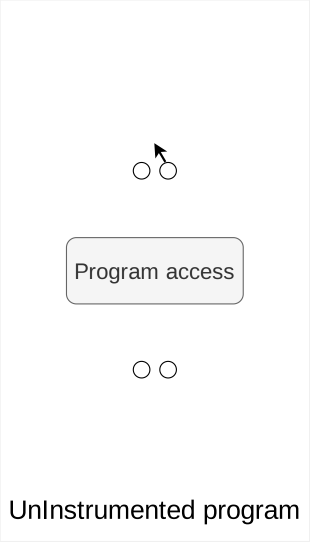
UnInstrumented program

analysis

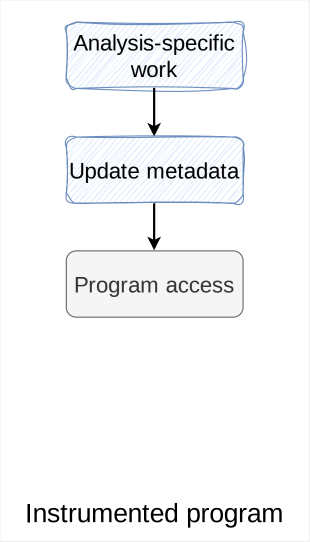
Program access

Instrumented program

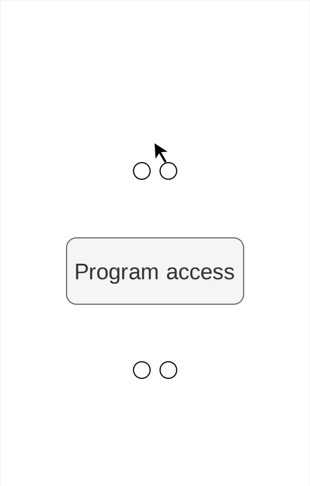
Precise Tracking is Expensive!



⇛

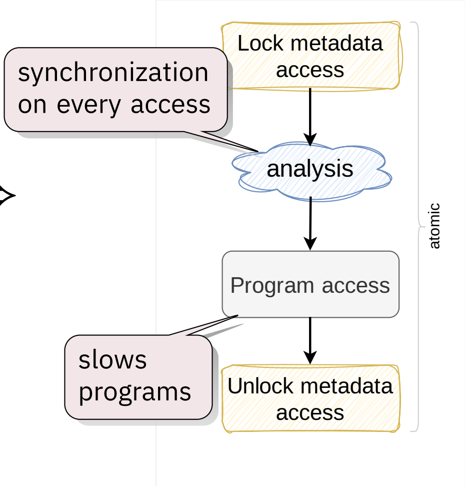


Synchronized Updates are Expensive!



UnInstrumented program

⇒



Instrumented program

Related Work on Atomicity Checking

Dynamic analysis

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

References

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. Chapters 26, 32, Online.