CS492B Analysis of Concurrent Programs

Atomicity Violation Detection

Prof. Moonzoo Kim

Data Race Free, Yet Race Bug (1/2)

```
public class Vector implements Collection {
 int elementCount ;
 Object [] elementData ;

 public Vector(Collection c){
  elementCount = c.size() ;
  elementData = new Object[elementCount] ;
  c.toArray(elementData) ; }

 public synchronized int size(){
  return elementCount ; }

 public synchronized Object[] toArray(Object a[]){
  ...
  System.arraycopy(elementData, 0, a, 0, elementCount) ;
  ... }

 public synchronized boolean add(Object o) {
   ...
  elementCount++; ...} }
```

Data race free (protected by this)

Should be executed atomically

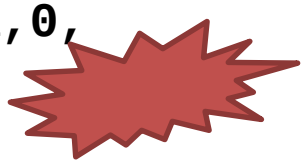
Thread 1	Thread 2
`Vector v2=new Vector(v1);`	`v1.add(obj);`

Data Race Free, Yet Race Bug (2/2)

Thread 1

```
//v1.size()
lock(v1)
t=v1.elementCount// 10
unlock(v1)
v2.elementCount=t
v2.elementData=new Object[10]
```

```
//v1.toArray()
lock(v1)
System.arraycopy(v1.elementData
 ,0,v2.elementData,0,
 v1.elementCount)
```



Array out of bound error

Thread2

```
//v1.add()
lock(v1)
…
v1.elementCount++ // 11
…
unlock(v1)
```

Time-Of-Check to Time-Of-Use (TOCTOU) Attack

Victim	Attacker

//a program installed setuid root
if (access("sym_link_file","w") != 0)
  exit(1) ;
 	
fd = fopen("sym_link_file", "w+");
// Do something about fd
	//after the access check
	symlink("/etc/passwd","sym_link_file");

An attacker can exploit the race condition between the access and open to trick the victim into reading/overwriting an entry in the system password database.

http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

http://cecs.wright.edu/~pmateti/InternetSecurity/Lectures/RaceConditions/index.html

Atomicity

- A code block is *atomic* if for every interleaved execution, there is an equivalent execution with the same result where the code block is executed serially (i.e., not interleaved with other threads).

- Programmers often assume the atomicity of a code block to reason its behavior as if it is a sequential code.
  - c.f. atomic block: code block defined as atomic by programmers

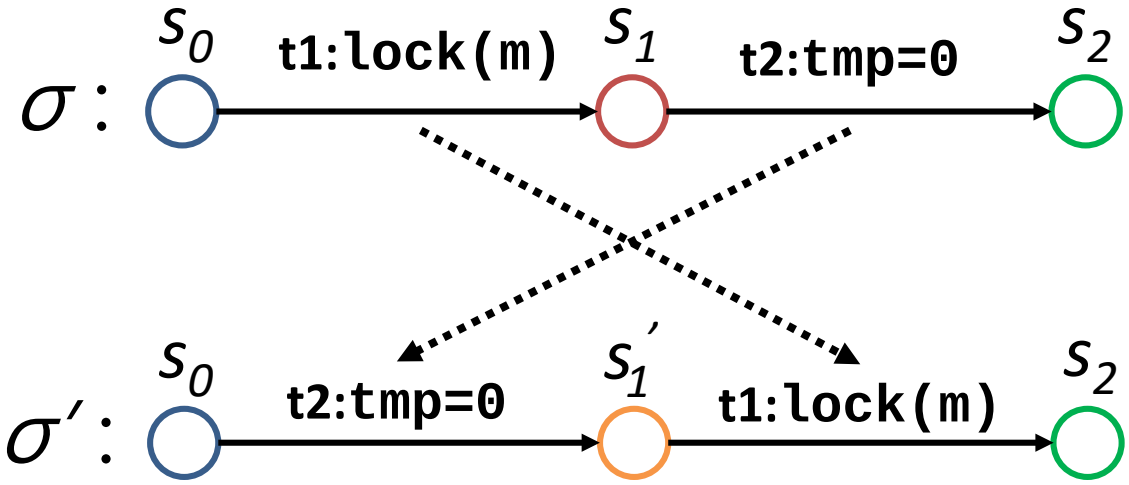
Atomicity Violation

- An atomicity violation (atomicity bug) is a race bug that violates the atomicity of an atomic block defined by programmers

- Atomicity violation detection techniques check if an observed execution is equivalent to any serial execution (i.e., serializable)
  - Reduction based technique
  - Access pattern based technique
  - Happens-before relation based techniques

Lipton's Reduction (1/2)

- An interleaved execution σ can be *reduced* to an equivalent interleaved execution σ' by swapping *commuting* operations.
- Two operations *a* and *b* in σ are *commuting* if
  - *a* and *b* are executed by different threads, and
  - *a* is immediately followed by *b, and*
  - their execution orders do **not** change the resulting state in an execution.



Lipton's Reduction (2/2)

- An operation *p* is *right-mover* if *p* commutes with the operation that immediately follows *p* and executed by another thread
- An operation *p* is *left-mover* if *p* is commutes with the operation immediately followed by *p* , executed by another thread
- An operation *p* is *both-mover* if *p* is right-mover and left-mover at the same time
  - *non-mover* if p is neither right-mover nor left-mover.

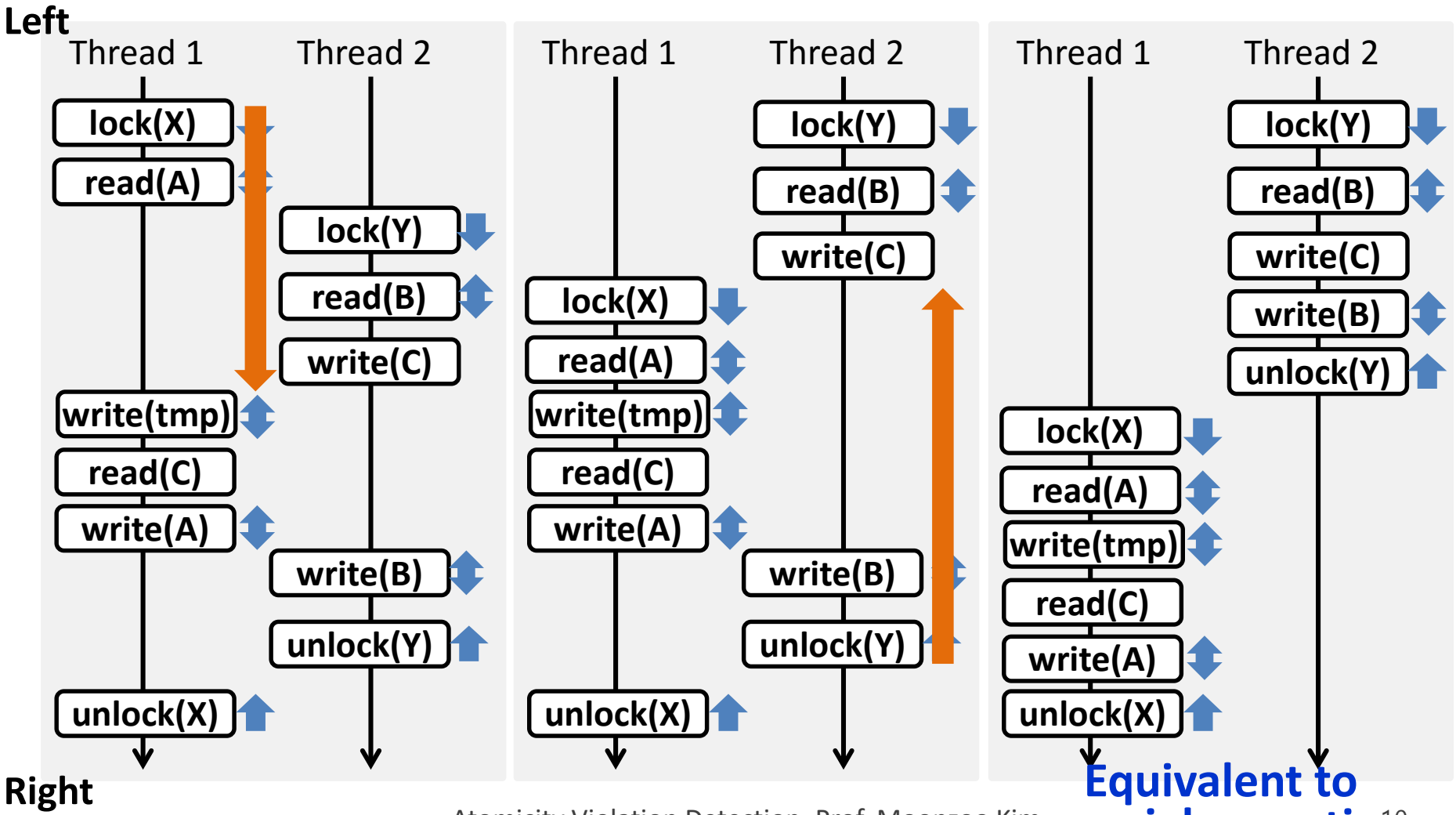
Atomizer [Flanagan, POPL 04]

- Atomizer checks if an observed execution σ can be reduced to an equivalent serial execution σ'
  - Atomizer assumes that every public methods are atomic blocks
  - Note that, in a serial execution, executions of atomic blocks do not overlap with each other.
- Reduction rule

Operation type	Right-mover and/or left-mover
lock(*m*) (i.e., lock *m* is held by a thread)	right-mover
unlock(*m*)	left-mover
read(*v*) or write(*v*) where *v* is thread-local	both-mover
read(*v*) or write(*v*) where *v* is shared, and consistently guarded by a lock	both-mover
read(*v*) or write(*v*) where *v* is shared, and not consistently guarded by a lock	non-mover

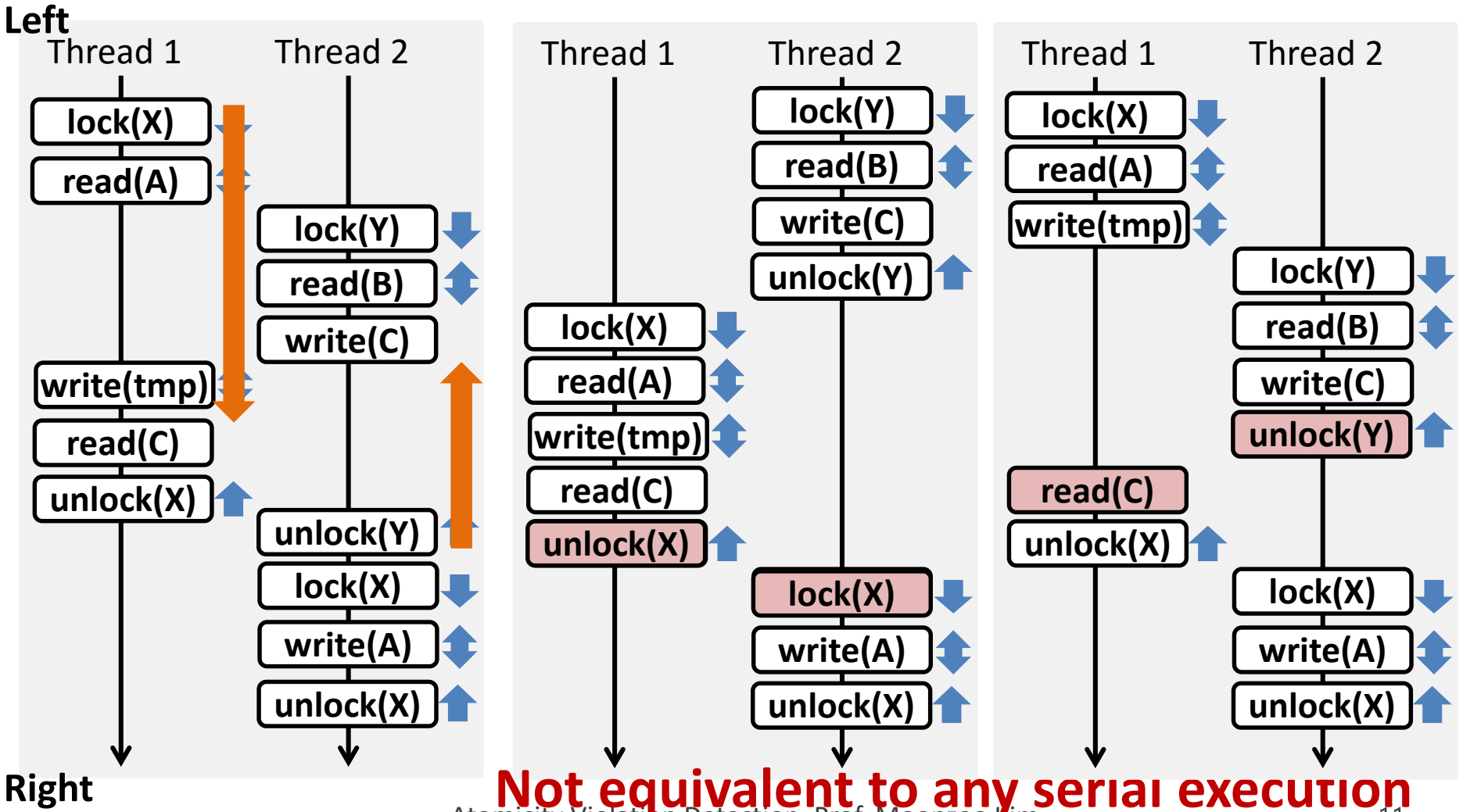
Serializable Execution Example

- Variable A and B are consistently guarded by lock X and Y, respectively
- Variable C is not consistently guarded by any lock



Non-serializable Execution Example

- Variables A and B are consistently guarded by lock X and Y, respectively
- Variable C is not consistently guarded by any lock



Atomicity Checking

- Atomizer's serializability checking with the reduction is the same as checking if every transaction is in the following pattern:

$$[\textit{Right-mover}]^* \; [\textit{Non-mover}]^? \; [\textit{Left-mover}]^*$$

  - e.g: Two-phase locking

```
lock(X)
  lock(Y)
    lock(Z)
    /* data accesses */
    unlock(Z)
  unlock(Y)
unlock(X)
```

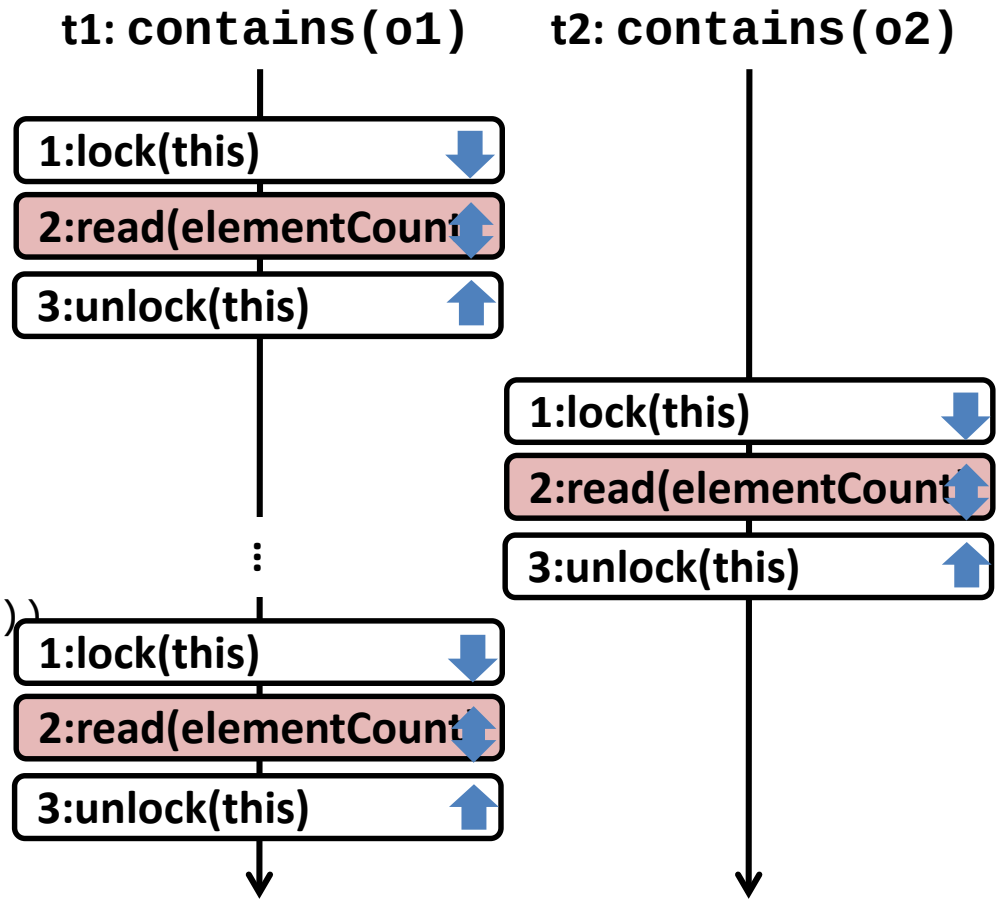
Expanding phase

Shrinking phase

L. Wang and S. D. Stoller: Runtime Analysis of Atomicity for Multithreaded Programs, IEEE TSE, 32(2), 2006

False Positive of Atomizer

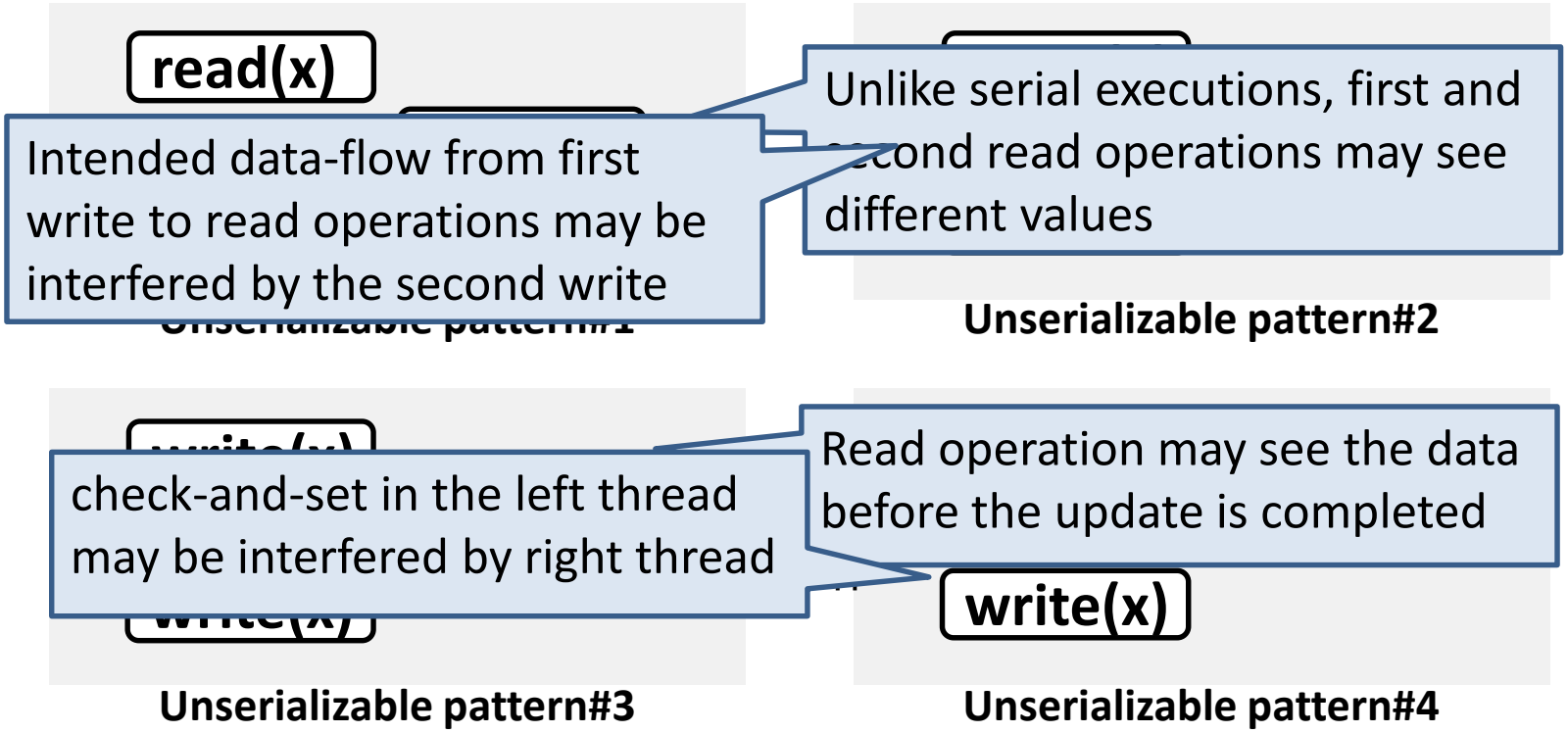
```
class Vector {
  Object [] elementData;
  int elementCount ;

  int size(){
1: synchronized(this){
2:  return elementCount;
3: }
  }

  boolean contains(Object x){
   //atomic block begins
4: int i=0;
5: while(i < size()){
6:  synchronized(this){
7:   if(elementData[i].equals(x))
8:    return true;
9:  }
10: i++;
11: }
12: return false;
   //atomic block ends
  }
}
```



Detected as atomicity violation, but "semantically" serializable

Access Pattern Based Detection

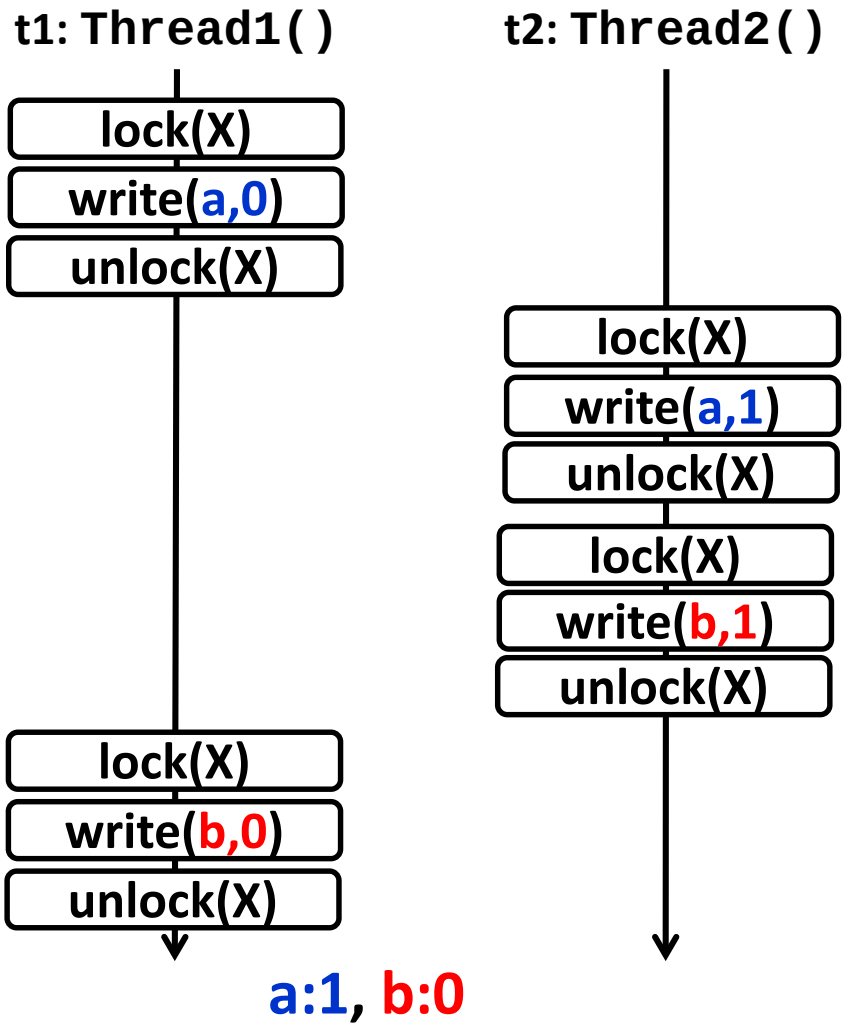
- Detect interleaved accesses on a shared variable that match to non-serializable patterns as atomicity violations
  - Select 4 among total 8 patterns as buggy based on experiences in real-bugs



S. Lu et al.: AVIO: Detecting Atomicity Violations via Access Interleaving Invariants, ASPLOS 2006

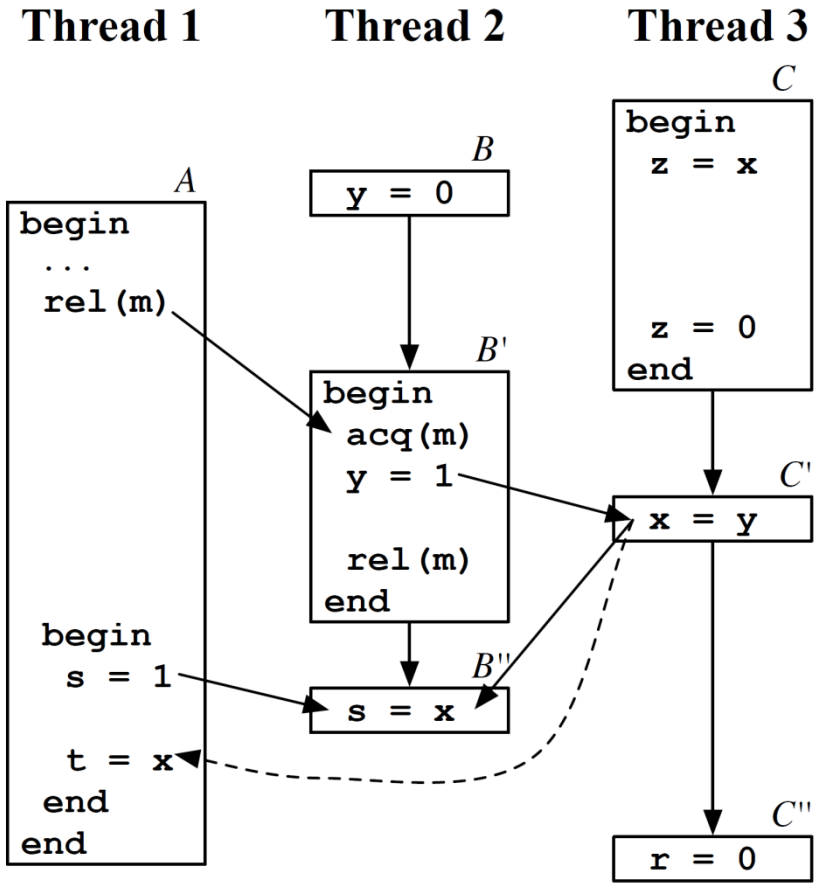
False Negative of Access Patterns

```
Thread1(){
 //atomic block begins
 synchronized(X){
  a = 0 ; }
 synchronized(Y){
  b = 0 ; }
 //atomic block ends
}

Thread2(){
 //atomic block begins
 synchronized(X){
  a = 1 ; }
 synchronized(Y){
  b = 1 ; }
 //atomic block ends
}
```



Velodrome [Flanagan, PLDI 08]

- Velodrome defines happens-before relation over atomic block executions
- Veldorome detects an inconsistency in the happens-before relation (i.e. cycle) as an atomicity violation



Happens-Before in Transactions (1/3)

- An *execution* is a sequence of operations

- A *transaction* in an execution is the sequence of operations in a thread, that correspond to an execution of an atomic block
  - Users can specify where an atomic block starts/ends in code

- An execution is *serial* if each transaction's operation execute contiguously, without interleaving

Happens-Before in Transactions (2/3)

- Two operations in an execution *conflict* if
  - these access the same variable, and at least one is writing, or
  - these operate on the same lock, or
  - these are executed in the same thread

  → If two operations do not conflict, they *commute*

- The happens-before relation over operations ($\prec$) in an execution satisfies the following properties:
  - $a \prec b$ if a occurs before b in the execution, and a conflicts with b
  - $a \prec b$ if a occurs before b, and a and b are in the same transaction
  - $a \prec c$ if $a \prec b$ and $b \prec c$

Happens-Before in Transactions (3/3)

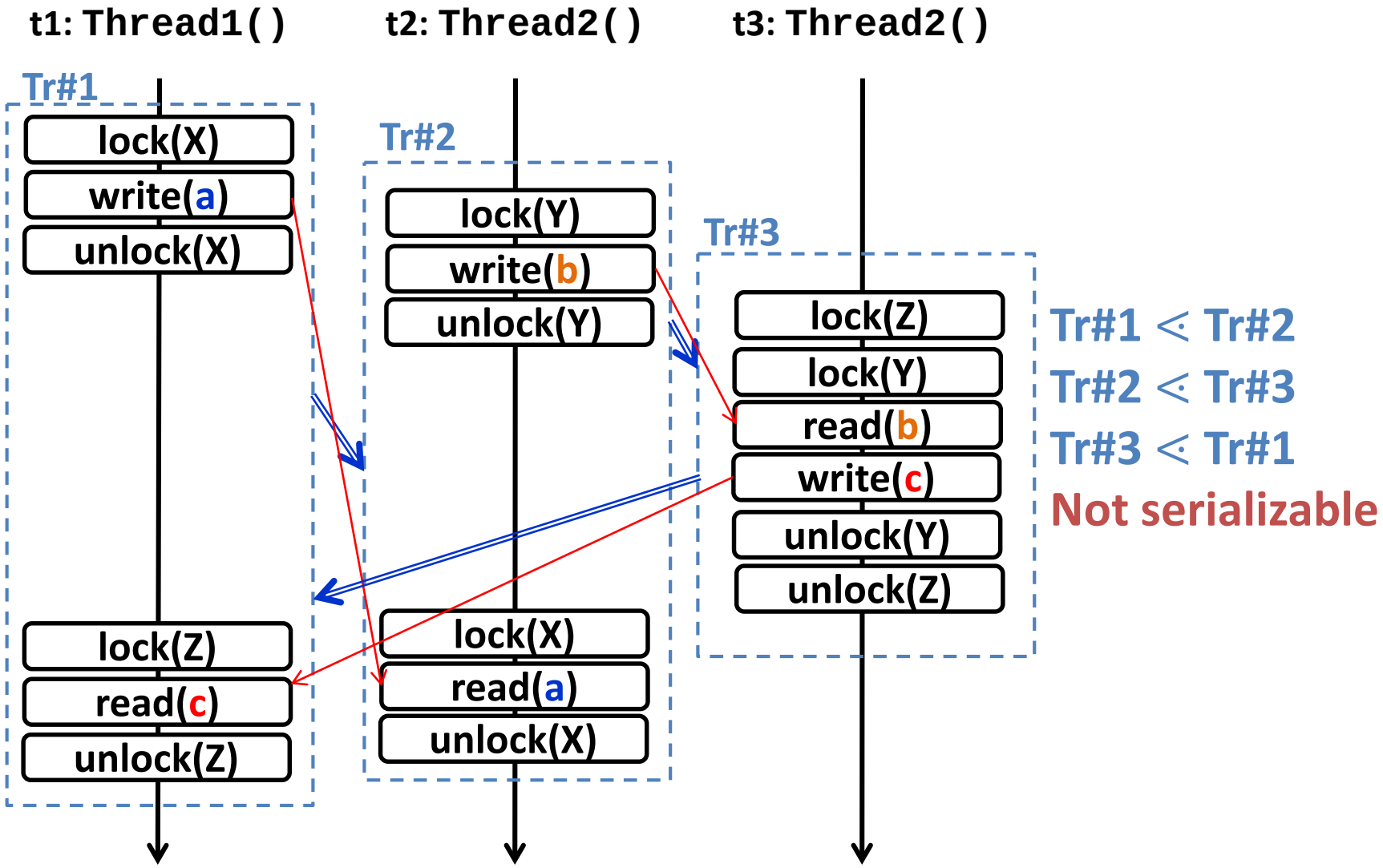
- A transaction P happens-before a transaction Q in an execution ($P \lessdot Q$) if
  - $P \neq Q$, and
  - $\exists\, p \in P, \exists q \in Q\colon\ p \prec q$

- An execution is serializable if and only if the transactional happens-before relation ($\lessdot$) is acyclic*

- A blame assignment is a set of happens-before relation over operations that construct a cycle in the transactional happens-before relation

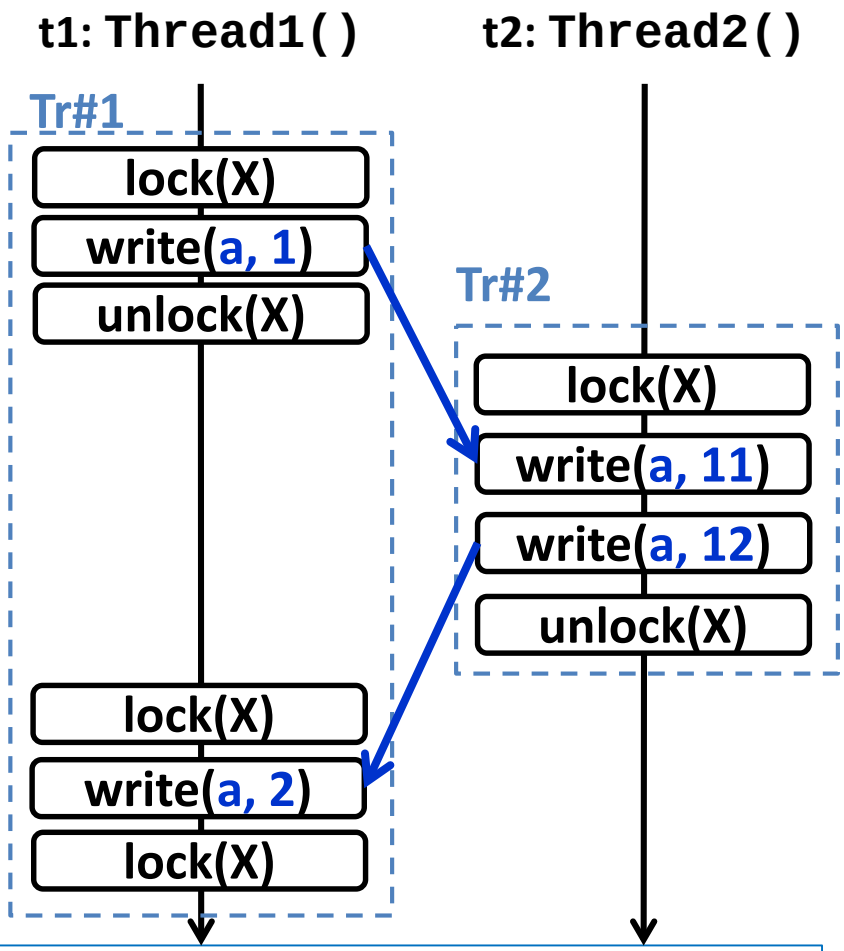
* P. A. Bernstein et al., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987

Example


t1: Thread1()
t2: Thread2()
t3: Thread2()
Tr#1
lock(X)
write(a)
unlock(X)
lock(Z)
read(c)
unlock(Z)
Tr#2
lock(Y)
write(b)
unlock(Y)
lock(X)
read(a)
unlock(X)
Tr#3
lock(Z)
lock(Y)
read(b)
write(c)
unlock(Y)
unlock(Z)
Tr#1 ≺ Tr#2
Tr#2 ≺ Tr#3
Tr#3 ≺ Tr#1
Not serializable

False Positive of Velodrome

```
Thread1(){
 //atomic block begins
 synchronized(X){
  a = 1}
 synchronized(X){
  a = 2 ; }
 //atomic block ends
}

Thread2(){
 //atomic block begins
 synchronized(X){
  a = 11 ;
  a = 12 ; }
 //atomic block ends
}
```



Execution order of write operations on a variable without read operations do not change result states, except for the last one

View-Serializability (1/2)

- Conflict-serializability (used by Velodrome)

  An execution is conflict-serializable if there exists a serial execution s.t.
  (1) the two executions have the same operations, and
  (2) for every pair of conflicting operations, the two operations have the same order in both executions

- View-serializability*

  An execution is view-serializable if there exists a serial execution s. t.
  (1) two executions have the same operations, and
  (2) each read operation has the same write predecessor in both executions, and
  - The write-predecessor of a read operation is the last write operation on the variable that the read operation reads in an execution

  (3) each variable has the same final write operation in both executions

*L. Wang et al.: Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs, PPoPP 2006

View-Serializability (2/2)

- We can implement the view-serializability checking by modifying the conflict definition
  - Two operations *p* and *q* in an execution *conflict* if
    - these access the same variable where
      - *p* is writing, and *q* is reading, or
      - *q* is the final write operation on the variable in the execution
    - these operate on the same lock, or
    - these are executed in the same thread