Goldilocks: A Race and Transaction-Aware Java Runtime

Tayfun Elmas, Serdar Tasiran
Koç University, İstanbul, Turkey

Shaz Qadeer
Microsoft Research, Redmond, WA

ACM SIGPLAN PLDI'07
June 12, San Diego, CA

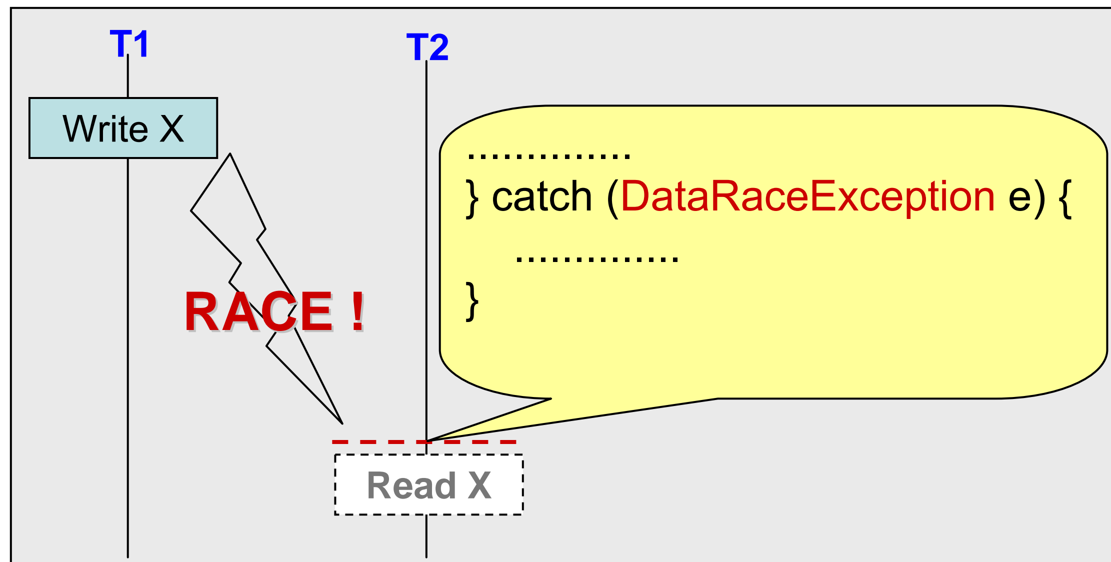
DataRaceException

Java Memory Model: Well-synchronized programs are free of races

- Run with sequential consistency semantics

DataRaceException: Runtime exception for race conditions

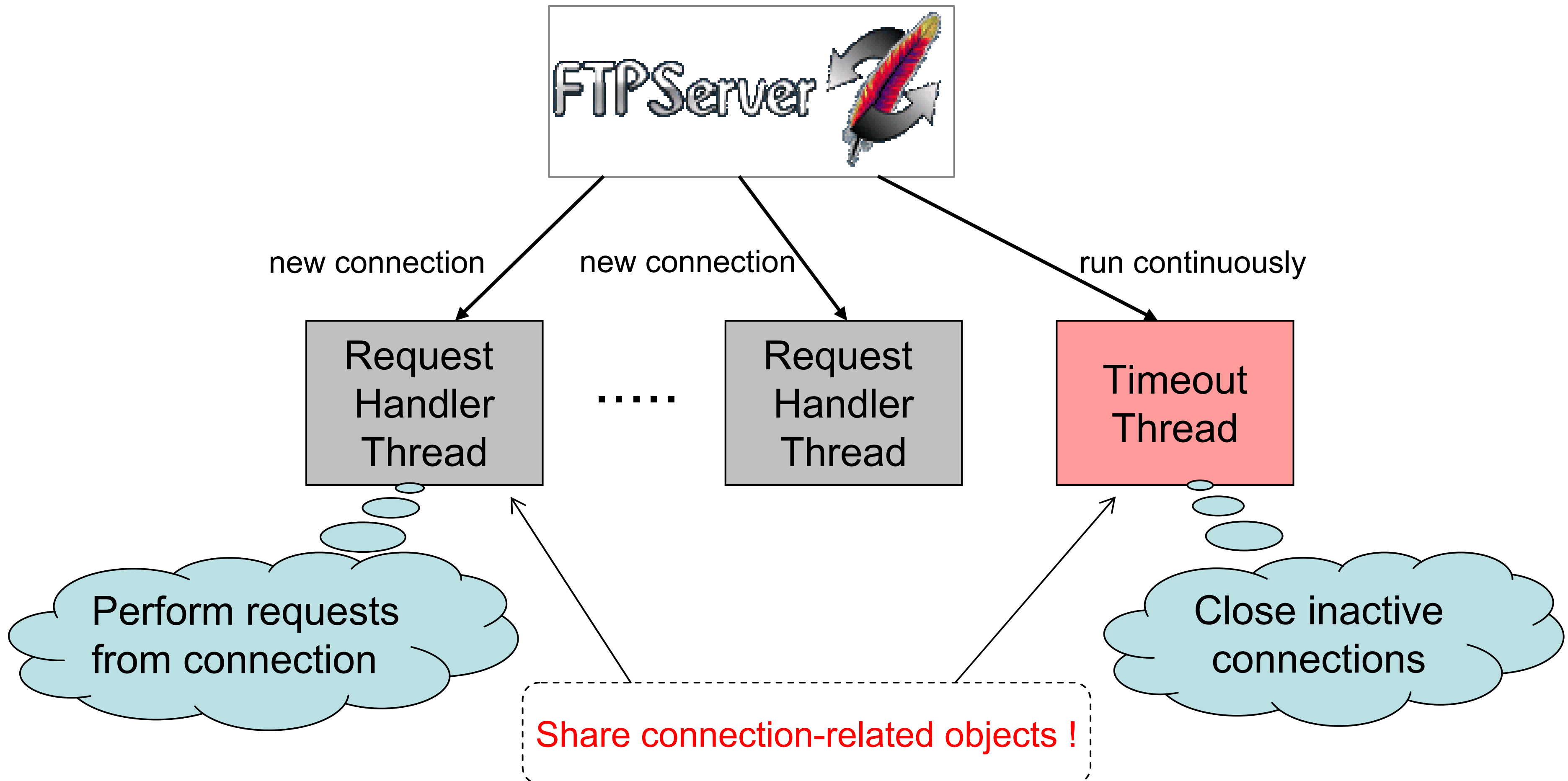
- Brings actual race condition to attention of programmer



1. Ensures sequential consistency of executions

2. Debugging tool

Handling DataRaceException



Request Handler Thread:

```
// Initialize connection
............
while ( ! m_isConnectionClo
  String command = m_reader.readLine();
  m_request.parse(command);
  if ( ! hasPermission ( ) ) {
    m_writer.send(530, "permission", null);
    continue;
   }
   // execute command
   service(m_request, m_writer);
}
```

NullPointerException !

Timeout Thread:

```
// Close inactive connection
(this) {
ectionClosed)
    return;
  m_isConnectionClosed = true;
}
...
m_request = null;
m_writer = null;
m_reader = null;
....
```

Request Handler Thread:

```
// Initialize connection
............
while ( ! m_isConnectionClosed) {
  String command = m_reader.readLine();
  m_request.parse(command);
  if ( ! hasPermission ( ) ) {
    m_writer.send(530, "permission", null);
    continue;
  }
- - - - - - - - - - - - - - - - - - - - - - - -
  // execute command
  service(m_request, m_writer);
}
```

NullPointerException !

(Far from source of bug)

Timeout Thread:

```
// Close inactive connection
synchronized(this) {
  if(m_isConnectionClosed)
    return;
  m_isConnectionClosed = true;
}
...
m_request = null;
m_writer = null;
m_reader = null;
....
```

Request Handler Thread:

```
............
try {
while ( ! m_isConnectionClosed) {
  String command = m_reader.readLine();
  m_request.parse(command);
  if ( ! hasPermission ( ) ) {
    m_writer.send(530, "permission", null);
    continue;
  }
  // execute command
  service(m_request, m_writer);
}
} catch (DataRaceException e) {
   addToLog ("Connection closed:" +
           e.getAccessedObject().toString());
   currentThread.stop ();
}
```

Timeout Thread:

```
// Close inactive connection
synchronized(this) {
  if(m_isConnectionClosed)
    return;
  m_isConnectionClosed = true;
}
...
m_request = null;
m_writer = null;
m_reader = null;
....
```

Race-aware runtime

What we need: Dynamic race-detection algorithm

- Integrated to Java virtual machine
- Sound, precise and efficient checking

Dynamic race-detection algorithms in the literature

- Lockset-based: Efficient but not precise
  – Reports false alarms
- Vector clock-based: Precise but not efficient
  – Generally not scalable with number of threads

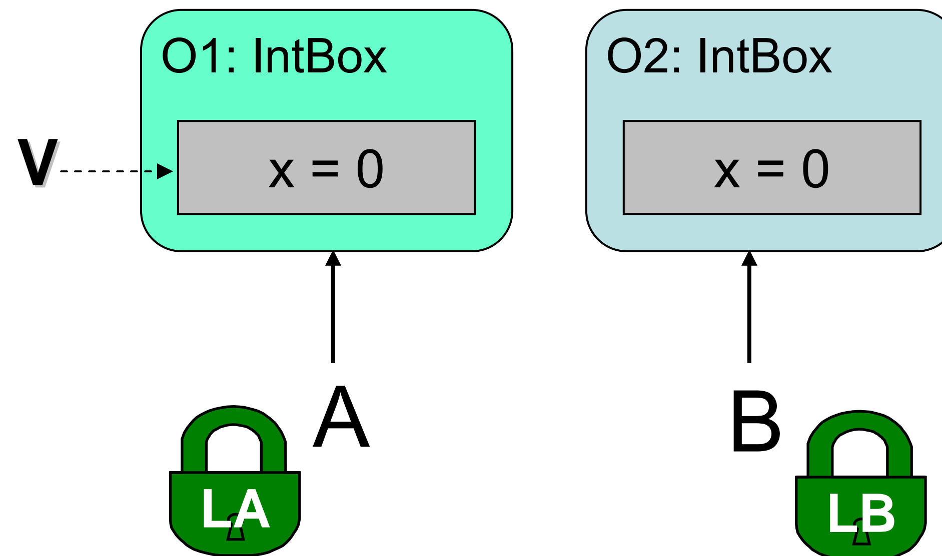
Goldilocks

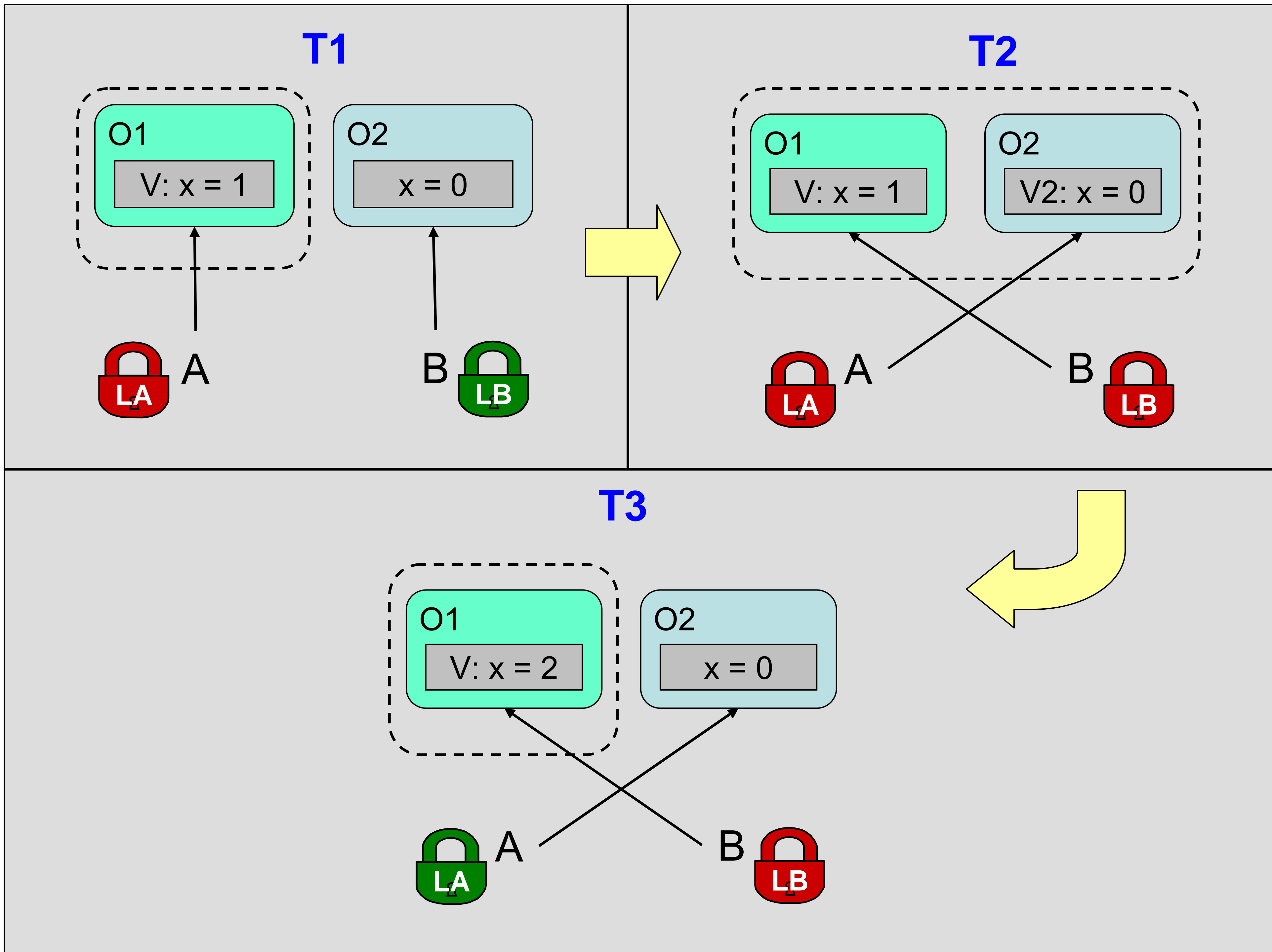
Novel lockset-based race detection algorithm

- Exactly captures JMM happens-before relation
  - Sound: Detects all actual races
  - Precise: No false alarms
- Uniformly handles all synchronization disciplines
  - Extended to software transactional memory
- Efficient: Many implementation features and optimizations
  - Implementations in Kaffe JVM and Chess (Microsoft Research)

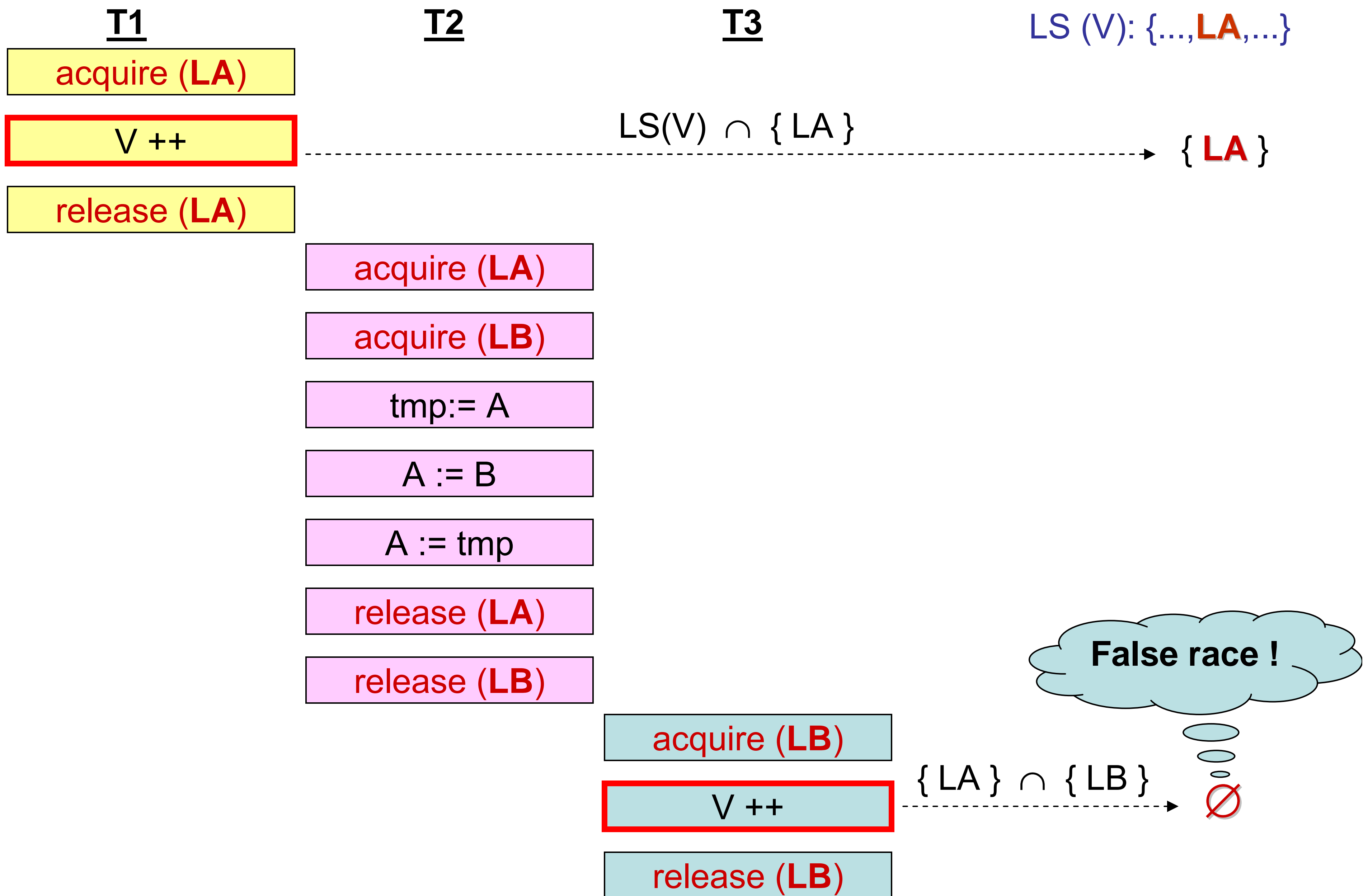
Example: Container-protected objects

```
class IntBox {
    int x;
}
```

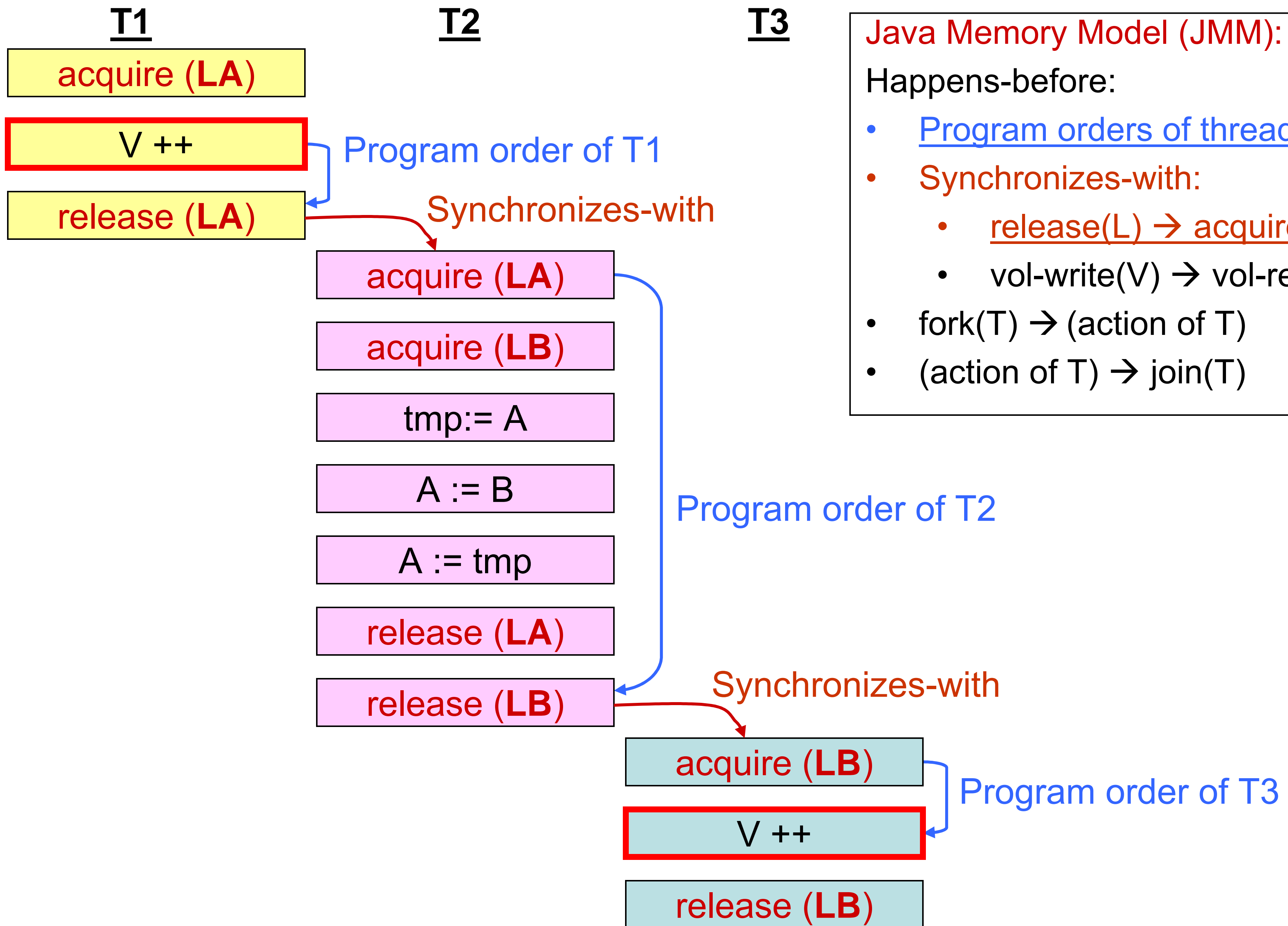



T1
O1
V: x = 1
O2
x = 0
LA
A
B
LB
T2
O1
V: x = 1
O2
V2: x = 0
LA
A
B
LB
T3
O1
V: x = 2
O2
x = 0
LA
A
B
LB

Eraser locksets



Why there is no race



Java Memory Model (JMM):

Happens-before:

- Program orders of threads
- Synchronizes-with:
  - release(L) → acquire(L)
  - vol-write(V) → vol-read(V)
- fork(T) → (action of T)
- (action of T) → join(T)

Goldilocks' intuition

Locksets contain

- Thread ids: $T \in LS(V)$ $\Rightarrow$ T is an owner of V

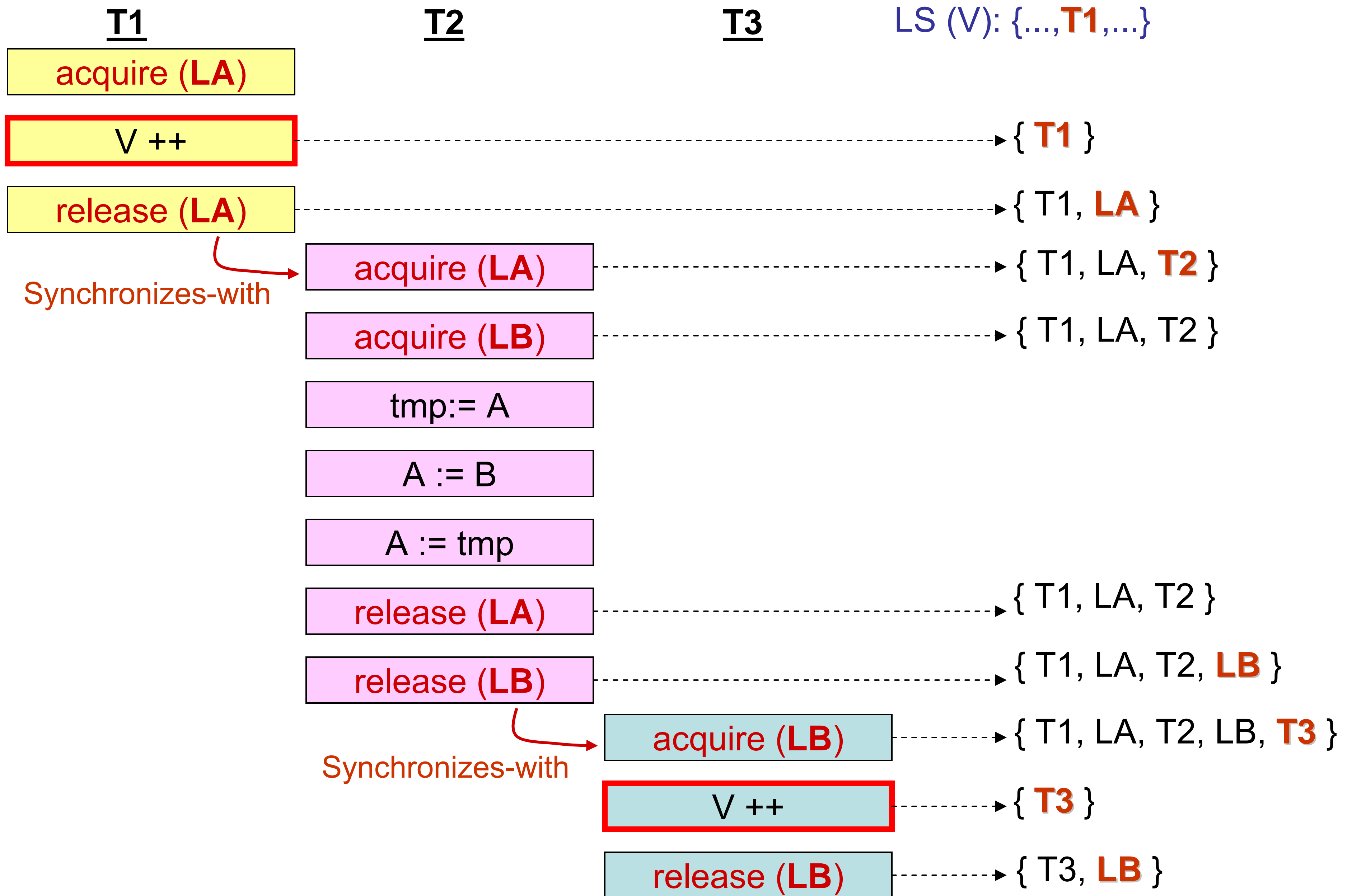
  $\Rightarrow$ T's accesses to V are race-free
- Locks: $L \in LS(V)$ $\Rightarrow$ L is a protecting lock
- Volatiles: $W \in LS(V)$ $\Rightarrow$ W is a synchronization variable

Locksets both shrink and grow

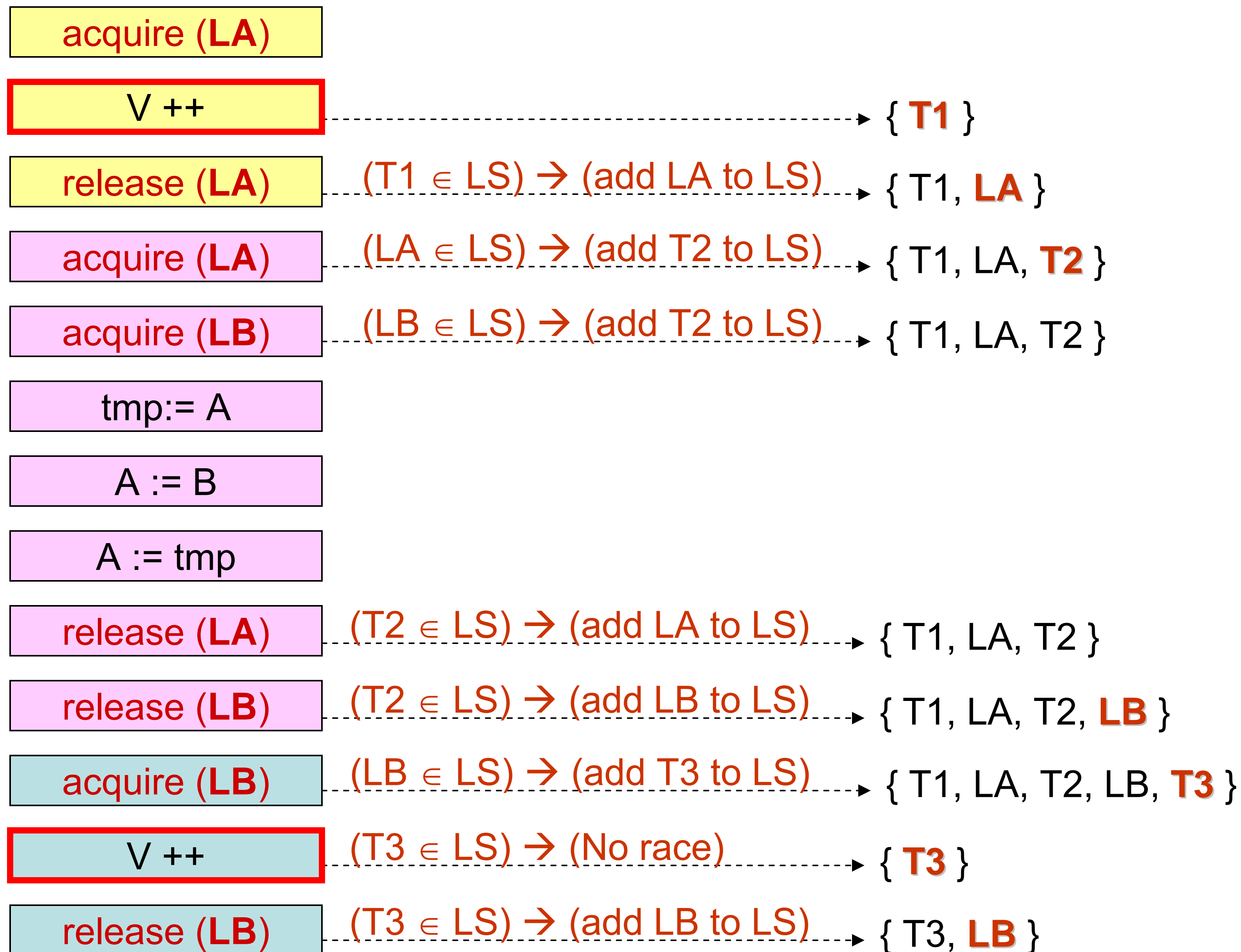
- After T accesses V: LS(V) = { T } → (T becomes the only owner)
- Synchronization operations by owners grows LS(V)
  - Example: release(L) → add L to lockset

Theorem [Soundness and precision]:
Goldilocks exactly captures happens-before in Java Memory Model

Goldilocks locksets



Goldilocks' lockset update rules



Goldilocks handles various scenarios uniformly

- Dynamically changing locksets
  - Container-protected objects
- Thread-locality
  - Permanent and temporary
- Fork/Join synchronization
- Wait/Notify(All)
  - No additional lockset update rules
- Volatile variables
  - Conditional branches on volatiles
- java.util.concurrent
  - Semaphores, barriers, ...

Implementation

Integrated into Kaffe JVM [kaffe.org]

Goldilocks allows efficient implementations:

1. Global event list

- Implicit, shared representation of locksets

2. Lazy evaluation of locksets

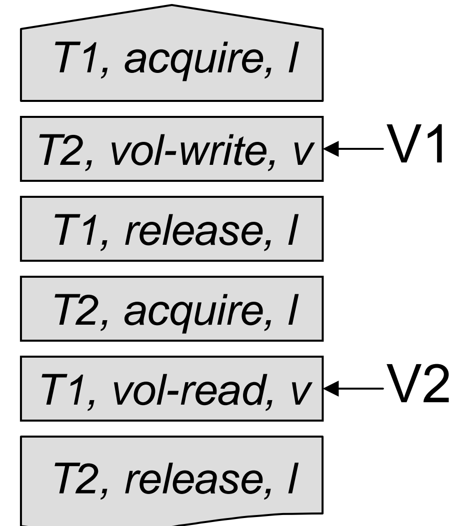
- Update and check lockset only at variable access

3. Short-circuit checks

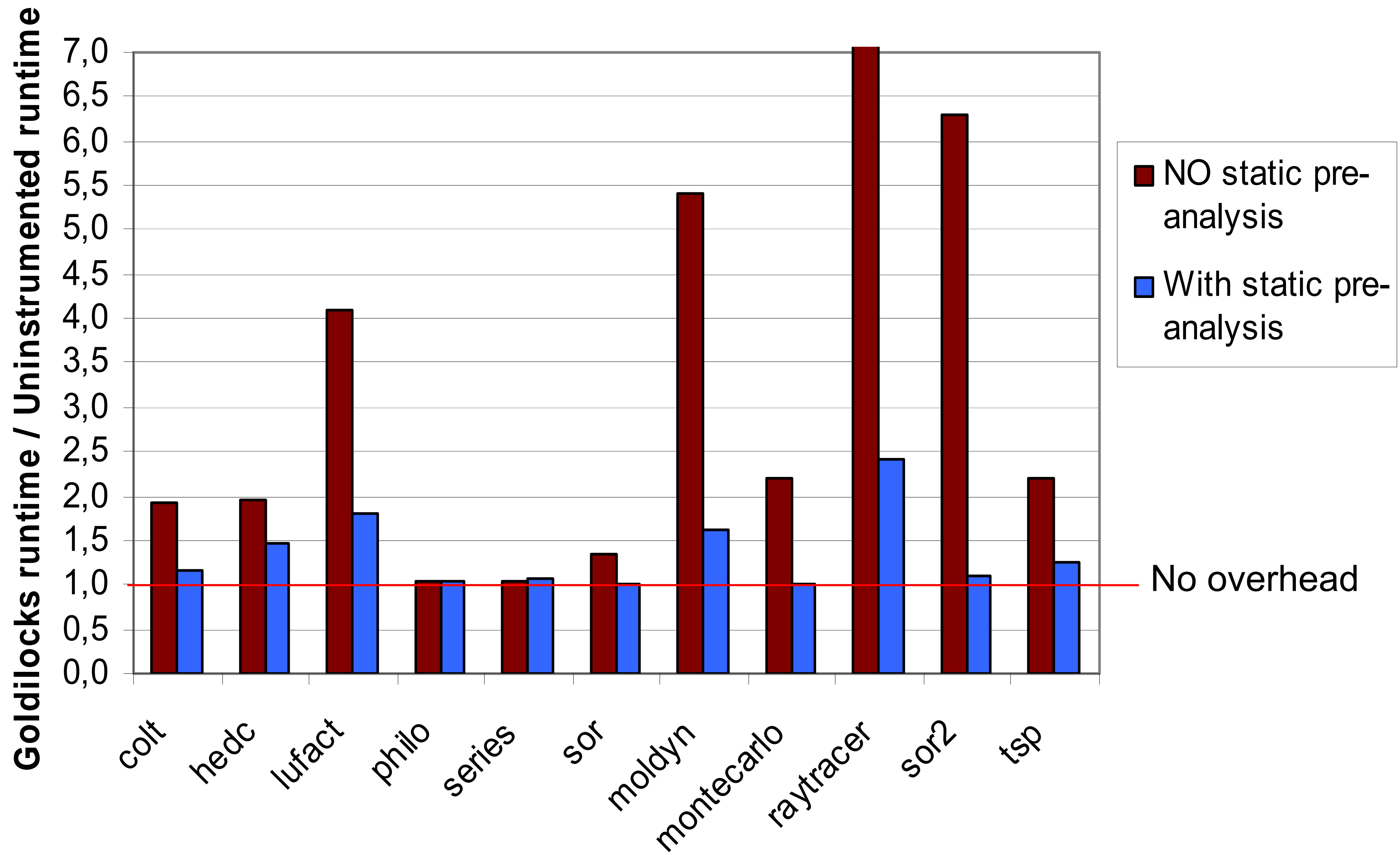
- Sufficient, constant-time checks for happens-before
- If success → Skip lockset computation (success rate: 30-90 %)

4. Static pre-analysis

- Do not check race-free accesses if proven statically
- Chord [Naik et. al., PLDI'06] and RccJava [Abadi et. al., ACM TOPLAS'06]



Goldilocks' performance



Transaction-aware runtime

What is a race when there are transactions ?

- Transactions coexist with other synchronization mechanisms
- Accesses to same shared variables inside and outside transactions

→ Extended definition of races

- Transactions as high-level language and synchronization primitive
  - No reference to internal synchronization of transactions at all

Based on extended happens-before relation

1. Existing JMM happens-before edges
2. Extra happens-before edges between transactions
   - From transaction semantics [Grossman et.al., MSPC'06]

New lockset update rules to capture extra happens-before edges

- High-level information from transaction package to Goldilocks

Contributions

DataRaceException:

- React to races before they happen
- Ensure sequential consistency semantics of programs

Goldilocks:

- Sound, precise, efficient race detection
- Naturally handles all synchronization disciplines

Goldilocks checks executions with software transactions

- Races redefined for transactional memory
- Independent of transaction implementation