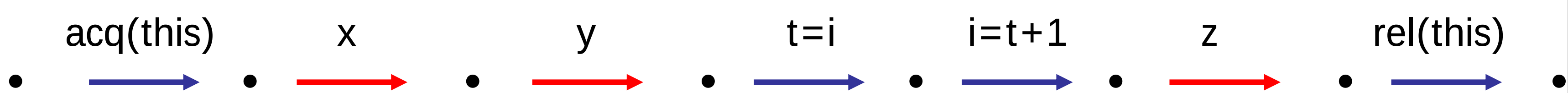
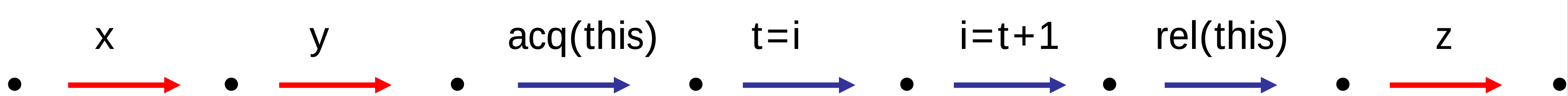
Atomicity

- The method inc() is atomic if concurrent threads do not interfere with its behavior

- Guarantees that for every execution



- there is a *serial* execution with same behavior



Tools for Checking Atomicity

- Calvin: ESC for multithreaded code
  - [Freund-Qadeer 03]

Experience with Calvin

```
 /*@ global_invariant (\forall int i; inodeLocks[i] == null ==>
                       0 <= inodeBlocknos[i] && inodeBlocknos[i] < Daisy.MAXBLOCK) */
//@ requires 0 <= inodenum && inodenum < Daisy.MAXINODE;
//@ requires i != null
//@ requires DaisyLock.inodeLocks[inodenum] == \tid
//@ modifies i.blockno, i.size, i.used, i.inodenum
//@ ensures i.blockno  == inodeBlocknos[inodenum]
//@ ensures i.size     == inodeSizes[inodenum]
//@ ensures i.used     == inodeUsed[inodenum]
//@ ensures i.inodenum == inodenum
//@ ensures 0 <= i.blockno && i.blockno < Daisy.MAXBLOCK

static void readi(long inodenum, Inode i) {
        i.blockno = Petal.readLong(STARTINODEAREA + (inodenum * Daisy.INODESIZE));
        i.size     = Petal.readLong(STARTINODEAREA +  (inodenum * Daisy.INODESIZE) + 8);
        i.used    = Petal.read(STARTINODEAREA + (inodenum * Daisy.INODESIZE) + 16) == 1;
        i.inodenum = inodenum;
        // read the right bytes, put in inode
}
```

Tools for Checking Atomicity

- Calvin: ESC for multithreaded code
  - [Freund-Qadeer 03]

- A type system for atomicity
  - [Flanagan-Qadeer 03,Flanagan-Freund-Lifshin 05]

Tools for Checking Atomicity

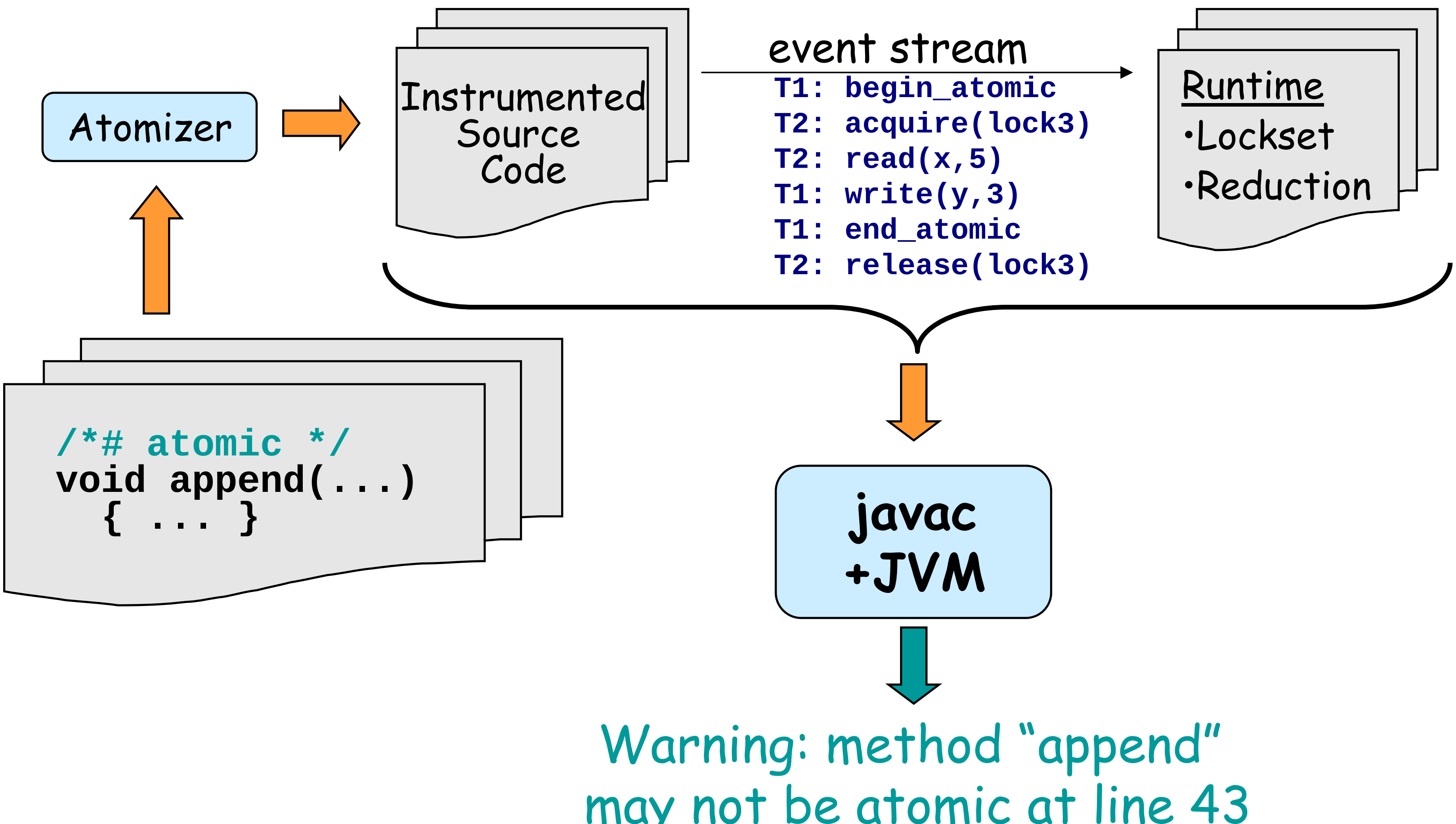
- Calvin: ESC for multithreaded code
  - [Freund-Qadeer 03]
- A type system for atomicity
  - [Flanagan-Qadeer 03,Flanagan-Freund-Lifshin 05]
- Atomizer: dynamic atomicity checker
  - [Flanagan-Freund 04]

`http://www.soe.ucsc.edu/~cormac/atom.html`

Atomizer


OFF!

Atomizer: Instrumentation Architecture



Atomizer: Dynamic Analysis

- Lockset algorithm
  - from Eraser [Savage et al. 97]
  - identifies race conditions

- Reduction [Lipton 75]
  - proof technique for verifying atomicity, using information about race conditions

Analysis 1: Lockset Algorithm

- Tracks *lockset* for each field
  - lockset = set of locks held on all accesses to field
- Dynamically infers protecting lock for each field
- Empty lockset indicates possible race condition

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

- First access to **`o.f`**:

LockSet(**`o.f`**) = Held(curThread)

= { x, y }

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

- Subsequent access to **`o.f`**:

$$\mathrm{LockSet}(\mathbf{o.f}) := \mathrm{LockSet}(\mathbf{o.f}) \cap \mathrm{Held}(\mathrm{curThread})$$

$$= \{ x, y \} \cap \{ x \}$$

$$= \{ x \}$$

Lockset Example

Thread 1

```
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

→

Thread 2

```
synchronized(y) {
  o.f = 2;
}
```

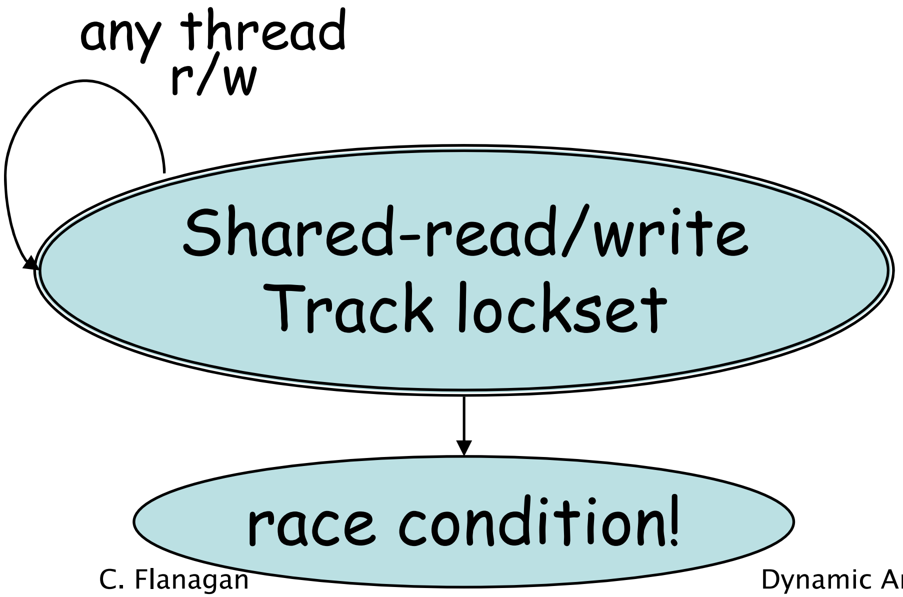
- Subsequent access to **`o.f`**:

$$\text{LockSet}(\mathbf{o.f}) := \text{LockSet}(\mathbf{o.f}) \cap \text{Held}(\text{curThread})$$

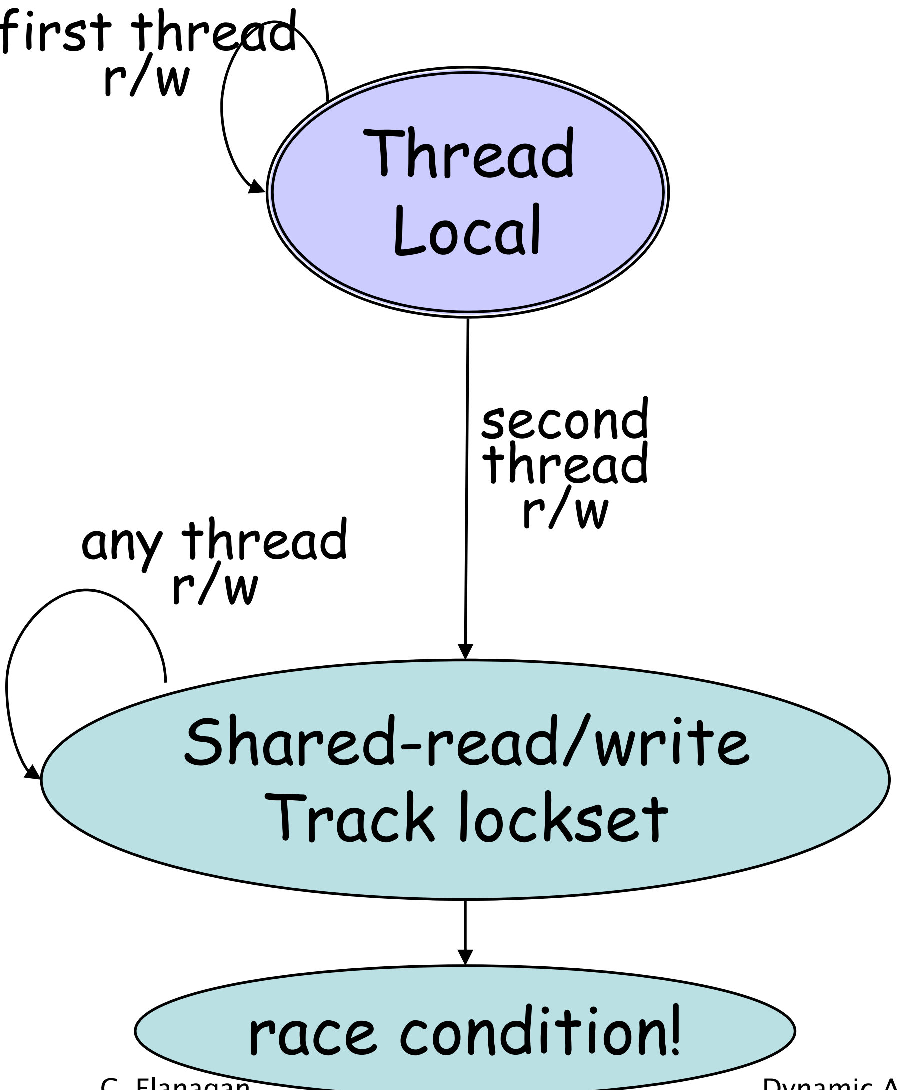
$$= \{ x \} \cap \{ y \}$$

$$= \{ \}$$ => race condition

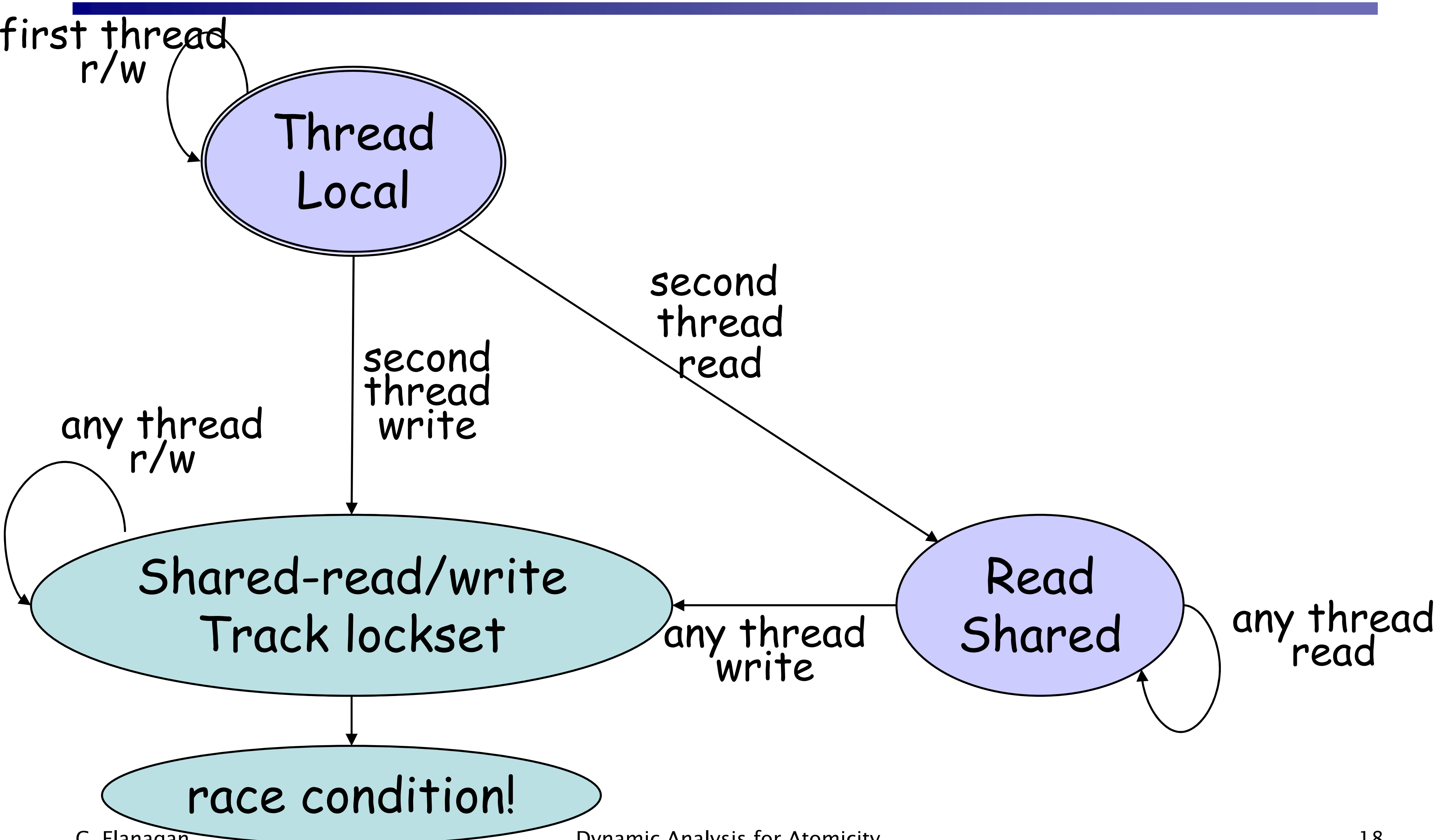
Lockset


any thread
r/w
Shared-read/write
Track lockset
race condition!

Lockset with Thread Local Data

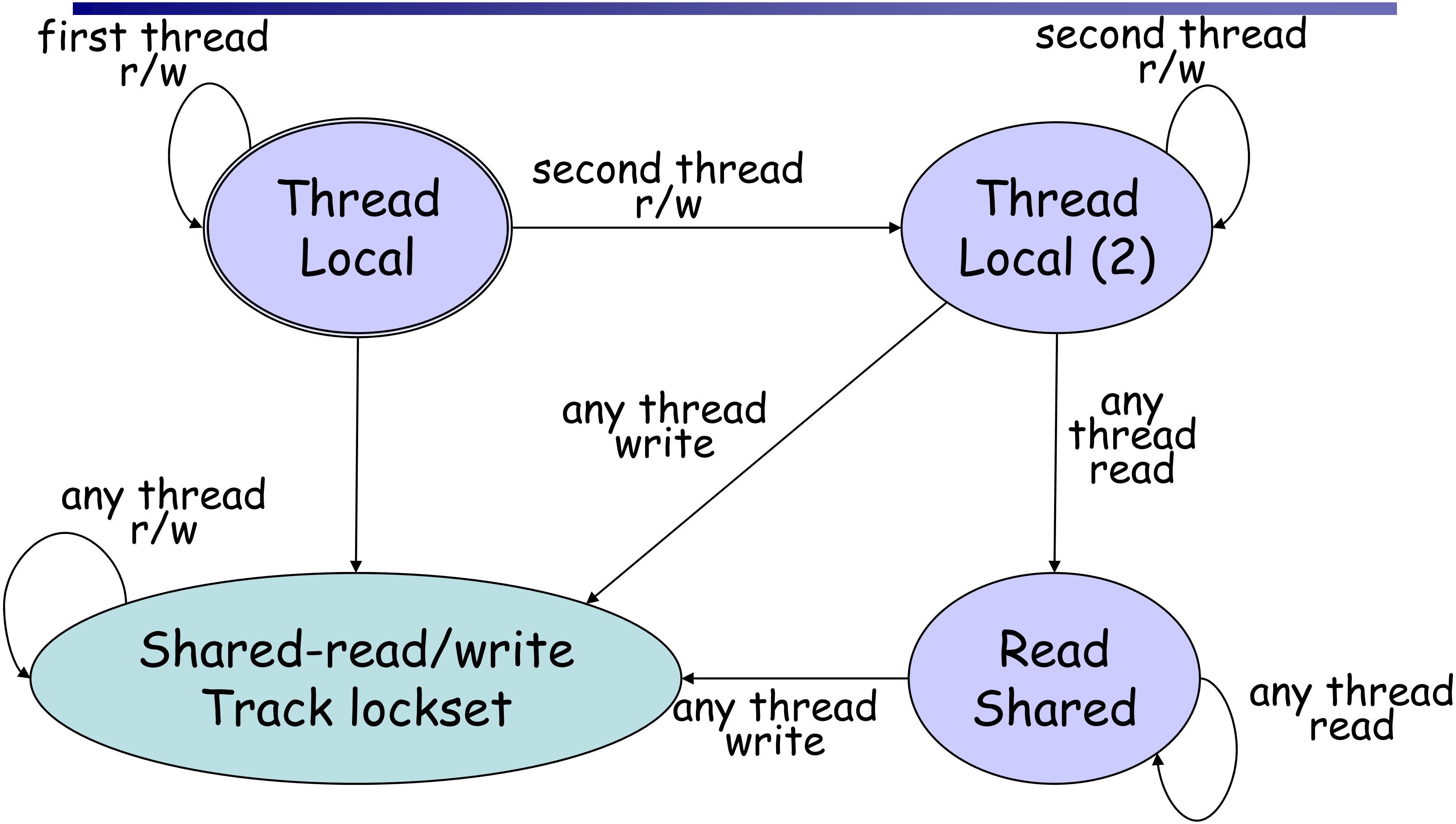


Lockset with Read Shared Data



Lockset for Producer-Consumer

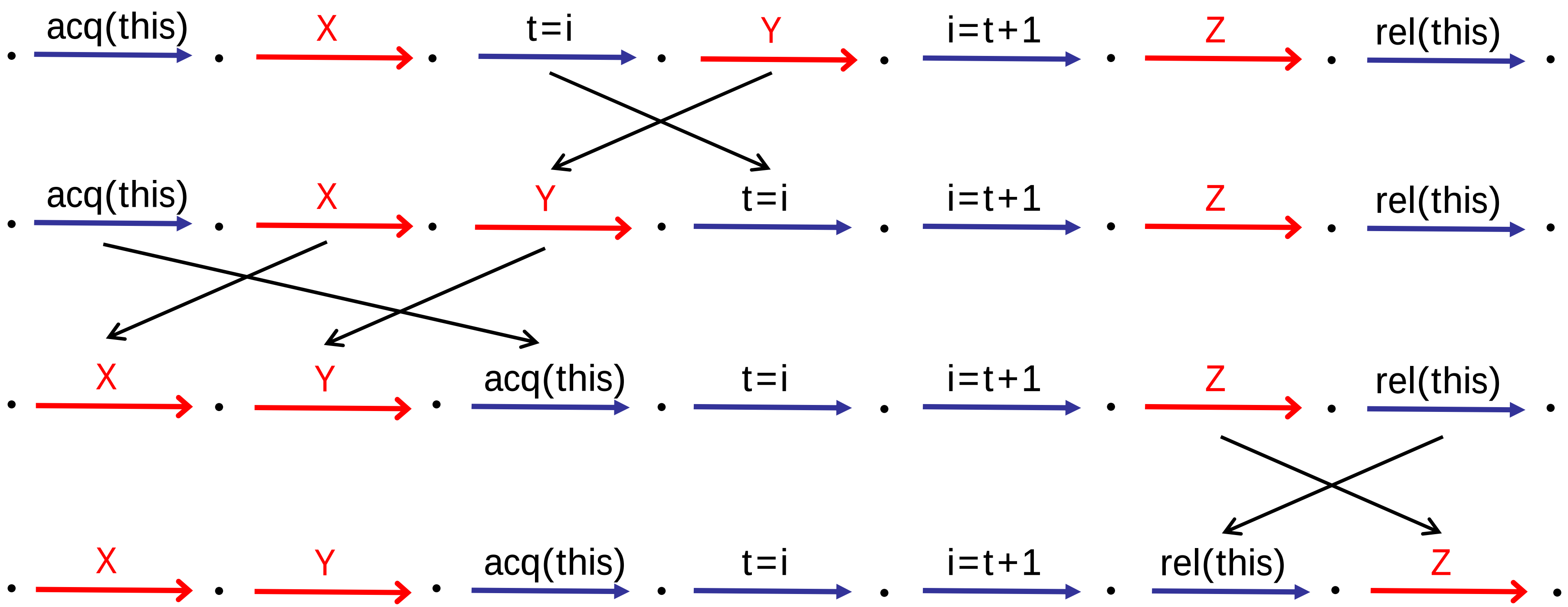
[Gross-Von Praun 01]



Atomizer: Dynamic Analysis

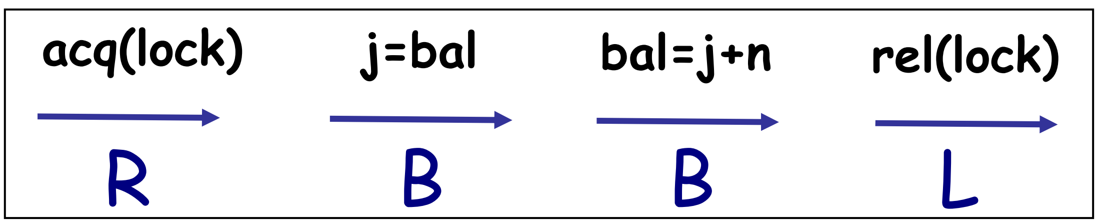
- Lockset algorithm
  - from Eraser [Savage et al. 97]
  - identifies race conditions
- Reduction [Lipton 75]
  - proof technique for verifying atomicity, using information about race conditions

Reduction [Lipton 75]

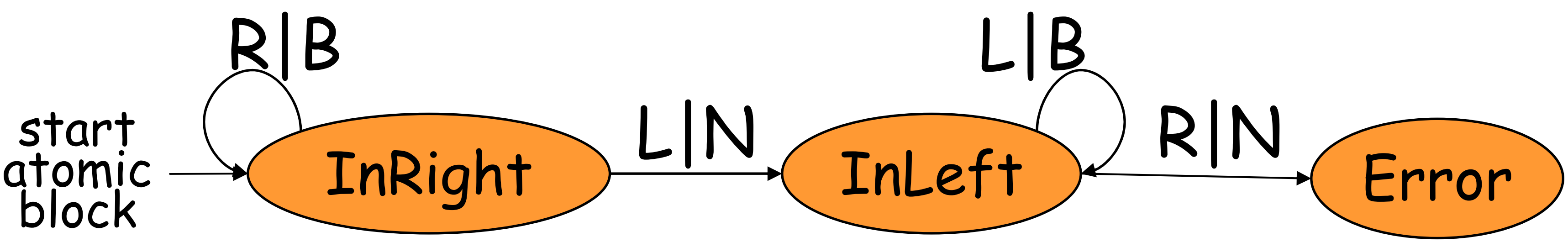


Performing Reduction Dynamically

- R: right-mover
  - lock acquire
- L: left-mover
  - lock release
- B: both-mover
  - race-free field access
- N: non-mover
  - access to "racy" fields



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



java.lang.StringBuffer

```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
  /*# atomic */
  public synchronized void append(StringBuffer sb){

    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(...,len,...);
    ...
  }
}
```

sb.length() acquires lock on sb, gets length, and releases lock

other threads can change sb

use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)

java.lang.StringBuffer

```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
  /*# atomic */
  public synchronized void append(StringBuffer sb){

    int len = sb.length();
    ...
    ...
    ...
    sb.getChars(...,len,...);
    ...
  }
}
```

```
StringBuffer.append is not atomic:
  Start:
    at StringBuffer.append(StringBuffe
    at Thread1.run(Example.java:17)

  Commit: Lock Release
    at StringBuffer.length(StringBuffe
    at StringBuffer.append(StringBuffe
    at Thread1.run(Example.java:17)

  Error: Lock Acquire
    at StringBuffer.getChars(StringBuf
    at StringBuffer.append(StringBuffe
    at Thread1.run(Example.java:17)
```

Atomizer Review

- Instrumented code calls Atomizer runtime
  - on field accesses, sync ops, etc
- Lockset algorithm identifies races
  - used to classify ops as movers or non-movers
- Atomizer checks reducibility of atomic blocks
  - warns about atomicity violations

Refining Race Information

- Discovery of races during reduction

```
    /*# atomic */
    void deposit(int n) {
R     synchronized (this) {
B       int j = bal;
        // other thread changes bal
A       bal = j + n;
L     }
```

Extensions

- Redundant lock operations
  - acquire is right-mover
  - release is left-mover
  - Want to treat them as both movers when possible

- Write-protected data
  - common idiom

Thread-Local Locks

```
class Vector {
  atomic synchronized Object get(int i) { ... }
  atomic synchronized void add(Object o) { ... }
}

class WorkerThread {

  atomic void transaction() {
    Vector v = new Vector();
    v.add(x1);
    v.add(x2);
    ...
    v.get(i);
  }
}
```

Reentrant Locks

```
class Vector {

  atomic synchronized Object get(int i) { ... }
  atomic synchronized Object add(Object o) { ... }

  atomic boolean contains(Object o) {
    synchronized(this) {
      for (int i = 0; i < size(); i++)
        if (get(i).equals(o)) return true;
    }
    return false;
  }
}
```

Layered Abstractions

```
class Set {
  Vector elems;

  atomic void add(Object o) {
    synchronized(this) {
      if (!elems.contains(o)) elems.add(o);
    }
  }
}
```

Redundant Lock Operations

- Acquire is right-mover
- Release is left-mover

- Redundant lock operations are both-movers
  - acquiring/releasing a thread-local lock
  - re-entrant acquire/release
  - acquiring/releasing lock A, if lock B always acquired before A

Write-Protected Data

```
class Account {
    volatile int bal;
    /*# atomic */ int read() { return bal; }
    /*# atomic */ void deposit(int n) {
R     synchronized (this) {
B       int j = bal;
N       bal = j + n;
L     }
    }
}
```

Write-Protected Data

```
class Account {
    int bal;
    /*# atomic */ int read() { return bal; }
    /*# atomic */ void deposit(int n) {
R     synchronized (this) {
B       int j = bal;
A       bal = j + n;
L     }
    }
}
```

- Lock **this** held whenever balance is updated
  - write must hold lock, and is non-mover
  - read without lock held is non-mover
  - read with lock held is both-mover

Lockset for Write-Protected Data

- Track *access* lockset and *write* lockset
  - access lockset = locks held on every access
  - write lockset = locks held on every write
- For regularly-protected data
  - access lockset = write lockset = { protecting lock }
- For write-protected data
  - access lockset = Ø
  - write lockset = { write-protecting lock }
- Read is both-mover if some *write* lock held
- Write is both-mover if *access* lockset nonempty

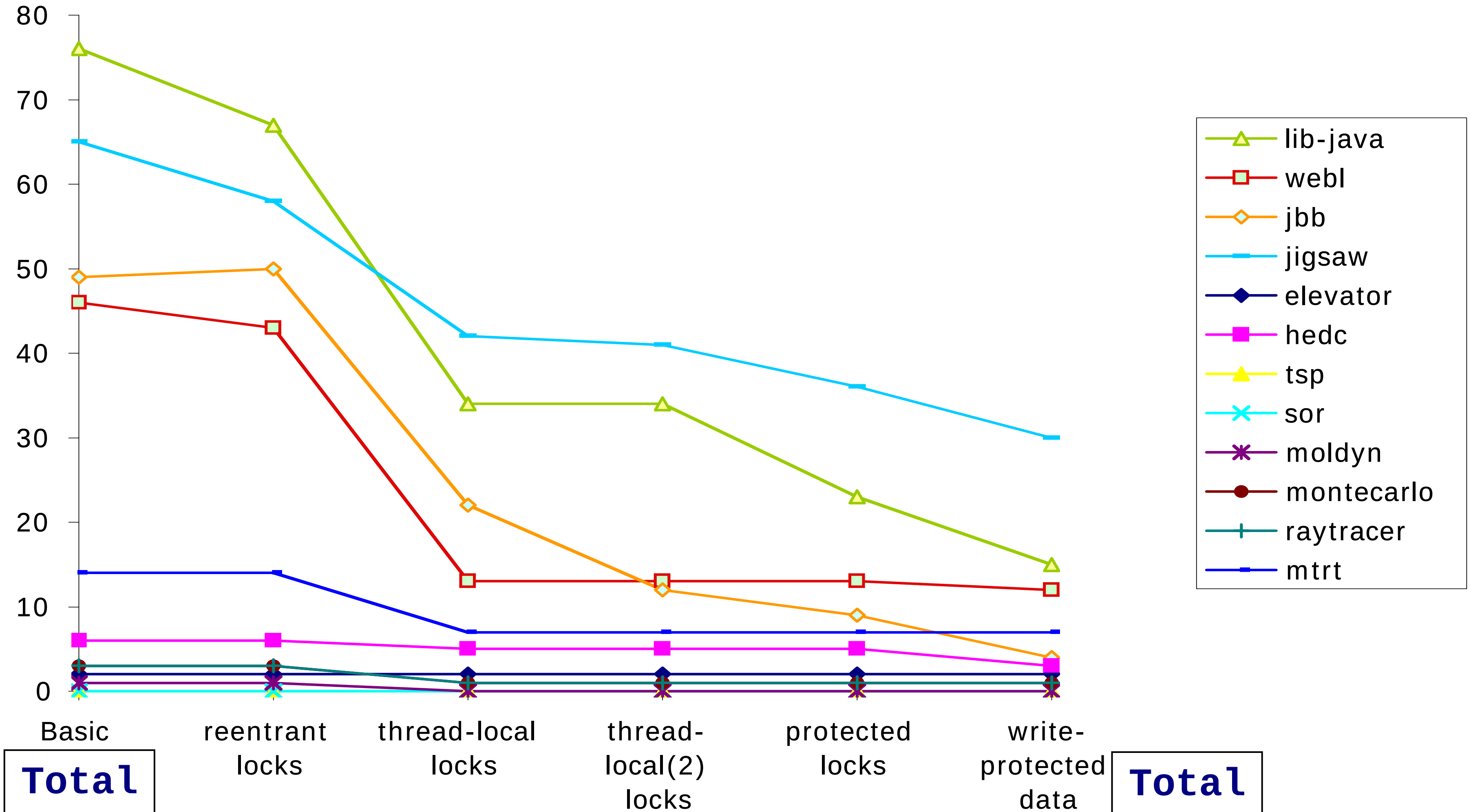
Evaluation

- 12 benchmarks
  - scientific computing, web server, std libraries, ...
  - 200,000+ lines of code
- Heuristics for atomicity
  - all synchronized blocks are atomic
  - all public methods are atomic, except `main` and `run`
- Slowdown: 1.5x – 40x

Performance

Benchmark	Lines	Base Time (s)	Slowdown
elevator	500	11.2	-
hedc	29,900	6.4	-
tsp	700	1.9	21.8
sor	17,700	1.3	1.5
moldyn	1,300	90.6	1.5
montecarlo	3,600	6.4	2.7
raytracer	1,900	4.8	41.8
mtrt	11,300	2.8	38.8
jigsaw	90,100	3.0	4.7
specJBB	30,500	26.2	12.1
webl	22,300	60.3	-
lib-java	75,305	96.5	-

Extensions Reduce Number of Warnings



Total 341

Total 97

Evaluation

- Warnings: 97 (from 200KLOC)
- At least 7 are real errors
- False alarms due to:
  - simplistic heuristics for atomicity
    - programmer should specify atomicity
  - false races
  - methods irreducible yet still "atomic"
    - eg caching, lazy initialization
- No warnings reported in more than 90% of exercised methods

Example Bugs

```
class PrintWriter {
    Writer out;
    public void println(String s) {
      synchronized(lock) {
        out.print(s);
        out.println();
      }
    }
}
class ResourceStoreManager {
    synchronized checkClosed() { ... }
    synchronized lookup(...) { ... }
    public ResourceStore loadResourceStore(...) {
      checkClosed();
      return lookup(...);
    }
}
```