Atomizer:
A Dynamic Atomicity Checker For Multithreaded Programs

Cormac Flanagan
University of California, Santa Cruz

Stephen Freund
Williams College

Testing and Debugging Multithreaded Software

- Race conditions

- Deadlock

Testing and Debugging Multithreaded Software

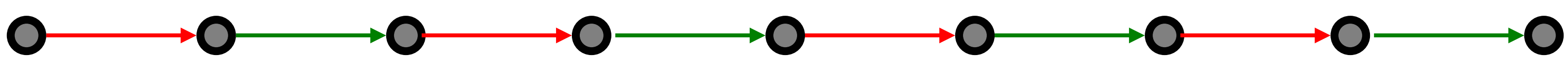
- Race conditions

- Deadlock

Bank Account Implementation

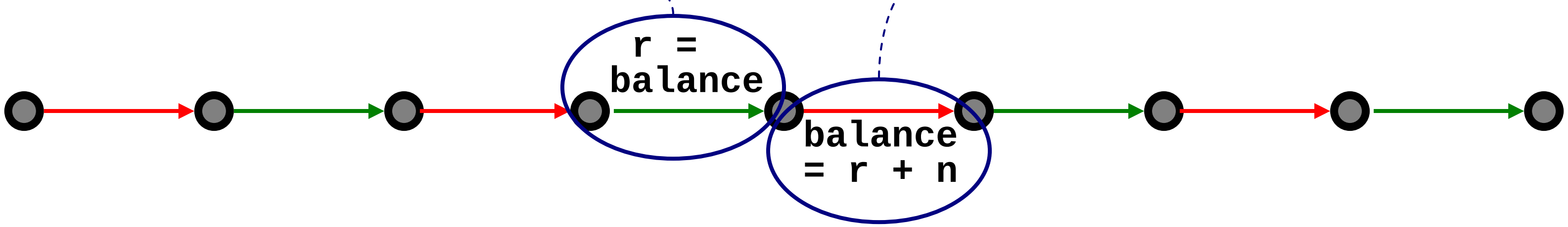
```
class Account {
  private int balance = 0;

  public read() {
    int r;
    r = balance;
    return r;
  }
}
```

```
public void deposit(int n) {
  int r = read();
  balance = r + n;
}
```



Bank Account Implementation

```
class Account {
  private int balance = 0;

  public read() {
    int r;
    r = balance;
    return r;
  }

  public void deposit(int n) {
    int r = read();
    balance = r + n;
  }
}
```



A *race condition* occurs if two threads access a shared variable at the same time, and at least one of the accesses is a write

Race Conditions

- Many tools for detecting race conditions
  - type systems
    - [Abadi-Flanagan 99, Flanagan-Freund 00, Boyapati-Rinard 01]
  - dynamic race detectors
    - Eraser [Savage et al 97]
  - static analyses
    - Warlock [Sterling 93]
- Is race-freedom the "right" property to check?

Race-Free Bank Account

```
class Account {
  private int balance = 0;

  public read() {
    int r;
    synchronized(this) {
      r = balance;
    }
    return r;
  }

  public void deposit(int n) {
    int r = read();
    other threads can update balance
    synchronized(this) {
      balance = r + n;
    }
  }

}
```

- Race-freedom is not sufficient

Fixed Bank Account

```
class Account {
  private int balance = 0;

  public read() {
    int r;
    synchronized(this) {
      r = balance;
    }
    return r;
  }

}
```

```
public void deposit(int n) {
  synchronized(this) {
    int r = balance;
    balance = r + n;
  }
}
```

Optimized Bank Account

```
class Account {
  private int balance = 0;

  public read() {
    return balance;
  }

  public void deposit(int n) {
    synchronized(this) {
      int r = balance;
      balance = r + n;
    }
  }

}
```

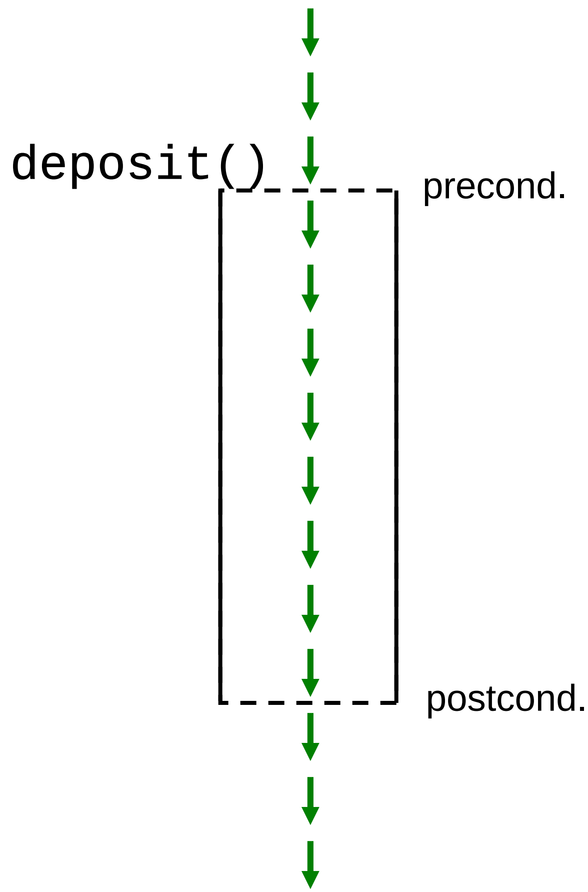
- Race-freedom is not necessary

Race-Freedom

- Race-freedom is neither *necessary* nor *sufficient* to ensure the absence of errors due to unexpected interactions between threads

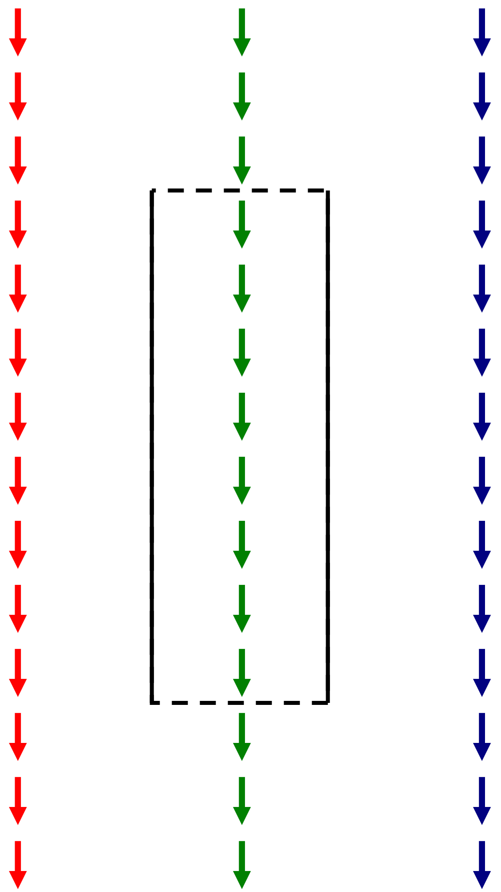
- Is there a more fundamental semantic correctness property?

Sequential Program Execution



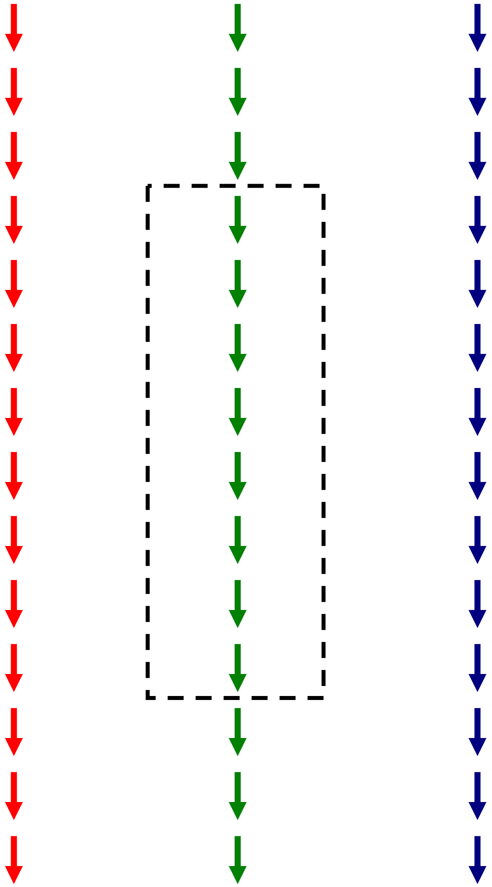

```
void deposit() {

   ..

   ..

}
```

Multithreaded Program Execution



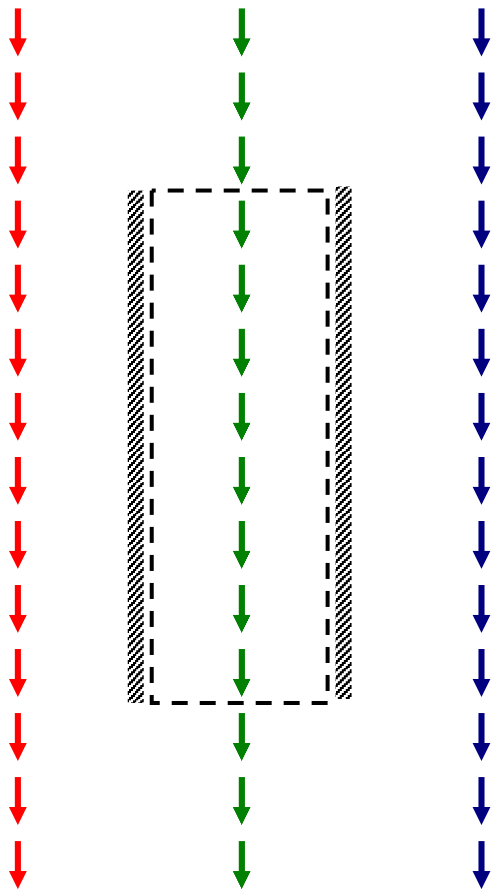
```
void deposit() {

  ..

  ..

}
```

Multithreaded Program Execution



```
void deposit() {

   ..


   ..


}
```

Multithreaded Program Execution



- Atomicity
  - maximal non-interference property
  - enables sequential reasoning
  - matches existing methodology

Definition of Atomicity

- Serial execution of deposit

$$\bullet \xrightarrow{x} \bullet \xrightarrow{y} \bullet \xrightarrow{acq(this)} \bullet \xrightarrow{r=bal} \bullet \xrightarrow{bal=r+n} \bullet \xrightarrow{rel(this)} \bullet \xrightarrow{z} \bullet$$

- Non-serial executions of deposit

$$\bullet \xrightarrow{acq(this)} \bullet \xrightarrow{x} \bullet \xrightarrow{r=bal} \bullet \xrightarrow{y} \bullet \xrightarrow{bal=r+n} \bullet \xrightarrow{z} \bullet \xrightarrow{rel(this)} \bullet$$

$$\bullet \xrightarrow{acq(this)} \bullet \xrightarrow{x} \bullet \xrightarrow{y} \bullet \xrightarrow{r=bal} \bullet \xrightarrow{bal=r+n} \bullet \xrightarrow{z} \bullet \xrightarrow{rel(this)} \bullet$$

```
public void deposit(int n) {
  synchronized(this) {
    int r = bal;
    bal = r + n;
  }
}
```

- deposit is atomic if for every non-serial execution, there is a serial execution with the same overall behavior (same final state)

Atomicity a Canonical Concept

- Strict serializability in databases
- Linearizability for concurrent objects
- Hoare's monitors
- Argus language [Liskov et al 87]
- Avalon language [Eppinger et al 91]
- Persistent languages [Atkinson et al 81]

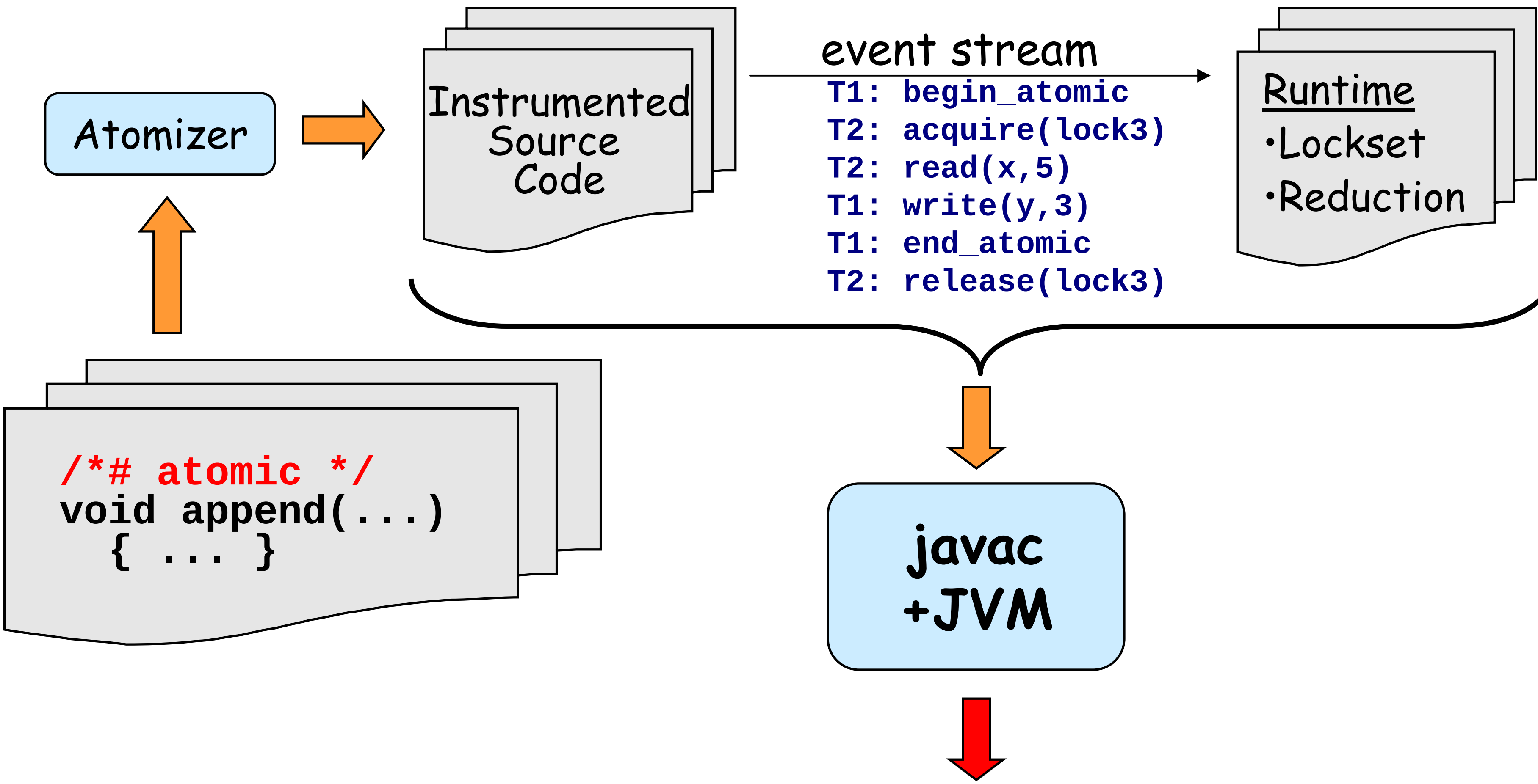
Tools for Checking Atomicity

- Calvin: ESC for multithreaded code
  - heavyweight static analysis (2 KLOC)

- A type system for Atomicity
  - lightweight static analysis (20 KLOC)

- Atomizer: A dynamic atomicity checker
  - lightweight dynamic analysis (200 KLOC)
  - "a purify-like tool for atomicity"

Atomizer: Instrumentation Architecture



Warning: method "append" may not be atomic at line 43

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

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`:

$$\text{LockSet}(\texttt{o.f}) := \text{LockSet}(\texttt{o.f}) \cap \text{Held}(\text{curThread})$$
$$= \{ x, y \} \cap \{ x \}$$
$$= \{ x \}$$

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`:

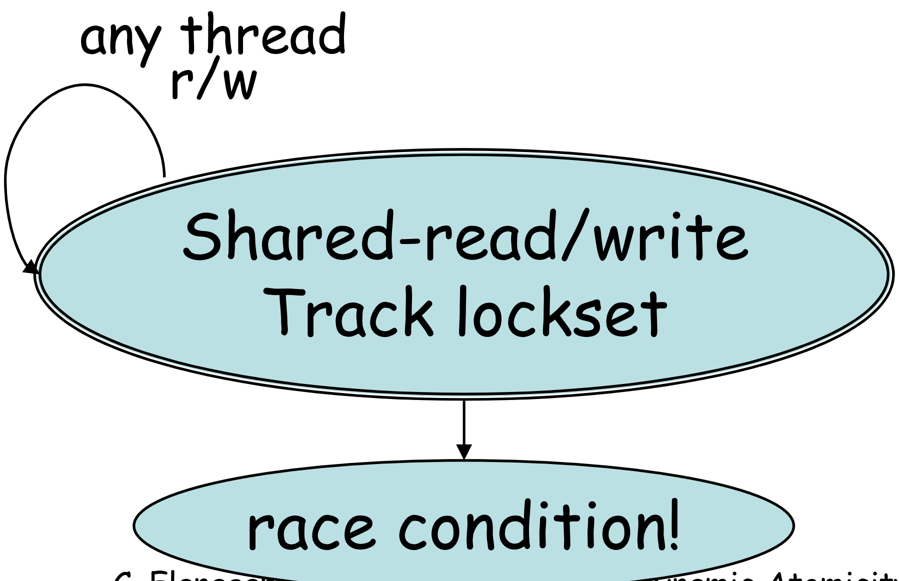
$$\text{LockSet}(\texttt{o.f}) := \text{LockSet}(\texttt{o.f}) \cap \text{Held}(\text{curThread})$$

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

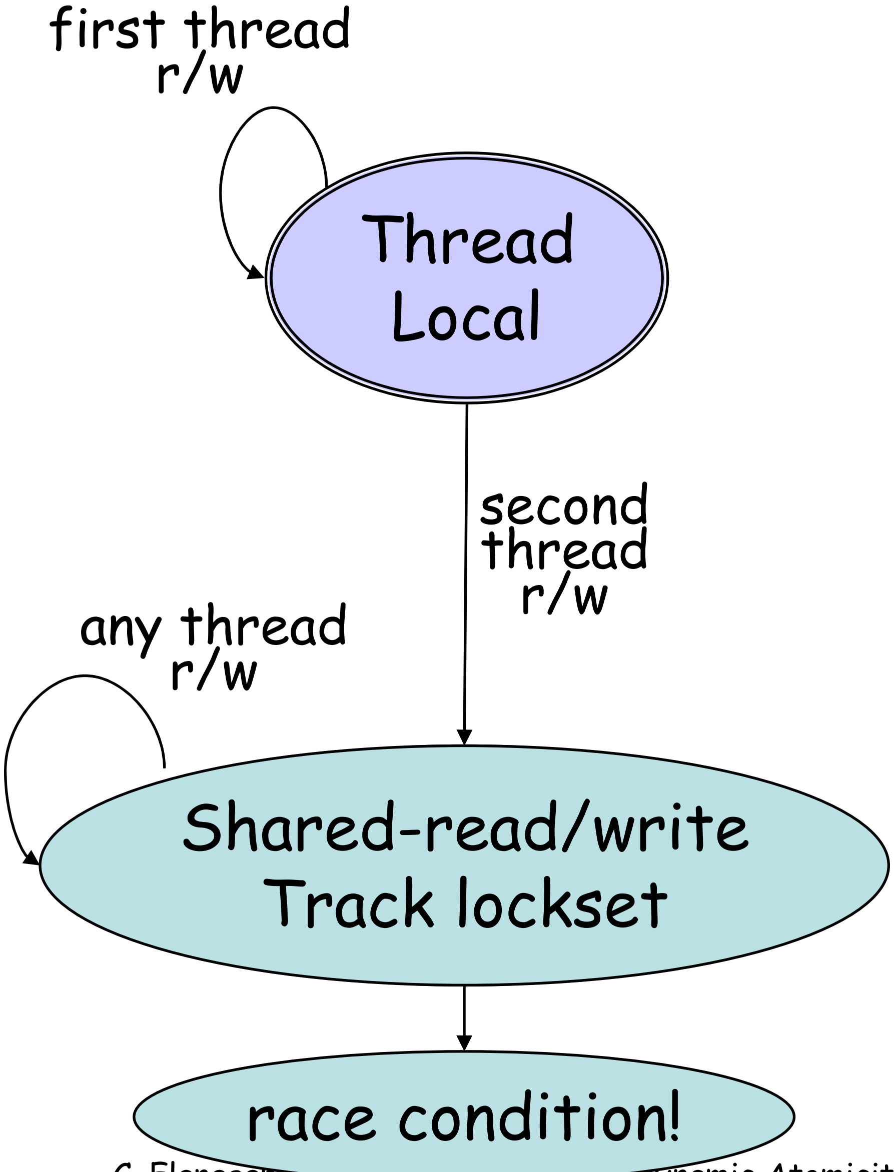
$$= \{ \}$$

=> race condition

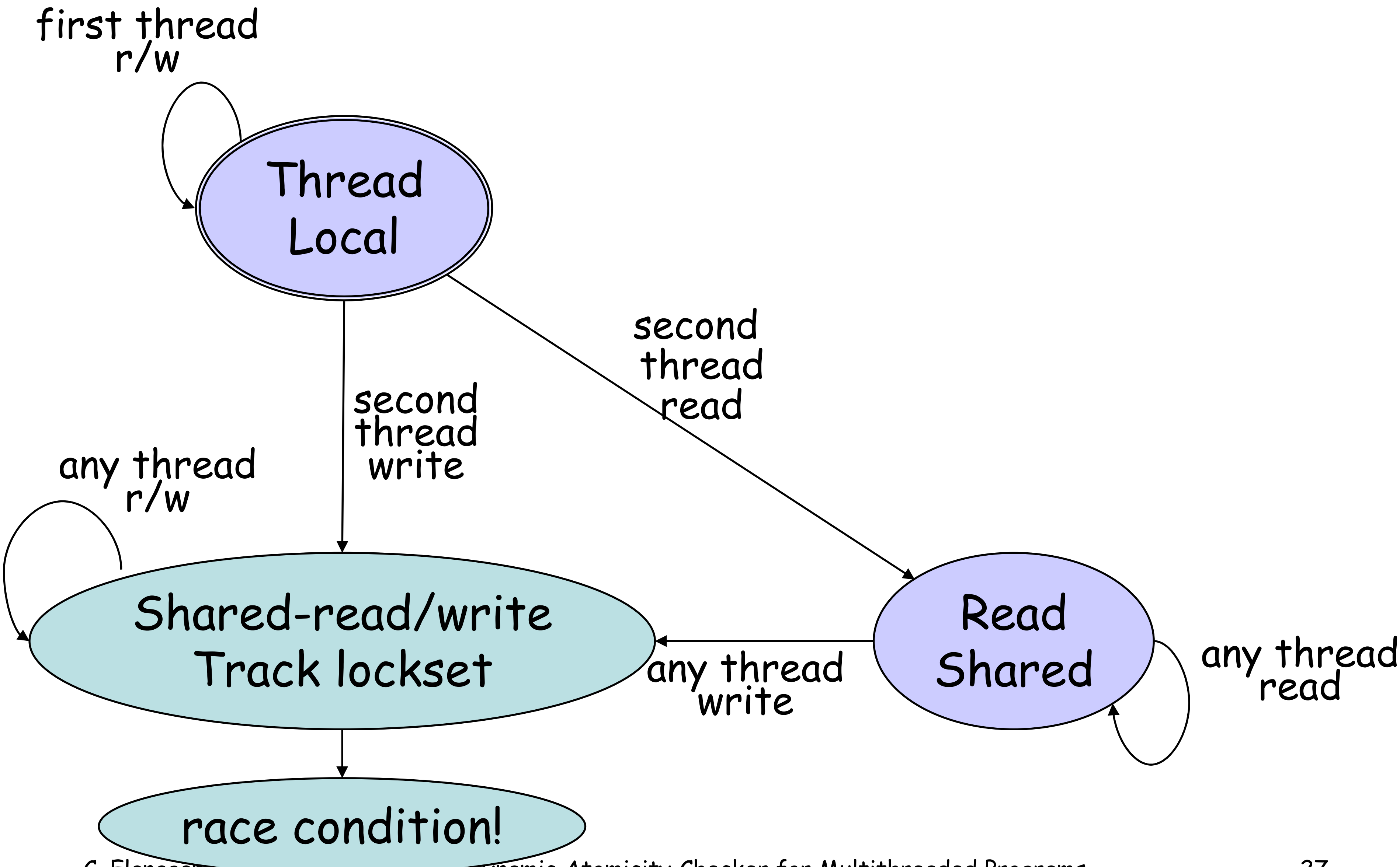
Lockset


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

Extending Lockset (Thread Local Data)



Extending Lockset (Read Shared Data)



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]

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

green thread holds lock

⇒ red thread does not hold lock

⇒ operation y does not access balance
(assuming balance protected by lock)

⇒ operations commute

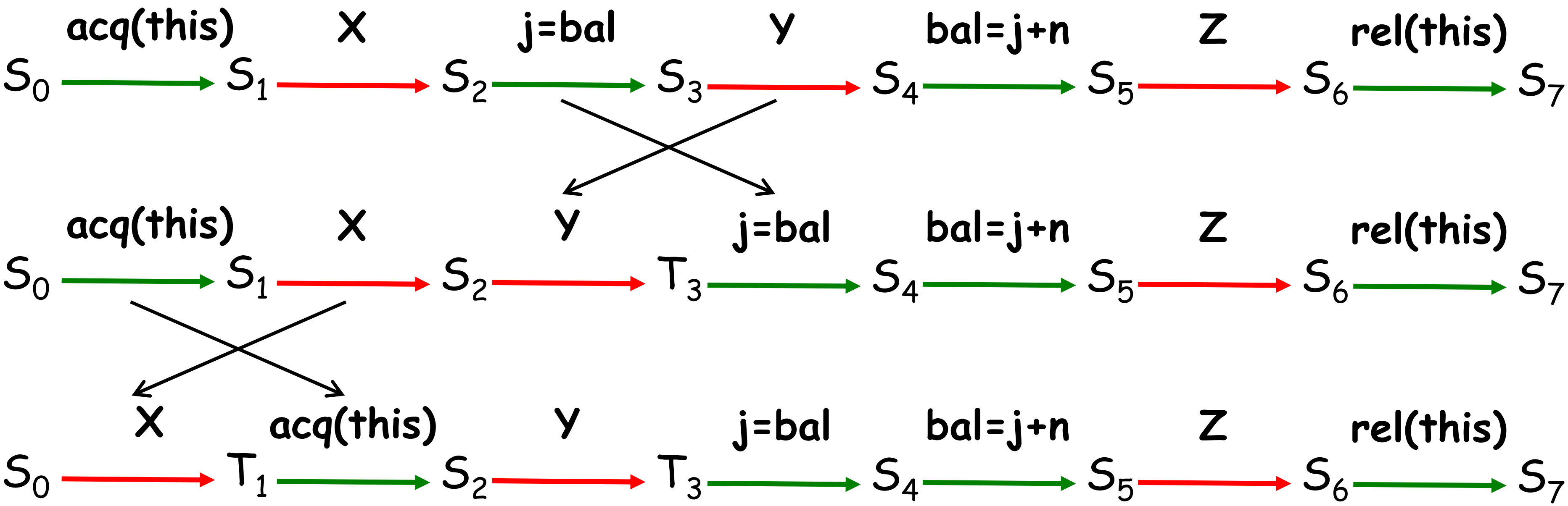
Reduction [Lipton 75]

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

$$S_0 \xrightarrow{\text{acq(this)}} S_1 \xrightarrow{\text{x}} S_2 \xrightarrow{\text{y}} T_3 \xrightarrow{\text{j=bal}} S_4 \xrightarrow{\text{bal=j+n}} S_5 \xrightarrow{\text{z}} S_6 \xrightarrow{\text{rel(this)}} S_7$$

green thread holds lock after acquire
$\Rightarrow$ operation x does not modify lock
$\Rightarrow$ operations commute

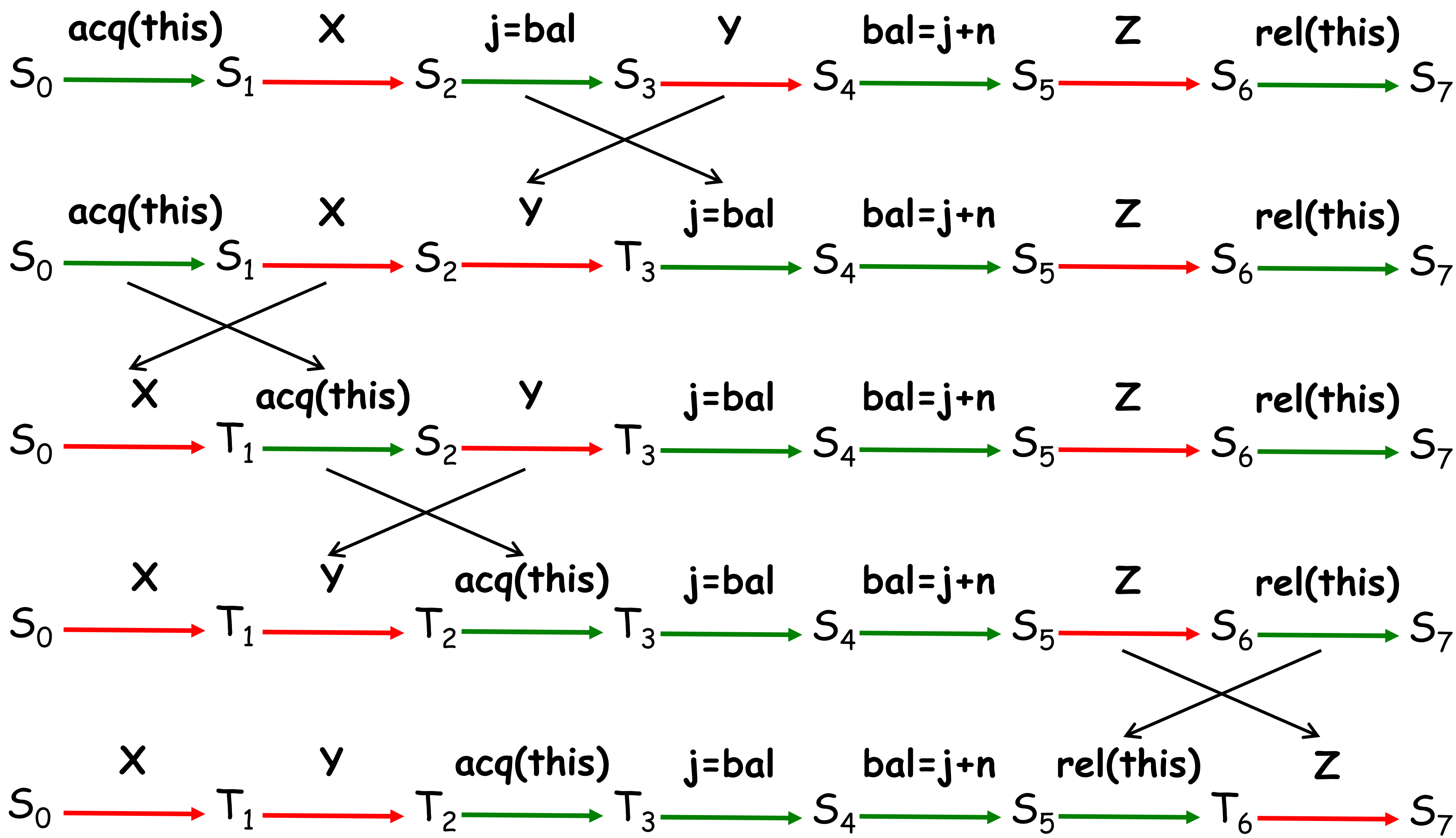
Reduction [Lipton 75]



Reduction [Lipton 75]

$$\xrightarrow{\textbf{acq(this)}} S_1 \xrightarrow{\textbf{X}} S_2 \xrightarrow{\textbf{j=bal}}$$

Reduction [Lipton 75]

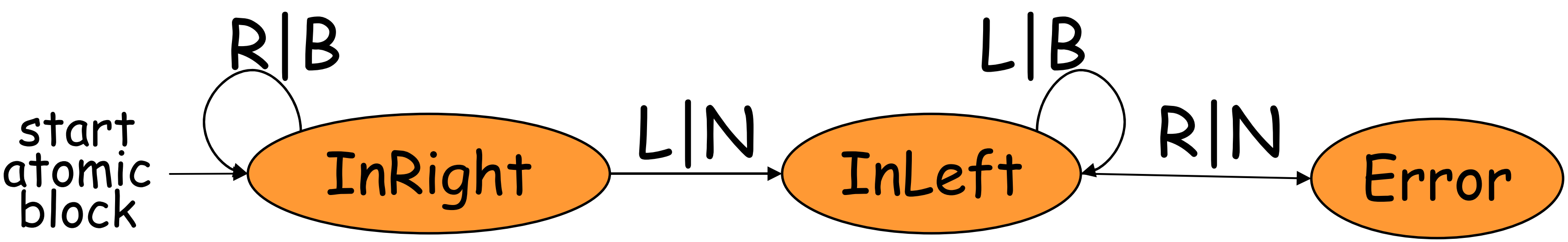


Performing Reduction Dynamically

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

acq(lock)	j=bal	bal=j+n	rel(lock)
R	B	B	L

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



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

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

Benchmark	Lines	Base Time (s)	Atomizer 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

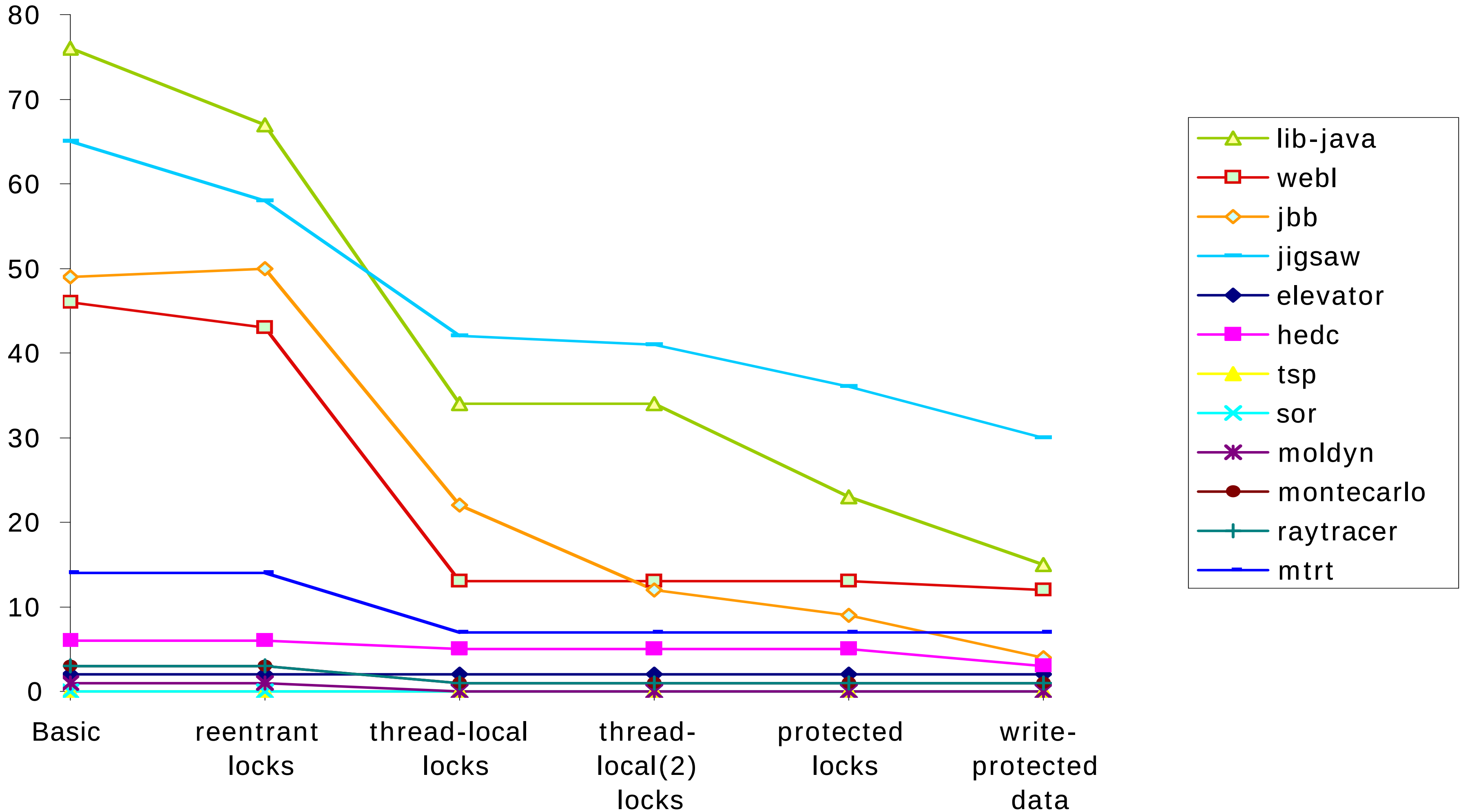
- Redundant lock operations are both-movers
  - re-entrant acquire/release
  - operations on thread-local locks
  - operations on lock A, if lock B always acquired before A

- Write-protected data

Write-Protected Data

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

Extensions Reduce Number of Warnings



Total 341

Total 97

Evaluation

- Warnings: 97 (down from 341)
- Real errors (conservative): 7
- 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

java.lang.StringBuffer

```
/**
   ... used by the compiler to implement the binary
   string concatenation operator ...

   String buffers are safe for use by multiple
   threads. The methods are synchronized so that
   all the operations on any particular instance
   behave as if they occur in some serial order
   that is consistent with the order of the method
   calls made by each of the individual threads
   involved.
*/
/*# atomic */ public class StringBuffer { ... }
```

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

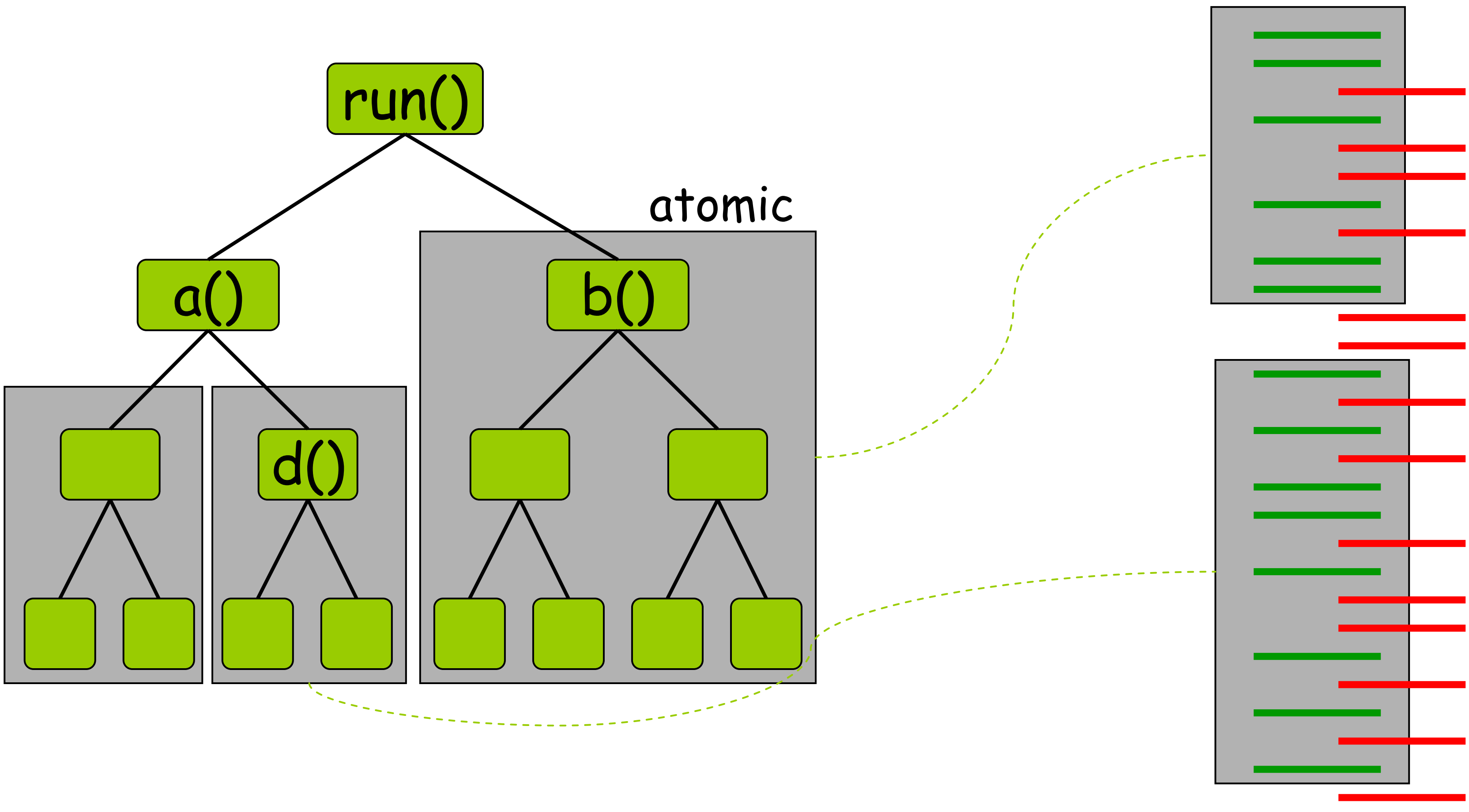
```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
  /*# atomic */
  public synchronized void append(StringBuffer sb){

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

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

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

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

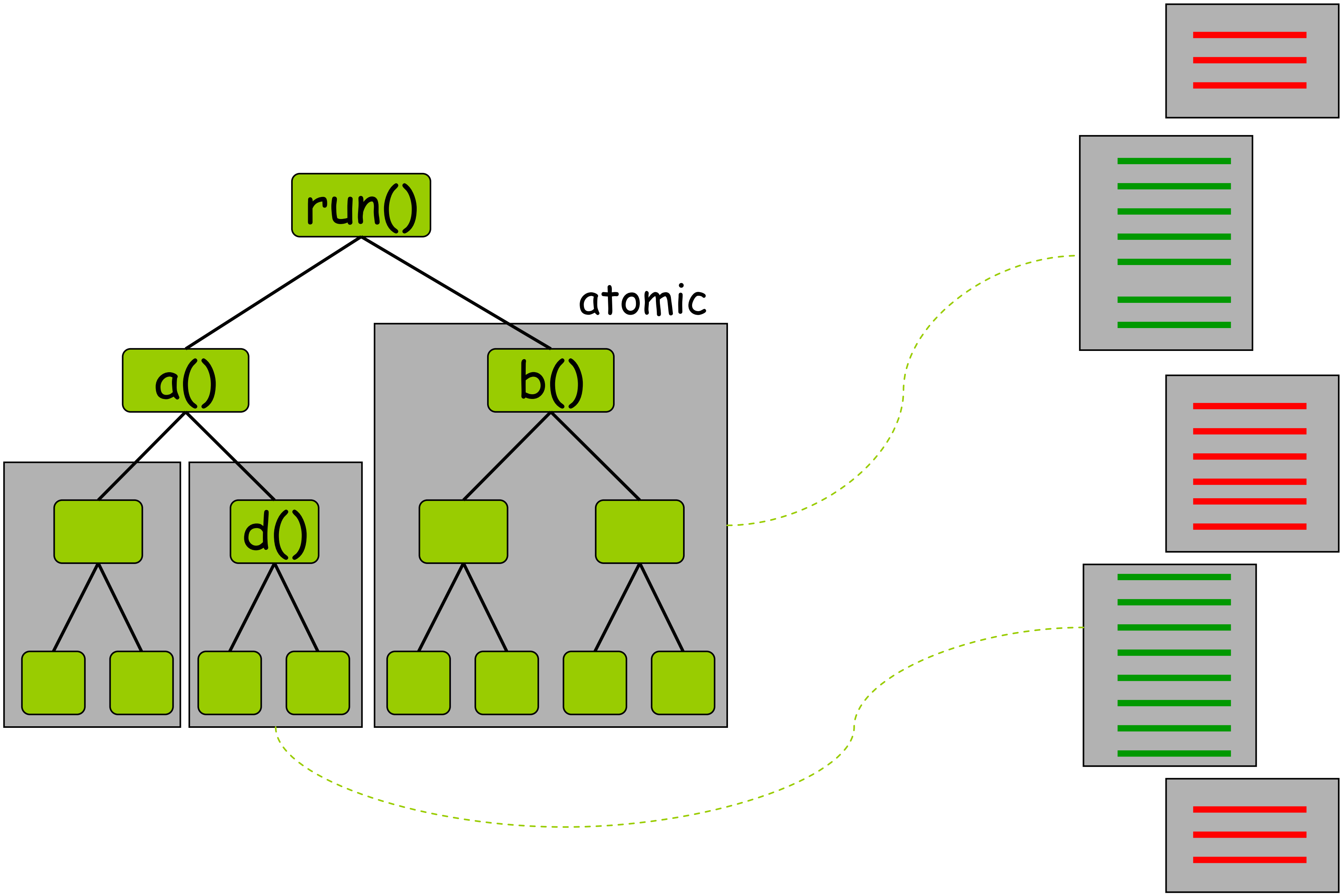
Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - type systems [Flanagan-Qadeer 03],
    model checking [Stoller-Cohen 03, Flanagan-Qadeer 03],
    procedure summaries [Qadeer et al 04]
- Other atomicity checkers
  - [Wang-Stoller 03], Bogor model checker [Hatcliff et al 03]
- Race detection / prevention
  - dynamic [Savage et al 97, O'Callahan-Choi 01, von Praun-Gross 01]
  - Warlock [Sterling 93], SPMD [Aiken-Gay 98]
  - type systems [Abadi-Flanagan 99, Flanagan-Freund 00, Boyapati-Rinard 01]
  - Guava [Bacon et al 01]
- View consistency [Artho-Biere-Havelund 03, von Praun-Gross 03]

Multithreaded Program Execution



Multithreaded Program Execution



Conclusions And Future Directions

- Atomicity
  - maximal non-interference property
  - enables sequential reasoning
  - matches practice
- Atomizer extends race detection techniques
  - catches "higher-level" concurrency errors
  - some benign races do not break atomicity
- Atomicity for distributed systems?