Parallel & Concurrent Programming:
Atomicity

Emery Berger

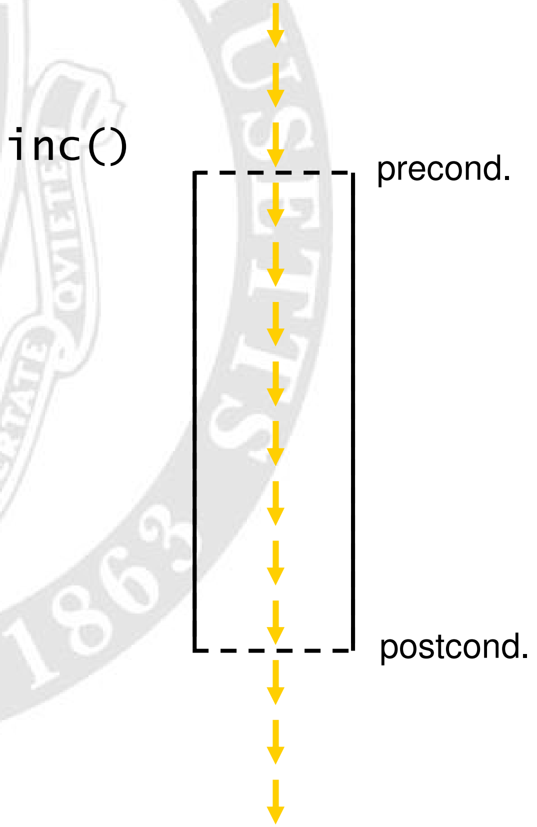
CMPSCI 691W
Spring 2006

Outline

- Last time:
  - Race detection
- This time:
  - Atomicity

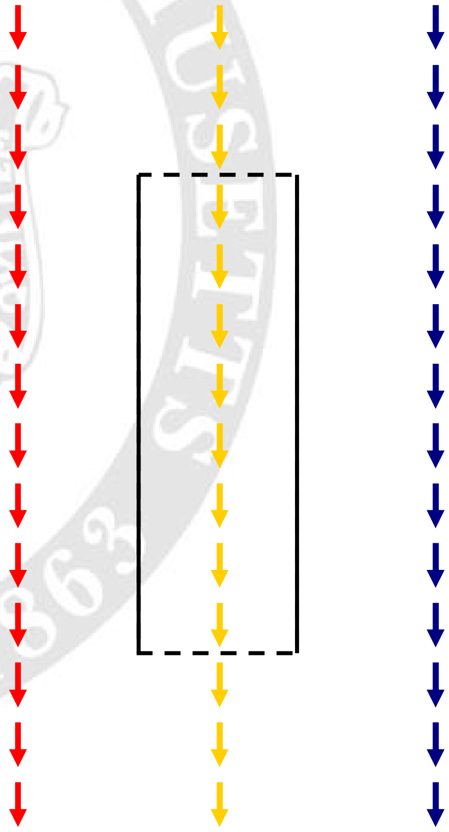
some slides adapted from Flanagan, PLDI 05

Sequential Execution



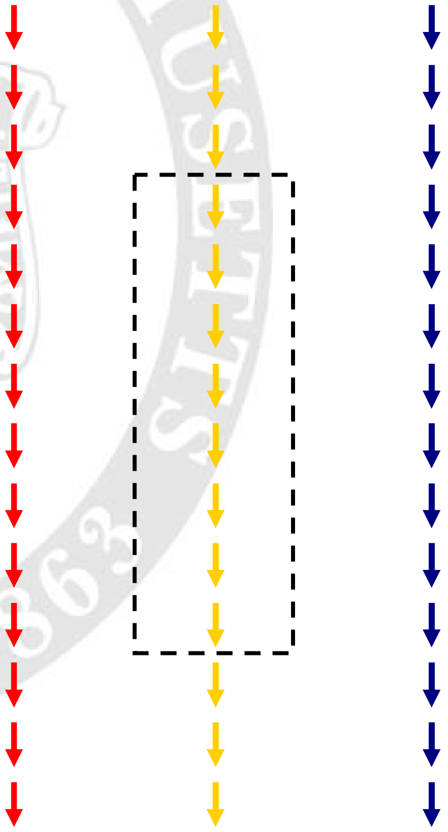
```
void inc() {
   ..

   ..

}
```

MT Execution



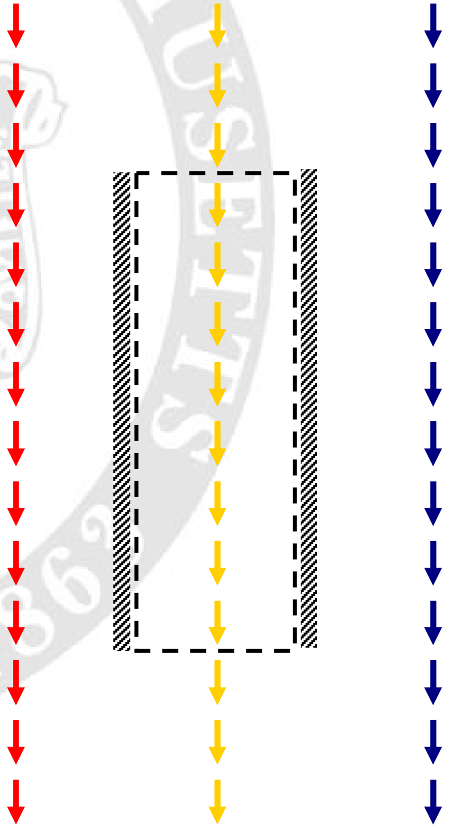
```
void inc() {

  ..


  ..


}
```

MT Execution



```
void inc() {
  ..

  ..

}
```

MT Execution



Atomicity

- guarantees concurrent threads do not interfere with atomic method
- enables sequential reasoning
- matches existing methodology

Definition of Atomicity

- Method (or code block) **atomic** if
  - ∀ arbitrarily interleaved executions: ∃ equivalent execution with same behavior when method executed serially
- Compare to **linearizability, serializability**

Atomicity Example

```
class Account {
  private int balance = 0;
  public int read() {
    return balance;
  }
}
```

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

possible serial executions:

1

```
int v = read();
```

```
deposit(10);
```

2

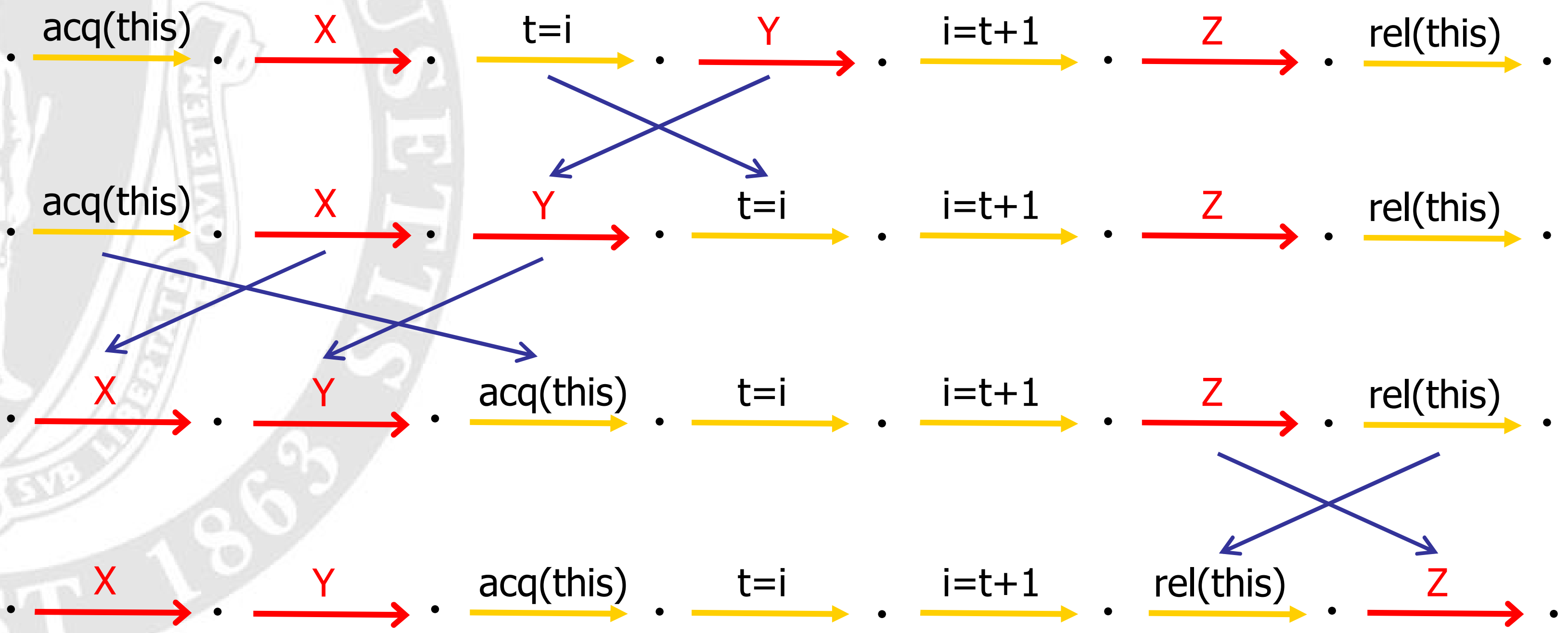
```
int v = read();
```

```
deposit(10);
```

Atomizer

- **Atomizer** [Flanagan & Freund, POPL 04]
  - Dynamic tool for atomicity violation detection
  - Builds on Eraser & **Lipton's theory of reduction**
- Results:
  - Finds more defects than race detectors
  - Few false positives
  - *Most exported methods atomic*

Reduction [Lipton 75]



acq(this) X t=i Y i=t+1 Z rel(this)

acq(this) X Y t=i i=t+1 Z rel(this)

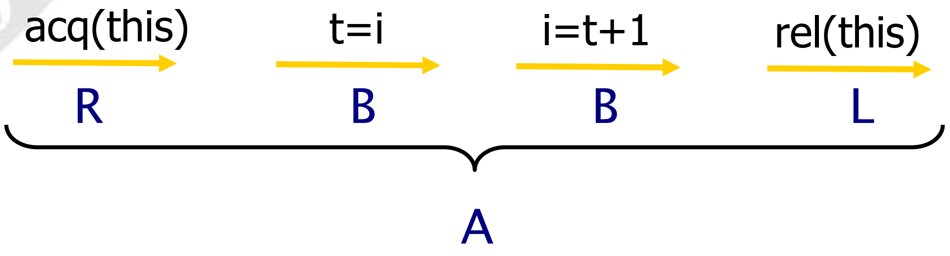
X Y acq(this) t=i i=t+1 Z rel(this)

X Y acq(this) t=i i=t+1 rel(this) Z

Checking Atomicity

```
atomic void inc() {
    int t;
    synchronized (this) {
      t = i;
      i = t + 1;
    }
  }
```

R: right-mover	lock acquire
L: left-mover	lock release
B: both-mover	race-free variable access
A: atomic access	conflicting variable

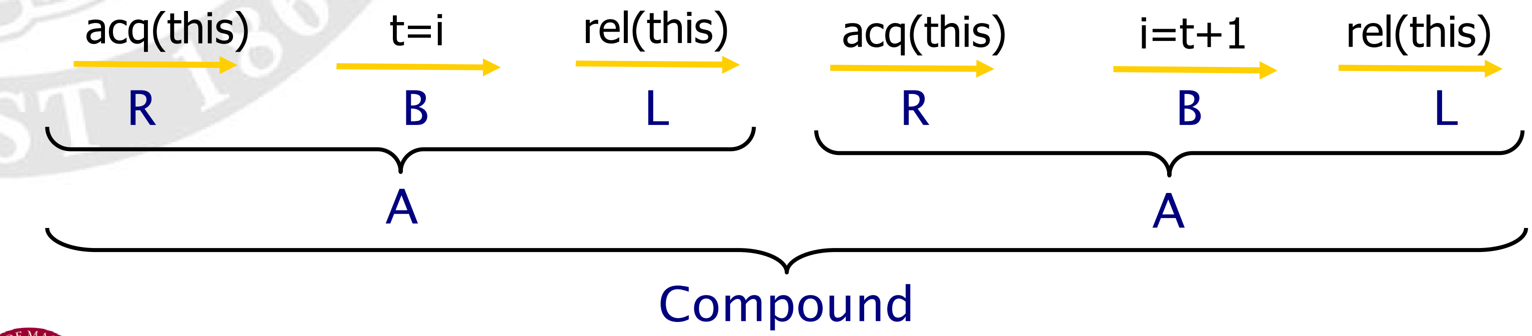


- Reducible blocks have form: (R|B)* [A] (L|B)*

Checking Atomicity II

```
~~atomic~~ void inc() {
    int t;
    synchronized (this) {
      t = i;
    }
    synchronized (this) {
      i = t + 1;
    }
}
```

R: right-mover	lock acquire
L: left-mover	lock release
B: both-mover	race-free variable access
A: atomic access	conflicting variable



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.
*/
```

FALSE

```
/*# 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

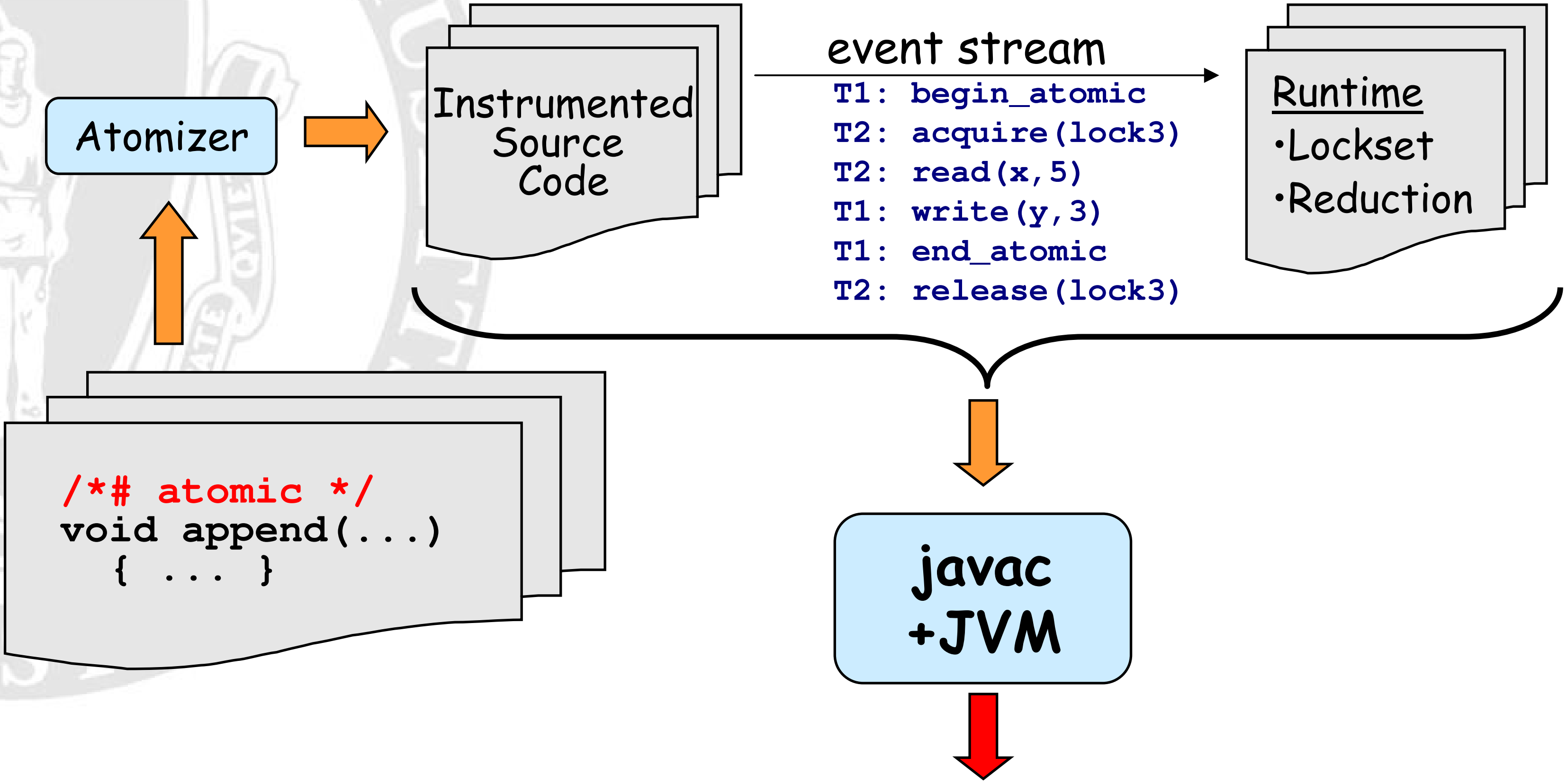
```
public class StringBuffer {
  private int count;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }

  atomic public synchronized void append(StringBuffer sb){

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

A

A

Compound

Atomizer Architecture



```
/*# atomic */
void append(...)
  { ... }
```

Atomizer

Instrumented Source Code

event stream

```
T1: begin_atomic
T2: acquire(lock3)
T2: read(x,5)
T1: write(y,3)
T1: end_atomic
T2: release(lock3)
```

Runtime
- Lockset
- Reduction

javac +JVM

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

Dynamic Analysis

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

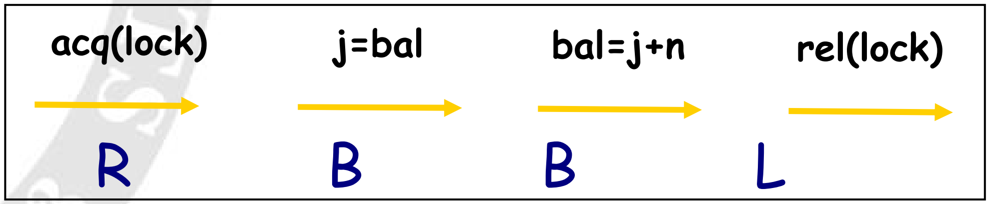
- Reduction [Lipton 75]
  - proof technique for verifying atomicity, using information about race conditions

Dynamic Analysis

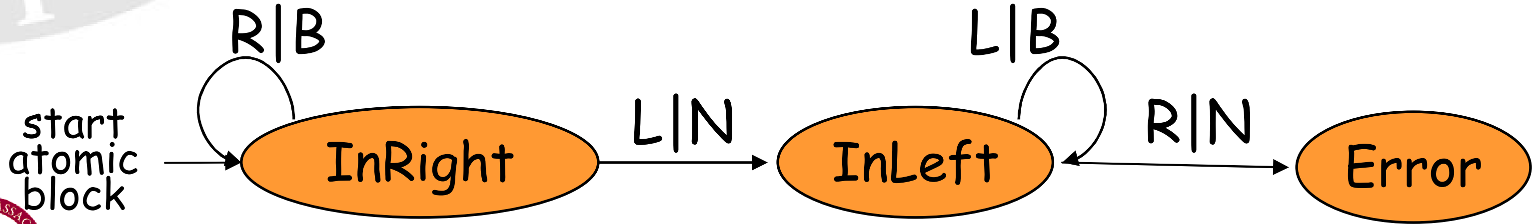
- Lockset algorithm
  - from Eraser [Savage et al. 97]
  - identifies race conditions
- Reduction [Lipton 75]
  - proof technique for verifying atomicity, using information about race conditions

Dynamic Reduction

- 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)*



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

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	-

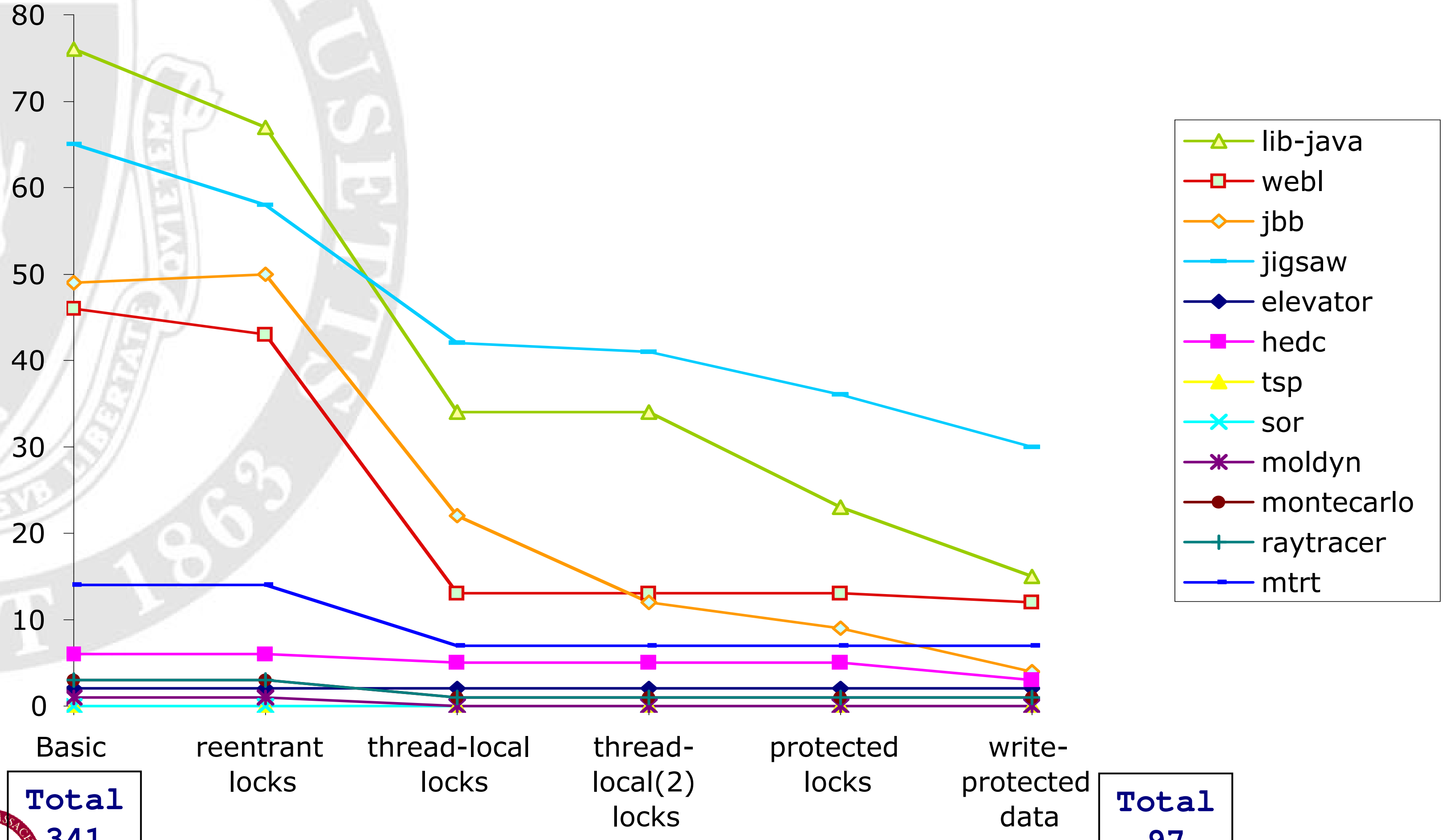
Extensions

- 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
  - Much like reader-writer locks

Write-Protected Data

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

Extensions & Warnings



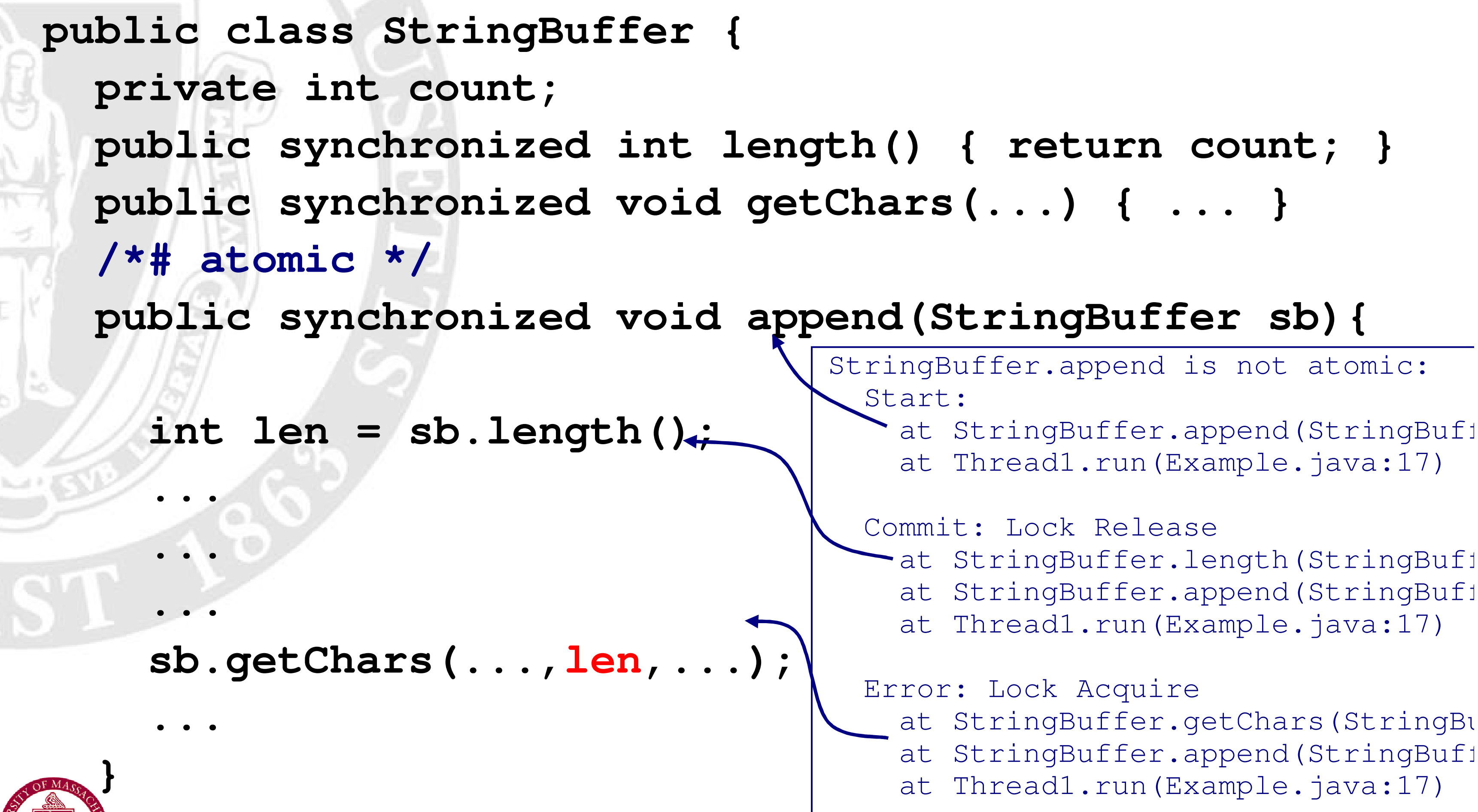
Total 341

Total 97

Evaluation

- Warnings: 97 (down from 341)
- Real errors: at least 7
- False alarms:
  - simplistic heuristics for atomicity
    - need programmer help to specify atomicity
  - false races
  - methods irreducible yet still “atomic”
    - e.g., caching, lazy initialization
- No warnings reported in more than 90% of exercised methods

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(StringBuf
    at Thread1.run(Example.java:17)

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

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

Static approaches

- **Types for atomicity**
  - Basic atomicity (atomic, left-mover, etc.)
  - **Conditional atomicity**
    - If lock(l) held, ...
  - Field **Guarded-by** lock, **Write-guarded-by** lock
  - Method **Requires** lock1, lock2...
- Uses constraint-based system to infer most precise types
  - Full inference often NP-complete
  - Better than undecidable...

The End