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() {
    ..
    ..
}
```
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]
Checking Atomicity

```java
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

```java
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       conflicting variable

Compound
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

```java
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(...)`
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, ...);
        ... 
    }
}
/** atomic */
void append(...) {
... }

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>Base Time (s)</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>elevator</td>
<td>500</td>
<td>11.2</td>
<td>-</td>
</tr>
<tr>
<td>hedc</td>
<td>29,900</td>
<td>6.4</td>
<td>-</td>
</tr>
<tr>
<td>tsp</td>
<td>700</td>
<td>1.9</td>
<td>21.8</td>
</tr>
<tr>
<td>sor</td>
<td>17,700</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>moldyn</td>
<td>1,300</td>
<td>90.6</td>
<td>1.5</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3,600</td>
<td>6.4</td>
<td>2.7</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,900</td>
<td>4.8</td>
<td>41.8</td>
</tr>
<tr>
<td>mtrt</td>
<td>11,300</td>
<td>2.8</td>
<td>38.8</td>
</tr>
<tr>
<td>jigsaw</td>
<td>90,100</td>
<td>3.0</td>
<td>4.7</td>
</tr>
<tr>
<td>specJBB</td>
<td>30,500</td>
<td>26.2</td>
<td>12.1</td>
</tr>
<tr>
<td>webl</td>
<td>22,300</td>
<td>60.3</td>
<td>-</td>
</tr>
<tr>
<td>lib-java</td>
<td>75,305</td>
<td>96.5</td>
<td>-</td>
</tr>
</tbody>
</table>
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
class Account {
    int bal;
    /*# atomic */ int read() { return bal; }
    /*# atomic */ void deposit(int n) {
        synchronized (this) {
            int j = bal;
            bal = j + n;
        }
    }
}
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
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, ...);
        ...
    }
}
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