Towards Reliable Multithreaded Software

- Multithreaded software
  - increasingly common (Java, C#, GUIs, servers)
  - decrease latency
  - exploit underlying hardware
    - multi-core chips
- Heisenbugs due to thread interference
  - race conditions
  - atomicity violations
- Need tools to verify atomicity
  - dynamic analysis
  - type systems

Motivations for Atomicity

1. Beyond Race Conditions

A race condition occurs if
- two threads access a shared variable at the same time
- at least one of those accesses is a write
Lock-Based Synchronization

- Field guarded by a lock
- Lock acquired before accessing field
- Ensures race freedom

Ref x = new Ref(0);
parallel {
    x.inc();
    x.inc();
}
assert x.i == 2;

• Field guarded by a lock
• Lock acquired before accessing field
• Ensures race freedom

Limitations of Race-Freedom

Ref x = new Ref(0);
parallel {
    x.inc();
    x.inc();
}
assert x.i == 2;

• race-free
• behaves correctly in a multithreaded context

Ref.read()
• has a race condition
• behaves correctly in a multithreaded context

Race freedom does not prevent errors due to unexpected interactions between threads

Motivations for Atomicity

2. Enables Sequential Reasoning

- 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

```c
void inc() {
    ..
    ..
}
```

### Multithreaded Execution

### Multithreaded Execution

```c
void inc() {
    ..
    ..
}
```

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

### Motivations for Atomicity

3. Simple Specification

### Model Checking of Software Models

Model Constructor

Specification for filesystem.c

filesystem model

Model Checker

Valid

Invalid
Model Checking of Software

Experience with Calvin Software Checker

Experience with Calvin Software Checker

The Need for Atomicity

Motivations for Atomicity

1. Beyond Race Conditions
2. Enables Sequential Reasoning
3. Simple Specification
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

Reduction [Lipton 75]

Checking Atomicity

Checking Atomicity (cont.)

java.lang.StringBuffer

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

String buffers are safe for use by multiple threads. The contents of a string buffer that allows operations on any part of the buffer behavior as if it were a lock-free variable. A method call is considered atomic if it is 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.length() acquires lock on sb,
        gets length, and releases lock
        ... other threads can change sb
        sb.getChars(...,len,...); // use of stale len may yield
        StringIndexOutOfBoundsException
        inside getChars(...)
    }
}
```

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();
        ... A
        ... Compound
        ... A
        ... sb.getChars(...,len,...); ...
    }
}
```

## Tutorial Outline

- Part 1
  - Introduction
  - Runtime analysis for atomicity
- Part 2
  - Model checking for atomicity
- Part 3
  - Type systems for concurrency and atomicity
- Part 4
  - Beyond reduction – atomicity via "purity"

### Part I continued:

Runtime Analysis for Atomicity

---

Atomizer: Instrumentation Architecture

```
/*# atomic */
void append(...) {
    // acquire(lock3)
    read(y,3)
    // release(lock3)
}
```

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

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 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

- First access to o.f:
  \[
  \text{LockSet}(o.f) = \text{Held(curThread)} = \{x, y\}
  \]

- Subsequent access to o.f:
  \[
  \text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held(curThread)} = \{x\} \cap \{y\} = \{\}
  \]

- Subsequent access to o.f:
  \[
  \text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held(curThread)} = \{\} \cap \{\} = \{\}
  \]

- => race condition

Lockset Example

- First access to o.f:
  \[
  \text{LockSet}(o.f) = \text{Held(curThread)} = \{x, y\}
  \]

- Subsequent access to o.f:
  \[
  \text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held(curThread)} = \{x\} \cap \{y\} = \{\}
  \]

- => race condition

Lockset Example

- First access to o.f:
  \[
  \text{LockSet}(o.f) = \text{Held(curThread)} = \{x, y\}
  \]

- Subsequent access to o.f:
  \[
  \text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held(curThread)} = \{x\} \cap \{y\} = \{\}
  \]

- => race condition

Lockset Example

- First access to o.f:
  \[
  \text{LockSet}(o.f) = \text{Held(curThread)} = \{x, y\}
  \]

- Subsequent access to o.f:
  \[
  \text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held(curThread)} = \{x\} \cap \{y\} = \{\}
  \]

- => race condition
Lockset with Thread Local Data

Thread

Local

Shared-read/write Track lockset

race condition!

Lockset with Read Shared Data

Thread

Local

Read

Shared

Shared-read/write Track lockset

race condition!

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

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)*

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>sst</td>
<td>17,700</td>
<td>1.3</td>
<td>11.7</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>6.6</td>
<td>2</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,000</td>
<td>4.8</td>
<td>41.5</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>39,500</td>
<td>26.3</td>
<td>12.1</td>
</tr>
<tr>
<td>web</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

Extensions Reduce Number of Warnings

- 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

Write-Protected Data

```java
class Account {
    int bal;
    /*# atomic */ int read() { return bal; }
    /*# atomic */ void deposit(int n) {
        synchronized (this) {
            int j = bal;
            bal = j + n;
        }
    }
}
```
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,...)
        ...
    }
}