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

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

• Race-freedom is not sufficient
Fixed Bank Account

class Account {
    private int balance = 0;

    public int read() {
        synchronized(this) {
            int 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 int 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() {
    ...
    ...
}

precond.

postcond.
Multithreaded Program Execution

```c
void deposit() {
    ...
    ...
}
```
Multithreaded Program Execution

```c
void deposit() {
    ..
    ..
}
```
Multithreaded Program Execution

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

• Serial execution of deposit

\[ x \rightarrow y \rightarrow \text{acq(this)} \rightarrow r=\text{bal} \rightarrow \text{bal}=r+n \rightarrow \text{rel(this)} \rightarrow z \rightarrow \]

• Non-serial executions of deposit

\[ \text{acq(this)} \rightarrow x \rightarrow r=\text{bal} \rightarrow y \rightarrow \text{bal}=r+n \rightarrow z \rightarrow \text{rel(this)} \]

\[ \text{public void deposit(int n) \{ \text{synchronized(this) \{ \text{int } r = \text{bal} ; \text{bal} = r + n; } \}} \]

• 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:

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

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

\[
\text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held(curThread)}
\]
\[
= \{ x \} \cap \{ y \}
\]
\[
= \{ \} \quad \Rightarrow \text{race condition}
\]
Lockset

any thread
r/w

Shared-read/write
Track lockset

race condition!
Extending Lockset (Thread Local Data)

Thread Local

first thread
r/w

second thread
r/w

any thread
r/w

Shared-read/write
Track lockset

race condition!
Extending Lockset (Read Shared Data)

- First thread: Read/Write
- Second thread: Read
- Any thread: Read/Write

Thread Local

Shared-read/write Track lockset

Read Shared

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
Reduction [Lipton 75]

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

- 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]

Green thread holds lock after acquire
⇒ operation x does not modify lock
⇒ operations commute
Reduction [Lipton 75]

\[ \text{acq(this)} \quad X \quad j=\text{bal} \quad Y \quad \text{bal}=j+n \quad Z \quad \text{rel(this)} \]

\[ S_0 \xrightarrow{} S_1 \xrightarrow{} S_2 \xrightarrow{} S_3 \xrightarrow{} S_4 \xrightarrow{} S_5 \xrightarrow{} S_6 \xrightarrow{} S_7 \]

\[ \text{acq(this)} \quad X \quad Y \quad j=\text{bal} \quad \text{bal}=j+n \quad Z \quad \text{rel(this)} \]

\[ S_0 \xrightarrow{} S_1 \xrightarrow{} S_2 \xrightarrow{} S_3 \xrightarrow{} S_4 \xrightarrow{} S_5 \xrightarrow{} S_6 \xrightarrow{} S_7 \]

\[ X \quad \text{acq(this)} \quad Y \quad j=\text{bal} \quad \text{bal}=j+n \quad Z \quad \text{rel(this)} \]

\[ S_0 \xrightarrow{} T_1 \xrightarrow{} S_2 \xrightarrow{} T_3 \xrightarrow{} S_4 \xrightarrow{} S_5 \xrightarrow{} S_6 \xrightarrow{} S_7 \]
Reduction [Lipton 75]

\(\text{acq(this)} \times j=\text{bal}\)

\[\xrightarrow{} S_1 \xrightarrow{} S_2\]
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: \((R|B)^* [N] (L|B)^*\)

Start atomic block

InRight \(\xrightarrow{\text{R|B}}\) InLeft \(\xrightarrow{\text{L|N}}\) Error

\(\text{acq(lock)} \quad j=\text{bal} \quad \text{bal}=j+n \quad \text{rel(lock)}\)
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
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>Base Time (s)</th>
<th>Atomizer 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
Write-Protected Data

class Account {
    int bal;
    /*# atomic */ int read() { return bal; }
    /*# atomic */ void deposit(int n) {
        synchronized (this) {
            int j = bal;
            bal = j + n;
        }
    }
}
Extensions Reduce Number of Warnings

Total 341

C. Flanagan

Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs

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
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, ...);
        ...
    }
}
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

run()

atomic

a()

d()

b()
Multithreaded Program Execution

C. Flanagan: Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs
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?