Lightweight Checkers For Atomicity

Stephen Freund
Williams College (on sabbatical at UCSC)

Cormac Flanagan
University of California, Santa Cruz

The Blue Screen of Death (BSOD)

More BSOD Embarrassments

French Guyana, June 4, 1996
$800 million software failure
Lost contact due to real-time priority inversion bug

$4 billion development effort
> 50% system integration & validation cost

400 horses
100 microprocessors

Software Reliability
- Computers are:
  - pervasive
  - complex (Windows XP is 50 million lines of code)
- Reliability is important
  - convenience
  - security
  - economic
  - safety

Towards Reliable Software
- Testing
  - how much, how long, at what cost?
- Code Reviews
- Programming Languages
  - GC, OOP, exceptions
- Extended Checkers
  - tools to find bugs not caught by compiler
  - static or dynamic
  - ex: Purify or Lint for C
  - earlier than testing, modular, better coverage

Programming With Threads
- Decompose into pieces that run in parallel
- Exploit multiple processors
  - Moore’s Law is over
  - desktops, multicore chips, ...
- Threads make progress, even if others are blocked
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-Free Bank Account

class Account {
    private int bal = 0;
    public void deposit(int n) {
        synchronized(this) {
            int j = bal;
            bal = j + n;
        }
    }
    public int read() {
        synchronized(this) {
            int j = bal;
            return j;
        }
    }
}

Preventing Race Conditions Using Locks

- Lock can be held by at most one thread at a time
- Associate a lock with each variable
- Acquire lock before accessing variable

Thread 1

synchronized(m) {
    int j1 = bal;
    bal = j1 + n;
}

Thread 2

synchronized(m) {
    int j2 = bal;
    bal = j2 + n;
}

// other thread can update bal
Optimized Bank Account

class Account {
    private int bal = 0;

    public int read() {
        return bal;
    }

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

Race-Freedom

- Race-freedom is neither necessary nor sufficient to ensure the absence of errors

- Is there a more fundamental semantic correctness property?

Atomicity

- A method is atomic if concurrent threads do not interfere with its behavior

- Informally, a method behaves as if it is the only method executing in the system

Atomicity

- Serialized execution of deposit

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

- Non-serialized executions of deposit

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

- deposit is \textit{atomic} if, for every non-serialized execution, there is a serialized execution with the same overall behavior

Tools for Checking Atomicity

- Benefits of Atomicity
  - stronger property than absence of data races
  - enables sequential reasoning
  - simple specification

- Tools
  - Calvin-R: ESC for multithreaded code (2 KLOC) [Freund-Qadeer 03]
  - A type system for atomicity (20 KLOC) [Flanagan-Qadeer 03, Flanagan-Freund-Lifshin 05]
  - Atomizer dynamic atomicity checker (200 KLOC) [Flanagan-Freund 04]
Calvin-R

```java
/* global_invariant (\forall int i; inodeLocks[i] == null ==> 0 <= inodeBlocknos[i] && inodeBlocknos[i] < Daisy.MAXBLOCK) */
//@ requires 0 <= inodenum && inodenum < Daisy.MAXINODE;
//@ requires i != null
//@ requires DaisyLock.
//@ modifies i.blockno, i.size, i.used, i.inodenum
//@ ensures i.blockno == inodeBlocknos[inodenum]
//@ ensures i.size == inodeSizes[inodenum]
//@ ensures i.used == inodeUsed[inodenum]
//@ ensures i.inodenum == inodenum
//@ ensures 0 <= i.blockno && i.blockno < Daisy.MAXBLOCK
static void readi(long inodenum, Inode i) {
    i.blockno = Petal.readLong(STARTINODEAREA + (inodenum * Daisy.INODESIZE));
    i.size = Petal.readLong(STARTINODEAREA + (inodenum * Daisy.INODESIZE) + 8);
    i.used = Petal.read(STARTINODEAREA + (inodenum * Daisy.INODESIZE) + 16) == 1;
    i.inodenum = inodenum;
    // read the right bytes, put in inode
}
//@ nowarn Post
```

Atomizer: Documenting Atomicity

- Manual annotations
  ```java
  //@ atomic */ void append(...) {
  ```
- Heuristics
  - all synchronized blocks are atomic
  - all public methods are atomic, except main and run
  - these heuristics are very effective

Warning: method "append" may not be atomic at line 43
Lockset Algorithm [Savage et al 97]

- 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

- Reduction algorithm leverages race information

Lockset Example

Thread 1

synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}

Thread 2

synchronized(y) {
  o.f = 2;
}

.o.f = 11;

• First access to o.f:
  LockSet(o.f) := Held(curThread)
  = \{x, y\}

• Subsequent access to o.f:
  LockSet(o.f) := LockSet(o.f) \cap 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;
}

.o.f = 11;

• Subsequent access to o.f:
  LockSet(o.f) := LockSet(o.f) \cap Held(curThread)
  = \{x\} \cap \{y\} = \{\} \rightarrow \text{RACE!}

Lockset

Extending Lockset (Thread Local Data)

Thread Local

first thread

second thread

race condition!
Extending Lockset (Read Shared Data)

Thread Local

Shared-read/write Track lockset

race condition!

Reduction [Lipton 75]

acq(this) X j-bal Y bal+j-n Z rel(this)

S0 → S1 → S2 → S3 → S4 → S5 → S6 → S7

Reduction [Lipton 75]

acq(this) X j-bal Y bal+j-n Z rel(this)

S0 → S1 → S2 → T1 → S4 → S5 → S6 → S7

Reduction [Lipton 75]

acq(this) X j-bal Y bal+j-n Z rel(this)

S0 → T1 → S2 → T1 → S4 → S5 → S6 → S7

Reduction [Lipton 75]

acq(this) X j-bal Y bal+j-n Z rel(this)

S0 → T1 → S2 → T1 → S4 → S5 → S6 → S7

blue thread holds lock after acquire
⇒ operation x does not modify lock
⇒ operations commute

blue thread holds lock
⇒ red thread does not hold lock
⇒ operation y does not access balance
⇒ operations commute
Reduction [Lipton 75]

\[
\begin{align*}
S_0 & \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_7 \\
acq(this) & \quad X \quad j=bal \quad Y \quad bal\rightarrow j+n \quad Z \quad rel(this)
\end{align*}
\]

Code Classification

- right: lock acquire
- left: lock release
- both-mover: race-free variable access
- atomic: conflicting variable access

- Reducible blocks match regular expression: \((right|both)^*\text{atomic} (left|both)^*\)

Dynamic Reduction

- Automata to check \((R|B)^* [A] (L|B)^*\)

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 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 on all the individual threads
 * involved.
 */
```
Reporting Errors

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 ...);
    ...
  }
}

Reporting Errors

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 ...);
    ...
  }
}

Atomizer Review

- Instrumented code calls Atomizer run time
  - 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

Redundant Lock Operations

- Acquire is right-mover
- Release is left-mover
- Redundant lock operations are both-movers
  - acquiring/releasing a thread-local lock
  - re-entrant acquire/release
  - operations lock A, if lock B always acquired before A

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>hadc</td>
<td>29,900</td>
<td>6.4</td>
<td>-</td>
</tr>
<tr>
<td>lap</td>
<td>700</td>
<td>1.9</td>
<td>21.8</td>
</tr>
<tr>
<td>sort</td>
<td>17,700</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>mX/Java</td>
<td>1,100</td>
<td>90.6</td>
<td>0.00</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>mnt</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>26.6</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>

All public methods are atomic, except main and run

Evaluation

- Warnings: 97  (down from 341- extensions necessary!)
- 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 (more later)
- No warnings reported in more than 90% of exercised methods
Example Bugs

```java
class PrintWriter {
    Writer out;
    public void println(String s) {
        synchronized(lock) {
            out.print(s);
            out.println();
        }
    }
}
```

```java
class ResourceStoreManager {
    synchronized checkClosed() { ... }
    synchronized lookup(...) { ... }
    public ResourceStore loadResourceStore(...) {
        checkClosed();
        return lookup(...);
    }
}
```

Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
- Other atomicity checkers
  - [Wang-Stoller 03], Bogor model checker [Hatcliff et al 03]
  - view consistency [Artho-Biere-Havelund 03, von Praun-Gross 03]
- Race conditions
  - dynamic: [Savage et al 97, O'Callahan-Choi 01, vonPraun-Gross 01]
  - static: Warlock [Sterling 93], Rcc/Java [Flanagan-Freund 00], ...
- Languages without data races
  - Guava [Bacon et al 01]

Atomizer Summary

- Atomicity
  - enables sequential analysis
  - matches practice

- Improvements over race detectors
  - catches "higher-level" concurrency errors
  - some benign races do not break atomicity