Atomicity

- The method inc() is atomic if concurrent threads do not interfere with its behavior
- Guarantees that for every execution
  
\[
acq(this) \quad x \quad t=i \quad y \quad i=s+1 \quad z \quad rel(this)
\]
- Acquire and release inc()

\[
acq(this) \quad x \quad y \quad t=i \quad i=s+1 \quad z \quad rel(this)
\]
- There is a serial execution with same behavior

\[
acq(this) \quad x \quad acq(this) \quad i=s \quad rel(this) \quad z
\]
- Atomic segment

Tools for Checking Atomicity

- Calvin: ESC for multithreaded code
  - [Freund–Qadeer 03]
- A type system for atomicity
  - [Flanagan–Qadeer 03, Flanagan–Freund–Lifshin 05]

Experience with Calvin

```c
/*@ 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.inodeLocks[inodenum] == 	id
//@ 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
}
```

Tools for Checking Atomicity

- Calvin: ESC for multithreaded code
  - [Freund–Qadeer 03]
- A type system for atomicity
  - [Flanagan–Qadeer 03, Flanagan–Freund–Lifshin 05]
- Atomizer: dynamic atomicity checker
  - [Flanagan–Freund 04]
  - [http://www.soe.ucsc.edu/~cormac/atom.html]
Atomizer: Instrumentation Architecture

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

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

event stream

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 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:

  LockSet(o.f) = Held(curThread)
  = { x, y }

Thread 1
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}

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

• Subsequent access to o.f:

  LockSet(o.f) := LockSet(o.f) \cap Held(curThread)
  = { x, y } \cap { x }
  = { x }             => race condition

Lockset

any thread

Shared-read/write
Track lockset

race condition!

Lockset with Thread Local Data

Thread Local

Shared-read/write
Track lockset

race condition!

Lockset with Read Shared Data

Thread Local

Shared-read/write
Track lockset

race condition!

Read Shared

any thread

any thread

second thread

write

any thread

second thread

read

any thread

any thread

write
Dynamic Analysis for Atomicity

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

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

**Reducible methods**: $(R|B)^* [N] (L|B)^*$

**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

---

Java.lang.StringBuffer

```java
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();
    // ... other threads can change sb
    sb.getChars(..., len, ...);
  }
}
```

**Use of stale len may yield**

StringIndexOutOfBoundsException inside getChars(...)
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

Refining Race Information

- Discovery of races during reduction

```java
/*# atomic */
void deposit(int n) {
    synchronized (this) {
        int j = bal;
        // other thread changes bal
        bal = j + n;
    }
}
```

Extensions

- Redundant lock operations
  - acquire is right-mover
  - release is left-mover
  - Want to treat them as both movers when possible
- Write-protected data
  - common idiom

Thread–Local Locks

```java
class Vector {
    atomic synchronized Object get(int i) { ... }
    atomic synchronized void add(Object o) { ... }
}
class WorkerThread {
    atomic void transaction() {
        Vector v = new Vector();
        v.add(x1);
        v.add(x2);
        ...
        v.get(i);
    }
}
```

Reentrant Locks

```java
class Vector {
    atomic synchronized Object get(int i) { ... }
    atomic synchronized Object add(Object o) { ... }
    atomic boolean contains(Object o) {
        synchronized(this) {
            for (int i = 0; i < size(); i++)
                if (get(i).equals(o)) return true;
        }
        return false;
    }
}
```

Layered Abstractions

```java
class Set {
    Vector elems;
    atomic void add(Object o) {
        synchronized(this) {
            if (!elems.contains(o)) elems.add(o);
        }
    }
}
```
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
  - acquiring/releasing lock A, if lock B always acquired before A

Class: Account

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

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;
    }
  }
}
```

- Lock `this` held whenever balance is updated
  - write must hold lock, and is non-mover
  - read without lock held is non-mover
  - read with lock held is both-mover

Lockset for Write-Protected Data

- Track access lockset and write lockset
  - access lockset = locks held on every access
  - write lockset = locks held on every write
- For regularly-protected data
  - access lockset = write lockset = { protecting lock }
- For write-protected data
  - access lockset = Ø
  - write lockset = { write-protecting lock }
- Read is both-mover if some write lock held
- Write is both-mover if access lockset nonempty

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>hbed</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>sin</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>raytracing</td>
<td>1,900</td>
<td>4.8</td>
<td>41.8</td>
</tr>
<tr>
<td>mini</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>26.4</td>
<td>12.1</td>
</tr>
<tr>
<td>webj</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 Reduce Number of Warnings

Evaluation

- Warnings: 97 (from 200KLOC)
- At least 7 are real errors
- 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

Example Bugs

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

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