Lightweight Analyses For Reliable Concurrency

Stephen Freund
Williams College (currently on leave at UCSC)

joint work with Cormac Flanagan (UCSC) and Shaz Qadeer (MSR)

---

Towards Reliable Multithreaded Software

- Multithreaded software
  - increasingly common (Java, C#, GUIs, servers)
  - trends will continue
    - multi-core chips

- Heisenbugs due to thread interference
  - race conditions
  - atomicity violations

---

Programming With Threads

- Decompose program into pieces that can run in parallel
- Advantages
  - exploit multiple processors
  - threads make progress, even if others are blocked

---

400 horses
100 microprocessors
class Account {
    private int bal = 0;
    public void deposit(int n) {
        int j = bal;
        bal = j + 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.

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.

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

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

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 due to unexpected interactions between threads.
- 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 the same regardless of what else is happening.

Motivations for Atomicity

1. Stronger property than absence of data races
   - bad race-free programs
   - good "racy" programs

2. Enables sequential reasoning

Sequential Program Execution

Multithreaded Execution
Multithreaded Execution

- Guarantees concurrent threads do not interfere with atomic method
- Enables sequential reasoning

Motivations for Atomicity
1. Stronger property than absence of data races
2. Enables sequential reasoning
3. Simple specification

Model Checking of Software Models

Specifying for filesystem.c

Model Construction

Model Checker

Experience with Calvin Software Checker

Calvin theorem proving

Calvin software checker
Experience with Calvin Software Checker

The Need for Atomicity

Sequential case: code inspection & testing mostly ok

Motivations for Atomicity

1. Stronger property than absence of data races
2. Enables sequential reasoning
3. Simple specification

Atomcity
• Serialized execution of deposit

```
class Account {
    int bal;
    void deposit(int n) {
        synchronized (this) {
            int j = bal;
            bal = j + n;
        }
    }
}
```

• Non-serialized executions of deposit

1. 

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

• deposit is atomic if, for every non-serialized execution, there is a serialized execution with the same overall behavior
Atomicity

- Canonical property
  - (linearizability, serializability, ...)
- Enables sequential reasoning
  - simplifies validation of multithreaded code
- Matches practice in existing code
  - most methods (>80%) are atomic
  - many interfaces described as "thread-safe"
- Can verify atomicity statically or dynamically
  - violations often indicate errors
  - leverages Lipton’s theory of reduction

Reduction [Lipton 75]

\[
\begin{array}{cccc}
S_0 & S_1 & S_2 & S_3 \\
\uparrow X & \downarrow j-bal & \downarrow Y & \downarrow bal+j-n \\
S_4 & S_5 & S_6 & S_7 \\
\end{array}
\]

blue thread holds lock
⇒ red thread does not hold lock
⇒ operation y does not access balance
⇒ operations commute

Reduction [Lipton 75]

\[
\begin{array}{cccc}
S_0 & S_1 & S_2 & S_3 \\
\uparrow X & \downarrow j-bal & \downarrow Y & \downarrow bal+j-n \\
S_4 & S_5 & S_6 & S_7 \\
\end{array}
\]

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

\[\text{acq(this)} \rightarrow S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7\]

\[\text{acq(this)} \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7\]

\[\text{acq(this)} \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7\]

\[\text{acq(this)} \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7\]

\[\text{acq(this)} \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7\]

\[\text{acq(this)} \rightarrow S_5 \rightarrow S_6 \rightarrow S_7\]

\[\text{acq(this)} \rightarrow S_6 \rightarrow S_7\]

\[\text{acq(this)} \rightarrow S_7\]

**Movers**

- **right-mover**
  - lock acquire

- **left-mover**
  - lock release

- **both-mover**
  - race-free field access

- **non-mover (atomic)**
  - access to “racy” fields
**Code Classification**

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

*reducible blocks have form:  
(right|both)[atomic](left|both)*

```java
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 threads all involved.  
*/  
/**@ atomic */ public class StringBuffer {  
    ...  
}
```

```java
java.lang.StringBuffer

/*@ atomic */ public class StringBuffer {
    private int count;
    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
    public synchronized void append(StringBuffer sb){
        int len = sb.length();  
        ...  
        ...  
        sb.getChars(...,len,....);
        ...  
    }
}
```

**Composing Atomicities**

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

```java
acq(this) j-bal bal=j+n rel(this)
right both both
left

atomic

compound
```

**Tools for Checking Atomicity**

- 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]

http://www.cs.williams.edu/~freund/atom.html
Calvin-R

/* 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]) ==tid
//@ 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
}

Lightweight Tools For Atomicity

• Part 1
  - Runtime analysis
  - practical aspects of building / validating tools
• Part 2
  - Type systems for concurrency and atomicity
• Part 3
  - Beyond reduction
  - "Purity", abstraction, commit-atomicity, ...

Atomizer: Documenting Atomicity

• Manual annotations
  - /*# atomic */ void append(...) { ... }

• Heuristics
  - all synchronized blocks are atomic
  - all public methods are atomic, except main and run
  - these heuristics are very effective

Atomizer: Instrumentation Architecture

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;
}

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

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

- Subsequent access to o.f:
  LockSet(o.f) := LockSet(o.f) ∩ Held(curThread)
  = {x, y} ∩ {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:
  LockSet(o.f) := LockSet(o.f) ∩ Held(curThread)
  = {x, y} ∩ {x} = {}

RACE CONDITION!
**Extending Lockset (Thread Local Data)**

- First thread: r/w
- Second thread: r/w

**Extending Lockset (Read Shared Data)**

- First thread: r/w
- Second thread: read

**Reduction [Lipton 75]**

- Reducible blocks have form \((R|B)^* \{A\} (L|B)^*\)

**Movers**

- **R**: right-mover
  - lock acquire
- **B**: both-mover
  - race-free field access
- **L**: left-mover
  - lock release
- **A**: atomic
  - access to "racy" fields

**Dynamic Reduction**

- Automata to check \((R|B)^* \{A\} (L|B)^*\)

**Reporting Errors**

```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(...)
    }
}
```
### 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

### 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 Data

```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

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

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;
        }
    }
}

- 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

Extending Lockset for Write-Prot 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 at least one write lock held
- Write is both-mover if access lockset not empty

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 - 45x

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>bace</td>
<td>29,900</td>
<td>6.4</td>
<td></td>
</tr>
<tr>
<td>sp</td>
<td>700</td>
<td>1.9</td>
<td>21.8</td>
</tr>
<tr>
<td>wr</td>
<td>17,500</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>meldyn</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>mnt</td>
<td>11,500</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>web3</td>
<td>22,500</td>
<td>60.3</td>
<td></td>
</tr>
<tr>
<td>lib-java</td>
<td>75,505</td>
<td>96.5</td>
<td></td>
</tr>
</tbody>
</table>
Extensions Reduce Number of Warnings

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”
    - e.g. 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();
        }
    }
}

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

Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - types [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]
  - view consistency [Artho-Biere-Havelund 03, von Praun-Gross 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, Grossman 03]
  - Guava [Bacon et al 01]

Software Transactions

- Language support for lightweight transactions
  - [Fraser-Harris 03], [Harris-Marlow-Jones-Herlihy 05]
- Example:
  ```java
  transaction (x > 10) {
      x = x - 10;
  }
  ```
  - Run-time waits until condition is true and then executes body “atomically”
    - no programmer-inserted concurrency control

Naive Lock-Based Implementation

- Acquire global lock when entering transaction, release lock when exiting
- Can try to use more fine-grained locking
  - hard to scale
  - hard to do automatically
- Database solutions?
Optimistic Concurrency

- Run transaction as normal code
- Log reads/writes to shared variables (but do not modify them)
- On commit, check whether interference has occurred
  - have shared variables been modified?
  - if so, discard log and retry
  - if not, commit logged changes
- STM (Software Transactional Memory)
  - run-time support for tracking shared variables, modification history, etc.

Transactions vs. Atomicity

- Orthogonal techniques
  - programmer vs. run-time control
- Transactions avoid:
  - deadlocks, priority inversion, ...
- Can transactions scale?
  - overhead, retry rate, non-"undoable" ops
  - large transactions
- Combining atomicity and transactions
  - optimize transactions
  - focus on most critical performance bottlenecks

Atomizer Summary

- Atomicity
  - enables sequential analysis
  - matches practice
- Improvements over race detectors
  - catches "higher-level" concurrency errors
  - some benign races do not break atomicity
- Next steps
  - expressiveness
  - hybrid tools

Looking Ahead

- Limitations of Atomizer
  - coverage
  - whole program
  - annotating large code bases
- Static type systems
  - modular checking
  - inferring specifications
  - computational / expressiveness issues