Runtime Atomicity Analysis of Multi-threaded Programs

Focus is on the paper:
“Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs”
by C. Flanagan and S. Freund

presented by Sebastian Burckhardt
University of Pennsylvania
CIS 700 – Runtime Verification Seminar
Wednesday, October 20, 2004
Outline of talk

• Verification of multithreaded programs in general
• Atomizer: the core concepts
  – Dynamic analysis
  – Reduction
  – Lock set algorithm
• Atomizer: the improvements
• Atomizer: evaluation
Correctness of Multithreaded Programs

• “Multithreaded” means: concurrent, communication by shared memory
• Reasoning quite challenging even for experts
• Typically, programmers use fairly low-level synchronization primitives
  – Mutex, Locks
  – Semaphores
  – Monitors (re-popularized by Java)
• To make it worse, performance matters
  (otherwise, why bother with multiple threads?)
Non-dynamic verification

- We won’t talk about these today.
  - Restrict design space
    - type systems
    - special-purpose languages
    - Design paradigms
  - Static analysis
    - Lexical
    - Control flow
    - Data flow
Checking concurrent executions

• Problem: number of possible concurrent executions very large
• Approach I: Check them all
  – means: model check the concurrent model
  – not practical without heavy abstraction
• Approach II: Check just one
  – this is the regular “testing” method
• Approach III: Check one, and extrapolate
  – look for bad things that “could” happen
What are the bad things we can look for?

• Deadlock

• Races
  – Definition of “race”: Two threads are allowed to access same variable at the same time, and at least one access is a write

• View inconsistency
  – Intuitive description: grouping of variables inconsistent among threads

• Lack of atomicity
What are we looking for?

• Deadlock
  – look for inconsistent order of lock acquisition

• Races
  – look for variables that aren’t consistently protected by some lock, by tracking locks held during each access (e.g. “Eraser” Lockset alg)

• View inconsistency
  – track variable sets associated which each lock (e.g. in JPaX, JNuke)

• Atomicity
  – Reduction-based (e.g. Atomizer)
  – Block based (e.g. Wang/Stoller’s tool)
Atomicity Checking: Advantages

• Can find bugs that are resistant to regular testing, and race detection

• Good correspondence with programming methodology
  – easy to understand the idea
  – can verify interfaces, encouraging code reuse
  – programmer can gain confidence in code by validating atomicity assumptions

• Scalable
  – has been applied to >100k lines of Java code
Example: java.lang.StringBuffer

```java
public final class StringBuffer {

    public synchronized StringBuffer append(StringBuffer sb) {
        int len = sb.length();
        ...
        ...
        ...
        ...
        sb.getChars(0, len, value, count);
        ...
    }

    public synchronized int length() { ... }  
    public synchronized void getChars(...) { ... }  
}
```
public final class StringBuffer {

    public synchronized StringBuffer append(StringBuffer sb) {
        int len = sb.length();
        ... // another thread can modify sb here
        ... // => len is no longer the correct length of len
        ... // but there is no race.
        sb.getChars(0, len, value, count);
        ...
    }

    public synchronized int length() { ... }
    public synchronized void getChars(...) { ... }
}
Definition

• A block of code is ‘atomic’ if for every legal execution of the program, there is an equivalent legal execution within which the entire block executes without preemption.

• Executions are “equivalent” iff
  – the (dynamic) instruction stream per thread is identical
  – the same read reads the value of the same write
How does it work? (1)

• Identify blocks that are supposed to be atomic
  – use heuristics
    • exported methods
    • synchronized methods
    • synchronized blocks
  – allow user annotations
    • can ‘turn off’ the checking if there are false bugs
    • can do additional checks by declaring atomic
      
      /*# atomic */ void getChars() { ... }
How does it work? (2)

• Perform instrumentation on the source code level
  – could also be done at the bytecode level
  – Instrumented source code produces event stream during execution

• Analyze event stream on-line (Atomizer) or off-line.
  – For each block that is supposed to be atomic, check whether there is an equivalent execution in which it is scheduled contiguously.
How does it work? (3)

• We can’t possibly check all possible executions to find an equivalent atomic one.
• Idea: Find a large class of instruction sequences for which we can always guarantee that it can be shuffled into an uninterrupted sequence by local, pairwise swaps.
• Then, warn user if supposedly atomic block does not belong to this class.
• -> Lipton’s theory of reduction (1975)
Semantic model

- Dynamic instruction stream of each thread consists of 4 types of instructions:
  - \( rd(x,v) \)  
    read value \( v \) from shared var \( x \)
  - \( wr(x,v) \)  
    write value \( v \) to shared var \( x \)
  - \( acq(m) \)  
    acquire lock \( m \)
  - \( rel(m) \)  
    release lock \( m \)
Left-movers

- Can always swap an \texttt{rel(m)} with an interleaved instruction \texttt{j1} of another thread to its left. Call this a \texttt{left-mover}.

- Reason
  - can always release lock earlier
  - read/write matching not affected by move
Right-movers

• Can always swap an \textit{acq(m)} with an interleaved instruction \textit{j1} of another thread to its right. Call this a \textbf{right-mover}.

\[ i_0 \quad \text{acq(m)} \quad i_2 \quad j_0 \quad j_1 \quad i_0 \quad \text{acq(m)} \quad i_2 \quad j_0 \quad j_1 \]

• Reason
  – lock is still available (\textit{j1} can not be \textit{acq(m)})
  – read/write matching not affected by move
Non-movers

- Neither rd(x,v) nor wr(x,v) can in general be swapped with an adjacent interleaved instruction of another thread. Call them non-movers.
Both-movers

- If an access \( \text{rd}(x,v), \text{wr}(x,v) \) goes to a variable protected by a lock which is held by this thread, it is a both-mover.

\[
\text{acq}(m) \quad \text{rd}(x,v) \quad \text{rel}(m) \]

- Reason
  - \( j1 \) can not be an access to \( x \)
- Suppose for now we know which locks can protect a variable
Lipton’s Reduction

• Let’s denote the instructions as follows: L for left-mover, R for right-mover, N for non-mover, B for both-mover

• Then any execution sequence matching the following regular expression is equivalent to an atomic one:

\[(R + B)^* (N + \varepsilon) (L + B)^*\]

• Examples: RL RBL NLLLB RNL BBB

• But not: NN LR
Example

• Say the method “copy()”

```java
public class A {
    private int x, y;
    public synchronized void copy() {
        y = x;
    }
    ...
    ...
}
```

• produces the dynamic instruction stream

```
acq(m)  rd(x,4)  wr(y,4)  rel(m)
```

• is it atomic?
  – For now, assume all methods of class A are synchronized
Example

\[
\begin{align*}
&\text{acq(m)} & \text{rd}(x,4) & \text{wr}(y,4) & \text{rel}(m) \\
& b1 & b2 & b3
\end{align*}
\]

\[
\begin{align*}
&\text{acq(m)} & \text{rd}(x,4) & \text{wr}(y,4) & \text{rel}(m) \\
& b1 & b2 & b3
\end{align*}
\]
Implementation

• Can efficiently check if blocks match 
  \((R + B)^* (N + \varepsilon) (L + B)^*\) 
  by using an online automaton.

• Problem: to classify variable accesses 
  correctly, we need to know which locks 
  protect which
Which locks with which field?

Fields may not be protected by this object’s lock

```java
public class A2 {
    private int x, y;
    public synchronized swap() { int z = y; y = x; x = z; }
    public int getX() { return x; }
    public int getY() { return y; }
    ...
}
```

Field may be protected by a different object’s lock

```java
public class A2 {
    private int x, y;
    Integer mylock = new Integer(0);
    public copy() { synchronized (mylock) { y = x; } }
    public int getX() { synchronized (mylock) { return x; } }
    public int getY() { synchronized (mylock) { return y; } }
    ...
}
```
Basic “Eraser” lockset algorithm

• Argue: “If a variable is consistently protected by some lock, this lock must be held during all accesses to that variable”

• Dynamically, we can look at the set of locks held during each access so far, and keep track of their intersection
  – If the intersection is empty, there seems to be no consistent locking discipline - classify access as a non-mover
  – Otherwise, there seems to be a consistent locking discipline - classify access as a both-mover

• What about re-classifying accesses if changes occur during runtime?
  – can’t be done on-line, but could be done off-line
Improve Classification (1)

- Avoid flagging some classic, safe usages
  - thread-local variables: need no lock to protect them
  - initialization: one thread initializes data, then passes it to another thread, thread-local from there on
  - Write once, read many times

- Track state for each field
  - use lock set for classification only if in state Shared Modified
Improve Classification (2)

- **Re-entrant locks**
  - re-entrant acquires and releases are both-movers

- **Redundant locks**
  - if a lock is only accessed by one thread, it is redundant (thread-local locks)
  - if lock B is always held while accessing lock A, lock A is redundant (protected locks)
  - redundant acquires and releases are both-movers
  - can classify locks using the same lockset and thread-access algorithms as for fields
Improve Classification (3)

• “Benign” read/write races

```java
public class A2 {
    private int x;
    public int read() { return x; }
    synchronized void inc() { x = x + 1; }
}
```

• `read()` and `inc()` are atomic... (more or less)
  – track separate lockset containing locks held during all writes (= superset of locks held during all accesses)
  – classify read as both-mover if current thread holds a write lock, even if access-protecting lockset is empty
It’s not that easy

- Unsynchronized reads and writes
  - are not atomic if more than 32 bit quantity
    - more rules exist (e.g. volatile vs. non-volatile)
  - are not guaranteed to proceed in order
    - only synchronization events are sequentially consistent.
    - memory model relative to hardware is specified (?)
    - memory model of hardware is not specified.
    - does anybody know?
  - does Atomizer need adjustments for non-sequentially consistent machines?
## Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines</th>
<th>Num. Threads</th>
<th>Num. Locks</th>
<th>Max. Locks Held</th>
<th>Num. Lock Set Pairs</th>
<th>Base Time (s)</th>
<th>Atomizer Slowdown</th>
<th>Atomicty Warnings</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>elevator</td>
<td>529</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>17</td>
<td>11.14</td>
<td>—</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>hadc</td>
<td>29,948</td>
<td>26</td>
<td>385</td>
<td>3</td>
<td>728</td>
<td>8.36</td>
<td>—</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>tsp</td>
<td>706</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>0.94</td>
<td>48.2</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>sor</td>
<td>17,690</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0.70</td>
<td>7.3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>moldyn</td>
<td>1,291</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3.62</td>
<td>11.8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3,557</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>7.94</td>
<td>2.2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,859</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>5.96</td>
<td>36.6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mtrt</td>
<td>11,315</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>2.33</td>
<td>46.4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>jigsaw</td>
<td>90,100</td>
<td>53</td>
<td>706</td>
<td>31</td>
<td>4,531</td>
<td>13.49</td>
<td>4.7</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>specJBB</td>
<td>30,490</td>
<td>10</td>
<td>262,000</td>
<td>6</td>
<td>340,088</td>
<td>18.01</td>
<td>11.2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>webl</td>
<td>22,284</td>
<td>5</td>
<td>402,445</td>
<td>3</td>
<td>452,685</td>
<td>60.35</td>
<td>—</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>lib-java</td>
<td>75,305</td>
<td>39</td>
<td>816,617</td>
<td>6</td>
<td>986,855</td>
<td>96.5</td>
<td>—</td>
<td>19</td>
<td>4</td>
</tr>
</tbody>
</table>
Effect of improvements
Atomizer paper: contributions

• Concise review of concepts
  – Formal semantics for multithreaded programs
  – Reduction idea, Lockset algorithm

• Description of the algorithm and some improvements
  – Formal description of the algorithm, formulation of theorem describing its correctness, in provable detail
  – Mentions optimizations: handle re-entrant locks, thread-local locks, protected Locks, write-protected data

• Experimental evaluation of the tool
  – performance, scale, usability
Bibliography


