SimpleLock: Fast and Accurate Hybrid Data Race Detector

Misun Yu
2013. 11. 20
Table of Contents

- Introduction
- Overview
  - Previous Tools
  - Goals and Solutions
- SimpleLock Algorithm
- Evaluation
- Conclusion
Multithread programing

- Nondeterministic thread scheduling
- Difficulty in finding and resolving concurrency bugs
  - Data races, deadlocks, atomicity violations, etc.
Data Race

❖ Occurs when

- There is no synchronization dependence between the two accesses of different threads, and
- At least one thread performs a write: write-write, write-read, read-write

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 ( x = 2; )</td>
<td>( y? )</td>
</tr>
<tr>
<td>1.2 ( y = x; )</td>
<td>( 2.1 \ x = 0; )</td>
</tr>
</tbody>
</table>

❖ Impacts

US Northeastern blackout in 2003

Radiation overdoes by Therac-25 in the 1980s
Dynamic Race Detection

- Happens-before (ordering-based) analysis
  - Traces program order and synchronization order among threads => detect unordered access pairs
  - Keeps vector clocks
  - Precise but scheduling sensitive

```
<table>
<thead>
<tr>
<th>VC1</th>
<th>VC2</th>
<th>Lm</th>
<th>Wx</th>
<th>W'x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;4,0&gt;</td>
<td>&lt;0,8&gt;</td>
<td>&lt;0,0&gt;</td>
<td>&lt;0,0&gt;</td>
<td>null</td>
</tr>
<tr>
<td>wr(0,x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;4,0&gt;</td>
<td>&lt;0,8&gt;</td>
<td>&lt;0,0&gt;</td>
<td>&lt;4,0&gt;</td>
<td>4@0</td>
</tr>
<tr>
<td>rel(0,m)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;5,0&gt;</td>
<td>&lt;0,8&gt;</td>
<td>&lt;4,0&gt;</td>
<td>&lt;4,0&gt;</td>
<td>4@0</td>
</tr>
<tr>
<td>acq(1,m)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;5,0&gt;</td>
<td>&lt;4,8&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wr(1,x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;5,0,..&gt;</td>
<td>&lt;4,8&gt;</td>
<td>&lt;4,0&gt;</td>
<td>&lt;4,0&gt;</td>
<td>4@0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Djit+    FastTrack
Dynamic Race Detection

- **Lockset analysis**
  - Locking consistency
  - Scheduling insensitive but many false positives

<table>
<thead>
<tr>
<th>Thread</th>
<th>t1</th>
<th>Thread</th>
<th>t2</th>
<th>Lockset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>lock m</td>
<td>1.2</td>
<td>write x</td>
<td>{m}</td>
</tr>
<tr>
<td>1.3</td>
<td>read x</td>
<td>1.4</td>
<td>unlock m</td>
<td>{m}</td>
</tr>
<tr>
<td>1.5</td>
<td>t2.start()</td>
<td>2.1</td>
<td>write x</td>
<td>∅</td>
</tr>
</tbody>
</table>

Eraser
Dynamic Race Detection

- Hybrid
  - Happens before + Lockset
  - Happens before: events that determine the specific thread execution order such as start, join, notify, etc.
  - Lockset: accesses of different threads that have no specific order
  - Small # of false positives but slow

Thread t1
1.1 write x
1.2 t2.start()
1.3 write x
1.4 lock m
1.5 write x
1.6 unlock m

Thread t2
2.1 lock m
2.1 read x
2.3 unlock m

A potential data race missed by happens-before detectors
Goals

- To detect real & potential data races: scheduling insensitivity
- To remove false positives
- To reduce execution overhead for runtime detection
Solutions

- We simplify the detection procedure to check locking inconsistency
  - By checking the existence of locks protecting a shared variable.
    - Low execution overhead
    - No false positives due to tracking nested locks

- We use a property of a data race.
  - The distance between the accesses is not long.
    - Low memory overhead
    - Low execution overhead
Detecting Write-Write races

- Queues keeping \(<\text{clock}_t, \# \text{ of locks}>\) pairs of threads.
- Length: Q\_LEN

\[ VC_1 \]
<4,0> wr(1,x)
<4,0> t2.start()
<5,0> acq(1,x)
<5,0> wr(1,x)
<5,0> rel(1,x)
<5,0>

\[ VC_2 \]
<4,1> wr(2,x)
<4,1> wr(1,x)
<4,1> rel(1,x)
<4,1>

\[ WC_{1x} \quad WC_{2x} \]
[4,0] \rightarrow [1,0]
[5,1]

Write-Write Race
Detecting Write-Read races
Detecting Read-Write races

VC₁

<5,0> acq(1,x)
<5,0> wr(1,x)
<5,0> rel(1,x)
<5,0> rd(1,x)

VC₂

<4,1> wr(2,x)
<4,1> wr(2,x)
<4,1> wr(2,x)
<4,1> wr(2,x)

WC₁ₓ

[4,0]

WC₂ₓ

[2,0]

RC₁ₓ

[5,1]

[5,0]

Read-Write Race
- **Missing data races**
  - When all accesses have at least one lock protecting them:

<table>
<thead>
<tr>
<th>Thread t1</th>
<th>Thread t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 lock m</td>
<td>2.1 lock n</td>
</tr>
<tr>
<td>1.2 write x</td>
<td>2.2 write x</td>
</tr>
<tr>
<td>1.3 unlock m</td>
<td>2.3 unlock n</td>
</tr>
</tbody>
</table>
Evaluation (1)

- Three data race detectors
  - FastTrack, AccuLock-Multi and two versions of SimpleLock with different QUEUE_LENs
  - All use RoadRunner

- Benchmarks
  - 11 multithreaded programs from PJBench: avrora, luindex, lusearch, sunflow, batik, etc.

- Platform
  - Intel Core i-7-3770K (quad core) CPU 3.50 GHz machine with 16GB RAM and operating system 64-bit Ubuntu 12.4.
## Benchmark results

<table>
<thead>
<tr>
<th>Program</th>
<th># of warnings</th>
<th>Execution overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FT</td>
<td>SL</td>
</tr>
<tr>
<td>avrora</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>luindex</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lusearch</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>sunflow</td>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>batik</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cache4j</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>elevator</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hdec</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>sor</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tsp</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>weblech</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Average</td>
<td>2.7</td>
<td>5.4</td>
</tr>
</tbody>
</table>

**FT:** FastTrack, **SL:** SimpleLock (Q_LEN:1), **SL’:** SimpleLock (Q_LEN: limitless), **AL:** Acculock-Multi
# Evaluation

## # of data races

- SimpleLock and Acculock-Multi detected the same data races.
  
  \[ \Rightarrow \text{"Most of data races are caused by accesses to shared variables not protected by lock"} \]

- SL and SL’ detected the same data races although they have different queue lengths.
  
  \[ \Rightarrow \text{"The distance between accesses causing a data race is not long"} \]

## Execution overhead

- SimpleLock << AccuLock-Multi
- \[ 1.34 \times \text{SimpleLock} \approx \text{FastTrack} \]
SimpleLock is a novel dynamic detector that provides broad coverage and a low execution overhead.

SimpleLock’s high performance stems from the simplified Lockset that only checks for the existence of accesses protected by no locks and the fact that the distance between two accesses causing a data race is not long.

The proposed algorithm may be useful for C/C++ program analysis.