Maximal Causality Reduction for TSO and PSO

Shiyou Huang    Jeff Huang
huangsy@tamu.edu
Parasol Lab, Texas A&M University
A Real PSO Bug – $12 million loss of equipment

```java
curPos = new Point(1, 2);

class Point {
  int x, y;
}

Thread 1:
newPos = new Point(curPos.x + 1, curPos.y + 1);

Thread 2:
while (newPos != null)
  if (newPos.x + 1 != newPos.y)
    ERROR
```

http://stackoverflow.com/questions/16159203/
Memory Consistencies

http://preshing.com/20120930/weak-vs-strong-memory-models/
TSO and PSO

Total Store Ordering (TSO)
For a write $w$ and a read $r$ by the same thread, the read $r$ can be reordered with the write $w$ if the two operations access different locations.

Partial Store Ordering (PSO)
For a write $w_1$ and a write $w_2$ by the same thread, the write $w_2$ can be reordered with the write $w_1$ if the two operations access different locations.
New State Generated under TSO/PSO

Init: \( x=y=0 \)

thread 1:  
\[
\begin{align*}
x &= 1 & // a1 \\
a &= y & // a2
\end{align*}
\]

thread 2:  
\[
\begin{align*}
y &= 1 & // b1 \\
b &= x & // b2
\end{align*}
\]

Assert \( (a==1 || b==1) \)

\[
b2 - a1 - a2 - b1 \quad (a=0, b=0)
\]
Huge Interleaving Space

#interleaving = \[ \prod_{i=1}^{M} \left( \sum_{j=i}^{M} N_j \right) \]  

(Lu et al. FSE’07)

(M : #threads and N_i : #accesses by thread i)

M=4, N1=N2=N3=N4=4, #interleavings > 60 million
Related Work

• Dynamic Partial Order Reduction (DPOR) [Flanagan et al., POPL’05]
• Maximal Causality Reduction [Huang, PLDI’15]
• rInspect [Zhang et al., PLDI’15]
• SATCheck [Demskey and Lam, OOPSLA’15]
Maximal Causality Reduction (MCR)

Given an executed trace, MCR generates new interleavings to explore the program state space. Each new interleaving (called seed interleaving) enforces at least one read to read a new value.
Workflow of MCR

1. Interleaving
2. Scheduler
3. Trace
4. Constraints Formula
5. SMT Solver

Seed Interleavings
Interleaving 1
Interleaving 2
...
Interleaving n

New Seed Interleavings

$\Phi_{mhb}$
$\Phi_{lock}$
$\Phi_{validity}$
$\Phi_{state}$
Workflow of MCR

Following a seed interleaving will produce a new state.
Constraints ($\phi$)

- **happens-before**

- **lock-mutual-exclusion**

  \[(l_1, u_1) \text{ and } (l_2, u_2): O_{u_1} < O_{l_2} \lor O_{u_2} < O_{l_1}\]

- **validity**

  \[
  \Phi_{\text{value}}(r, v) \equiv \bigvee_{w \in W^x_v} (\Phi_{\text{validity}}(w) \land O_w < O_r \\
  \land (O_{w'} < O_w \lor O_r < O_{w'}))
  \]

- **new state**
Constraints (φ)

- **happens-before**

- **lock-mutual-exclusion**

\[(l_1, u_1) \text{ and } (l_2, u_2): O_{u_1} < O_{l_2} \lor O_{u_2} < O_{l_1}\]

- **validity**

An event is feasible if every read in the seed interleaving returns the same value as that in the previous trace.
Constraints ($\phi$)

- happens-before
- lock-mutual-exclusion

$$(l_1, u_1) \text{ and } (l_2, u_2): O_{u_1} < O_{l_2} \lor O_{u_2} < O_{l_1}$$

- validity

$$\Phi_{value}(r, v) \equiv \bigvee_{w \in W^x_v} (\Phi_{validity}(w) \land O_w < O_r)$$

$$\land_{w \neq w' \in W^x} (O_{w'} < O_w \lor O_r < O_{w'})$$

- new state
An Example

Init: \( x = y = 0 \)

thread 1:
\[
\begin{align*}
x &= 1 & //a1 \\
a &= y & //a2 \\
\end{align*}
\]

thread 2:
\[
\begin{align*}
y &= 1 & //b1 \\
b &= x & //b2 \\
\end{align*}
\]

S0: \( a1-a2-b1-b2 \) (\( a=0, b=1 \))

\[
\begin{align*}
O_{b1} &< O_{a2} \\
O_{a1} &< O_{a2} \\
O_{b2} &< O_{a1} \\
O_{b1} &< O_{b2} \\
\end{align*}
\]

S1: \( a1- b1 - a2 \) (\( a=1, b=1 \))

S2: \( b1 - b2 \) (\( a=1, b=0 \))

3 executions
Limitation of MCR

The original MCR only checks the program under sequential consistency.
Limitation of MCR

Init: \( x=y=0 \)

thread 1:
\[
\begin{align*}
x &= 1 \quad // a1 \\
a &= y \quad // a2
\end{align*}
\]

thread 2:
\[
\begin{align*}
y &= 1 \quad // b1 \\
b &= x \quad // b2
\end{align*}
\]

Assert \((a==1 \text{ || } b==1)\)
Contributions

- Extend MCR for TSO and PSO
- Present a new replay algorithm
- Evaluation on various applications
- Explore 5x – 10x fewer executions than DPOR
Two Challenges

1. Relax the happens-before constraints
2. Replay a schedule out of the program order
Happens-before Relaxation

Relax the happens-before relation of the write-read and write-write events by the same thread:

\[ \phi_{hb} = \begin{cases} 
\phi_{rr} & r_1 < r_2, \text{ iff } r_1, r_2 \in \text{Reads} \\
\phi_{addr} & e_1 < e_2, \text{ iff } \text{addr}(e_1) = \text{addr}(e_2) \\
\phi_{r-w} & r < w, \text{ iff } r \in \text{Reads} \land w \in \text{Writes} \\
\phi_{w-w} & w_1 < w_2, \text{ iff } w_1, w_2 \in \text{Writes} 
\end{cases} \]
Example

Init: x=y=0
thread 1:  
x = 1    //a1
a = y    //a2

thread 2:  
y = 1    //b1
b = x    //b2

Under SC:
O_{a1} < O_{a2}
O_{b1} < O_{b2}

Under TSO/PSO
O_{a1}, O_{a2}, O_{b1}, O_{b2}
Replay

Expecting: b2 – a1 – a2 – b1

Actual: b1 – a1 – a2 – b2

Can’t decide whether to buffer
Replay

Interleaving: a sequence of schedule choices, with each schedule choice c(tid, addr).

Case 1: when addr(e) ≠ addr(c), buffer e

Case 2: when addr(c) = addr(w), w is buffered, update w
A feasible schedule: 1-2-3-6-7-8-4-5 that can trigger the error!
Replay

thread 1: thread 2:
1. z=0   7. if (z>0)
2. x=0   8. assert (x+1 == y)
3. y=0
4. x=1
5. y=2
6. z=1

Scheduler

Addr doesn’t match

Addr doesn’t match

Replay: 1 - 2 - 3 - 6 - 7 - 8 - 4 - 5

1: z=0
2: x=0
3: y=0
6: z=1
7: z>0
8: x+1
4: x=1
Evaluation

• Java implementation using ASM and Z3
• Compared with rInspec [Zhang et al., PLDI’15] and SATCheck [Demsky and Lam, OOPSLA’15]
  ➢ States pace exploration effectiveness
  ➢ Efficiency of finding errors
• A collection of benchmarks with known errors
Benchmarks

- 7 popular small benchmarks
- 6 real Java applications including a large one weblech

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>#Thrd</th>
<th>#Evt</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker</td>
<td>119</td>
<td>3</td>
<td>56</td>
<td>Two critical sections with 3 shared variables.</td>
</tr>
<tr>
<td>Lamport</td>
<td>162</td>
<td>3</td>
<td>40</td>
<td>Two critical sections with 4 variables.</td>
</tr>
<tr>
<td>bakery</td>
<td>119</td>
<td>3</td>
<td>27</td>
<td>n critical sections using 2n shared variables. We take n=2.</td>
</tr>
<tr>
<td>Peterson</td>
<td>94</td>
<td>3</td>
<td>72</td>
<td>Two critical sections with 3 variables</td>
</tr>
<tr>
<td>StackUnsafe</td>
<td>135</td>
<td>3</td>
<td>34</td>
<td>Unsafe operations on a stack by two threads, which cause the stack underflow.</td>
</tr>
<tr>
<td>RVExample</td>
<td>79</td>
<td>3</td>
<td>32</td>
<td>An example from original MCR [21], which contains a very tricky error</td>
</tr>
<tr>
<td>Example</td>
<td>73</td>
<td>2</td>
<td>44</td>
<td>The example program from Figure 6 with loop number from 1 to 4.</td>
</tr>
<tr>
<td>Account</td>
<td>373</td>
<td>5</td>
<td>51</td>
<td>Concurrent account deposits and withdrawals suffering from atomicity violations.</td>
</tr>
<tr>
<td>Airline</td>
<td>136</td>
<td>6</td>
<td>67</td>
<td>A race condition causing the tickets oversold.</td>
</tr>
<tr>
<td>Allocation</td>
<td>348</td>
<td>3</td>
<td>125</td>
<td>An atomicity violation causing the same block allocated or freed twice.</td>
</tr>
<tr>
<td>PingPong</td>
<td>388</td>
<td>6</td>
<td>44</td>
<td>The player is set to null by one thread and dereferenced by another throwing NPE.</td>
</tr>
<tr>
<td>StringBuf</td>
<td>1339</td>
<td>3</td>
<td>70</td>
<td>An atomicity violation in Java StringBuffer causing StringIndexOutOfBoundsException.</td>
</tr>
<tr>
<td>Weblech</td>
<td>35K</td>
<td>3</td>
<td>2045</td>
<td>A tool for downloading websites and enumerating standard web-browser behavior.</td>
</tr>
</tbody>
</table>
## State Space Exploration

<table>
<thead>
<tr>
<th>Program</th>
<th>DPOR (rInspect)</th>
<th>MCR (our approach)</th>
<th>#Executions Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SC</td>
<td>TSO</td>
<td>PSO</td>
</tr>
<tr>
<td>Dekker</td>
<td>248</td>
<td>252</td>
<td>508</td>
</tr>
<tr>
<td>Lamport</td>
<td>128</td>
<td>208</td>
<td>2672</td>
</tr>
<tr>
<td>Bakery</td>
<td>350</td>
<td>1164</td>
<td>2040</td>
</tr>
<tr>
<td>Peterson</td>
<td>36</td>
<td>95</td>
<td>120</td>
</tr>
<tr>
<td>StackUnsafe</td>
<td>252</td>
<td>252</td>
<td>252</td>
</tr>
<tr>
<td>RVExample</td>
<td>1959</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Example (N=1 to 4)</td>
<td>4</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>105</td>
<td>105</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4282</td>
<td>4282</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>14840</td>
<td>14840</td>
<td>-</td>
</tr>
<tr>
<td>Avg.</td>
<td>435</td>
<td>394</td>
<td>1118</td>
</tr>
</tbody>
</table>
State Space Exploration

Our approach explores 5x – 10x fewer executions than DPOR.

<table>
<thead>
<tr>
<th>Program</th>
<th>DPOR (rlnspect)</th>
<th>MCR (our approach)</th>
<th>#Executions Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SC</td>
<td>TSO</td>
<td>PSO</td>
</tr>
<tr>
<td>Dekker</td>
<td>248</td>
<td>252</td>
<td>508</td>
</tr>
<tr>
<td>Lamport</td>
<td>128</td>
<td>208</td>
<td>2672</td>
</tr>
<tr>
<td>Bakery</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peterson</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StackUnsafe</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RVExample</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example (N=1 to 4)</td>
<td>105</td>
<td>105</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4282</td>
<td>4282</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>14840</td>
<td>14840</td>
<td>-</td>
</tr>
<tr>
<td>Avg.</td>
<td>435</td>
<td>394</td>
<td>1118</td>
</tr>
</tbody>
</table>
# Finding Bugs

<table>
<thead>
<tr>
<th>Program</th>
<th>DPOR</th>
<th></th>
<th></th>
<th>SATCheck</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SC</td>
<td>TSO</td>
<td>PSO</td>
<td>SC</td>
<td>TSO</td>
<td>SC</td>
</tr>
<tr>
<td>Dekker</td>
<td>22</td>
<td>28</td>
<td>29</td>
<td>32!</td>
<td>68735!</td>
<td>10</td>
</tr>
<tr>
<td>Lamport</td>
<td>6</td>
<td>8</td>
<td>24</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Bakery</td>
<td>12</td>
<td>15</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>Peterson</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>19*</td>
<td>34282!</td>
<td>7</td>
</tr>
<tr>
<td>StackUnsafe</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>RVExample</td>
<td>301</td>
<td>-</td>
<td>-</td>
<td>60564!</td>
<td>70365!</td>
<td>53</td>
</tr>
<tr>
<td>Example</td>
<td>14840*</td>
<td>14840*</td>
<td>-</td>
<td>1*</td>
<td>1*</td>
<td>2767*</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>10</strong></td>
<td><strong>12</strong></td>
<td><strong>16</strong></td>
<td>-</td>
<td>-</td>
<td><strong>6</strong></td>
</tr>
</tbody>
</table>

!: repeat the same execution  
*: finish without finding the bug
## Finding Bugs

<table>
<thead>
<tr>
<th>Program</th>
<th>DPOR</th>
<th>SATCheck</th>
<th>MCR (our approach)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SC</td>
<td>TSO</td>
<td>PSO</td>
</tr>
<tr>
<td>Dekker</td>
<td>32</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>Lamport</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bakery</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peterson</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StackUnsafe</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RVExample</td>
<td>301</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Example</td>
<td>14840*</td>
<td>14840*</td>
<td>-</td>
</tr>
<tr>
<td>Avg.</td>
<td>10</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

Our approach needs 2X-3X fewer executions than DPOR and SATCheck to find the bugs.

!: repeat the same execution
*: finish without finding the bug
Conclusion

1. MCR for TSO and PSO
   • Relax the happens-before constraints
   • Faithfully replay the TSO/PSO interleavings

2. Explore 5X – 10X fewer executions than DPOR

3. Take fewer executions to find the bugs
Acknowledgement

Brian Demsky
University of California, Irvine

Patrick Lam
University of Waterloo
Thank you
&
Questions?