CONLOCK

A Constraint-Based approach to dynamic checking on Deadlocks in multithreaded programs

Yan Cai, Shangru Wu, W.K. Chan
ICSE 14
Problem outline

Find (real) deadlocks

**Static techniques**
- Analyze code
- Many false positives

**Dynamic techniques**
- „Educated“ scheduling
- Still many false positives
Goals

(a) Find potential deadlocks

(b) Automatically confirm potential deadlocks
   ✓ Eliminate false positives
   ✓ Do \textit{not} eliminate true positives
**CONLOCK**

## Concepts

### Events
- Lock acquisition or release

### Lockset
- Set of locks hold by one thread

### Cycles
- Chain of events $\varepsilon$, that build a circular dependency
- Potential deadlock

![Diagram](image-url)
### Example

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Events</strong></td>
<td><strong>Events</strong></td>
</tr>
<tr>
<td>01 acq(n)</td>
<td>10 acq(a)</td>
</tr>
<tr>
<td>02 rel(n)</td>
<td>11 rel(a)</td>
</tr>
<tr>
<td>03 acq(a)</td>
<td>12 acq(n)</td>
</tr>
<tr>
<td>04 acq(n)</td>
<td>13 acq(p)</td>
</tr>
<tr>
<td>05 rel(n)</td>
<td></td>
</tr>
<tr>
<td>06 acq(p)</td>
<td></td>
</tr>
<tr>
<td>07 acq(m)</td>
<td></td>
</tr>
<tr>
<td>08 acq(n)</td>
<td></td>
</tr>
</tbody>
</table>

Cycle c

$\epsilon_1$

$\epsilon_2$
Approach

Phase 0: Identify cycles

Phase 1: Generate constraints
  ▶ Analyze order of operations
  ▶ Provoke deadlock

Phase 2: Educated scheduling of execution
  ▶ No violation of any constraint
  ▶ Trigger deadlock (if any)
Phase 1: Constraints

Should happen before relation: $e_1 \sim e_2$

**Rule 1:**
Deadlock event $\varepsilon_a$ on $t_a$ is an acq. of lock $o$.
⇒ All operations of any thread $t_{b \neq a}$ on $o$ must happen before $\varepsilon_a$.

**Rule 2:**
Thread $t_a$ holds lock on object $o$ from $e_1$ until its deadlock event $\varepsilon_a$.
⇒ All operations (except $\varepsilon_b$) of any thread $t_{b \neq a}$ on $o$ must happen before $e_1$. 
**Phase 1: Example**

<table>
<thead>
<tr>
<th>thread 1</th>
<th>Events</th>
<th>Locksets</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>acq(n)</td>
<td>{ , , , , }</td>
</tr>
<tr>
<td>02</td>
<td>rel(n)</td>
<td>{ , ,n, , }</td>
</tr>
<tr>
<td>03</td>
<td>acq(a)</td>
<td>{ , , , , }</td>
</tr>
<tr>
<td>04</td>
<td>acq(n)</td>
<td>{a, , , , }</td>
</tr>
<tr>
<td>05</td>
<td>rel(n)</td>
<td>{a, ,n, , }</td>
</tr>
<tr>
<td>06</td>
<td>acq(p)</td>
<td>{a, , , , }</td>
</tr>
<tr>
<td>07</td>
<td>acq(m)</td>
<td>{a, , ,p}</td>
</tr>
<tr>
<td>08</td>
<td>acq(n)</td>
<td>{a,m, ,p}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 2</th>
<th>Events</th>
<th>Locksets</th>
</tr>
</thead>
<tbody>
<tr>
<td>09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>acq(a)</td>
<td>{ , , , , }</td>
</tr>
<tr>
<td>11</td>
<td>rel(a)</td>
<td>{a, , , , }</td>
</tr>
<tr>
<td>12</td>
<td>acq(n)</td>
<td>{ , , , , }</td>
</tr>
<tr>
<td>13</td>
<td>acq(p)</td>
<td>{ , ,n, , }</td>
</tr>
</tbody>
</table>

**Rule 1:**

Deadlock event $\varepsilon_a$ on $t_a$ is an acq. of lock $o$.

$\Rightarrow$ All operations of any thread $t_{\beta \neq a}$ on $o$ must happen before $\varepsilon_a$. 

### Phase 1: Example

| 01 | acq(n) | \{ , , , \} |
|    | rel(n) | \{ , ,n, \} |
| 03 | acq(a) | \{ , , , \} |
| 04 | acq(n) | \{a, , , \} |
| 05 | rel(n) | \{a, ,n, \} |
| 06 | acq(p) | \{a, , , \} |
| 07 | acq(m) | \{a, , ,p\} |
| 08 | acq(n) | \{a,n, ,p\} |
|    | acq(a) | \{ , , , \} |
|    | rel(a) | \{a, , , \} |
|    | acq(n) | \{ , , , \} |
|    | acq(p) | \{ , ,n, \} |

**Rule 2:**

Thread $t_\alpha$ holds lock on object $o$ from $e_1$ until its deadlock event $\varepsilon_\alpha$.

$\Rightarrow$ All operations (except $\varepsilon_\beta$) of any thread $t_{\beta \neq \alpha}$ on $o$ must happen before $e_1$. 
Phase 1: Optimization

Reduce Constraint set
- Transitivity
- Program Locking Order

Nearest Scheduling points
- Nearest operation where lockset is empty
- Only consider operations from NSPs
Phase 2: Scheduling

Schedule randomly (by OS)

- Keep track of constraints
- Only „non-violating“ operations get performed

No progress possible?

- Deadlock? Output trace and halt. Success!
Analysis

Experiment

- Code from JDBC Connector, SQLite and MySQL Server
- Compare with other deadlock detectors (PCT, MagicScheduler, DeadlockFuzzer)
- 100 runs each

- Test precision and efficiency for known deadlock
- Test efficiency for false positives
Analysis

Deadlock detection probability

Runtime
Analysis

False positives

- Analysed 87 false positives
- All other deadlock detection algorithms timed out
- All but one run of Conlock showed scheduling violations

→ Probabilistic method to discard cycles
Conclusion

Limitations

- Sufficient test set?
- False positives? → Manual inspection!
- Can it find unknown bugs?

Contributions

- Successful new approach
- Significantly improved precision
Thank you.