Correct Parallel Software
UNDER RELAXED MEMORY MODELS
JACOB BURNIM, KOUSHIK SEN, CHRISTOS STERGIOU

Introduction

- Software developers write programs with intentional data races
  - Highly-concurrent libraries, lock-free data structures
  - Custom synchronization operations
  - Avoid cost of synchronization on certain frequent operations
- In the presence of data races, sequential consistency is no longer guaranteed
- Sequential Consistency (SC)
  - Lamport: “... the result of any execution is the same as if the operations of all of the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

- Relaxed memory consistency:
  - Total Store Order (TSO): allows stores to be reordered past later loads, but maintains a total order over stores
  - Partial Store Order (PSO): TSO + allows stores to be reordered past later stores of different addresses
- Reasoning about memory models can be hard:
  - Initially x = y = 0
    thread1:  thread2:
    1: x = 1  3: y = 1
    2: t1 = y  4: t2 = x
    assert(t1 == 1 || t2 == 1)

- assertion can fail under TSO or PSO

Problem

- Model-checking is intractable with added non-determinism from underlying memory-model

Idea

- Despite ah-hoc synchronization, programmers expect their program to be sequentially consistent
- Sequential Consistency (SC) violations are likely to be bugs
- Can we find SC violations just by exploring SC executions of a program? [Burckhardt et al.]

Our Approach

- Devise monitoring algorithms for TSO and PSO
- Monitor algorithms are sound and complete
- Given SC violation, re-execute program and check if violation exposes a bug or not
- Based on intuitive operational simulation instead of complex axiomatic semantics
- Yields simple algorithms (complex proofs)

Monitoring

Active Testing

Problem

- Quickly find and reproduce memory model bugs
- Model checking can be expensive even with monitor
- Violations of sequential consistency are not always bugs

Our Solution: Active Testing

- 2-phase analysis and testing approach for predicting and confirming concurrency bugs
- Phase I: run program once and predict potential violations of sequential consistency
- Phase II: attempt to create potential violation by actively controlling thread schedule and underlying memory

Our Solution: Active Testing

- Initially x = y = done = 0
  thread1 {
    1: x = 1;
    2: y = 1;
    3: done = 1;
  }
  thread2 {
    4: if (done) {
      5: if (true) {
        6: ERROR;
        7: local = y;
      }
    }

Phase I predicts cycle : (1,3,4,5)
Phase II creates cycle by buffering write to x by thread1 and delaying thread2 at instruction 4

<table>
<thead>
<tr>
<th>Bench mark</th>
<th>Cycles predicted</th>
<th>Cycles Confirmed</th>
<th># of Bugs</th>
<th>Probability of confirming cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TSO</td>
<td>PSO</td>
<td>PSLO</td>
<td>TSO</td>
</tr>
<tr>
<td>dekker</td>
<td>112</td>
<td>47</td>
<td>45</td>
<td>69</td>
</tr>
<tr>
<td>bakery</td>
<td>222</td>
<td>36</td>
<td>75</td>
<td>100</td>
</tr>
<tr>
<td>min</td>
<td>459</td>
<td>0</td>
<td>177</td>
<td>144</td>
</tr>
<tr>
<td>ma2</td>
<td>75</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>lazylist</td>
<td>192</td>
<td>0</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>harris</td>
<td>172</td>
<td>0</td>
<td>54</td>
<td>49</td>
</tr>
<tr>
<td>mark</td>
<td>1800</td>
<td>0</td>
<td>647</td>
<td>404</td>
</tr>
</tbody>
</table>