An Effective Dynamic Analysis for Detecting Generalized Deadlocks
Pallavi Joshi*  Mayur Naik†  Koushik Sen*  David Gay‡
* UC Berkeley  † Intel Labs Berkeley  ‡ Google Inc

Motivation

- Deadlocks are a common problem in today’s multithreaded software.
  - 6,500/198,000 of bug reports in Sun’s bug database are deadlocks
- Two categories of deadlocks
  - Resource deadlocks
  - Communication deadlocks
- Most of previous work has exclusively focused on resource deadlocks
- In this work, we address both kinds of deadlocks and also deadlocks that are hybrid between these two

Example

Program: An Effective Dynamic Analysis for Detecting Generalized Deadlocks

<table>
<thead>
<tr>
<th>Program</th>
<th># condition</th>
<th>Original Program LOC</th>
<th>Trace Program LOC</th>
<th>Original Program Run Time</th>
<th>Time to generate trace program</th>
<th># of errors</th>
<th>Potential Deadlocks (HC/W)</th>
<th>Confirmed Deadlocks (HC/W)</th>
<th>Known Deadlocks (HC/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>groove</td>
<td>1</td>
<td>45,796</td>
<td>59 0.118h</td>
<td>1s &gt; 1h</td>
<td>3.3s</td>
<td>5</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
</tr>
<tr>
<td>log4j</td>
<td>2</td>
<td>48,023</td>
<td>235 0.116h</td>
<td>3s &gt; 1h</td>
<td>6.7s</td>
<td>107</td>
<td>1/0</td>
<td>1/0</td>
<td>0/0</td>
</tr>
<tr>
<td>pool</td>
<td>4</td>
<td>48,024</td>
<td>136 0.116h</td>
<td>3s &gt; 1h</td>
<td>3.3s</td>
<td>41</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
</tr>
<tr>
<td>felix</td>
<td>4</td>
<td>48,011</td>
<td>292 0.123s</td>
<td>3s &gt; 1h</td>
<td>9s</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>lucene</td>
<td>9</td>
<td>81,071</td>
<td>1,534 0.296s</td>
<td>3s &gt; 1h</td>
<td>30s</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>jgroups</td>
<td>12</td>
<td>92,034</td>
<td>318 0.228s</td>
<td>4s &gt; 1h</td>
<td>39s</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>javan</td>
<td>17</td>
<td>122,006</td>
<td>3,509</td>
<td>-</td>
<td>&gt; 1h</td>
<td>7894</td>
<td>2/7</td>
<td>2/7</td>
<td>1/2</td>
</tr>
<tr>
<td>jruby</td>
<td>18</td>
<td>134,476</td>
<td>966 1.1s</td>
<td>&gt; 1h</td>
<td>5.9s</td>
<td>18</td>
<td>1/0</td>
<td>1/0</td>
<td>1/0</td>
</tr>
<tr>
<td>jgroups</td>
<td>15</td>
<td>180,644</td>
<td>2,545 9.88h</td>
<td>21s &gt; 1h</td>
<td>124</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>java-log</td>
<td>0</td>
<td>43,795</td>
<td>131 0.177s</td>
<td>2s &gt; 1h</td>
<td>37.7h</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>apache</td>
<td>0</td>
<td>90,821</td>
<td>400 0.74s</td>
<td>1s &gt; 1h</td>
<td>96s</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>spring</td>
<td>0</td>
<td>264,528</td>
<td>3,155 0.98s</td>
<td>17.6h</td>
<td>105s</td>
<td>0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
</tbody>
</table>

Results

- Implementation
  - Implemented our technique in a prototype tool for Java called CHECKMATE
  - Experimented with a number of Java libraries and applications: log4j, pool, felix, lucene, jgroups, jruby, ...
  - Found both previously known and unknown deadlocks (17 in total)

Future/Ongoing Work

- Trace program approach would also be effective for other concurrency errors that cannot be detected using an idiom
  - e.g. deadlocks because of errors or exceptions

  // Thread 1
  while (lb) {
    try {
      sync (l) {
        foo();
        L.wait();
        b = true;
        sync (l) (L.notify());
        } catch (Exception e) […]
  } // Thread 2

b is initially false

Motivation

- Observe a program execution
- Retain only the synchronization operations observed during execution
- Throw away all other operations like method invocations and memory updates
- Create a program from the retained operations (trace program)
- Model check trace program
- Trace program is usually much more tractable to model check than the original program

Example

public class TraceProgram {
  static Object L = new Object();
  static boolean b;
  static Thread main = new Thread();
  public void run() {
    if (!b) {
      t1.start();
      t2.start();
    }
    static Thread t1 = new Thread() {
      public void run() {
        b = true;
        synchronized (L) {
          L.wait();
        }
      }
    }
    public static void main(String[] args) {
      t1.run();
      t2.run();
      t1.stop();
    }
  }

Implementation

- Implemented our technique in a prototype tool for Java called CHECKMATE
- Experimented with a number of Java libraries and applications: log4j, pool, felix, lucene, jgroups, jruby, ...
- Found both previously known and unknown deadlocks (17 in total)