Effective static analysis to find concurrency bugs in Java

Zhi Da (Daniel) Luo, Linda Hillis, Raja Das, Yao Qi

IBM China Software Development Lab
Background

- **Multicore processors have become mainstream**
  - Need to develop concurrent software to fully exploit hardware performance

- **Difficult and error prone to write Java concurrent program**
  - Concurrency bugs often not reproducible, due to non-deterministic thread scheduling
  - Fundamental misconceptions about concurrency in Java
  - Intentionally fragile code is created to improve performance

- **Practical analysis techniques that identify concurrent bugs are valuable!**
Existing Analysis Techniques for Concurrency Bugs

- **Dynamic analysis**
  - Can reveal most concurrent bugs, such as data races, deadlock
  - Limited to finding bugs in the program paths that are actually executed
  - Incurs runtime overhead, thus prevented from running frequently

- **Model checking**
  - Systematically explores all possible thread schedules
  - Depends on the construction of a good model
  - Suffers from state-space explosion

- **Static analysis**
  - Deep analysis based on graphs
    - Gives fewer false negatives
    - Reports many false positives (infeasible paths and imprecise program information)
    - Non-scalable to large real-world applications
  
  - Bug patterns matching
    - Effective to find real bugs
    - Efficient analysis, scalable to large applications
    - Inaccurate, finds both false negatives and false positives

Area we focus to improve
Static Analysis for Concurrency Bug Patterns

Our solution: Practical Static Concurrency Bug Patterns Detector for Real-world Applications (RSAR)

- **Define Concurrency Bug Patterns**
  - Code idioms that violate correct Java multithreaded programming practices
  - 7 commonly-seen Java multithreaded bug patterns
  - Bug pattern variants that cheat detectors

- **Approaches for Different Bug Patterns**
  - Syntactically match source code with Abstract Syntax Tree
    - Novel but simple heuristics and enhancements for analysis precision and performance
    - e.g. Estimate whether a class is multithreaded or not by searching synchronization primitives
  - Inter-procedural data flow analysis based on WALA
    - For efficiency, prune call graph to include only a subset of necessary methods
    - Alias analysis using selective equality predicates without whole-program alias analysis

---

### SUMMARY OF BUG PATTERNS

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VF</td>
<td>Non-atomic Operations on Volatile Field Without A Lock Held</td>
</tr>
<tr>
<td>IS</td>
<td>Inconsistent Synchronized Monitor and Receiver of wait()/notify()/notifyAll()</td>
</tr>
<tr>
<td>LL</td>
<td>java.util.concurrent Lock Leak</td>
</tr>
<tr>
<td>DC</td>
<td>Double Checked Locking</td>
</tr>
<tr>
<td>SN</td>
<td>Synchronized and Null Check on The Same Field</td>
</tr>
<tr>
<td>SW</td>
<td>Spin Wait</td>
</tr>
<tr>
<td>SS</td>
<td>Synchronized Setter Method Non-synchronized Similarly-name Getter Method</td>
</tr>
</tbody>
</table>
Accuracy & Performance

- **Accuracy**
  - Tested with 4 large real-world applications.
  - Over 65% warnings are harmful.

- **Performance**
  - Fast analysis, 16 sec to analyze 5M LOC for the slowest rule.

**Experimental Environment:**

Intel(R) Pentium(R) 4 CPU 2.60GHz, 1.5G memory, Windows XP Professional, RSAR 7.1.0

<table>
<thead>
<tr>
<th>ID</th>
<th>warnings</th>
<th>Jetty-7.0.2 (160KLOC, 677 files)</th>
<th>Derby-10.5 (542KLOC, 1950 files)</th>
<th>Glassfish-2.1-B60 (2235KLOC, 9751 files)</th>
<th>Commercial Software (&gt;5000KLOC, 19199 files)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>warnings</td>
<td>harmful</td>
<td>harmless</td>
<td>false pos</td>
</tr>
<tr>
<td>VF</td>
<td>2</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>IS</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>684</td>
</tr>
<tr>
<td>DC</td>
<td>1</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>174</td>
</tr>
<tr>
<td>SN</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>SW</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>190</td>
</tr>
<tr>
<td>SS</td>
<td>4</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>437</td>
</tr>
</tbody>
</table>
Comparison with existing tool

- **RSAR (IBM)**
  - Accurate and Efficient
  - Experiment with 4 large real-world applications
  - Over 65% warnings are real bugs.
  - Slowest rule takes 16 sec to analyze 5MLOC application.
  - Inter-procedural data flow analysis

- **FindBugs (Open source)**
  - Rich set of multithreaded bug patterns
  - Fast analysis
  - Intra-procedural data flow analysis
  - Numerous false positives and false negatives
    - Linear scan through the byte code,
    - Coarse-grained code match
    - Fail to consider bug pattern variants

---

### Table: Comparison of RSAR and FindBugs

<table>
<thead>
<tr>
<th>ID</th>
<th>Small test examples</th>
<th>Derby-10.5</th>
<th>Jetty-7.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real Bugs</td>
<td>RSAR</td>
<td>FindBugs</td>
</tr>
<tr>
<td>DC</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>SN</td>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>SW</td>
<td>11</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>SS</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

---

- **Typical spin wait that is not reported in FindBugs**
  ```java
  if (streamLength <= 0)
    return streamLength;
  
  boolean pushStack = false;
  try {
    // we have a stream
    synchronized (getConnectionSynchronization()) {
      pushStack = !getEmbedConnection().isClosed();
      if (pushStack) setupContextStack();
    }
  }

  while (!flag);
  
  synchronized (obj) {
    if (obj == null) {
      // swallow
    }
  }
  
  Simple sync-null-check bug that is not reported in FindBugs

- **False DCL alarms in FindBugs**

Bugs found in real applications

- **Jetty**: Non-atomic self-increment operation on volatile field _set in class SelectorManager (JETTY-1187) is confirmed.

```java
// Jetty 7.1.0,
// org.eclipse.jetty.io.nio,
// SelectorManager.java, line 103
private volatile int _set;
...
public void register(SocketChannel channel, Object att)
{
    int sm_set++;
    ...
}
public void addChange(Object point)
{
    synchronized (_changes)
    {
        ...
    }
}
```

- **Eclipse, Glassfish**: Broken double checked locking bugs are confirmed.
  - 13 broken double checked locking bugs found in Glassfish, confirmed by community developers (Bug-11383)
  - 1 bug found in Eclipse IDE source code, confirmed and bug state was changed from “Unconfirmed” to “New” (Bug 302536)

- **Derby**: Uses an instance lock to protect static shared data in EmbedPooledConnection (DERBY-4723) is fixed.

EmbedPooledConnection has the unsafe synchronization as follow.

```java
private static int idCounter = 0;
private synchronized int nextId()
{
    return idCounter++;
}
```

Kristian Vagaan added a comment - 28Jul10 08:56 AM
Attached patch 1a, which removes the code using incorrect synchronization.

Widely-used commercial concurrent software:
- Spin wait

```java
appM.uninstallApplicationLocal(
    appName, options, this,
    opContext.getSessionID());
while (_waitTarget != null)
......; // wait for notification
```
Conclusion

- Building an accurate and efficient Java concurrency bug patterns detector is not so difficult.
  - Combine simple code matching analysis with novel heuristics and enhancements
  - Use inter-procedural data flow analysis with optimized techniques

- Bug patterns detector is very effective at finding real bugs.

- Concurrency bugs widely exist in real-world applications!
Controversial Statement

- Simple analysis tools (e.g. static concurrency bug patterns detector) suffices to most software developers in practice.

Discussion

- Security vulnerabilities related to concurrency?
Questions?