Effective Static Deadlock Detection

Mayur Naik*
Chang-Seo Park†, Koushik Sen†, David Gay*

*Intel Research, Berkeley †UC Berkeley
What is a Deadlock?

- An unintended condition in a shared-memory, multi-threaded program in which:
  - a set of threads blocks forever
  - because each thread in the set waits to acquire a lock being held by another thread in the set

- This work: ignore other causes (e.g., wait/notify)

Example

```c
// thread t1
sync (l1) {
    sync (l2) {
        ...
    }
}

// thread t2
sync (l2) {
    sync (l1) {
        ...
    }
}
```
Motivation

• Today’s concurrent programs are rife with deadlocks
  – 6,500/198,000 (~ 3%) of bug reports in Sun’s bug database at http://bugs.sun.com are deadlocks

• Deadlocks are difficult to detect
  – Usually triggered non-deterministically, on specific thread schedules
  – Fail-stop behavior not guaranteed (some threads may be deadlocked while others continue to run)

• Fixing other concurrency bugs like races can introduce new deadlocks
  – Our past experience with reporting races: developers often ask for deadlock checker
Previous Work

- Based on detecting cycles in program’s dynamic or static lock order graph

- Dynamic approaches
  - Inherently unsound
  - Inapplicable to open programs
  - Can be ineffective without sufficient test input data

- Static approaches
  - Type systems (e.g., Boyapati-Lee-Rinard OOPSLA’02)
    - Annotation burden often significant
  - Model checking (e.g., SPIN)
    - Does not currently scale beyond few KLOC
  - Dataflow analysis (e.g., Engler & Ashcraft SOSP’03; Williams-Thies-Ernst ECOOP’05)
    - Scalable but highly imprecise
Challenges to Static Deadlock Detection

- Deadlock freedom is a complex property
  - can $t_1,t_2$ denote different threads?
  - can $l_1,l_4$ denote same lock?
  - can $t_1$ acquire locks $l_1 \rightarrow l_2$?
  - some more ...
Deadlock freedom is a complex property:
- can $t_1, t_2$ denote different threads?
- can $l_1, l_4$ denote same lock?
- can $t_1$ acquire locks $l_1 \rightarrow l_2$?

Existing static deadlock checkers cannot check all conditions simultaneously and effectively.

But each condition can be checked separately and effectively using existing static analyses.

- Existing static deadlock checkers cannot check all conditions simultaneously and effectively.
- Can $l_1, l_4$ denote same lock?
- Can $t_1$ acquire locks $l_1 \rightarrow l_2$?
- Some more ...
Our Approach

- Consider all candidate deadlocks in closed program
- Check each of six necessary conditions for each candidate to be a deadlock
- Report candidates that satisfy all six conditions
- Note: Finds only deadlocks involving 2 threads/locks
  - Deadlocks involving > 2 threads/locks rare in practice
Example: jdk1.4 java.util.logging

class LogManager {
    static LogManager manager = new LogManager();

    Hashtable loggers = new Hashtable();

    boolean addLogger(Logger l) {
        String name = l.getName();
        if (!loggers.put(name, l))
            return false;
        // ensure l’s parents are instantiated
        for (...) {
            String pname = ...;
            Logger.getLogger(pname);
        }
        return true;
    }

    Logger getLogger(String name) {
        return (Logger) loggers.get(name);
    }
}

class Logger {
    static sync Logger getLogger(String name) {
        LogManager lm = LogManager.manager;
        Logger l = lm.getLogger(name);
        if (l == null) {
            l = new Logger(...);
            lm.addLogger(l);
        }
        return l;
    }
}

class Harness {
    static void main(String[] args) {
        new Thread() { void run() {
            Logger.getLogger(...);
        }}.start();
        new Thread() { void run() {
            LogManager.manager.addLogger(...);
        }}.start();
    }
}
Example Deadlock Report

*** Stack trace of thread <Harness.java:11>:
LogManager.addLogger (LogManager.java:280)
   - this allocated at <LogManager.java:155>
   - waiting to lock {<LogManager.java:155>}
Logger.getLogger (Logger.java:231)
   - holds lock {<Logger.java:0>}
Harness$1.run (Harness.java:13)

*** Stack trace of thread <Harness.java:16>:
Logger.getLogger (Logger.java:226)
   - waiting to lock {<Logger.java:0>}
LogManager.addLogger (LogManager.java:314)
   - this allocated at <LogManager.java:155>
   - holds lock {<LogManager.java:155>}
Harness$2.run (Harness.java:18)
Our Approach

- Six necessary conditions identified experimentally:
  1. Reachable
  2. Aliasing
  3. Escaping
  4. Parallel
  5. Non-reentrant
  6. Non-guarded

- Checked using four incomplete but sound whole-program static analyses:
  1. Call-graph analysis
  2. May-alias analysis
  3. Thread-escape analysis
  4. May-happen-in-parallel analysis

- Relatively language independent
- Incomplete but sound checks
- Widely-used Java locking idioms
- Incomplete and unsound checks
- Sound needs must-alias analysis
Property: In some execution:
- can a thread abstracted by t1 reach l1
- and after acquiring lock at l1, proceed to reach l2 while holding that lock?
- and similarly for t2, l3, l4

Solution: Use call-graph analysis
- k-object-sensitive [Milanova-Rountev-Ryder ISSTA’03]
Example: jdk1.4 java.util.logging

```java
class LogManager {
    static LogManager manager = \new\ LogManager();
    Hashtable loggers = new Hashtable();

    sync boolean addLogger(Logger l) {
        String name = l.getName();
        if (!loggers.put(name, l))
            return false;
        // ensure l's parents are instantiated
        for (...) {
            String pname = ...;
        }
        return true;
    }

class Logger {
    static sync Logger getLogger(String name) {
        LogManager lm = LogManager.manager;
        Logger l = lm.getLogger(name);
        if (l == null) {
            l = new Logger(...);
            lm.addLogger(l);
        }
        return l;
    }

    class Harness {
        static void main(String[] args) {
            new Thread() { void run() {
            Logger.getLogger(...);
            }}.start();
            new Thread() { void run() {
            LogManager.manager.addLogger(...);
            }}.start();
        }
    }
}
```
Condition 2: Aliasing

- Property: In some execution:
  - can a lock acquired at $l_1$ be the same as a lock acquired at $l_4$?
  - and similarly for $l_2$, $l_3$

- Solution: Use may-alias analysis
  - k-object-sensitive [Milanova-Rountev-Ryder ISSTA’03]
Example: jdk1.4 java.util.logging

```java
class LogManager {
    static LogManager manager = new LogManager();

    Hashtable loggers = new Hashtable();

    boolean addLogger(Logger l) {
        String name = l.getName();
        if (!loggers.put(name, l))
            return false;
        // ensure l’s parents are instantiated
        for (...) {
            String pname = ...
        }
        return true;
    }

    Logger getLogger(String name) {
        return (Logger) loggers.get(name);
    }
}

class Logger {
    static sync Logger getLogger(String name) {
        LogManager lm = LogManager.manager;
        Logger l = lm.getLogger(name);
        if (l == null) {
            l = new Logger(...);
            lm.addLogger(l);
        }
        return l;
    }
}

class Harness {
    static void main(String[] args) {
        new Thread() { void run() {
            Logger.getLogger(...);
        }}.start();
        new Thread() { void run() {
            LogManager.manager.addLogger(...);
        }}.start();
    }
}
```
Condition 3: Escaping

- Property: In some execution:
  - can a lock acquired at $l_1$ be thread-shared?
  - and similarly for each of $l_2$, $l_3$, $l_4$

- Solution: Use thread-escape analysis
Example: jdk1.4 java.util.logging

class LogManager {
    static LogManager manager =
        new LogManager();
    Hashtable loggers = new Hashtable();

    public boolean addLogger(Logger l) {
        String name = l.getName();
        if (!loggers.put(name, l))
            return false;

        // ensure l’s parents are instantiated
        for (...) {
            String pname = ...
            Logger.getLogger(pname);
        }

        return true;
    }
}

class Logger {
    static sync Logger getLogger(String name) {
        LogManager lm = LogManager.manager;
        Logger l = lm.getLogger(name);
        if (l == null) {
            l = new Logger(...);
            lm.addLogger(l);
        }
        return l;
    }
}

class Harness {
    static void main(String[] args) {
        new Thread() { void run() {
            Logger.getLogger(...);
        }}.start();
        new Thread() { void run() {
            LogManager.manager.addLogger(...);
        }}.start();
    }
}
Condition 4: Parallel

- **Property:** In some execution:
  - can different threads abstracted by \( t_1 \) and \( t_2 \)
  - simultaneously reach \( l_2 \) and \( l_4 \)?

- **Solution:** Use may-happen-in-parallel analysis
  - Does not model full happens-before relation
  - Models only thread fork construct
  - Other conditions model other constructs
Example: jdk1.4 java.util.logging

class LogManager {
    static LogManager manager =
        new LogManager();
    HasTable loggers = new HasTable();
    boolean addLogger(Logger l) {
        String name = l.getName();
        if (!loggers.put(name, l))
            return false;
        // ensure l’s parents are instantiated
        for (...) {
            String pname = ...;
            Logger.getLogger(pname);
        }
        return true;
    }
}

class Logger {
    static Logger getLogger(String name) {
        LogManager lm = LogManager.manager;
        Logger l = lm.getLogger(name);
        if (l == null) {
            l = new Logger(...);
            lm.addLogger(l);
        }
        return l;
    }
}

class Harness {
    static void main(String[] args) {
        new Thread() {
            void run() {
                Logger.getLogger(...);
            }
        }.start();
        new Thread() {
            void run() {
                LogManager.manager.addLogger(...);
            }
        }.start();
    }
}
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>Classes</th>
<th>Methods</th>
<th>Syncs</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>moldyn</td>
<td>31,917</td>
<td>63</td>
<td>238</td>
<td>12</td>
<td>4m48s</td>
</tr>
<tr>
<td>montecarlo</td>
<td>157,098</td>
<td>509</td>
<td>3447</td>
<td>190</td>
<td>7m53s</td>
</tr>
<tr>
<td>raytracer</td>
<td>32,576</td>
<td>73</td>
<td>287</td>
<td>16</td>
<td>4m51s</td>
</tr>
<tr>
<td>tsp</td>
<td>154,288</td>
<td>495</td>
<td>3335</td>
<td>189</td>
<td>7m48s</td>
</tr>
<tr>
<td>sor</td>
<td>32,247</td>
<td>57</td>
<td>208</td>
<td>5</td>
<td>4m48s</td>
</tr>
<tr>
<td>hedc</td>
<td>160,071</td>
<td>530</td>
<td>3552</td>
<td>204</td>
<td>21m15s</td>
</tr>
<tr>
<td>weblech</td>
<td>184,098</td>
<td>656</td>
<td>4620</td>
<td>238</td>
<td>32m02s</td>
</tr>
<tr>
<td>jspider</td>
<td>159,494</td>
<td>557</td>
<td>3595</td>
<td>205</td>
<td>15m34s</td>
</tr>
<tr>
<td>jigsaw</td>
<td>154,584</td>
<td>497</td>
<td>3346</td>
<td>184</td>
<td>15m23s</td>
</tr>
<tr>
<td>ftp</td>
<td>180,904</td>
<td>642</td>
<td>4383</td>
<td>252</td>
<td>35m55s</td>
</tr>
<tr>
<td>dbcp</td>
<td>168,018</td>
<td>536</td>
<td>3602</td>
<td>227</td>
<td>16m04s</td>
</tr>
<tr>
<td>cache4j</td>
<td>34,603</td>
<td>72</td>
<td>218</td>
<td>7</td>
<td>4m43s</td>
</tr>
<tr>
<td>logging</td>
<td>167,923</td>
<td>563</td>
<td>3852</td>
<td>258</td>
<td>9m01s</td>
</tr>
<tr>
<td>collections</td>
<td>38,961</td>
<td>124</td>
<td>712</td>
<td>55</td>
<td>5m42s</td>
</tr>
</tbody>
</table>
# Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Deadlocks (0-cfa)</th>
<th>Deadlocks (k-obj.)</th>
<th>Lock type pairs (total)</th>
<th>Lock type pairs (real)</th>
</tr>
</thead>
<tbody>
<tr>
<td>moldyn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tsp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sor</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hedic</td>
<td>7,552</td>
<td>2,358</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>weblech</td>
<td>4,969</td>
<td>794</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>jspider</td>
<td>725</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>jigsaw</td>
<td>23</td>
<td>18</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ftp</td>
<td>16,259</td>
<td>3,020</td>
<td>33</td>
<td>24</td>
</tr>
<tr>
<td>dbcp</td>
<td>320</td>
<td>16</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>cache4j</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>logging</td>
<td>4,134</td>
<td>4,134</td>
<td>98</td>
<td>94</td>
</tr>
<tr>
<td>collections</td>
<td>598</td>
<td>598</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
Individual Analysis Contributions
Conclusion

• Novel approach to static deadlock detection for Java
  – Checks six necessary conditions for a deadlock
  – Uses four off-the-shelf static analyses

• Neither sound nor complete, but effective in practice
  – Applied to suite of 14 multi-threaded Java programs comprising over 1.5 MLOC
  – Found all known deadlocks as well as previously unknown ones, with few false alarms