Symbolic Deadlock Analysis for Concurrent Libraries and their Clients

Jyotirmoy V. Deshmukh\textsuperscript{1} E. Allen Emerson\textsuperscript{1} Sriram Sankaranarayanan\textsuperscript{2}
\{deshmukh,emerson\}@cs.utexas.edu, srirams@colorado.edu

\textsuperscript{1}University of Texas at Austin
\textsuperscript{2}University of Colorado at Boulder

Automated Software Engineering 2009
Thread Safety

Re: [PATCH 2/4] ext3: Fix possible deadlock between ext3_truncate() and ext3_get_blocks()... low transaction start (and it can lead to a real deadlock with ext3_get_blocks() allocating new blocks from ... (inode)); +/* + * Drop truncate_mutex to avoid deadlock with ext3_get_blocks_handle + * At this moment,..., can lead to a real deadlock with ext3_get_blocks() allocating new blocks from...

Aug 17, 2009 - Jan Kara - org.kernel.vger.linux-ext4

mciavi: fix deadlock
+ /* To avoid deadlock deal with the window only after the owning thread... + /* To avoid deadlock deal with the window only after the owning thread is

Mar 13, 2009 - Kirill K. Smirnov - org.winehq.wine-patches

[dwr-user] Java-level deadlock
Java-level deadl... http://8085-exec-20... at java.lang.Thread.run(Thread.java:619) Found 1 deadlock. Heap PSYoungGen total 218752K, used 213...

Mar 25, 2009 - Иван Трофимов - net.java.dev.dwr.users
Thread Safety

Deadlocks increasingly important

[Graph showing increasing number of messages related to deadlocks]

Deadlocks increasingly important

[Excerpt from a forum thread about deadlocks]

...] ext3: Fix possible deadlock between ext3_truncate() and ext3_get_blocks() ... elow transaction start (and it can lead to a real deadlock with ext3_get_blocks() allocating new blocks from ... (inode)); +/* + * Drop truncate_mutex to avoid deadlock with ext3_get_blocks_handle + * At this moment,......t can lead to a real deadlock with ext3_get_blocks() allocating new blocks from...

Aug 17, 2009 - Jan Kara - org.kernel.vger.linux-ext4 -

mciavi: fix deadlock
... + /* To avoid deadlock deal with the window only after the owning thread... + /* To avoid deadlock deal with the window only after the owning thread is
Mar 13, 2009 - Kirill K. Smirnov - org.winehq.wine-patches -

[dwr-user] Java-level deadlock
Thread Safety

Deadlocks increasingly important

Deshmukh

Symbolic Deadlockability Analysis
Thread Safety

Deadlocks increasingly important

Library-level deadlocks abundant
Concurrent Software is Modular

- **Concurrent Library**: methods concurrently invokable.
- **Multi-threaded Client**: each thread invokes library methods.
- “Whole-program approach” too expensive.
Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP '05]
Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP ’05]
- **Aliasing** information for improved accuracy.
Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP ’05]
- Aliasing information for improved accuracy.
- Interface Contracts on methods to ensure deadlock-freedom.
Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP ’05]
- **Aliasing** information for improved accuracy.
- **Interface Contracts** on methods to ensure deadlock-freedom.
- Use interface contracts when analyzing Client code.
Outline

1. Deadlockability Analysis
2. Problem Size Reduction
3. Symbolic Computation
4. Results
Outline

1. Deadlockability Analysis
2. Problem Size Reduction
3. Symbolic Computation
4. Results
java.awt.EventQueue

EventQueue nextQueue;

void postEventPrivate(Event e) {
  ...
  synchronized (this) {
    ...
    nextQueue.postEventPrivate(e);
    ...
  }
  ...
}

void wakeup(boolean f) {
  ...
  synchronized (this) {
    ...
    nextQueue.wakeup(f);
    ...
  }
  ...
}
void postEventPrivate(Event e) {
    ...
    synchronized(this) {
        nextQueue.postEventPrivate(e);
        ...
    }
    ...
}

void wakeup(boolean f) {
    ... 
    synchronized(this) {
        nextQueue.wakeup(f);
        ...
    }
    ...
}
Aliasing Pattern leading to Deadlock?

```java
ob1 = ob2.nextQueue, ob2 = ob1.nextQueue

void postEventPrivate(Event e)
{
    ...
    synchronized(this)
    {
        nextQueue.postEventPrivate(e);
        ...
    }
    ...
}

void wakeup(boolean f)
{
    ...
    synchronized(this)
    {
        nextQueue.wakeup(f);
        ...
    }
    ...
}
```

Symbolic Deadlockability Analysis
Such weird aliasing comes from...

```java
EventQueue nextQueue;
void push (EventQueue eq) {
    ... 
    nextQueue = eq;
    ... 
}
```

Sequence of method calls

```
eq1.push(eq2);
:  
eq1.wakeup(...);
:  
eq2.push(eq1);
:  
eq2.postEventPrivate(...);
:  
```
Deadlock-causing Aliasing Pattern

Aliasing Pattern between $lg(\text{postEventPrivate}), lg(\text{wakeup})$

$$\alpha = \text{isAliased}(\text{ob1}, \text{ob2} . \text{nextQueue}) \land \text{isAliased}(\text{ob2}, \text{ob2} . \text{nextQueue})$$
### Interface Contract

```java
void postEventPrivate (Event e) {
    ...
    synchronized (this) {
        ...
        nextQueue.postEventPrivate(e);
        ...
    }
    ...
}

void wakeup(boolean f) {
    ...
    synchronized (this) {
        ...
        nextQueue.wakeup(f);
        ...
    }
    ...
}
```

For `postEventPrivate`, `wakeup`

\[
\neg \text{isAliased}(ob1, ob2.nextQueue) \lor \\
\neg \text{isAliased}(ob2, ob1.nextQueue)
\]

**Call-site \[\models I \Rightarrow postEventPrivate \parallel wakeup\] is deadlock-free.**
Approach: View from 10,000 feet

Compute:
- Lock-graphs for library methods (static analysis)
- DL-causing patterns for combinations of 2 or more methods.
- Derive Interface Contracts.
Outline

1. Deadlockability Analysis
2. Problem Size Reduction
   - Lock-graph Size Reduction
   - Smarter Enumeration
3. Symbolic Computation
4. Results
Prune Lock-graphs:
Remove nodes that cannot be part of cycle

Terminal nodes that may alias only to other terminal nodes.
Prune Lock-graphs:
Remove nodes that cannot be part of cycle

Terminal nodes that may alias only to other terminal nodes.
Prune Lock-graphs:
Remove nodes that cannot be part of cycle

- Terminal nodes that may alias only to other terminal nodes.
- Initial nodes that may alias only to other initial nodes.
Smarter Enumeration by Subsumption

a (T1) --> x (T1)

b (T2) ← y (T2) ← z (T2)

c (T1) --> x (T1)
Deadlock-causing Aliasing Pattern ($\alpha_1$)

$\text{isAliased}(b, y) \land \text{isAliased}(c, x)$
Problem Size Reduction
Smarter Enumeration

Smarter Enumeration by Subsumption

Deadlock-causing Aliasing Pattern ($\alpha_2$)

\[
isAliased(b, y) \land isAliased(c, x) \land isAliased(a, x)
\]
Subsumption

- $\alpha_2$ subsumes $\alpha_1$: $\alpha_2$ has more aliasing.
- DL with lesser aliasing $\Rightarrow$ DL with more aliasing.
- Only enumerate “minimally” unsafe patterns.
- Disregard subsuming patterns.
Explicit Enumeration

- $U$ empty?
  - yes: Output $D$
  - no: Pick $\alpha$
    - $\alpha \triangleright G$ acyclic?
      - yes: Add $(u, v)$ to $\alpha$ till maximally safe
      - no: Restrict $\alpha$ to cycle-edges in $G$
        - $D = D \cup \alpha'$
        - $U = U - \{\beta | \beta \subseteq \alpha\}$
        - $U = U - \{\beta | \alpha' \subseteq \beta\}$
Outline

1. Deadlockability Analysis
2. Problem Size Reduction
3. Symbolic Computation
4. Results
Aliasing Pattern Enumeration with SMT

Theorem

Enumerating all deadlock-causing aliasing patterns is NP-complete.

Symbolic Computation

- Encode Lock-Order Graphs as Inequality Constraints.
- Encode Aliasing as Equality Constraints.
- Transform Cycle Detection in a Graph to SAT of a Constraint.
- Use SMT solvers to check SAT.
Symbolic Encoding

\[(a < b \land b < c)\]
Symbolic Encoding

\[(a < b \land b < c) \land (x < y \land y < z)\]
Symbolic Encoding

\[(a < b \land b < c) \land (x < y \land y < z) \land (c = x \land b = y)\]
Symbolic Computation

Symbolic Encoding

\[(a < b \land b < c) \land (x < y \land y < z) \land (c = x \land b = y)\]

Cycle $\equiv$ UNSAT $\equiv$ deadlock!
Symbolic Algorithm

\[ \Psi_U \]

1. Is \( \Psi_U \) unSAT?
   - yes: Output \( D \)
   - no: \( \alpha = \) solution of \( \Psi_U \)

2. Is \( \Psi(\alpha, G) \) SAT?
   - yes: \( \alpha = \alpha \land (x(u) = x(v)) \) till maximally safe
     - \( \Psi_U = \Psi_U \land \bigvee_{(e_i, e_j) \notin \alpha}(x(e_i) = x(e_j)) \)
   - no: Restrict \( \alpha \) to UnSAT core of \( \Psi(\alpha, G) \)
     - \( D = D \cup \alpha \)
     - \( \Psi_U = \Psi_U \land \bigvee_{(e_i, e_j) \in \alpha}(x(e_i) \neq x(e_j)) \)
Outline

1. Deadlockability Analysis
2. Problem Size Reduction
3. Symbolic Computation
4. Results
## Experimental Results

<table>
<thead>
<tr>
<th>Library Name</th>
<th>LOC (K)</th>
<th>Time Taken (secs)</th>
<th>False + ves</th>
<th>Potential Deadlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftpproxy</td>
<td>1.0</td>
<td>13.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JavaFTP</td>
<td>2.6</td>
<td>9.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cache4j</td>
<td>2.6</td>
<td>15.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>netty</td>
<td>11.0</td>
<td>14.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>apache-log4j</td>
<td>33.3</td>
<td>130.1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>oddjob</td>
<td>41.3</td>
<td>250.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>hsqldb</td>
<td>157.6</td>
<td>806.8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>javax 1.6 sdk</td>
<td>534.3</td>
<td>629.0</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>java 1.6 sdk</td>
<td>551.8</td>
<td>1011.6</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>

Total:

- LOC > 1.3M
- Time < 2880
- False + ves 24
- Potential Deadlocks 18
### Vindication

Most deadlocks identified correspond to real, live bug reports by developers!

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Method names</th>
<th>Bug Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt (EventQueue)</td>
<td>postEventPrivate, wakeup</td>
<td>Sun Bug DB ids: 4913324, 6424157, 6542185.</td>
</tr>
<tr>
<td>java.awt (Container)</td>
<td>removeAll, addPropertyChangeListener</td>
<td>OS-dir mail archive.</td>
</tr>
<tr>
<td>java.util (LogManager) (Logger)</td>
<td>addLogger, getLogger</td>
<td>Sun Bug DB id: 6487638.</td>
</tr>
<tr>
<td>javax.swing (JComponent)</td>
<td>setFont, paintChildren</td>
<td>Bug in Jajuk player</td>
</tr>
<tr>
<td>hsqldb (Session)</td>
<td>isAutoCommit, close</td>
<td>OS-dir mail archive</td>
</tr>
</tbody>
</table>
With Interface Contracts, we get . . .

- better specification of (deadlock-free) thread-safe behavior,
- useful documentation for client developers,
- plug-in for statically analyzing existing client code, and,
- compositional flavor in reasoning about deadlocks.
Thank You!
# Lock-order Graphs

## Definition
Access Expression (a.e.): \( \text{ob} \) or sequence of nested fields of \( \text{ob} \).

## Definition (Lock-order Graph \( G(V, E) \) for method \( m \))

\((v_1, v_2) \in E \iff:\)

- \( v_1 \) aliased to some a.e. \( x \),
- \( v_2 \) aliased to some a.e. \( y \),
- Path \( \text{lock}(x) \rightarrow \ldots \rightarrow \text{lock}(y) \) in \( \text{cfg}(m) \)
Computing Lock-order Graphs

Summary = State after each program statement
- which locks currently held ($ls$)
- lock-order graph ($lg$)
- root nodes ($rs$), and,
- aliasing information,
Computing Lock-order Graphs

- Standard interprocedural summary-based forward static analysis.
- \(\text{lock}(x) = \text{add } x \text{ to } ls, \forall y \in ls \text{ add } (y, x) \text{ to } lg.\)
- \(\text{unlock}(x) = \text{remove } x \text{ from } ls.\)
- Branch merge = union of summaries.
- Invocation of \(m = \text{concatenate } lg(m) \text{ to current } lg.\)
Deadlockability Analysis

Given library \( \mathcal{L} = \{ C_1, \ldots, C_m \} \)

Methods \( m_1, \ldots, m_k \) spread across classes \( C_1, \ldots, C_m \).

Compute for all \( m_1, \ldots, m_k \)

Lock-order graphs \( \text{lg}(m_1), \ldots, \text{lg}(m_k) \).

Check for each pair \( m_i, m_j \)

Is there any aliasing pattern s.t. \( \text{lg}(m_i) \cup \text{lg}(m_j) \) has cycles?

Compute

\( \mathcal{D} \): set of all \emph{deadlock-causing aliasing patterns}. 
So far . . .

Model Checking

- Generate global state graph.
- Explore all possible interleavings.

But...

May not scale after abstraction and partial order reduction.
So far . . .

Static Analysis
- Lock-acquisition order graph \((lg)\) for each thread.
- Conservatively merge \(lg\) for concurrent threads.
- Cycle in merged graph \(\Rightarrow\) possible deadlock.

But...
Too many false positives if analysis coarse, unscalable otherwise.
Deadlock-causing Aliasing Pattern Enumeration

**Definition (Subsumption)**

\[ \alpha_2 \text{ subsumes } \alpha_1 \quad (\alpha_1 \subseteq \alpha_2) \text{ iff } \forall (u, v) : (u, v) \in \alpha_1 \Rightarrow (u, v) \in \alpha_2. \]

**Lemma (Given } \alpha_1 \subseteq \alpha_2 \text{)**

\[ \alpha_1 \text{ is deadlock-causing } \Rightarrow \alpha_2 \text{ is deadlock-causing.} \]

**Definition (Minimally Unsafe)**

\[ \alpha \text{ minimally unsafe iff for any } (u, v), \alpha - (u, v) \text{ is safe.} \]

We only need to consider minimally unsafe patterns.
Deadlock-causing Aliasing Pattern Enumeration

Subsumption

- $\alpha_2$ subsumes $\alpha_1 \Rightarrow \alpha_2$ has more aliasing than $\alpha_1$.
- $\alpha_1 \subseteq \alpha_2$: $\alpha_1$ is deadlock-causing $\Rightarrow \alpha_2$ is deadlock-causing.
- $\alpha$ minimally unsafe if removing any aliasing makes it safe.

We only need to enumerate minimally unsafe patterns!
Encoding Lock-Graph $G(V, E)$

- $x(v_i)$: topological rank of $v_i \in V$.
- $\Psi(G) = \bigwedge_{(v_i, v_j) \in E} (x(v_i) < x(v_j))$.

Encoding Aliasing Pattern $\alpha$

$\Psi(\alpha) = \bigwedge_{(v_i, v_j) \in \alpha} (x(v_i) = x(v_j))$

Reduction to SAT

$\alpha \triangleright G$ has a cycle iff $\Psi(\alpha, G) = \Psi(G) \land \Psi(\alpha)$ is unsatisfiable.
A few more (sound) filters...

**Prune ...**

- locks corresponding to `final` fields.
- `private` fields not accessed outside constructor/finalizer.
- immutable constants.
- `private` objects that cannot escape scope of methods.
```java
void postEventPrivate(Event e) {
    ... ,
    synchronized (this) {
        nextQueue.postEventPrivate(e);
        ...
    }
    ...
}

void wakeup(boolean f) {
    ...
    synchronized (this) {
        nextQueue.wakeup(f);
        ...
    }
    ...
}
```

Joint Lock-Order Graph without Aliasing
Derive Interface Contracts

**Definition (Interface Contract)**

Compute $\mathcal{D}$: all deadlock-causing aliasing patterns.

$$\mathcal{I}(m_i, m_j): \bigwedge_{\alpha \in \mathcal{D}} \bigvee_{(e_i, e_j) \in \alpha} \neg \text{isAliased}(e_i, e_j).$$

Call-site of $m_i, m_j$ satisfies $\mathcal{I} \Rightarrow m_i \parallel m_j$ is deadlock-free.