Sound Static Deadlock Analysis for C/Pthreads

Daniel Kroening\textsuperscript{1}, Daniel Poetzl\textsuperscript{1}, Peter Schrammel\textsuperscript{2}, Bjoern Wachter\textsuperscript{3}

\textsuperscript{1}University of Oxford, \textsuperscript{2}University of Sussex, \textsuperscript{3}SSW-Trading GmbH

ASE 2016
Deadlocks

Thread $T_1$  Thread $T_2$

\[ \ldots \]
\[ \text{lock}(m1) \]
\[ x = 1 \]
\[ \text{lock}(m2) \]
\[ \ldots \]

\[ \ldots \]
\[ \text{lock}(m2) \]
\[ y = 2 \]
\[ \text{lock}(m1) \]
\[ \ldots \]

Execution

Lock assignment graph

$m_1$

$m_2$

$T_1$

$T_2$
Deadlocks

Thread $T_1$          Thread $T_2$

...                    ...

lock (m1)              lock (m2)

$x = 1$                 $y = 2$

lock (m2)              lock (m1)

...                    ...

Execution

lock (m1)

Lock assignment graph

- $m_1$
- $m_2$
- $T_1$
- $T_2$
Thread $T_1$          Thread $T_2$

...                      ...
lock(m1)                  lock(m2)
x = 1                     y = 2
lock(m2)                  lock(m1)
...                      ...

Execution

lock(m1)
x = 1

Lock assignment graph

assigned

$m_1$

$T_1$

$m_2$

$T_2$
Deadlocks

Thread $T_1$  Thread $T_2$

...  
`lock(m1)`  `lock(m2)`
$x = 1$  $y = 2$
`lock(m2)`  `lock(m1)`
...  ...  

Execution  

`lock(m1)`  
$x = 1$
`lock(m2)`

Lock assignment graph

$m_1$  assigned  $m_2$
$T_1$  

assigned  $T_2$
Deadlocks

Thread $T_1$  
...  
lock(m1)  
x = 1  
lock(m2)  
...  

Thread $T_2$  
...  
lock(m2)  
y = 2  
lock(m1)  
...  

Execution

lock(m1)  
x = 1  
lock(m2)  
y = 2  

Lock assignment graph

$T_1$  
$T_2$  
$m_1$  
$m_2$
Deadlocks

Thread $T_1$          Thread $T_2$
...
`lock(m1)`          `lock(m2)`
$x = 1$             $y = 2$
`lock(m2)`          `lock(m1)`
...

Execution

`lock(m1)`
$x = 1$
`lock(m2)`
$y = 2$
`lock(m2)`

Lock assignment graph

$T_1$  $T_2$

$m_1$
assigned

$m_2$
requested
assigned
Deadlocks

Thread $T_1$  

...  

lock($m_1$)  

$x = 1$  

lock($m_2$)  

...  

Execution  

lock($m_1$)  

$x = 1$  

lock($m_2$)  

$y = 2$  

lock($m_2$)  

lock($m_1$)

Thread $T_2$  

...  

lock($m_2$)  

$y = 2$  

lock($m_1$)  

...  

Lock assignment graph

$T_1$  

$m_1$  

requested  

assigned  

$m_2$  

requested  

assigned  

$T_2$
Overview

• Static deadlock analysis
  – Sound (i.e., no false negatives) for defined programs
  – New abstract interpretation framework
    • Context-sensitive
    • Thread-sensitive
  – Pipeline of several analyses that result in a lock graph
A context-sensitive approach is necessary for accurate deadlock analysis:

```c
void vlc_mutex_lock(vlc_mutex_t *p)
{
    int val = pthread_mutex_lock(p);
    VLC_THREAD_ASSERT("locking mutex");
}
```
Context- and Thread-Sensitivity

- Our analysis framework computes an abstract state for each place.
- A place represents a function call and thread creation history.

```plaintext
main() {
    create(thread);
    func();
}

thread() {
    func();
}

func() {
    lock(L);
    unlock(L);
}
```

- Examples of places:
  - (create: 2, call: 6, location: 9)
  - (call: 3, location: 9)

- Associated abstract thread IDs:
  - (create: 2)
  - ()
May Lockset Analysis

dependency analysis

pointer analysis

may lockset analysis

lock graph construction

non-concurrency analysis

cycle search
May Lockset Analysis

1. main() {
   2. create(thread);
   3. func();
   4. }

5. thread() {
   6. func();
   7. }

8. func() {
   9. lock(L);
  10. unlock(L);
  11. }

• May lockset analysis result:

(create: 2, call: 6, location: 9) \rightarrow \{ L \}
(call: 3, location: 9) \rightarrow \{ L \}
... \rightarrow \{ \}
Lock Multigraph

- dependency analysis
- pointer analysis
- may lockset analysis
- lock graph construction
- non-concurrency analysis
- cycle search
The lock multigraph contains context/thread information as the edges are annotated with places:

- Edge \((L, p, L')\) indicates that a thread might try to lock \(L'\) while holding \(L\) in the context represented by place \(p\).
- If there is a cycle such that the corresponding places are concurrent then there is a potential deadlock.
Dependency Analysis

- Dependency analysis
- Pointer analysis
- May lockset analysis
- Lock graph construction
- Non-concurrency analysis
- Cycle search
Dependency Analysis

- The dependency analysis identifies:
  1) Assignments that could affect lock expressions
  2) Functions relevant to deadlock analysis

- Only the identified statements need to be analysed by subsequent analysis passes
Dependency Analysis

*r = a

lock(*p)
Dependency Analysis

\[ q = \text{malloc}(...) \]

\[ *r = a \quad \text{lock}(\ast p) \]
Dependency Analysis

q = malloc(...)

\[
\begin{align*}
q & = \text{malloc}(...) \\
p & = q \\
*{r} & = a \\
\text{lock}(*p)
\end{align*}
\]
Dependency Analysis

\[
q = \text{malloc}(\ldots)
\]

\[
q \quad q
\]

\[
r = q \quad p = q
\]

\[
r \quad p
\]

\[
*r = a \quad \text{lock}(*p)
\]
We can compute in **linear time** an overapproximation of the assignments that could affect the lock expressions.
Dependency Analysis – Evaluation

- The dependency analysis pruned on average:
  - 60% of assignments
  - 37% of functions

- Effect on the overall analysis:
  - 81% reduction of runtime
  - 67% reduction of memory consumption
Deadlock Analysis – Evaluation

• Implemented our approach on top of CPROVER

• Evaluation of the precision and scalability:
  – 997 C/Pthreads programs using locks from Debian
  – 292 programs proved deadlock-free
  – 113 programs with false alarms

<table>
<thead>
<tr>
<th>KLOC</th>
<th>Analysed</th>
<th>Proved</th>
<th>Alarms</th>
<th>Timeout*</th>
<th>Memout+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>250</td>
<td>52 %</td>
<td>22 %</td>
<td>18 %</td>
<td>8 %</td>
</tr>
<tr>
<td>5-10</td>
<td>272</td>
<td>33 %</td>
<td>13 %</td>
<td>38 %</td>
<td>15 %</td>
</tr>
<tr>
<td>10-15</td>
<td>152</td>
<td>14 %</td>
<td>9 %</td>
<td>63 %</td>
<td>14 %</td>
</tr>
<tr>
<td>15-20</td>
<td>181</td>
<td>15 %</td>
<td>3 %</td>
<td>54 %</td>
<td>27 %</td>
</tr>
<tr>
<td>20-50</td>
<td>142</td>
<td>14 %</td>
<td>4 %</td>
<td>79 %</td>
<td>4 %</td>
</tr>
</tbody>
</table>

* timeout 1800s
* memory limit 24 GB
Evaluation – Observations

• Pointer analysis:
  – context-sensitivity is crucial
  – takes most of the time (~80%)
  – imprecision is the main source of false positives

• Dependency analysis:
  – fast (< 1s)
  – prunes a significant number of assignments and functions

• Non-concurrency analysis:
  – no reduction of false positives on the Debian benchmarks
Conclusions

• Sound static deadlock analysis approach
  – Based on abstract interpretation
  – For real-world C/Pthreads programs
  – Proved deadlock-freedom of 292 programs from Debian

• Extended version:
  – arxiv.org/abs/1607.06927
  – Includes proofs and additional experimental results

• Tool and benchmarks:
  – www.cprover.org/deadlock-detection
Conclusions

• Sound static deadlock analysis approach
  – Based on abstract interpretation
  – For real-world C/Pthreads programs
  – Proved deadlock-freedom of 292 programs from Debian

• Extended version:
  – arxiv.org/abs/1607.06927
  – Includes proofs and additional experimental results

• Tool and benchmarks:
  – www.cprover.org/deadlock-detection

Thank you!