Validating Optimizations of Concurrent C/C++ Programs

Soham Chakraborty  Viktor Vafeiadis

MPI-SWS

CGO 2016
int X = 0; int Y = 0;

Y = 4; \quad \parallel \quad \text{if (X)} \\
X = 1; \quad \parallel \quad r = Y;
```c
int X = 0; int Y = 0;
Y = 4; if (X)  
X = 1; r = Y;
```

Race on $X \sim$ undefined semantics
$X == 1 \land r \neq 4$ is possible
(i.e., the program is wrong)
atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1, mo_release);

if (atomic_load(&X, mo_acquire))
    r = Y;
atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1, mo_release);

if (atomic_load(&X, mo_acquire))
  r = Y;
atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1, mo_release);

if (atomic_load(&X, mo_acquire))
  r = Y;
atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1, mo_release);

if (atomic_load(&X, mo_acquire))
    r = Y;
atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1, mo_release);
if (atomic_load(&X, mo_acquire))
    r = Y;
atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1,
mo_release);

if (atomic_load(&X,
mo_acquire))

r = Y;
atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1, mo_release);
if (atomic_load(&X, mo_acquire))
    r = Y;
Concurrent Programming in C11

atomic_int X = 0; int Y = 0;

Y = 4;
atomic_store(&X, 1, mo_release);

if (atomic_load(&X, mo_acquire))
  r = Y;

X = Y = 0;
Y = 4;
if (Xacq)
  r = Y;
X_rel = 1;  
  r = Y;
An Unsafe Reordering

\[ X = Y = 0; \]
\[ Y = 4; \]
\[ X_{rel} = 1; \]

Always returns \( r == 4 \)

May return \( r == 0 \)

Optimizations for sequential programs are NOT always safe for concurrent programs.
Optimizations for sequential programs are **NOT** always safe for concurrent programs.
Another Example

\[
X = Y = 0; \\
Y = 4; \\
X_{rel} = 1; \\
f = false; \\
\cdots \\
a = f \ ? \ Y : 0; \\
b = X_{acq} \ ? \ Y : 4;
\]
\[ X = Y = 0; \]
\[ f = false; \]
\[ Y = 4; \]
\[ \cdots \]
\[ X_{rel} = 1; \]
\[ a = f \ ? \ Y : 0; \]
\[ b = X_{acq} \ ? \ Y : 4; \]

Output: \( b == 4 \) always
\( X = Y = 0; \)
\( f = false; \)
\( \ldots \)
\( a = f ? Y : 0; \)
\( b = X_{\text{acq}} ? Y : 4; \)

Output \( b == 0 \) possible in target.
\[
X = Y = 0; \\
f = \text{false}; \\
\ldots \\
a = f \ ? \ Y : 0; \\
b = X_{\text{acq}} \ ? \ Y : 4;
\]

\[
(1) \quad \begin{align*}
X &= Y = 0; \\
f &= \text{false}; \\
\ldots \\
s &= Y; \\
a &= f \ ? \ s : 0; \\
t &= X_{\text{acq}}; \\
r &= Y; \\
b &= t \ ? \ r : 4;
\end{align*}
\]

\[
(2) \quad \begin{align*}
X &= Y = 0; \\
f &= \text{false}; \\
\ldots \\
s &= Y; \\
a &= f \ ? \ s : 0; \\
t &= X_{\text{acq}}; \\
r &= Y; \\
b &= t \ ? \ s : 4;
\end{align*}
\]
\[
\begin{align*}
X &= Y = 0; \\
f &= false; \\
\ldots \\
a &= f \ ? \ Y : 0; \\
b &= X_{acq} \ ? \ Y : 4;
\end{align*}
\]

(1) \[
\begin{align*}
X &= Y = 0; \\
f &= false; \\
\ldots \\
s &= Y; \\
a &= f \ ? \ s : 0; \\
t &= X_{acq}; \\
r &= Y; \\
b &= t \ ? \ r : 4;
\end{align*}
\]

(2) \[
\begin{align*}
X &= Y = 0; \\
f &= false; \\
\ldots \\
s &= Y; \\
a &= f \ ? \ s : 0; \\
t &= X_{acq}; \\
r &= Y; \\
b &= t \ ? \ s : 4;
\end{align*}
\]
LLVM Compilation Bug in More Detail

\[ X = Y = 0; \]
\[ f = \text{false}; \]
\[ \ldots \]
\[ a = f \ ? \ Y : 0; \]
\[ b = X_{\text{acq}} \ ? \ Y : 4; \]

\[ X = Y = 0; \]
\[ f = \text{false}; \]
\[ \ldots \]
\[ (1) \ s = Y; \]
\[ a = f \ ? \ s : 0; \]
\[ t = X_{\text{acq}}; \]
\[ r = Y; \]
\[ b = t \ ? \ r : 4; \]

\[ X = Y = 0; \]
\[ f = \text{false}; \]
\[ \ldots \]
\[ (2) \ s = Y; \]
\[ a = f \ ? \ s : 0; \]
\[ t = X_{\text{acq}}; \]
\[ r = Y; \]
\[ b = t \ ? \ s : 4; \]

C11: (1) \textbf{Error}
\[
X = Y = 0; \quad f = false; \quad X = Y = 0; \quad f = false;
\]
\[
\begin{align*}
\ldots & \\
a = f ? Y : 0; & a = f ? s : 0; \\
b = X_{acq} ? Y : 4; & t = X_{acq}; \\
\end{align*}
\]
\[
\begin{align*}
(1) \quad s &= Y; & (2) \quad s &= Y; \\
a &= f ? s : 0; & a &= f ? s : 0; \\
t &= X_{acq}; & t &= X_{acq}; \\
r &= Y; & r &= Y; \\
b &= t ? r : 4; & b &= t ? s : 4; \\
\end{align*}
\]

C11: (1) Error (2) Correct
\( X = Y = 0; \)
\( f = false; \)
\( \ldots \)
\( a = f ? Y : 0; \)
\( b = X_{acq} ? Y : 4; \)

\( X = Y = 0; \)
\( f = false; \)
\( \ldots \)
(1) \( s = Y; \)
(2) \( s = Y; \)
\( a = f ? s : 0; \)
\( t = X_{acq}; \)
\( r = Y; \)
\( b = t ? r : 4; \)

C11: (1) Error (2) Correct
LLVM: (1) Correct
$X = Y = 0$;
$f = false$;
...

(1) $s = Y$;
$a = f ? s : 0$;
$t = X_{acq}$;
$r = Y$;
$b = t ? r : 4$;

(2) $s = Y$;
$a = f ? s : 0$;
$t = X_{acq}$;
$r = Y$;
$b = t ? s : 4$;

C11: (1) Error (2) Correct
LLVM: (1) Correct (2) Error
Define a set of safe reorderings & eliminations:

- For the LLVM model
- For the C11 model [POPL’15]
Can be used in validating other compilers.

Steps:
- Identify corresponding program paths
- Compute deletability of accesses
- Match access sequences and analyze
s_1 = X
s_2 = X
V = 1
s_4 = Z_{acq}
Y = 1
Y = 2
\begin{align*}
\checkmark \quad s_1 &= X \\
      s_2 &= X \\
      V &= 1 \\
      s_4 &= \mathcal{Z}_{\text{acq}} \\
      Y &= 1 \\
      Y &= 2
\end{align*}
Compiler Independent Matching

\[ \checkmark \quad s_1 = X \]

\[ \times \quad s_2 = X \]

\[ V = 1 \]

\[ s_4 = Z_{acq} \]

\[ Y = 1 \]

\[ Y = 2 \]
Compiler Independent Matching

- ✓ $s_1 = X$
- × $s_2 = X$
- ✓ $s_4 = Z_{acq}$
- ✓ $V = 1$
- ✓ $Y = 1$
- ✓ $Y = 2$
Compiler Independent Matching

✓ $s_1 = X$

✗ $s_2 = X$

$V = 1$

✓ $s_4 = Z_{acq}$

$Y = 1$

✓ $Y = 2$

Correct Check that unmatched accesses are deletable
Check that reorderings are allowed
Compiler Independent Matching

\[ s_1 = X \]
\[ s_2 = X \]
\[ V = 1 \]
\[ s_4 = Z_{acq} \]
\[ Y = 1 \]
\[ Y = 2 \]
Compiler Independent Matching

✓ \( s_1 = X \)
✓ \( V = 1 \)
✓ \( s_4 = Z_{\text{acq}} \)
✓ \( Y = 2 \)

✗ \( s_2 = X \)
✗ \( Y = 1 \)
Compiler Independent Matching

- ✓ $s_1 = X$
- X $s_2 = X$
- ✓ $V = 1$
- ✓ $s_4 = Z_{acq}$
- X $Y = 1$
- ✓ $Y = 2$
- $t_1 = X$
- $t_2 = Z_{acq}$
- $Y = 2$
- $V = 1$
Compiler Independent Matching

- ✓ \( s_1 = X \)
- ✗ \( s_2 = X \)
- ✓ \( V = 1 \)
- ✓ \( s_4 = Z_{\text{acq}} \)
- ✗ \( Y = 1 \)
- ✓ \( Y = 2 \)

\( t_1 = X \)
\( t_2 = Z_{\text{acq}} \)
\( Y = 2 \)
\( V = 1 \)
Compiler Independent Matching

\[ s_1 = X \]
\[ \times s_2 = X \]
\[ \checkmark V = 1 \]
\[ \checkmark s_4 = Z_{\text{acq}} \]
\[ \times Y = 1 \]
\[ \checkmark Y = 2 \]
\[ t_1 = X \]
\[ t_2 = Z_{\text{acq}} \]
\[ Y = 2 \]
\[ V = 1 \]
Compiler Independent Matching

\[ \begin{align*}
\checkmark & \quad s_1 = X \\
\times & \quad s_2 = X \\
\checkmark & \quad V = 1 \\
\checkmark & \quad s_4 = Z_{\text{acq}} \\
\times & \quad Y = 1 \\
\checkmark & \quad Y = 2
\end{align*} \]
Compiler Independent Matching

- $s_1 = X$
- $s_2 = X$
- $V = 1$
- $s_4 = Z_{acq}$
- $Y = 2$
- $Y = 1$
- $Y = 2$

Correct

Check that unmatched accesses are deletable
Check that reorderings are allowed
Check that unmatched accesses are deletable
Check that reorderings are allowed
Compiler Independent Matching

\[ s_1 = X \]
\[ s_2 = X \]
\[ V = 1 \]
\[ s_4 = Z_{acq} \]
\[ Y = 1 \]
\[ Y = 2 \]

\[ t_1 = X \]
\[ t_2 = Z_{acq} \]
\[ Y = 2 \]
\[ V = 1 \]

- Check that unmatched accesses are deletable
- Check that reorderings are allowed
Compiler Independent Matching

Correct

- Check that unmatched accesses are deletable
- Check that reorderings are allowed
Use branching conditions to match the paths
Unroll loops a fixed number of times
Validated according to LLVM memory model

<table>
<thead>
<tr>
<th>Test class</th>
<th>LLVM 3.6</th>
<th>LLVM 3.7rc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100 prog./class)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Straightline</td>
<td>95</td>
<td>0</td>
</tr>
<tr>
<td>With branches</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>With dead paths</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>With loops</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>Smaller tests</td>
<td>32</td>
<td>0</td>
</tr>
</tbody>
</table>

- Examples frequently expose errors in LLVM 3.6
- No false positives!
Experimental Evaluation

Validated according to C11 memory model

<table>
<thead>
<tr>
<th>Test class</th>
<th># Reported errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100 prog./class)</td>
<td>LLVM 3.6</td>
</tr>
<tr>
<td>Straightline</td>
<td>0</td>
</tr>
<tr>
<td>With branches</td>
<td>13</td>
</tr>
<tr>
<td>With dead paths</td>
<td>6</td>
</tr>
<tr>
<td>With loops</td>
<td>6</td>
</tr>
<tr>
<td>Smaller tests</td>
<td>7</td>
</tr>
</tbody>
</table>

- Errors often masked by adjacent accesses
Masking of Errors by Adjacent Accesses

\[ s_2 = Z_{acq} \quad t_2 = X \]
\[ s_3 = X \quad t_3 = Z_{acq} \]
Masking of Errors by Adjacent Accesses

\[ s_1 = X \]
\[ s_2 = Z_{\text{acq}} \]
\[ s_3 = X \]
\[ s_1 = X \]
\[ t_1 = X \]
\[ t_2 = X \]
\[ t_3 = Z_{\text{acq}} \]
\[ t_4 = X \]
Masking of Errors by Adjacent Accesses

\[ s_1 = X \quad \quad t_1 = X \]
\[ s_2 = Z_{acq} \quad \quad t_2 = X \]
\[ s_3 = X \quad \quad t_3 = Z_{acq} \]
\[ s_1 = X \quad \quad t_4 = X \]
Metadata-Based Matching

\[ s_1 = X \ !A \quad t_1 = X \ !A \]
\[ s_2 = Z_{acq} \ !B \quad t_2 = X \ !C \]
\[ s_3 = X \ !C \quad t_3 = Z_{acq} \ !B \]
\[ s_4 = X \ !D \quad t_4 = X \ !D \]
Summary

- C11 and LLVM semantics are different
- Reported three LLVM concurrency compilation bugs; all were fixed.
- Validator: http://plv.mpi-sws.org/validc/

Future Work

- Handle arrays, pointers, sequential optimisations
- Integrate with sequential validator
- Formalize the LLVM concurrency model