Robustness Against Release/Acquire Semantics

Ori Lahav
Roy Margalit
A short story: Peterson’s algorithm in C++

• In 1981, Peterson proposed a simple algorithm for critical section in shared memory.

• It assumes sequential consistent shared memory (SC).

• Q: How to implement Peterson’s algorithm in C/C++11?
A short story: Peterson’s algorithm in C++

```cpp
Peterson::Peterson() {
    _victim.store(0, memory_order_release);
    _interested[0].store(false, memory_order_release);
    _interested[1].store(false, memory_order_release);
}
void Peterson::lock() {
    int me = threadID; // either 0 or 1
    int he = 1 - me; // the other thread
    _interested[me].exchange(true, memory_order_acq_rel);
    _victim.store(me, memory_order_release);
    while (! _interested[he].load(memory_order_acquire)
        && _victim.load(memory_order_acquire) == me) {
        continue; // spin
    }
}
```

7. Dmitriy V'jukov Says:

December 3, 2008 at 4:55 am

Memory ordering in your implementation of Peterson’s algo is both insufficient and excessive at the same time. In both permits races and contains unnecessary fences.

11. Bartosz Milewski Says:

So even though I don’t have a formal proof, I believe my implementation of Peterson lock is correct. For all I know, Dmitriy’s implementation might also be correct, but it’s much harder to prove.

C++ atomics and memory ordering, blog post by Bartosz Milewski
https://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/
A subsequent post by Anthony Williams analyzed both algorithms:

- Bartosz’s implementation is indeed wrong.
- Dmitriy’s implementation is correct.

https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html

"Any time you deviate from SC, you increase the complexity of the problem by orders of magnitude."
Goal

Automatically establish **robustness** of programs against a weak memory model

\[
\text{verification under weak memory} = \text{verification under sequential consistency} + \text{robustness}
\]

- Key ingredient in **automatic fence insertion**
- Our focus: C/C++11’s **Release/Acquire** fragment
- Previous work: hardware models (especially **x86-TSO**)

---

**Goal**

Automatically establish **robustness** of programs against a weak memory model

\[
\text{verification under weak memory} = \text{verification under sequential consistency} + \text{robustness}
\]

- Key ingredient in **automatic fence insertion**
- Our focus: C/C++11’s **Release/Acquire** fragment
- Previous work: hardware models (especially **x86-TSO**)
Our Contribution

- Theorem
  - Execution-graph robustness against Release/Acquire is **decidable** and **PSPACE-complete**.
  - as verification under SC
  - as robustness against x86-TSO

- **A tool** for verifying execution-graph robustness

- **Evaluation** on several challenging synchronization algorithms
Release/Acquire in C/C++11

Implementability
- allows *cheaper implementation* (w.r.t. SC):
  - x86-TSO: use primitive accesses
  - IBM Power: use “lightweight” fences

Programmability
- ensures the DRF property
- often *sufficiently strong*:
  - *but not always*...
    (e.g., Peterson’s algorithm)
  - supports “message passing” idiom
**Syntax**

```
atomic_store_explicit(&x, r, memory_order_release)

r = atomic_load_explicit(&x, memory_order_acquire)

atomic_fetch_add_explicit(&x, r, memory_order_acq_rel)

b = atomic_compare_exchange_strong_explicit(&x, &r1, r2, memory_order_acq_rel, memory_order_acquire)

atomic_thread_fence(memory_order_seq_cst)
```
Semantics (one-slide course)

- A form of **causal consistency**
- Defined **declaratively** using **execution graphs**

\[
\text{happens-before} = (\text{program-order} \cup \text{reads-from})^+
\]

**modification-order** - total order on writes to the same location

**inconsistent** execution graph
**disallowed** program outcome
Operational version

machine state $(q, G)$

machine step $(q, G) \rightarrow_{RA} (q', G')$

program left to run

local store

finite state space

infinite state space

current consistent execution graph

$a = y \quad b = x$

$a = 0 \quad b = 0$
Example: “store-buffer” litmus test

\[
\begin{align*}
  x &= 1 & y &= 1 & \text{litmus test} \\
  a &= y & b &= x \\
  a &= 0 & b &= 0 \\
  \text{Initial state} & \quad (q_0, G_0) & \quad \xrightarrow{Wx1_{T1}}_{RA} & \quad (q_1, G_1) & \quad \xrightarrow{Ry0_{T1}}_{RA} & \quad (q_2, G_2) & \quad \xrightarrow{Wy1_{T2}}_{RA} & \quad (q_3, G_3) & \quad \xrightarrow{Rx0_{T2}}_{RA} & \quad (q_4, G_4) & \quad \text{final state}
\end{align*}
\]
Robustness

∀ q . \left( \exists G . (q_0, G_0) \rightarrow^*_{\text{RA}} (q, G) \right) \implies \left( \exists M . (q_0, M_0) \rightarrow^*_{\text{SC}} (q, M) \right)

Bad news…

• Reduction from state reachability [Bouajjani, Derevenetc, Meyer ESOP’13]
• State-reachability for Release/Acquire is undecidable! [Abdulla, Arora, Atig, Krishna PLDI’19]
Execution-graph robustness

∀q, G . (q₀, G₀) →*ₐ(q, G) ⟹ G describes an SC-history

State robustness
∀q . (∃G . (q₀, G₀) →*ₐ(q, G)) ⟹ (∃M . (q₀, M₀) →*ₚ(q, M))

hb∪mo can be linearized to an execution order of an SC-run
Theorem

Execution-graph robustness against Release/Acquire is **decidable** and **PSPACE-complete**.

Reduction to *reachability* under *an instrumented SC semantics*

A "minimal" robustness violation:

\[(q_0, G_0) \xrightarrow{\text{RA}} (q_1, G_1) \xrightarrow{\text{RA}} (q_2, G_2) \xrightarrow{\text{RA}} \cdots \xrightarrow{\text{RA}} (q_n, G_n) \xrightarrow{\text{RA}} (q_{n+1}, G_{n+1})\]

Can take an RA-step to a non-SC execution graph

\[(q_0, M_0) \xrightarrow{\text{SC}} (q'_1, M_1) \xrightarrow{\text{SC}} (q'_2, M_2) \xrightarrow{\text{SC}} \cdots \xrightarrow{\text{SC}} (q_n, M_n)\]

\[I_0 \quad I_1 \quad I_2 \quad \cdots \quad I_n\]

A "minimal" robustness violation is allowed by SC but disallowed by SC.
For \( w \) = the mo-maximal write to \( x \) (\( Wx1 \)):

- \( w \) has no \( hb \) \( T2 \)
- Every SC-run producing \( G_3 \) executes \( w \) before the current last event of \( T2 \)
Instrumented SC Semantics

Figure 5. Maintaining $V_{SC}$, $M_{SC}$ and $W_{SC}$ in SCM transitions.

Figure 6. Maintaining $V$, $W$, $V_{RWI}$, and $W_{RWI}$ in SCM transitions.
Complications

- **Read-modify-write (RMW) instructions**

  require much more *refined instrumentation* (depends on values being read)

- **Masking benign violations**

  masked using blocking instructions:

    - Not robust:
      
      \[
      \begin{align*}
      X &= Y = 0 \\
      X &= 1, & Y &= 1 \\
      \text{do } a &= Y \text{ while } (a \neq 1) \quad | \quad \text{do } b &= X \text{ while } (b \neq 1)
      \end{align*}
      \]

    - Robust:
      
      \[
      \begin{align*}
      X &= Y = 0 \\
      X &= 1, & Y &= 1 \\
      \text{wait } (Y &= 1) \quad | \quad \text{wait } (X &= 1)
      \end{align*}
      \]

- **Sequentially consistent fences**

  modelled as RMWs
<table>
<thead>
<tr>
<th>number of threads</th>
<th>number of lines</th>
<th>robust?</th>
<th>Time (sec)</th>
<th>w/o robustness instrumentation</th>
<th>Trencher (TSO)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SC (sec)</td>
<td>Result</td>
</tr>
<tr>
<td>spin-lock</td>
<td>2</td>
<td>34</td>
<td>✓</td>
<td>1.6</td>
<td>1.2</td>
</tr>
<tr>
<td>seq-lock</td>
<td>4</td>
<td>49</td>
<td>✓</td>
<td>20.7</td>
<td>3.4</td>
</tr>
<tr>
<td>Peterson</td>
<td>2</td>
<td>28</td>
<td>✗</td>
<td>2.5</td>
<td>1.2</td>
</tr>
<tr>
<td>Peterson for x86-TSO</td>
<td>2</td>
<td>30</td>
<td>✗</td>
<td>3.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Peterson - Dmitriy</td>
<td>2</td>
<td>36</td>
<td>✓</td>
<td>4.3</td>
<td>1.2</td>
</tr>
<tr>
<td>Peterson - Bartosz</td>
<td>2</td>
<td>28</td>
<td>✗</td>
<td>3.4</td>
<td>1.1</td>
</tr>
<tr>
<td>RCU</td>
<td>4</td>
<td>74</td>
<td>✓</td>
<td>67.6</td>
<td>2.2</td>
</tr>
<tr>
<td>RCU (offline)</td>
<td>3</td>
<td>215</td>
<td>✓</td>
<td>137.9</td>
<td>18.3</td>
</tr>
</tbody>
</table>

Note: * requires blocking instructions against x86-TSO.
Summary

- We developed a **sound and precise reduction** from execution-graph robustness against Release/Acquire semantics to a reachability problem under SC.
- Execution-graph robustness against Release/Acquire is **PSPACE-complete**.
- We **implemented** the reduction and verified several challenging algorithms, demonstrating in particular that execution-graph robustness is not overly strong.

\[
\text{verification under weak memory} = \text{verification under sequential consistency} + \text{robustness}
\]

input program with Release/Acquire atomics and non-atomics \(\xrightarrow{\text{Rocker}}\) verification problem in Promela \(\xrightarrow{\text{SPIN model checker}}\) not robust

---

Thank you!
<table>
<thead>
<tr>
<th>( X = Y = 0 )</th>
<th>( X = 1 )</th>
<th>( Y = 1 )</th>
<th>( X = 0 )</th>
<th>( Y = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = Y ) // 0</td>
<td>( b = X ) // 0</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>

**state robustness**  | **execution graph robustness**

- \( X = Y = 0 \)
- \( X = 1 \)
- \( a = Y \) // 0
- \( b = X \) // 0