Taming release-acquire consistency

Ori Lahav  Nick Giannarakis  Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

POPL 2016
Weak memory models provide formal sound semantics for realistic high-performance concurrency.
Weak memory models provide formal sound semantics for realistic high-performance concurrency.
Weak memory models provide formal sound semantics for realistic high-performance concurrency.

Sequential consistency (SC) vs. release-acquire memory model (RA) on a spectrum of programmability vs. performance.
C11’s release-acquire memory model

C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release.
C11’s release-acquire memory model

C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

### Store buffering

\[
\begin{align*}
x &= y = 0 \\
x &:= 1; \quad | \quad y := 1; \\
\text{print } y \quad | \quad \text{print } x \\
\end{align*}
\]

both threads may print 0

### Message passing

\[
\begin{align*}
x &= m = 0 \\
m &:= 42; \quad | \quad \text{while } x = 0 \\
x &:= 1 \quad | \quad \text{skip;} \\
\end{align*}
\]

only 42 may be printed
C11’s release-acquire memory model

C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

---

Store buffering

\[
\begin{align*}
x &= y = 0 \\
x &:= 1; \quad & y &:= 1; \\
\text{print } y & \quad & \text{print } x \\
\text{both threads may print } 0
\end{align*}
\]

---

Message passing

\[
\begin{align*}
x &= m = 0 \\
m &:= 42; \quad & \text{while } x = 0 \\
& \quad & \text{skip;} \\
x &:= 1 \quad & \text{print } m \\
\text{only 42 may be printed}
\end{align*}
\]
C11’s release-acquire memory model

C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

Store buffering

\[ x = y = 0 \]
\[ x := 1; \parallel y := 1; \]
\[ \text{print } y \parallel \text{print } x \]
both threads may print 0

Message passing

\[ x = m = 0 \]
\[ m := 42; \parallel \text{while } x = 0 \]
\[ \quad \text{skip;} \]
\[ \quad \text{print } m \]
only 42 may be printed
Formal definition

An execution is *consistent* if there are relations:

- **reads-from** \((rf)\): maps every read to a corresponding write, such that:
  
  \[ \text{happens-before} = (\text{program-order} \cup \text{reads-from})^+ \text{ is irreflexive.} \]

- **modification-order** \((mo)\): total order on same-location writes, such that none of the following occur:

\[
\begin{align*}
&W_x, v \\ 
&\text{mo} \downarrow \text{hb} \\ 
&W_x, v' \\
&W_x, v \\ &\text{mo} \rightarrow W_x, v' \\
&W_x, v \\ &\text{rf} \rightarrow R_x, v \\
&W_x, v \\ &\text{mo} \downarrow \text{mo} \\
&U_x, v, v''
\end{align*}
\]
Good news

- **Verified compilation schemes:**
  - x86-TSO (trivial compilation) [Batty el al. '11]
  - Power [Batty el al. '12] [Sarkar el al. '12]

- **RA supports intended optimizations:**
  - In particular, write-read reordering (unlike SC):
    \[ \text{Wx} \rightarrow \text{Ry} \quad \sim \quad \text{Ry} \rightarrow \text{Wx} \]

- **DRF theorem (unlike full C11):**
  - No data races under SC ensures no weak behaviors

- **Monotonicity (unlike full C11, and x86-TSO):**
  - Adding synchronization does not introduce new behaviors

- **Program logics:**
  - RSL [Vafeiadis and Narayan '13]
  - GPS [Turon et al. '14]
  - OGRA [L and Vafeiadis '15]
Bad news

1. Allowed dubious behaviors

2. Overly weak SC-fences

3. No intuitive operational semantics
Bad news

1. Allowed dubious behaviors

2. Overly weak SC-fences

3. No intuitive operational semantics
Unobservable behaviors

\[ x = y = a = b = 0 \]

\[ x := 1; \quad \text{print } a; \quad y := 1; \]
\[ y := 2; \quad \text{print } b; \quad x := 2; \]
\[ a := 1 \quad \text{print } x; \quad b := 1 \]
\[ \text{print } y \]

Can this program print \(1, 1, 1, 1\)?
Unobservable behaviors

\[ x = y = a = b = 0 \]

\[ x := 1; \quad \text{print } a; \quad y := 1; \]
\[ y := 2; \quad \text{print } b; \quad x := 2; \]
\[ a := 1 \quad \text{print } x; \quad b := 1 \quad \text{print } y \]

Can this program print \(1, 1, 1, 1\)?
Unobservable behaviors

\[ x = y = a = b = 0 \]

\[ x := 1; \quad \text{print } a; \quad y := 1; \]
\[ y := 2; \quad \text{print } b; \quad x := 2; \]
\[ a := 1 \quad \text{print } x; \quad b := 1 \quad \text{print } y \]

Can this program print 1, 1, 1, 1?
An execution is \textit{SRA-consistent} if it is RA-consistent and $hb \cup \bigcup_x mo_x$ is acyclic.
Strong release/acquire consistency

Definition (SRA-consistency)

An execution is *SRA-consistent* if it is RA-consistent and \( \text{hb} \cup \bigcup_x \text{mo}_x \) is acyclic.

If there are no write-write races then SRA and RA coincide.
Better product, same price

- Same compiler optimizations are sound.

- Compilation to x86-TSO and Power is still correct.

* No better deal for Power:

Power model restricted to RA accesses = SRA

(based on Power’s declarative model of [Alglave et al. ’14])
2. Using Fences to Recover SC
The strongest fence instruction provided by C11 is \textit{SC-fence}.

\textbf{Example (Store Buffering)}

\begin{align*}
x &= y = 0 \\
x &= 1; & y &= 1; \\
fence(); & fence(); \\
print y & print x
\end{align*}

\begin{itemize}
\item printing 0 in both threads is disallowed
\end{itemize}

\textbf{Inconsistent:} \((F \times F) \cap (\text{po}^?; (\text{hb} \cup \text{mo} \cup \text{fr}); \text{po}^?)\) is cyclic.

Using the semantics of [Batty et al. ’16].
SC-fences are overly weak

Independent reads, independent writes

\[ x = y = 0 \]
\[ x := 1 \]
\[ \text{print } x; \quad \text{print } y; \]
\[ \text{print } y \quad \text{print } x \]

both threads may print 1, 0

consistent execution
SC-fences are overly weak

Independent reads, independent writes

\[ x = y = 0 \]
\[ x := 1 \]
\[ \text{print } x; \]
\[ \text{fence();} \]
\[ \text{print } y; \]
\[ \text{fence();} \]
\[ \text{print } y \]
\[ \text{print } x \]

both threads may print 1, 0

consistent execution
SC-fences are overly weak

Our suggestion

▶ Model SC-fences as acquire/release atomic updates of a distinguished fence location.
▶ RA semantics enforces all fence events to be ordered by $hb$. 

consistent execution
SC-fences are overly weak

Our suggestion

- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by $hb$.

\[
[x = y = F = 0]
\]

\[
\begin{align*}
W_x, 1 & \xrightarrow{rf} U F, 0, 0 \\
R_x, 1 & \xrightarrow{rf} Ry, 0 \\
R_y, 1 & \xrightarrow{rf} U F, 0, 0 \\
W_y, 1 & \xrightarrow{rf} Rx, 0
\end{align*}
\]
SC-fences are overly weak

Our suggestion

- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by $hb$. 

\[
x = y = F = 0
\]
SC-fences are overly weak

Our suggestion

- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by $hb$. 

$$[x = y = F = 0]$$

Inconsistent:

$Wx, 1 \xrightarrow{rf} Rx, 1 \xleftarrow{rf} Ry, 1 \xrightarrow{rf} Wy, 1$

$U F, 0, 0 \xrightarrow{mo_x} Rx, 1 \xrightarrow{mo_y} Ry, 1 \xleftarrow{mo_x} Wx, 1$

$Rx, 0 \xleftarrow{rf} Ry, 0 \xrightarrow{rf} Rx, 0 \xrightarrow{rf} Wx, 1$
SC-fences are overly weak

Our suggestion

- Model SC-fences as acquire/release atomic updates of a distinguished fence location.
- RA semantics enforces all fence events to be ordered by $hb$.

Inconsistent: $Rx, 0$ reads an overwritten value
Compilation schemes are not affected

**x86-TSO**
- SC-fence $\rightarrow$ mfence
- Atomic update $\rightarrow$ lock xchg
- mfence and lock xchg of an unused value have the same operational effect.

**Power**
- sync events are **equivalent** to a acquire/release atomic updates from an otherwise-unused location.
Reductions to SC

Theorem (basic reduction)

If a program includes a fence between every two racy accesses to different shared variables, then it has only SC behaviors.

Under x86-TSO, it suffices to have a fence between racy writes and subsequent racy reads.

\[
\begin{align*}
x & := e; \\
fence(); \\
r & := y
\end{align*}
\]

Theorem (advanced reduction, simplified version)

For client-server programs, it suffices to place fences as under x86-TSO.

In the paper: application to an RCU implementation.
3. An Operational Presentation

- Easier to understand by simulating step-by-step progress
- Foundation for traditional verification techniques
Example: the SRA machine (first attempt)

Message passing

\[ m = x = 0 \]

\[ \text{while } x = 0 \]
\[ \text{skip;} \]
\[ \text{print } m \]

\[ m := 42; \]
\[ x := 1 \]
Example: the SRA machine (first attempt)

Message passing

\[ m = x = 0 \]

\[ m := 42; \]

\[ x := 1 \]

\[ \text{while } x = 0 \]

\[ \text{skip;} \]

\[ \text{print } m \]

CPU 1

\[ m = 42 \]

\[ x = 0 \]

CPU 2

\[ m = 0 \]

\[ x = 0 \]

outgoing message buffer
Example: the SRA machine (first attempt)

Message passing

\[ m = x = 0 \]

\[ m := 42; \]
\[ x := 1 \]

\[ \text{while } x = 0 \]
\[ \text{skip;} \]
\[ \text{print } m \]
Example: the SRA machine (first attempt)

Message passing

\[ m = x = 0 \]

\[ m := 42; \]
\[ x := 1 \]

\[ \text{while } x = 0 \]
\[ \text{skip; } \]
\[ \text{print } m \]

CPU 1

\[ m = 42 \]
\[ x = 1 \]

CPU 2

\[ m = 42 \]
\[ x = 0 \]

outgoing message buffer
Example: the SRA machine (first attempt)

Message passing

\[ m = x = 0 \]

\[ m := 42; \]
\[ x := 1 \]

\[ \text{while } x = 0 \]
\[ \text{skip;} \]
\[ \text{print } m \]

CPU 1

\[ m = 42 \]
\[ x = 1 \]

CPU 2

\[ m = 42 \]
\[ x = 1 \]

Outgoing message buffer

\[ m = 42 \]
\[ x = 1 \]
Example: the SRA machine (first attempt)

Message passing

\[ m = x = 0 \]

\[ m := 42; \]
\[ x := 1 \]

while \( x = 0 \)

skip;

\[ \text{print } m \]

CPU 1

\[ m = 42 \]
\[ x = 1 \]

CPU 2

\[ m = 42 \]
\[ x = 1 \]

outgoing message buffer

\[ m = 42 \]
\[ x = 1 \]
\( x := 6; \)
\( \text{if } x = 7 \)
\( \text{print } 'go' \)

\( x := 7; \)
\( \text{if } x = 6 \)
\( \text{print } 'go' \)

‘go’ should be printed at most once
\[ x := 6; \]
\[ \text{if } x = 7 \]
\[ \text{print 'go'} \]
\[ x := 7; \]
\[ \text{if } x = 6 \]
\[ \text{print 'go'} \]

'go' should be printed at most once
\[ x := 6; \]
\[ \textbf{if } x = 7 \]
\[ \textbf{print} 'go' \]

\[ x := 7; \]
\[ \textbf{if } x = 6 \]
\[ \textbf{print} 'go' \]

'go' should be printed at most once
\[
x := 6; \quad \text{if } x = 7 \quad \text{print 'go'}
\]

\[
x := 7; \quad \text{if } x = 6 \quad \text{print 'go'}
\]

'go' should be printed at most once
$x := 6$;
\[\begin{align*}
\text{if } x &= 7 \\
\text{print 'go'}
\end{align*}\]

$\text{'go' should be printed at most once}$
$x := 6;$

$\begin{align*}
\text{CPU 1} & \quad x=7 @2 \\
\text{Global timestamp table} & \quad x=6 @1 \\
& \quad x=7 @2
\end{align*}$

$\begin{align*}
\text{CPU 2} & \quad x=7 @2 \\
& \quad x=7 @2
\end{align*}$

$\begin{align*}
\text{Global timestamp table} & \quad x=2 @2
\end{align*}$

$\text{if } x = 7$

print ‘go’

$\text{if } x = 6$

print ‘go’

‘go’ should be printed at most once
\[ x := 6; \]
\[ \quad \text{if } x = 7 \]
\[ \quad \text{print 'go'} \]
\[ x := 7; \]
\[ \quad \text{if } x = 6 \]
\[ \quad \text{print 'go'} \]

'go' should be printed at most once
The release/acquire fragment of the C/C++11 memory model, strikes a good balance between performance and programmability.

We propose a strengthening of this memory model that:
- forbids weak behaviors, unobservable in any implementation
- has simple fence semantics, with SC-reduction theorems
- admits intuitive operational semantics

The stronger model has no additional implementation cost.

Summary

- The release/acquire fragment of the C/C++11 memory model, strikes a good balance between performance and programmability.

- We propose a strengthening of this memory model that:
  - forbids weak behaviors, unobservable in any implementation
  - has simple fence semantics, with SC-reduction theorems
  - admits intuitive operational semantics

- The stronger model has no additional implementation cost.


Thank you!