Reasoning about
the C/C++ weak memory model

Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS)
17 July 2014
Understanding weak memory consistency

Read the architecture/language specs?
  ▶ Too informal, often wrong.

Read the formalisations?
  ▶ Fairly complex.

Run benchmarks / Litmus tests?
  ▶ Observe only subset of behaviours.

We need a better methodology...
The C11 memory model

Two types of locations: ordinary and atomic

- Races on ordinary accesses \(\leadsto\) error

A spectrum of atomic accesses:

- Relaxed \(\leadsto\) no fence
- Consume reads \(\leadsto\) no fence, but preserve deps
- Release writes \(\leadsto\) no fence (x86); lwsync (PPC)
- Acquire reads \(\leadsto\) no fence (x86); isync (PPC)
- Seq. consistent \(\leadsto\) full memory fence

Explicit primitives for fences
Relaxed behaviour: store buffering

Initially $x = y = 0$.

$x\text{.store}(1, rlx); \quad y\text{.store}(1, rlx);$

$t_1 = y\text{.load}(rlx); \quad t_2 = x\text{.load}(rlx);$

This can return $t_1 = t_2 = 0$.

**Justification:**

$[x = y = 0]$

$W_{rlx}(x, 1) \quad W_{rlx}(y, 1)$

$\downarrow \quad \downarrow$

$R_{rlx}(y, 0) \quad R_{rlx}(x, 0)$

Behaviour observed on x86/Power/ARM
Release-acquire synchronization: message passing

Initially $a = x = 0$.

\[
a = 5; \quad \text{while } (x.\text{load}(acq) == 0); \quad \text{print}(a);
\]

This will always print 5.

**Justification:**

\[
\begin{align*}
W_{\text{na}}(a, 5) & \quad R_{\text{acq}}(x, 1) \\
\downarrow & \quad \downarrow \\
W_{\text{rel}}(x, 1) & \quad R_{\text{na}}(a, 5)
\end{align*}
\]

Release-acquire synchronization
Relaxed accesses don’t synchronize

Initially \( a = x = 0 \).

\[
\begin{align*}
    a &= 5; & \textbf{while} & (x.load(rlx) == 0); \\
    x.store(1, rlx); & \textbf{print}(a);
\end{align*}
\]

The program is racy \( \sim \) undefined semantics.

**Justification:**

\[
\begin{align*}
    W_{na}(a, 5) & \quad R_{rlx}(x, 1) \\
    W_{rlx}(x, 1) & \quad R_{na}(a, ?) & \text{Relaxed accesses don’t synchronize}
\end{align*}
\]
Initially $x = y = 0$.

\[
\text{if } (x.load(rlx) == 1) \quad \text{if } (y.load(rlx) == 1)
\]

\[
y.store(1, rlx); \quad x.store(1, rlx);
\]

C11 allows the outcome $x = y = 1$.

**Justification:**

\[
R_{rlx}(x, 1) \quad R_{rlx}(y, 1)
\]

\[
\downarrow \quad \downarrow
\]

\[
W_{rlx}(y, 1) \quad W_{rlx}(x, 1)
\]

Relaxed accesses don’t synchronize
Given a memory model definition

1. Check that the model is *mathematically sane.*
   - For example, it is monotone.

2. Check that it is *not too weak.*
   - Provides useful reasoning principles.

3. Check that it is *not too strong.*
   - Can be implemented efficiently.

4. Check that it is *actually useful.*
   - Admits the intended program optimisations.
How does the C11 definition rate? (1/2)

Let’s start with some good news...

Verified compilation of atomic accesses to x86 and Power/ARM.

[Batty et al., POPL’11]
[Batty et al., POPL’12]
[Sarkar et al., PLDI’12]

⇒ The C11 model is not too strong.
1. Check that the model is *mathematically sane*.
   \[ \times \text{ No, it is not monotone.} \]

2. Check that it is *not too weak*.
   \[ \times \text{ No, due to dependency cycles.} \]

3. Check that the model is *not too strong*.
   \[ \checkmark \text{ OK, prior work.} \]

4. Check that it is *actually useful*.
   \[ \times \text{ No, it disallows intended program transformations.} \]
Part I. Mathematical sanity

- Monotonicity
- Prefix closure
Monotonicity

“Adding synchronisation should not introduce new behaviours”

Examples:

► Adding a memory fence
► Strengthening the access mode of an operation
► Reducing parallelism, $C_1 \parallel C_2 \leadsto C_1 ; C_2$
► Expression evaluation linearisation:

$$x = a + b ; \leadsto t_1 = a ; t_2 = b ; x = t_1 + t_2 ;$$

► (Roach motel reorderings)
Obstacles to monotonicity

1. The axiom for non-atomic reads

\[ rf(b) = a \land (isNA(a) \lor isNA(b)) \implies hb(a, b) \]

(in combination with dependency cycles)

2. The axiom for SC reads
Sequentionalisation is invalid

\[ a = 1; \begin{aligned}
\text{if} \ (x.\text{load}(rlx) == 1) \\
\text{if} \ (a == 1) \\
\ y.\text{store}(1, rlx); \\
\end{aligned} \begin{aligned}
\text{if} \ (y.\text{load}(rlx) == 1) \\
\ x.\text{store}(1, rlx); \\
\end{aligned} \]

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

\[ W_{na}(a, 1) \quad R_{rlx}(x, 1) \quad R_{rlx}(y, 1) \]

\[ R_{na}(a, 1) \quad W_{rlx}(x, 1) \quad W_{rlx}(y, 1) \]

\[ rf(b) = a \land (\text{isNA}(a) \lor \text{isNA}(b)) \implies hb(a, b) \]
SC read restriction

There shall be a single total order $S$ on all seq_cst operations \[\ldots\] such that each seq_cst operation $B$ that loads a value from an atomic object $M$ observes one of the following values:

- the result of the last modification $A$ of $M$ that precedes $B$ in $S$, if it exists, or
- if $A$ exists, the result of some modification of $M$ in the visible sequence of side effects with respect to $B$ that is not seq_cst and that does not happen before $A$, or
- if $A$ does not exist, \[\ldots\]

\[
\text{rf}(b) = c \land \text{isSC}(b) \implies \\
\text{iscr}(c, b) \lor \neg \text{isSC}(c) \land \nexists a. \, \text{hb}(c, a) \land \text{iscr}(a, b)
\]

where $\text{iscr}(c, b) \overset{\text{def}}{=} \text{scr}(c, b) \land \nexists d. \, \text{scr}(c, d) \land \text{scr}(d, b)$

$\text{scr}(c, b) \overset{\text{def}}{=} \text{iswrite}_{\text{locs}}(b)(c) \land \text{sc}(c, b)$
Strengthening is invalid

\[ x.\text{store}(1, rlx); \]
\[ x.\text{store}(2, sc); \]
\[ y.\text{store}(1, sc); \]
\[ x.\text{store}(3, rlx); \]
\[ y.\text{store}(2, sc); \]
\[ y.\text{store}(3, sc); \]
\[ r = x.\text{load}(sc); \]
\[ s_1 = x.\text{load}(rlx); \]
\[ s_2 = x.\text{load}(rlx); \]
\[ s_3 = x.\text{load}(rlx); \]
\[ t_1 = y.\text{load}(rlx); \]
\[ t_2 = y.\text{load}(rlx); \]
\[ t_3 = y.\text{load}(rlx); \]

\[ r = s_1 = t_1 = 1 \land s_2 = t_2 = 2 \land s_3 = t_3 = 3 \quad \text{— Disallowed} \]
Strengthening is invalid

\[
x.\text{store}(1, \text{rlx});
\]
\[
x.\text{store}(2, \text{sc});
\]
\[
y.\text{store}(1, \text{sc});
\]
\[
x.\text{store}(3, \text{sc});
\]
\[
y.\text{store}(2, \text{sc});
\]
\[
y.\text{store}(3, \text{sc});
\]
\[
r = x.\text{load}(\text{sc});
\]
\[
s_1 = x.\text{load}(\text{rlx});
\]
\[
s_2 = x.\text{load}(\text{rlx});
\]
\[
s_3 = x.\text{load}(\text{rlx});
\]
\[
t_1 = y.\text{load}(\text{rlx});
\]
\[
t_2 = y.\text{load}(\text{rlx});
\]
\[
t_3 = y.\text{load}(\text{rlx});
\]
\[
r = s_1 = t_1 = 1 \land s_2 = t_2 = 2 \land s_3 = t_3 = 3 — \text{Allowed}
\]
“Removing \((hb \cup rf)\)-maximal events should preserve consistency”

- Maximal events should not affect other events
- Does not hold because of release sequences
Release sequences too strong (relaxed writes)

Initially \( x = y = 0 \).

\[
\begin{align*}
  a &= 1; \\
  x.\text{store}(1, \text{release}); &\quad \textbf{while} \ (x.\text{load(}}\text{acq}) \neq 3); \\
  x.\text{store}(3, \text{rlx}); &\quad a = 2;
\end{align*}
\]

This program is not racy.
The acquire synchronizes with the release.
Initially $x = y = 0$.

\[
\begin{align*}
a &= 1; \\
x.\text{store}(1, \text{release}); \\
x.\text{store}(3, rlx); \\
\text{while } (x.\text{load}(acq) \neq 3); \\
&\quad a = 2;
\end{align*}
\]

But this one is racy according to C11. The acquire no longer synchronizes with the release. Same if (*) is in a different thread.
Part II. Not overly weak

- High-level reasoning principles
Some basic high-level reasoning principles

**DRF:** Race-free programs have SC semantics  
≈ Ownership-based reasoning

**Coherence:** SC for single-variable programs  
≈ Non-relational invariants; e.g., \( x \geq 0 \land y \geq 0 \).

**Cumulativity:** Transitive visibility for Rel-Acq  
- Ownership transfer possible
Initially $a = x = 0$.

\[
\begin{align*}
a & = 5; \\
x & .store(\text{release}, 1); \\
\text{while} & (x.load(\text{acq}) == 0); \\
\text{print} & (a);
\end{align*}
\]

This will always print 5.

**Justification:**

\[
\begin{align*}
W_{na}(a, 5) & \rightarrow R_{acq}(x, 1) \\
W_{rel}(x, 1) & \rightarrow R_{na}(x, 5)
\end{align*}
\]

Release-acquire synchronization
Rules for release/acquire accesses
Relaxed separation logic [OOPSLA’13]

Ownership transfer by rel-acq synchronizations.

► Atomic allocation \( \leadsto \) pick loc. invariant \( Q \).

\[
\{ Q(v) \} \ x = \text{alloc}(v); \quad \{ W_Q(x) \ast R_Q(x) \}
\]

► Release write \( \leadsto \) give away permissions.

\[
\{ Q(v) \ast W_Q(x) \} \ x.\text{store}(v, rel); \quad \{ W_Q(x) \}
\]

► Acquire read \( \leadsto \) gain permissions.

\[
\{ R_Q(x) \} \ t = x.\text{load} (acq); \quad \{ Q(t) \ast R_{Q[t:=\text{emp}]}(x) \}
\]
Initially \( a = x = 0 \). Let \( J(v) \overset{\text{def}}{=} v = 0 \lor \&a \mapsto 5 \).

\[
\begin{align*}
\{ \&a \mapsto 0 \ast W_J(x) \} & \quad \{ R_J(x) \} \\
\text{a} = 5; & \quad \text{while } (x.\text{load}(acq) == 0); \\
\{ \&a \mapsto 5 \ast W_J(x) \} & \quad \{ \&a \mapsto 5 \} \\
x.\text{store}(\text{release}, 1); & \quad \text{print}(a); \\
\{ W_J(x) \} & \quad \{ \&a \mapsto 5 \}
\end{align*}
\]

\textbf{PL consequences:} Ownership transfer works!
Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

\[
\{R_Q(x)\} \quad t := x.\text{load}(rlx) \quad \{R_Q(x)\}
\]

- Relaxed writes:

\[
Q(v) = \text{emp} \\
\{W_Q(x)\} \quad x.\text{store}(v, rlx) \quad \{W_Q(x)\}
\]

Unsound because of dependency cycles!
Dependency cycles

Initially $x = y = 0$.

\[
\text{if } (x.\text{load}(rlx) == 1) \quad \text{if } (y.\text{load}(rlx) == 1)
\]

\[
y.\text{store}(1, rlx);
\]

\[
x.\text{store}(1, rlx);
\]

C11 allows the outcome $x = y = 1$.

**Justification:**

\[
R_{rlx}(x, 1) \quad R_{rlx}(y, 1)
\]

\[
W_{rlx}(y, 1) \quad W_{rlx}(x, 1)
\]

Relaxed accesses don’t synchronize
Initially $x = y = 0$.

\[
\text{if } (x.\text{load}(rlx) == 1) \quad \text{if } (y.\text{load}(rlx) == 1)
\]
\[
y.\text{store}(1, rlx); \quad x.\text{store}(1, rlx);
\]

C11 allows the outcome $x = y = 1$.

**What goes wrong:**
Non-relational invariants are unsound.

\[
x = 0 \land y = 0
\]

The DRF-property does not hold.
Initially $x = y = 0$.

\[
\text{if } (x.\text{load}(rlx) == 1) \quad \text{if } (y.\text{load}(rlx) == 1) \\
y.\text{store}(1, rlx); \quad x.\text{store}(1, rlx);
\]

C11 allows the outcome $x = y = 1$.

**How to fix this:**

- Don’t use relaxed writes
- Strengthen the model
Initially \( a = x = 0 \).

\[
a = 5; \\
x.\text{store}(\text{release, } \& a); \\
t = x.\text{load}(\text{consume}); \\
\text{if } (t \neq 0) \text{ print}(*t);
\]

This program cannot crash nor print 0.

\[\begin{align*}
W_{na}(a, 5) & \quad R_{con}(x, \& a) \\
\downarrow & \quad \downarrow \\
W_{rel}(x, \& a) & \quad R_{na}(a, 5)
\end{align*}\]

Release-consume synchronization

\textbf{Justification:}
Release-consume synchronization

Initially \( a = x = 0 \). Let \( J(t) \overset{\text{def}}{=} t = 0 \lor t \mapsto 5 \).

\[
\begin{align*}
\{ &a \mapsto 0 \ast W_J(x) \} &\quad \{ &R_J(x) \} \\
\quad a = 5; &\quad t = x.\text{load}(\text{consume}); \\
\{ &a \mapsto 5 \ast W_J(x) \} &\quad \{ \nabla_t (t = 0 \lor t \mapsto 5) \} \\
x.\text{store}(\text{release}, &a); &\quad \text{if } (t \neq 0) \text{ print}(\ast t);
\end{align*}
\]

This program cannot crash nor print 0.

**PL consequences:**
Needs funny modality, but otherwise OK.
Proposed rules for consume accesses

\( \{ R_Q(x) \} \ t := x.\text{load}(\text{cons}) \ \{ R_Q[t:=\text{emp}](x) * \nabla_t Q(t) \} \)

\( \{ P \} \ C \ \{ Q \} \)

C is basic command mentioning t

\( \{ \nabla_t P \} \ C \ \{ \nabla_t Q \} \)

Question: Is the following valid?

\( \{ W_Q(x) * \nabla_t Q(v) \} x.\text{store}(v, \text{rel}); \ \{ W_Q(x) \} \)
Release-acquire too weak in the presence of consume

Initially $x = y = 0$.

$\begin{align*}
a &= 1; \\
x &. \text{store}(1, \text{release}); \\
y &. \text{store}(1, \text{release}); \\
\text{while } (x . \text{load}(\text{consume}) \neq 1); \\
\text{while } (y . \text{load}(\text{acquire}) \neq 1); \\
(*) a &= 2;
\end{align*}$

C11 deems this program racy.

- Only different thread rel-acq synchronize.

What goes wrong in PL:

On ownership transfers, we must prove that we don’t read from the same thread.
Initially $x = y = 0$.

\[
\begin{align*}
& a = 1; \\
& x.\text{store}(1, \text{release}); \\
& y.\text{store}(1, \text{release}); \\
\end{align*}
\]

while $(x.\text{load}(\text{consume}) \neq 1); \\
\text{while} (y.\text{load}(\text{acquire}) \neq 1); \\
(\ast) a = 2;
\]

C11 deems this program racy. But, it is not racy:

- On x86-TSO, Power, ARM, and Itanium.
- Or if we move the $(\ast)$ lines to a new thread.

So, drop the “different thread” restriction.
Part III. Actual usefulness

- Verify source-to-source program transformations
A study of optimisations under C11

- “Roach motel” reorderingings
  (depends on how we fix dependency cycles)

- Elimination of redundant accesses
  (overwritten write, read after same R/W)
  (write after same read is invalid)

- Introduction of unused reads
  (invalid → may race)

- Elimination of unused reads
  (only non-atomic, others may synchronise)
Valid instruction reorderings \( a ; b \leadsto b ; a \)

| \( \downarrow a \ \backslash \ b \rightarrow \) | \( R_{\not= sc} \) | \( R_{sc} \) | \( W_{na} \) | \( W_{rlx} \) | \( W_{\sqsupset rel} \) | \( C_{rlx|acq} \) | \( C_{\sqsupset rel} \) | \( F_{acq} \) | \( F_{rel} \) |
|----------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| \( R_{na} \)   | \( \checkmark \) | \( \checkmark \) | (\( \checkmark \)) | (\( \checkmark \)) | \( X \) | (\( \checkmark \)) | \( X \) | \( \checkmark \) | \( X \) |
| \( R_{rlx} \)  | \( \checkmark \) | \( \checkmark \) | (\( \checkmark \)) | (\( \times \)) | \( X \) | (\( \times \)) | \( X \) | \( X \) | \( X \) |
| \( R_{\sqsupset acq} \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) |
| \( W_{\not= sc} \) | \( \checkmark \) | \( \checkmark \) | \( \checkmark \) | \( \checkmark \) | \( X \) | \( \checkmark \) | \( X \) | \( \checkmark \) | \( X \) |
| \( W_{sc} \)   | \( \checkmark \) | \( \times \) | \( \checkmark \) | \( \checkmark \) | \( X \) | \( \checkmark \) | \( X \) | \( \checkmark \) | \( X \) |
| \( C_{rlx|rel} \) | \( \checkmark \) | \( \checkmark \) | (\( \checkmark \)) | (\( \times \)) | \( X \) | (\( \times \)) | \( X \) | \( X \) | \( X \) |
| \( C_{\sqsupset acq} \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) |
| \( F_{acq} \)  | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) |
| \( F_{rel} \)  | \( \checkmark \) | \( \checkmark \) | \( \checkmark \) | \( \times \) | \( \checkmark \) | \( \times \) | \( \checkmark \) | \( \checkmark \) | \( \times \) |
Redundant instruction eliminations

Overwritten write:
\[
x . \text{store}(v, M) ; C ; x . \text{store}(v', M) \quad C \text{ has no rel}
\]
\[
\sim C ; x . \text{store}(v', M) \quad & \text{no } x \text{ accesses}
\]

Read after write:
\[
x . \text{store}(v, M) ; C ; t = x . \text{load}(M') \quad C \text{ has no acq}
\]
\[
\sim x . \text{store}(v, M) ; C ; t = v \quad & \text{no } x \text{ accesses}
\]

Read after read:
\[
t = x . \text{load}(M) ; C ; t' = x . \text{load}(M) \quad C \text{ has no acq}
\]
\[
\sim t = x . \text{load}(M) ; C ; t' = t \quad & \text{no } x \text{ accesses}
\]
Write-after-read elimination is invalid

\[ t = x.\text{load}(M); x.\text{store}(t, rlx) \]
\[ \not\Rightarrow t = x.\text{load}(M) \]

There could be a CAS “in between”

\[ x = y = 0; \]
\[ y.\text{store}(1, rlx); \]
\[ \text{fence(release)}; \quad t_2 = x.\text{CAS}(0, 1, \text{acq}); \]
\[ t_1 = x.\text{load}(rlx); \quad t_3 = y.\text{load}(rlx); \]
\[ x.\text{store}(t_1, rlx); \quad t_4 = x.\text{load}(rlx); \]

Can we get \( t_1 = t_2 = t_3 = 0 \) and \( t_4 = 1 \)?
What have we learnt?

The C11 memory model is broken
  ▶ But is largely fixable

Tools for understanding weak memory models:
  ▶ Source-to-source program transformations
  ▶ Relaxed program logics