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 $\rightsquigarrow$ error

A spectrum of atomic accesses:

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

Explicit primitives for fences

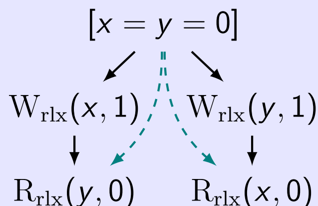
Relaxed behaviour: store buffering

Initially $x = y = 0$.

$$\begin{array}{l} x.\mathsf{store}(1, rlx); \\ t_1 = y.\mathsf{load}(rlx); \end{array} \quad \Bigg\| \quad \begin{array}{l} y.\mathsf{store}(1, rlx); \\ t_2 = x.\mathsf{load}(rlx); \end{array}$$

This can return $t_1 = t_2 = 0$.

Justification:



Behaviour observed on x86/Power/ARM

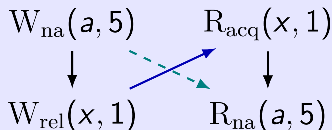
Release-acquire synchronization: message passing

Initially $a = x = 0$.

$a = 5;$ $x.\text{store}(1, \mathit{release});$	‖	**while** $(x.\text{load}(\mathit{acq}) == 0);$ $\text{print}(a);$

This will always print 5.

Justification:



Release-acquire synchronization

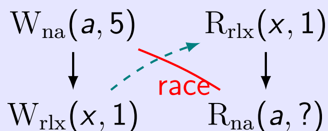
Relaxed accesses don't synchronize

Initially $a = x = 0$.

$a = 5;$ $x.\text{store}(1, rlx);$	**while** $(x.\text{load}(rlx) == 0);$ $\text{print}(a);$

The program is racy $\rightsquigarrow$ undefined semantics.

Justification:



Relaxed accesses don't synchronize

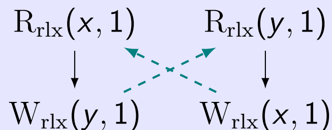
Dependency cycles

Initially $x = y = 0$.

$$\textbf{if } (x.load(rlx) == 1) \quad y.store(1, rlx); \;\Big\|\; \textbf{if } (y.load(rlx) == 1) \quad x.store(1, rlx);$$

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

Justification:



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]

$\Longrightarrow$ The C11 model is *not too strong*.

How does the C11 definition rate? (2/2)

1. Check that the model is *mathematically sane*.

   ✗ No, it is not monotone.

2. Check that it is *not too weak*.

   ✗ No, due to dependency cycles.

3. Check that the model is *not too strong*.

   ✓ OK, prior work.

4. Check that it is *actually useful*.

   ✗ 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, $C_1 \| C_2 \rightsquigarrow C_1 \,;\, C_2$
- Expression evaluation linearisation:

$$x = a + b\,; \quad \rightsquigarrow \quad t_1 = a\,; t_2 = b\,; x = t_1 + t_2\,;$$

- (Roach motel reorderings)

Obstacles to monotonicity

1. The axiom for non-atomic reads

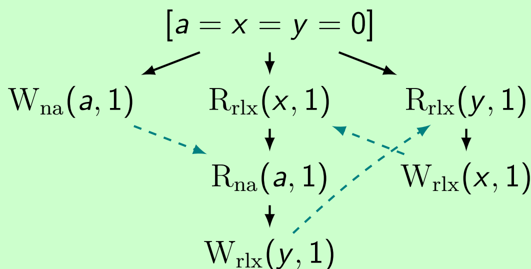
$$\mathsf{rf}(b) = a \wedge (\mathsf{isNA}(a) \vee \mathsf{isNA}(b)) \implies \mathsf{hb}(a, b)$$

   (in combination with dependency cycles)

2. The axiom for SC reads

Sequentionalisation is invalid

$$a = 1; \;\Bigg\|\; \begin{array}{l} \textbf{if } (x.\mathsf{load}(rlx) == 1) \\ \quad \textbf{if } (a == 1) \\ \quad\quad y.\mathsf{store}(1, rlx); \end{array} \;\Bigg\|\; \begin{array}{l} \textbf{if } (y.\mathsf{load}(rlx) == 1) \\ \quad x.\mathsf{store}(1, rlx); \end{array}$$



$$\mathsf{rf}(b) = a \wedge (\mathsf{isNA}(a) \vee \mathsf{isNA}(b)) \implies \mathsf{hb}(a, b)$$

SC read restriction

There shall be a single total order S on all `seq_cst` operations [...] 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, [...]

[N1570, §7.17.3.6]

$$\begin{aligned}&\mathrm{rf}(b) = c \wedge \mathrm{isSC}(b) \implies \\ &\qquad \mathrm{iscr}(c, b) \;\vee\; \neg\mathrm{isSC}(c) \wedge \nexists a.\ \mathrm{hb}(c, a) \wedge \mathrm{iscr}(a, b)\end{aligned}$$

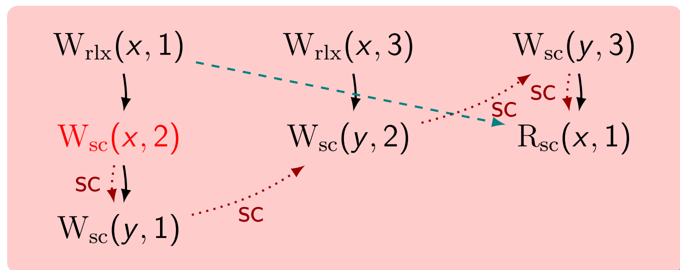
where $\mathrm{iscr}(c, b) \stackrel{\mathrm{def}}{=} \mathrm{scr}(c, b) \wedge \nexists d.\ \mathrm{scr}(c, d) \wedge \mathrm{scr}(d, b)$

$\qquad \mathrm{scr}(c, b) \stackrel{\mathrm{def}}{=} \mathrm{iswrite}_{\mathrm{locs}(b)}(c) \wedge \mathrm{sc}(c, b)$

Strengthening is invalid

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

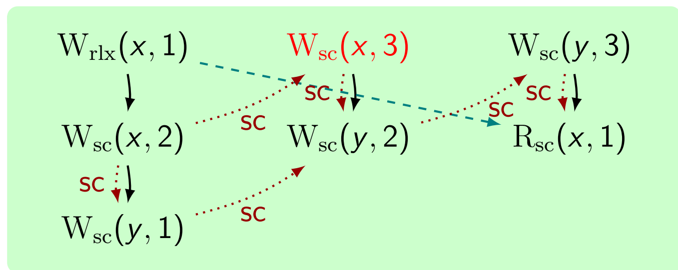
$r = s_1 = t_1 = 1 \wedge s_2 = t_2 = 2 \wedge s_3 = t_3 = 3$ — Disallowed



Strengthening is invalid

x.store(1, *rlx*); x.store(2, *sc*); y.store(1, *sc*);	x.store(3, *sc*); y.store(2, *sc*);	y.store(3, *sc*); $r = x$.load(*sc*);	$s_1 = x$.load(*rlx*); $s_2 = x$.load(*rlx*); $s_3 = x$.load(*rlx*); $t_1 = y$.load(*rlx*); $t_2 = y$.load(*rlx*); $t_3 = y$.load(*rlx*);

$r = s_1 = t_1 = 1 \wedge s_2 = t_2 = 2 \wedge s_3 = t_3 = 3$ — Allowed



Prefix closure

"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 $x = y = 0$.

$a = 1;$ $x.\text{store}(1, \textit{release});$ $x.\text{store}(3, \textit{rlx});$	$\textbf{while}\ (x.\text{load}(\textit{acq}) \neq 3);$ $a = 2;$

This program is not racy.

The acquire synchronizes with the release.

Release sequences too strong (relaxed writes)

Initially $x = y = 0$.

$a = 1;$	x.store$(2, rlx)$; $(*)$
x.store$(1, release)$;	**while** $(x$.load$(acq) \neq 3)$;
x.store$(3, rlx)$;	$a = 2;$

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

$\approx$ Ownership-based reasoning

Coherence: SC for single-variable programs

$\approx$ Non-relational invariants; e.g., $x \geq 0 \wedge y \geq 0$.

Cumulativity: Transitive visibility for Rel-Acq

- Ownership tranfer possible

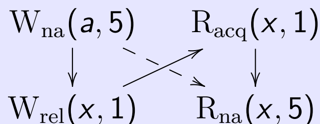
Release-acquire synchronization: message passing

Initially $a = x = 0$.

$a = 5;$ $x.\text{store}(\mathit{release}, 1);$	‖	**while** $(x.\text{load}(\mathit{acq}) == 0);$ $\text{print}(a);$

This will always print 5.

Justification:



Release-acquire synchronization

Ownership transfer by rel-acq synchronizations.

- Atomic allocation $\rightsquigarrow$ pick loc. invariant $\mathcal{Q}$.

$$\{\mathcal{Q}(v)\}\ x = \mathsf{alloc}(v);\ \{\mathbf{W}_{\mathcal{Q}}(x) * \mathbf{R}_{\mathcal{Q}}(x)\}$$

- Release write $\rightsquigarrow$ give away permissions.

$$\{\mathcal{Q}(v) * \mathbf{W}_{\mathcal{Q}}(x)\}\ x.\mathsf{store}(v, \mathit{rel});\ \{\mathbf{W}_{\mathcal{Q}}(x)\}$$

- Acquire read $\rightsquigarrow$ gain permissions.

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t = x.\mathsf{load}(\mathit{acq});\ \{\mathcal{Q}(t) * \mathbf{R}_{\mathcal{Q}[t:=\mathsf{emp}]}(x)\}$$

Release-acquire synchronization: message passing

Initially $a = x = 0$. Let $J(v) \stackrel{\text{def}}{=} v = 0 \vee \&a \mapsto 5$.

$\{\&a \mapsto 0 * \mathbf{W}_J(x)\}$	$\{\mathbf{R}_J(x)\}$
$a = 5;$	**while** $(x.\text{load}(acq) == 0);$
$\{\&a \mapsto 5 * \mathbf{W}_J(x)\}$	$\{\&a \mapsto 5\}$
$x.\text{store}(release, 1);$	$\text{print}(a);$
$\{\mathbf{W}_J(x)\}$	$\{\&a \mapsto 5\}$

PL consequences:

Ownership transfer works!

Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t := x.\mathsf{load}(\mathit{rlx})\ \{\mathbf{R}_{\mathcal{Q}}(x)\}$$

- Relaxed writes:

$$\frac{\mathcal{Q}(v) = \mathsf{emp}}{\{\mathbf{W}_{\mathcal{Q}}(x)\}\ x.\mathsf{store}(v, \mathit{rlx})\ \{\mathbf{W}_{\mathcal{Q}}(x)\}}$$

Unsound because of dependency cycles!

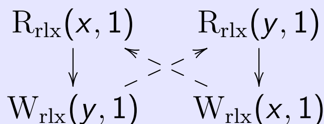
Dependency cycles

Initially $x = y = 0$.

$$\begin{array}{c||c} \textbf{if } (x.load(rlx) == 1) & \textbf{if } (y.load(rlx) == 1) \\ y.store(1, rlx); & x.store(1, rlx); \end{array}$$

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

Justification:



Relaxed accesses don't synchronize

Dependency cycles

Initially $x = y = 0$.

$$\begin{array}{c|c} \textbf{if } (x.load(rlx) == 1) & \textbf{if } (y.load(rlx) == 1) \\ y.store(1, rlx); & x.store(1, rlx); \end{array}$$

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.

Dependency cycles

Initially $x = y = 0$.

$$\textbf{if } (x.load(rlx) == 1) \quad y.store(1, rlx); \quad \Big\| \quad \textbf{if } (y.load(rlx) == 1) \quad x.store(1, rlx);$$

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

How to fix this:

Don't use relaxed writes

$\vee$

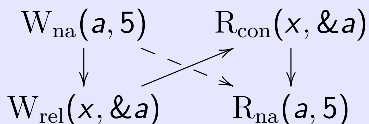
Strengthen the model

Initially $a = x = 0$.

$a = 5;$	$t = x.load(consume);$
$x.store(release, \&a);$	**if** $(t \neq 0)$ $print(*t);$

This program cannot crash nor print 0.

Justification:



Release-consume synchronization

Release-consume synchronization

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

$\{\&a \mapsto 0 * \mathbf{W}_J(x)\}$	$\{\mathbf{R}_J(x)\}$
$a = 5;$	$t = x.load(consume);$
$\{\&a \mapsto 5 * \mathbf{W}_J(x)\}$	$\{\nabla_t(t = 0 \lor t \mapsto 5)\}$
$x.store(release, \&a);$	$\textbf{if } (t \neq 0)\ print(*t);$

This program cannot crash nor print 0.

PL consequences:

Needs funny modality, but otherwise OK.

Proposed rules for consume accesses

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t := x.\mathrm{load}(cons)\ \{\mathbf{R}_{\mathcal{Q}[t:=\mathrm{emp}]}(x) * \nabla_t\, \mathcal{Q}(t)\}$$

$$\frac{\{P\}\ C\ \{Q\} \qquad C \text{ is basic command mentioning } t}{\{\nabla_t\, P\}\ C\ \{\nabla_t\, Q\}}$$

Question: Is the following valid?

$$\{\mathbf{W}_{\mathcal{Q}}(x) * \nabla_t \mathcal{Q}(v)\}\ x.\mathrm{store}(v, rel);\ \{\mathbf{W}_{\mathcal{Q}}(x)\}$$

Release-acquire too weak in the presence of consume

Initially $x = y = 0$.

$$
\begin{array}{l||l}
 & \quad \textbf{while } (x.load(consume) \neq 1); \\
a = 1; & \quad y.store(1, release); \\
x.store(1, release); & (*) \ \textbf{while } (y.load(acquire) \neq 1); \\
 & (*) \ a = 2;
\end{array}
$$

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.

Release-acquire too weak in the presence of consume

Initially $x = y = 0$.

$$
\begin{array}{l||rl}
 & & \textbf{while } (x.load(consume) \neq 1); \\
a = 1; & & y.store(1, release); \\
x.store(1, release); & (*) & \textbf{while } (y.load(acquire) \neq 1); \\
 & (*) & a = 2;
\end{array}
$$

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

- On x86-TSO, Power, ARM, and Itanium.
- Or if we move the $(*)$ 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" reorderings
  (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 $\leadsto$ may race)
- Elimination of unused reads
  (only non-atomic, others may synchronise)

Valid instruction reorderings $a\,;b \leadsto b\,;a$

$\downarrow a \setminus b \rightarrow$	$\mathsf{R}_{\neq \mathrm{sc}}$	R_{sc}	W_{na}	$\mathsf{W}_{\mathrm{rlx}}$	$\mathsf{W}_{\sqsupseteq \mathrm{rel}}$	$\mathsf{C}_{\mathrm{rlx}\mid\mathrm{acq}}$	$\mathsf{C}_{\sqsupseteq \mathrm{rel}}$	$\mathsf{F}_{\mathrm{acq}}$	$\mathsf{F}_{\mathrm{rel}}$
R_{na}	✓	✓	(✓)	(✓)	✗	(✓)	✗	✓	✗
$\mathsf{R}_{\mathrm{rlx}}$	✓	✓	(✓)	(✗)	✗	(✗)	✗	✗	✗
$\mathsf{R}_{\sqsupseteq \mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	✓	✗
$\mathsf{W}_{\neq \mathrm{sc}}$	✓	✓	✓	✓	✗	✓	✗	✓	✗
W_{sc}	✓	✗	✓	✓	✗	✓	✗	✓	✗
$\mathsf{C}_{\mathrm{rlx}\mid\mathrm{rel}}$	✓	✓	(✓)	(✗)	✗	(✗)	✗	✗	✗
$\mathsf{C}_{\sqsupseteq \mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	✓	✗
$\mathsf{F}_{\mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	=	✗
$\mathsf{F}_{\mathrm{rel}}$	✓	✓	✓	✗	✓	✗	✓	✓	=

Redundant instruction eliminations

Overwritten write:

$$x.\text{store}(v, M)\,;C\,;x.\text{store}(v', M) \quad C \text{ has no rel}$$
$$\leadsto C\,;x.\text{store}(v', M) \quad \text{\& no } x \text{ accesses}$$

Read after write:

$$x.\text{store}(v, M)\,;C\,;t = x.\text{load}(M') \quad C \text{ has no acq}$$
$$\leadsto x.\text{store}(v, M)\,;C\,;t = v \quad \text{\& no } x \text{ accesses}$$

Read after read:

$$t = x.\text{load}(M)\,;C\,;t' = x.\text{load}(M) \quad C \text{ has no acq}$$
$$\leadsto t = x.\text{load}(M)\,;C\,;t' = t \quad \text{\& no } x \text{ accesses}$$

Write-after-read elimination is invalid

$$t = x.\text{load}(M)\,;x.\text{store}(t, rlx)$$
$$\not\rightsquigarrow t = x.\text{load}(M)$$

There could be a CAS "in between"

$$x = y = 0;$$
$$\begin{array}{l|l} y.\text{store}(1, \text{rlx}); & \\ \text{fence}(\text{release}); & t_2 = x.\text{CAS}(0, 1, \text{acq}); \\ t_1 = x.\text{load}(\text{rlx}); & t_3 = y.\text{load}(\text{rlx}); \\ x.\text{store}(t_1, \text{rlx}); & \end{array}$$
$$t_4 = x.\text{load}(\text{rlx});$$

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