Software verification under weak memory consistency

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

January 2016

Joint work with Soham Chakraborty, Derek Dreyer, Marko Doko, Nick Giannarakis, Ori Lahav, Chinmay Narayan, Joseph Tassarotti, Aaron Turon.
Sequential consistency (SC):
- The standard model for concurrency.
- Interleave each thread’s atomic accesses.
- Almost all verification work assumes it.

Initially, \( x = y = 0 \).

\[
\begin{align*}
x & := 1; \quad \| \quad y & := 1; \\
a & := y \quad \| \quad b & := x
\end{align*}
\]

In SC, this program cannot return \( a = b = 0 \).
Initially, $x = y = 0$.

\[
\begin{align*}
  x &:= 1; \quad \mid \quad y := 1; \\
  a &:= y; \quad \mid \quad b := x;
\end{align*}
\]

This program can return $a = b = 0$. 

Viktor Vafeiadis

Software verification under weak memory consistency
Owicki-Gries method (1976)

\[ \{P_1\} c_1 \{Q_1\} \quad \{P_2\} c_2 \{Q_2\} \]

the two proofs are non-interfering

\[ \{P_1 \land P_2\} c_1 \parallel c_2 \{Q_1 \land Q_2\} \]

Non-interference

\[ R \land P \vdash R\{u/x\} \text{ for every:} \]

▶ assertion \( R \) in the proof outline of one thread
▶ assignment \( x := u \) with precondition \( P \) in the proof outline of the other thread
Standard OG is unsound for WM

\[
\begin{align*}
\{ a \neq 0 \} \\
 x := 1; & \quad y := 1; \\
 a := y & \quad b := x \\
\{ a \neq 0 \lor b \neq 0 \}
\end{align*}
\]
Standard OG is unsound for WM

\[
\begin{align*}
\{ a \neq 0 \} & \quad \{ a \neq 0 \} & \quad \{ \top \} \\
\{ x \neq 0 \} & \quad \{ y \neq 0 \} \\
\{ x \neq 0 \} & \quad \{ y \neq 0 \} \wedge (a \neq 0 \lor b = x) \\
\{ a \neq 0 \lor b \neq 0 \} & \quad \{ \top \}
\end{align*}
\]

To regain soundness, strengthen the non-inference check.
Standard OG is unsound for WM

\[
\begin{align*}
\{a \neq 0\} &\quad \{a \neq 0\} \quad \{\top\} \\
\{x \neq 0\} &\quad \{y \neq 0\} \\
a := y &\quad b := x \\
\{x \neq 0\} &\quad \{y \neq 0 \land (a \neq 0 \lor b = x)\} \\
\{a \neq 0 \lor b \neq 0\} &
\end{align*}
\]

To regain soundness, strengthen the non-inference check.

\[\implies \text{OGRA: Owicki-Gries for release-acquire (ICALP’15)}\]
Outline

- Introduction to the C11 weak memory model
  - Release-acquire synchronization
  - Per-location coherence

- Reasoning about WMM using program logics
  - RSL (relaxed separation logic)
  - FSL (fenced separation logic)
  - GPS (ghosts, protocols and separation)

- Avoiding to reason about WMM
  - Use reduction theorems
The C11 memory model: Atomics

Two types of locations

Ordinary (Non-Atomic)

Races are errors

Atomic

Welcome to the expert mode
A spectrum of atomic accesses:

- **Seq. consistent**
  - full memory fence

- **Release write**
  - no fence (x86); lwsync (PPC)

- **Acquire read**
  - no fence (x86); isync (PPC)

- **Relaxed**
  - no fence

Explicit primitives for fences
Store buffering in C11

Initially $x = y = 0$.

\[
\begin{align*}
  x &.\text{store}(1, rlx); \\
  \text{print}(y.\text{load}(rlx)); \\
  y &.\text{store}(1, rlx); \\
  \text{print}(x.\text{load}(rlx));
\end{align*}
\]

Can print 00 with the following execution:

$$[x = y = 0]$$

$$W_{rlx}(x, 1)$$

$$R_{rlx}(y, 0)$$

$$W_{rlx}(y, 1)$$

$$R_{rlx}(x, 0)$$
Release-acquire synchronization

Initially $a = x = 0$.

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

One possible execution:

![Diagram showing the order of operations with $W_{na}(a, 0)$, $W_{na}(x, 0)$, $W_{na}(a, 5)$, $W_{rel}(x, 1)$, $R_{acq}(x, 0)$, $R_{acq}(x, 1)$, and $R_{na}(a, 5)$, with $hb = (po \cup sw)^+$]

Happens before: $hb = (po \cup sw)^+$
Coherence

Programs with a single shared variable behave as under SC.

\[
\begin{align*}
    &x.\text{store}(1, rlx); \quad \parallel \quad x.\text{store}(2, rlx); \\
    &a = x.\text{load}(rlx); \quad \parallel \quad b = x.\text{load}(rlx);
\end{align*}
\]

The outcome \( a = 2 \land b = 1 \) is forbidden.

\[
\begin{align*}
    &W_{rlx}(x, 1) \xrightarrow{mo} W_{rlx}(x, 2) \\
    &R_{rlx}(x, 2) \quad \quad R_{rlx}(x, 1)
\end{align*}
\]

- Modification order, \( mo_x \), total order of writes to \( x \).
- Reads-before : \( rb \triangleq (rf^{-1}; mo) \cap (\neq) \)
- Coherence : \( hb \cup rf_x \cup mo_x \cup rb_x \) is acyclic for all \( x \).
Relaxed program logics

- RSL (relaxed separation logic, OOPSLA’13)
- FSL (fenced separation logic, VMCAI’16)
- GPS (ghosts & protocols, OOPSLA’14, PLDI’15)
Separation logic

Key concept of ownership:

- Resourceful reading of Hoare triples.

\[
\{ P \} \ C \ \{ Q \}
\]

- To access a non-atomic location, you must own it:

\[
\{ x \mapsto v \} \star x \{ t. \ t = v \land x \mapsto v \}
\]

\[
\{ x \mapsto v \} \star x = v'; \ \{ x \mapsto v' \}
\]

- Disjoint parallelism:

\[
\begin{array}{c}
\{ P_1 \} \ C_1 \ \{ Q_1 \} \\
\{ P_2 \} \ C_2 \ \{ Q_2 \}
\end{array}
\]

\[
\begin{array}{c}
\{ P_1 \star P_2 \} \ C_1 \parallel C_2 \ \{ Q_1 \star Q_2 \}
\end{array}
\]
Ownership transfer by rel-acq synchronizations.

- Atomic allocation $\leadsto$ pick loc. invariant $Q$.

$$\{Q(v)\} \ x = \text{alloc}(v); \ \{W_Q(x) * R_Q(x)\}$$

- Release write $\leadsto$ give away permissions.

$$\{Q(v) * W_Q(x)\} \ x.\text{store}(v, rel); \ \{W_Q(x)\}$$

- Acquire read $\leadsto$ gain permissions.

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

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

**PL consequences:**
Ownership transfer works!
Relaxed accesses

Basically, disallow ownership transfer.

▶ Relaxed reads:

\[
\{ R_Q(x) \} \quad t := x.\text{load}(rlx) \quad \{ R_Q(x) \land (Q(t) \not= false) \}
\]

▶ 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 $x = y = 0$.

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

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

**Justification:**

\[
\begin{align*}
R_{rlx}(x, 1) & \quad R_{rlx}(y, 1) \\
W_{rlx}(y, 1) & \quad W_{rlx}(x, 1)
\end{align*}
\]

Relaxed accesses don’t synchronize.
Dependency cycles

Initially $x = y = 0$.

$$\begin{align*}
\textbf{if} & \ (x.\text{load}(rlx) == 1) \quad \textbf{if} \ (y.\text{load}(rlx) == 1) \\
& y.\text{store}(1, rlx); \quad x.\text{store}(1, rlx);
\end{align*}$$

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 $x = y = 0$.

\[
\begin{align*}
\text{if } (x#.\text{load}(rlx) == 1) & \quad \text{if } (y#.\text{load}(rlx) == 1) \\
y#.\text{store}(1, rlx); & \quad x#.\text{store}(1, rlx);
\end{align*}
\]

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

**How to fix this:**

Don’t use relaxed writes

\[\lor\]

Strengthen the model
Incorrect message passing

\[
\begin{align*}
\text{int } a; & \quad \text{atomic\_int } x = 0; \\
& \quad (a = 5; \quad \text{if } (x.\text{load}(rlx) \neq 0) \{ \\
& \quad \quad x.\text{store}(1, rlx); \quad \quad \text{print}(a); \})
\end{align*}
\]

\[
\begin{align*}
W_{na}(a, 5) & \quad R_{rlx}(x, 1) \\
\text{race} & \\
W_{rlx}(x, 1) & \quad R_{na}(a, ?)
\end{align*}
\]
Message passing with C11 memory fences

\[
\begin{align*}
\text{int } a; \quad \text{atomic\_int } x = 0; \\
(a = 5; \quad \text{if } (x.\text{load}(\text{rlx}) \neq 0)\
\text{fence(\text{release});} \\
\text{x.\text{store}(1, \text{rlx});} \\
&\quad \text{fence(\text{acq});} \\
&\quad \text{print}(a); \} 
\end{align*}
\]

\[
\begin{array}{ccc}
W_{na}(a, 5) & & R_{rlx}(x, 1) \\
\downarrow & & \downarrow \\
F_{rel} & \overset{sw}{\leftrightarrow} & F_{acq} \\
\downarrow & & \downarrow \\
W_{rlx}(x, 1) & & R_{na}(a, 5)
\end{array}
\]
Reasoning about fences

- Introduce two ‘modalities’ in the logic

\[
\{ P \} \text{fence}(\text{release}) \{ \bigtriangleup P \}
\]

\[
\{ W_Q(x) \ast \bigtriangleup Q(v) \} \ x.\text{store}(v, rlx) \{ W_Q(x) \}
\]

\[
\{ R_Q(x) \} \ t := x.\text{load}(rlx) \ \{ R_Q[t:=\text{emp}](x) \ast \bigtriangledown Q(t) \}
\]

\[
\{ \bigtriangledown P \} \text{fence}(\text{acq}) \{ P \}
\]
Reasoning about fences

Let \( Q(v) \triangleq v = 0 \lor &a \mapsto 5. \)

\[
\begin{align*}
\{ &a \mapsto 0 \star W_Q(x) \star R_Q(x) \} \\
\{ &a \mapsto 0 \star W_Q(x) \} \\
\{ &a \mapsto 5 \star W_Q(x) \} \\
fence(release); \\
\{ \triangle(&a \mapsto 5) \star W_Q(x) \} \\
x.\text{store}(1, rlx); \\
\{ \top \}
\end{align*}
\]

\[
\begin{align*}
t &= x.\text{load}(rlx); \\
\{ \nabla(t = 0 \lor &a \mapsto 5) \} \\
\text{if } (t \neq 0) \\
fence(acq); \\
\{ &a \mapsto 5 \} \\
\text{print}(a); \\
\{ \top \}
\end{align*}
\]
GPS: A better logic for release-acquire

Three key features:

▶ Location protocols

▶ Ghost state/tokens

▶ Escrows for ownership transfer

Example (Racy message passing)

Initially, $x = y = 0$.

$$
\begin{align*}
&x.\text{store}(1, rlx) ; \quad \parallel \quad x.\text{store}(1, rlx) ; \quad \parallel \quad t = y.\text{load}(acq);
&y.\text{store}(1, rel) ; \quad \parallel \quad y.\text{store}(1, rel) ; \quad \parallel \quad t' = x.\text{load}(rlx);
\end{align*}
$$

Cannot get $t = 1 \land t' = 0$. 
Racy message passing in GPS

Protocol for \( x \):

\[ A: x = 0 \quad \rightarrow \quad B: x = 1 \]

Protocol for \( y \):

\[ C: y = 0 \quad \rightarrow \quad D: y = 1 \land x.st \geq B \]

Acquire reads gain knowledge, not ownership.

\[
\begin{align*}
\{ x.st \geq A \land y.st \geq C \} & \quad \rightarrow \quad \{ x.st \geq A \land y.st \geq C \} \\
x.\text{store}(1, \text{rlx}); & \quad \rightarrow \quad \{ t = y.\text{load}(acq); \} \\
\{ x.st \geq B \land y.st \geq C \} & \quad \rightarrow \quad \{ t = 0 \land x.st \geq A \} \\
y.\text{store}(1, \text{rel}); & \quad \rightarrow \quad \{ \lor t = 1 \land x.st \geq B \} \\
\{ x.st \geq B \land y.st \geq D \} & \quad \rightarrow \quad \{ t' = x.\text{load}(rlx); \} \\
& \quad \rightarrow \quad \{ t = 0 \lor (t = 1 \land t' = 1) \}
\end{align*}
\]
GPS ghosts and escrows

To gain ownership, we use ghost state & escrows.

\[
P \ast P \Rightarrow false \quad Esc(P, Q) \Rightarrow Q \quad Esc(P, Q) \ast P \Rightarrow Q
\]

Example (Message passing using escrows)

Invariant for \(x\): \(x = 0 \lor Esc(K, &a \mapsto 7)\).

\[
\begin{align*}
\{&a \mapsto 0\} \\
a = 7; \\
\{&a \mapsto 7\} \\
\{Esc(K, \&a \mapsto 7)\} \\
x.\text{store}(1, \text{rel}); \\
\{K\} \\
\text{if} \ (x.\text{load}(acq) \neq 0) \\
\{K \ast Esc(K, &a \mapsto 7)\} \\
\{&a \mapsto 7\} \\
\text{print}(a);
\end{align*}
\]
Avoiding weak memory reasoning

- DRF theorem
- Enough fences to guarantee SC
Data race freedom

Theorem (DRF)

If $[Prg]_{SC}$ contains no data races on non-SC accesses, then $[Prg]_{C11} = [Prg]_{SC}$.

- Requires strengthened semantics for relaxed accesses.
- Program logics that disallow data races are trivially sound.
- What about racy programs?
C11’s SC-fences

- The strongest fence instruction provided by C11 is SC-fence.
- Can also be used to regain sequential consistency.

Example (Store Buffering)

\[
\begin{align*}
\text{\texttt{x = y = 0}} \\
\text{x.store(1, rlx); } & \text{y.store(1, rlx);} \\
\text{a = y.load(rlx); } & \text{b = x.load(rlx);}
\end{align*}
\]

\[
\begin{align*}
\text{[x = y = 0]} & \\
\text{W}_{rlx}(x, 1) & \text{W}_{rlx}(y, 1) \\
\text{R}_{rlx}(y, 0) & \text{R}_{rlx}(x, 0)
\end{align*}
\]
C11’s SC-fences

- The strongest fence instruction provided by C11 is \textit{SC}-fence.
- Can also be used to regain sequential consistency.

Example (Store Buffering)

\[
\begin{align*}
\text{x} &= \text{y} = 0 \\
\text{x}.\text{store}(1, \text{rlx}); &\parallel \text{y}.\text{store}(1, \text{rlx}); \\
\text{fence(sc)}; &\parallel \text{fence(sc)}; \\
\text{a}=\text{y}.\text{load}(\text{rlx}); &\parallel \text{b}=\text{x}.\text{load}(\text{rlx});
\end{align*}
\]

Inconsistent: \((\text{F}_{\text{sc}} \times \text{F}_{\text{sc}}) \cap (\text{po}^?; (\text{hb} \cup \text{rf} \cup \text{mo} \cup \text{rb}); \text{po}^?)\) is cyclic.
SC-fences are overly weak

Initially, $x = y = 0$.

$x$.store $(1, rlx)$;
$a = x$.load$(rlx)$; $c = y$.load$(rlx)$;
$b = y$.load$(rlx)$; $d = x$.load$(rlx)$;
$y$.store $(1, rlx)$;

The outcome $a = c = 1 \land b = d = 0$ is allowed.
SC-fences are overly weak

Initially, $x = y = 0$.

$$x \text{.store } (1, rlx); \quad \begin{align*}
a &= x \text{.load}(rlx); \\
fence(sc); \\
b &= y \text{.load}(rlx);
\end{align*}$$

$$y \text{.store } (1, rlx); \quad \begin{align*}
c &= y \text{.load}(rlx); \\
fence(sc); \\
d &= x \text{.load}(rlx);
\end{align*}$$

The outcome $a = c = 1 \land b = d = 0$ is allowed.

$$(F_{sc} \times F_{sc}) \cap (po?; (hb \cup rf \cup mo \cup rb); po?)$$ is acyclic.
Our suggestion

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

\[ x = y = f = 0 \]

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

$U_{acq-rel}(f) \rightarrow R_{rlx}(y, 0) \rightarrow R_{rlx}(x, 0)$

$rb_x \rightarrow \rightarrow rb_y$
Our suggestion

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

Inconsistent: $\text{R}_{\text{rlx}}(x, 0)$ reads an overwritten value.

\[
[x = y = f = 0] \\
W_{\text{rlx}}(x, 1) \rightarrow \text{R}_{\text{rlx}}(x, 1) \rightarrow \text{R}_{\text{rlx}}(y, 1) \rightarrow W_{\text{rlx}}(y, 1) \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\text{U}_{\text{acq-rel}}(f) \rightarrow \text{R}_{\text{rlx}}(y, 0) \rightarrow \text{R}_{\text{rlx}}(x, 0) \rightarrow \text{U}_{\text{acq-rel}}(f) \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
r_{bx} \rightarrow \text{R}_{\text{rlx}}(y, 0) \rightarrow \text{R}_{\text{rlx}}(x, 0) \\
r_{by} \rightarrow W_{\text{rlx}}(y, 1) \rightarrow W_{\text{rlx}}(x, 1)
Our suggestion

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

\[
\begin{align*}
[x = y = f = 0] & \\
W_{rlx}(x, 1) & \rightarrow R_{rlx}(x, 1) & R_{rlx}(y, 1) & \rightarrow W_{rlx}(y, 1) \\
U_{acq-rel}(f) & \rightarrow U_{acq-rel}(f) & R_{rlx}(y, 0) & \rightarrow R_{rlx}(x, 0) \\
\end{align*}
\]
Our suggestion

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

Inconsistent: $R_{rlx}(x, 0)$ reads an overwritten value.
Basic reduction theorem

**Theorem**

*If in a program $P$,*

- *all shared accesses are atomic (relaxed or stronger), and*
- *there is a fence between every two shared accesses to different shared variables,*

*then $P$ has only SC behaviors.*
Advanced reduction theorem

- For x86-TSO, it suffices to have a fence between every racy write and subsequent racy read.
- Generally, C11 requires more fences.

Theorem (Advanced reduction to SC, simplified)

If in a client-server program $P$,
- all shared accesses are release/acquire, and
- there is a fence between every store to a shared location and subsequent shared location load,

then $P$ has only SC behaviors.
Applying the theorem to RCU

\[
\begin{align*}
\text{rcu\_quiescent\_state}() : & \quad \text{rc}[\text{get\_my\_tid()}] := \text{gc}; \\
& \quad \text{fence()}
\end{align*}
\]

\[
\begin{align*}
\text{rcu\_thread\_offline}() : & \quad \text{rc}[\text{get\_my\_tid()}] := 0; \\
& \quad \text{fence()}
\end{align*}
\]

\[
\begin{align*}
\text{rcu\_thread\_online}() : & \quad \text{rc}[\text{get\_my\_tid()}] := \text{gc}; \\
& \quad \text{fence()}
\end{align*}
\]

\[
\begin{align*}
\text{synchronize\_rcu}() : & \quad \text{gc} := \text{gc} + 1; \\
& \quad \text{fence();} \\
& \quad \text{for } i := 1 \text{ to } N \text{ do} \\
& \quad \quad \text{wait (rc[i] } \in \{0, \text{gc}\})
\end{align*}
\]
Conclusion

Reasoning about weak memory is challenging and often unavoidable.

Two approaches:
- Use relaxed program logics to reason about weak memory.
- Use reduction theorems to avoid such reasoning.

Relaxed program logics also useful for understanding weak memory.