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

Sequential consistency (SC):

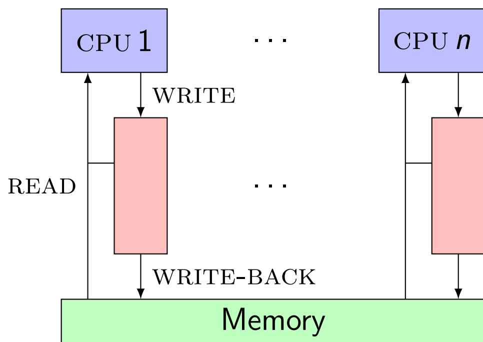
- The standard model for concurrency.
- Interleave each thread's atomic accesses.
- Almost all verification work assumes it.

Initially, $x = y = 0$.

$$\begin{array}{l} x := 1; \\ a := y \end{array} \;\Bigg\|\; \begin{array}{l} y := 1; \\ b := x \end{array}$$

In SC, this program cannot return $a = b = 0$.

Store buffering in x86-TSO



Initially, $x = y = 0$.

$$\begin{array}{l} x := 1; \\ a := y; \end{array} \quad \Bigg\| \quad \begin{array}{l} y := 1; \\ b := x; \end{array}$$

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

Owicki-Gries method (1976)

OG = Hoare logic + rule for parallel composition

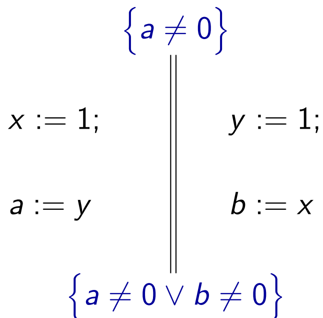
$$\frac{\{P_1\}\, c_1\, \{Q_1\} \qquad \{P_2\}\, c_2\, \{Q_2\} \qquad \text{the two proofs are } \textit{non-interfering}}{\{P_1 \wedge P_2\}\, c_1 \parallel c_2\, \{Q_1 \wedge Q_2\}}$$

Non-interference

$R \wedge P \vdash R\{u/x\}$ 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



Standard OG is unsound for WM

$$\{a \neq 0\}$$

$$\begin{array}{l|l} \{a \neq 0\} & \{\top\} \\ x := 1; & y := 1; \\ \{x \neq 0\} & \{y \neq 0\} \\ a := y & b := x \\ \{x \neq 0\} & \{y \neq 0 \wedge (a \neq 0 \vee b = x)\} \end{array}$$

$$\{a \neq 0 \vee b \neq 0\}$$

Standard OG is unsound for WM

$$\{a \neq 0\}$$

$$\begin{array}{l|l}
\{a \neq 0\} & \{\top\} \\
x := 1; & y := 1; \\
\{x \neq 0\} & \{y \neq 0\} \\
a := y & b := x \\
\{x \neq 0\} & \{y \neq 0 \wedge (a \neq 0 \vee b = x)\}
\end{array}$$

$$\{a \neq 0 \vee b \neq 0\}$$

To regain soundness, strengthen the non-inference check.
$\Longrightarrow$ 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

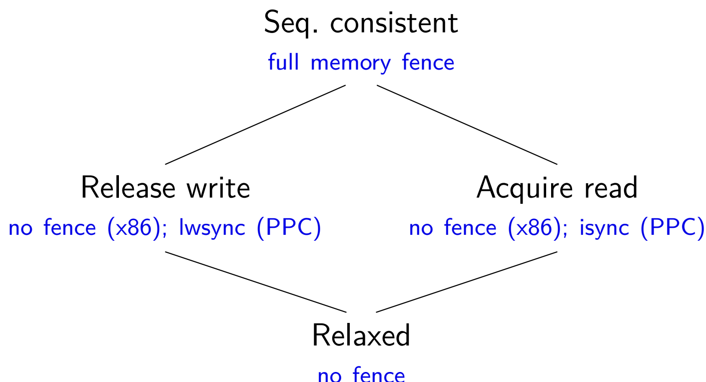
The C11 memory model: Atomics

Two types of locations

Ordinary (Non-Atomic)	**Atomic**
Races are errors	Welcome to the expert mode

The C11 memory model: Hierarchy of atomics

A spectrum of atomic accesses:



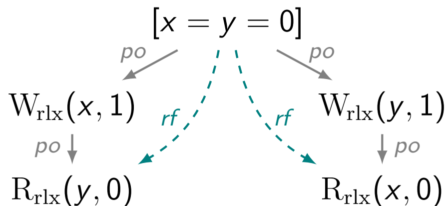
Explicit primitives for fences

Store buffering in C11

Initially $x = y = 0$.

$$\begin{array}{l||l} x.\text{store}(1, \mathit{rlx}); & y.\text{store}(1, \mathit{rlx}); \\ \mathit{print}(y.\text{load}(\mathit{rlx})); & \mathit{print}(x.\text{load}(\mathit{rlx})); \end{array}$$

Can print 00 with the following execution:

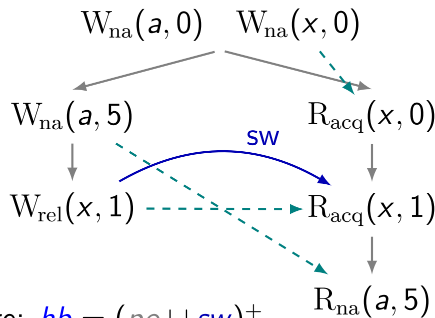


Release-acquire synchronization

Initially $a = x = 0$.

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

One possible execution:



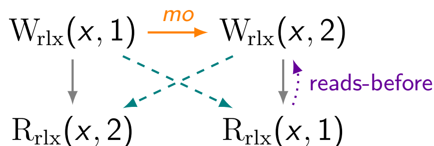
Happens before: $hb = (po \cup sw)^+$

Coherence

Programs with a single shared variable behave as under SC.

$$\begin{array}{l|l} x.\text{store}(1, rlx); & x.\text{store}(2, rlx); \\ a = x.\text{load}(rlx); & b = x.\text{load}(rlx); \end{array}$$

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



- 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:

$$\{x \mapsto v\}\ {*x}\ \{t.\ t = v \wedge x \mapsto v\}$$
$$\{x \mapsto v\}\ {*x} = v';\ \{x \mapsto v'\}$$

- Disjoint parallelism:

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}$$

Rules for release/acquire accesses

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 $J(v) \triangleq v = 0 \vee \&a \mapsto 5$.

$\{\&a \mapsto 0 * \mathbf{W}_J(x)\}$	$\{\mathbf{R}_J(x)\}$
$a = 5;$	**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:

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t := x.\mathrm{load}(\mathit{rlx})\ \{\mathbf{R}_{\mathcal{Q}}(x) \land (\mathcal{Q}(t) \not\equiv \mathit{false})\}$$

- Relaxed writes:

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

Unsound because of dependency cycles!

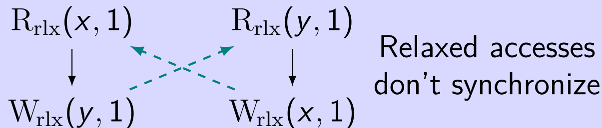
Dependency cycles

Initially $x = y = 0$.

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

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

Justification:



Relaxed accesses don't synchronize

Dependency cycles

Initially $x = y = 0$.

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

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 $x = y = 0$.

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

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

How to fix this:

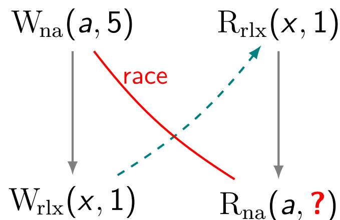
Don't use relaxed writes

Strengthen the model

Incorrect message passing

$$\textbf{int } a; \ \textbf{atomic_int } x = 0;$$

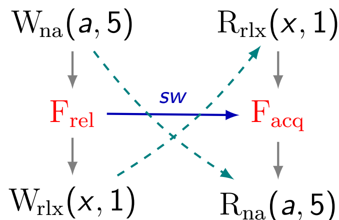
$$\left(\begin{array}{l||c} a = 5; & \textbf{if } (x.\mathrm{load}(rlx) \neq 0)\{ \\ & \\ x.\mathrm{store}(1, rlx); & \textbf{print}(a); \} \end{array}\right)$$



Message passing with C11 memory fences

$$\textbf{int}\ a;\ \ \textbf{atomic_int}\ x = 0;$$

$$\left(\begin{array}{l} a = 5; \\ \text{fence}(\mathit{release}); \\ x.\text{store}(1, \mathit{rlx}); \end{array} \middle\| \begin{array}{l} \textbf{if}\ (x.\text{load}(\mathit{rlx}) \neq 0)\{ \\ \qquad \text{fence}(\mathit{acq}); \\ \qquad \textbf{print}(a); \} \end{array}\right)$$



Reasoning about fences

- Introduce two ‘modalities’ in the logic

$$\{P\}\ \text{fence}(\mathit{release})\ \{\triangle P\}$$

$$\{\mathbf{W}_{\mathcal{Q}}(x) * \triangle \mathcal{Q}(v)\}\ x.\text{store}(v, \mathit{rlx})\ \{\mathbf{W}_{\mathcal{Q}}(x)\}$$

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t := x.\text{load}(\mathit{rlx})\ \{\mathbf{R}_{\mathcal{Q}[t:=\text{emp}]}(x) * \triangledown \mathcal{Q}(t)\}$$

$$\{\triangledown P\}\ \text{fence}(\mathit{acq})\ \{P\}$$

Reasoning about fences

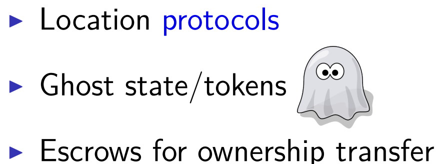
Let $\mathcal{Q}(v) \triangleq v = 0 \vee \&a \mapsto 5$.

$$\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x) * \mathbf{R}_{\mathcal{Q}}(x)\}$$

$$\left(\begin{array}{l|l}
\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x)\} & t = x.\mathrm{load}(\mathit{rlx}); \\
a = 5; & \{\triangledown(t = 0 \vee \&a \mapsto 5)\} \\
\{\&a \mapsto 5 * \mathbf{W}_{\mathcal{Q}}(x)\} & \mathbf{if}\ (t \neq 0) \\
\mathrm{fence}(\mathit{release}); & \quad \mathrm{fence}(\mathit{acq}); \\
\{\triangle(\&a \mapsto 5) * \mathbf{W}_{\mathcal{Q}}(x)\} & \quad \{\&a \mapsto 5\} \\
x.\mathrm{store}(1, \mathit{rlx}); & \quad \mathbf{print}(a);\ \} \\
\{\top\} & \{\top\}
\end{array}\right)$$

GPS: A better logic for release-acquire

Three key features:

- Location protocols
- Ghost state/tokens 
- Escrows for ownership transfer

Example (Racy message passing)

Initially, $x = y = 0$.

$$\begin{array}{l} x.\mathrm{store}(1, rlx); \\ y.\mathrm{store}(1, rel); \end{array} \left\| \begin{array}{l} x.\mathrm{store}(1, rlx); \\ y.\mathrm{store}(1, rel); \end{array} \right\| \begin{array}{l} t = y.\mathrm{load}(acq); \\ t' = x.\mathrm{load}(rlx); \end{array}$$

Cannot get $t = 1 \wedge t' = 0$.

Racy message passing in GPS

Protocol for x: **A:** $x = 0$ → **B:** $x = 1$

Protocol for y: **C:** $y = 0$ → **D:** $y = 1 \land x.st \geq \mathbf{B}$

Acquire reads gain knowledge, not ownership.

$\{x.st \geq \mathbf{A} \land y.st \geq \mathbf{C}\}$ $x.\text{store}(1, rlx);$ $\{x.st \geq \mathbf{B} \land y.st \geq \mathbf{C}\}$ $y.\text{store}(1, rel);$ $\{x.st \geq \mathbf{B} \land y.st \geq \mathbf{D}\}$	$\{x.st \geq \mathbf{A} \land y.st \geq \mathbf{C}\}$ $t = y.\text{load}(acq);$ $\left\{\begin{array}{l} t = 0 \land x.st \geq \mathbf{A} \\ \lor\ t = 1 \land x.st \geq \mathbf{B} \end{array}\right\}$ $t' = x.\text{load}(rlx);$ $\{t = 0 \lor (t = 1 \land t' = 1)\}$

GPS ghosts and escrows

To gain ownership, we use ghost state & escrows.

$$\frac{P * P \Rightarrow \mathit{false}}{Q \Rrightarrow \mathbf{Esc}(P, Q)} \qquad \frac{}{\mathbf{Esc}(P, Q) * P \Rrightarrow Q}$$

Example (Message passing using escrows)

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

Thread 1	Thread 2
$\{\&a \mapsto 0\}$	$\{K\}$
$a = 7;$	**if** $(x.\mathrm{load}(acq) \neq 0)$
$\{\&a \mapsto 7\}$	$\quad \{K * \mathbf{Esc}(K, \&a \mapsto 7)\}$
$\{\mathbf{Esc}(K, \&a \mapsto 7)\}$	$\quad \{\&a \mapsto 7\}$
$x.\mathrm{store}(1, rel);$	$\quad$ **print**$(a);$

Avoiding weak memory reasoning

- DRF theorem
- Enough fences to guarantee SC

Data race freedom

Theorem (DRF)

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

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

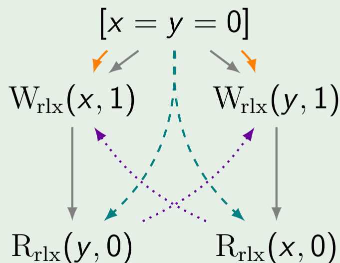
C11's SC-fences

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

Example (Store Buffering)

$$x = y = 0$$

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



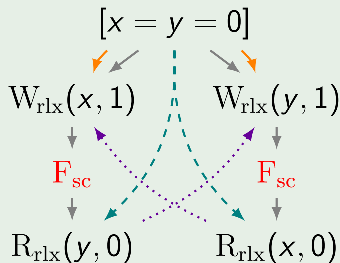
C11's SC-fences

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

Example (Store Buffering)

$$x = y = 0$$

x.store(1, *rlx*);	y.store(1, *rlx*);
`fence(sc);`	`fence(sc);`
a=y.load(*rlx*);	b=x.load(*rlx*);



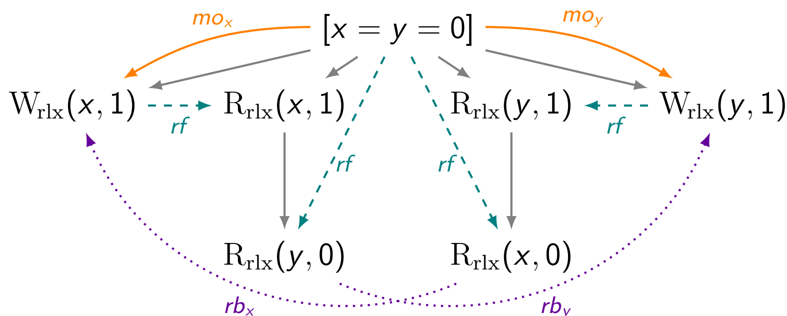
Inconsistent: $(\mathrm{F_{sc}} \times \mathrm{F_{sc}}) \cap (po^?; (hb \cup rf \cup mo \cup rb); po^?)$ is cyclic.

SC-fences are overly weak

Initially, $x = y = 0$.

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

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

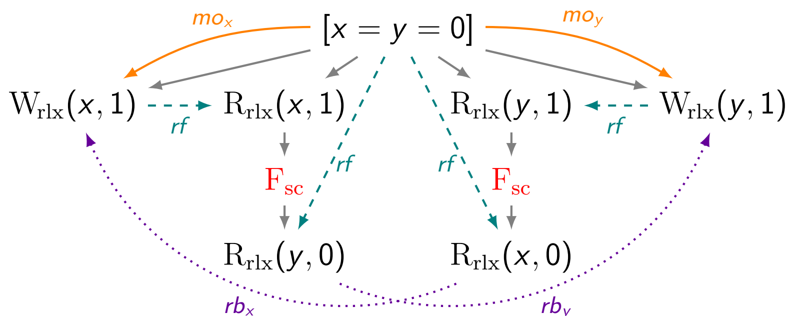


SC-fences are overly weak

Initially, $x = y = 0$.

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

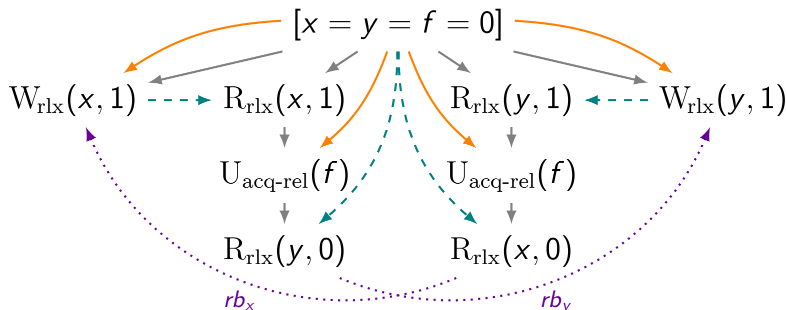
The outcome $a = c = 1 \wedge b = d = 0$ is allowed.



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

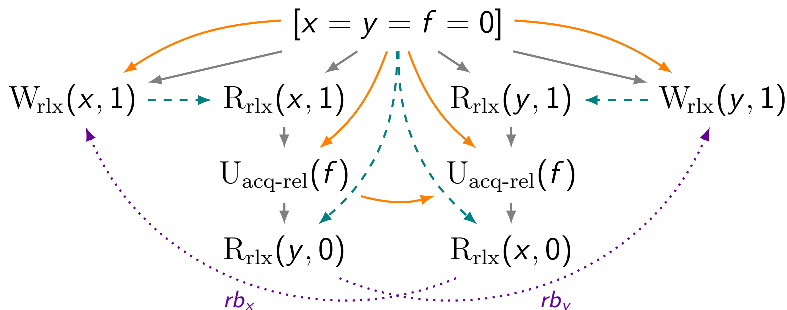
Our suggestion

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



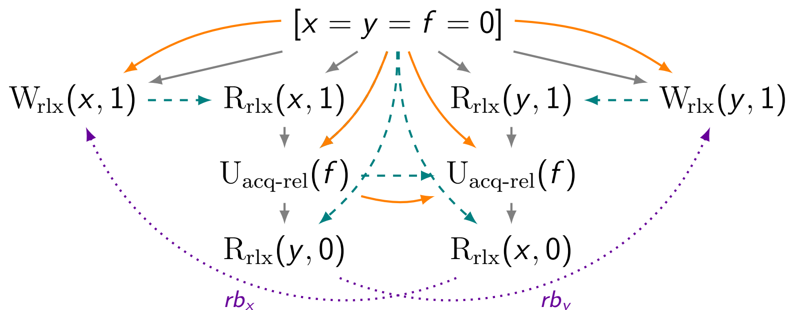
Our suggestion

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



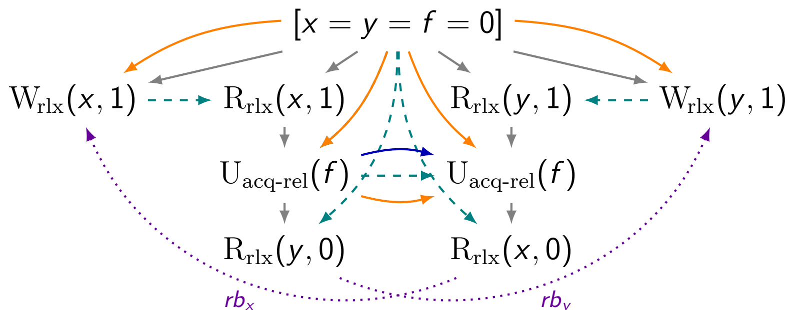
Our suggestion

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



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.

$x := e;$
```
fence();
```
$r := y$

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

```
rcu_quiescent_state() :
  rc[get_my_tid()] := gc;
  fence()

rcu_thread_offline() :
  rc[get_my_tid()] := 0;
  fence()

rcu_thread_online() :
  rc[get_my_tid()] := gc;
  fence()
```

```
synchronize_rcu() :
  gc := gc + 1;
  fence();
  for i := 1 to N do
    wait (rc[i] ∈ {0, gc})
```

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.