A case for relaxed program logics

Viktor Vafeiadis

MPI-SWS

10 July 2014

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 better tools... *Relaxed program logics*

Which memory model?

Hardware or language models?

- Want to reason at "high level"
- TSO $\rightsquigarrow$ good robustness theorems

C/C++ or Java?

- JMM is broken [Sevcik et al.]
- So, only C/C++11 left

Goals:

- Understand the memory model
- Verify intricate concurrent programs

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

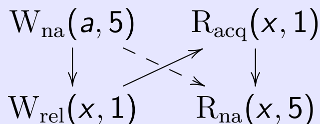
Release-acquire synchronization: message passing

Initially $a = x = 0$.

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

This will always print 5.

Justification:



Release-acquire synchronization

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

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

PL consequences:

Ownership transfer works!

Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

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

- Relaxed writes:

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

Unsound because of dependency cycles!

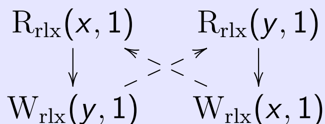
Dependency cycles

Initially $x = y = 0$.

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

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

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.

$$x = 0 \wedge y = 0$$

The DRF-property does not hold.

Dependency cycles

Initially $x = y = 0$.

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

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

How to fix this:

Don't use relaxed writes

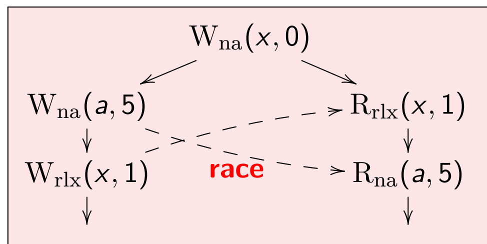
$\vee$

Strengthen the model

Incorrect message passing

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

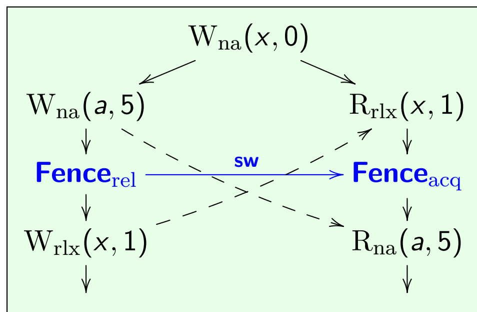
$$\left(\begin{array}{l} a = 5; \\ x.\mathrm{store}(1, rlx); \end{array} \middle\| \begin{array}{c} \textbf{if } (x.\mathrm{load}(rlx) \neq 0)\{ \\ \textbf{print}(a);\ \} \end{array}\right)$$



Message passing with C11 memory fences

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

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



Reasoning about fences

- Introduce two 'modalities' in the logic

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

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

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

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

Reasoning about fences

Let $\mathcal{Q}(v) \stackrel{\text{def}}{=} v = 0 \lor \&a \mapsto 5$.

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

$$\left(\begin{array}{l||l}
\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x)\} & t = x.\mathsf{load}(\mathit{rlx}); \\
a = 5; & \{\nabla(t = 0 \lor \&a \mapsto 5)\} \\
\{\&a \mapsto 5 * \mathbf{W}_{\mathcal{Q}}(x)\} & \textbf{if } (t \neq 0) \\
\mathsf{fence}(\mathit{release}); & \qquad \mathsf{fence}(\mathit{acq}); \\
\{\triangle(\&a \mapsto 5) * \mathbf{W}_{\mathcal{Q}}(x)\} & \qquad \{\&a \mapsto 5\} \\
x.\mathsf{store}(1, \mathit{rlx}); & \qquad \textbf{print}(a); \ \} \\
\{\mathsf{true}\} & \{\mathsf{true}\}
\end{array}\right)$$

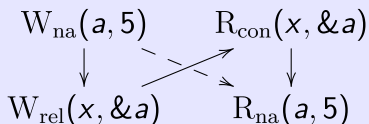
Release-consume synchronization

Initially $a = x = 0$.

$$a = 5; \quad \Big\| \quad t = x.load(consume);$$
$$x.store(release, \&a); \quad \Big\| \quad \textbf{if } (t \neq 0)\ print(*t);$$

This program cannot crash nor print 0.

Justification:



Release-consume synchronization

Release-consume synchronization

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

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

This program cannot crash nor print 0.

PL consequences:

Needs funny modality, but otherwise OK.

Proposed rules for consume accesses

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

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

Release-acquire too weak in the presence of consume

Initially $x = y = 0$.

$a = 1;$ $x.store(1, release);$	‖	$\textbf{while}\ (x.load(consume) \neq 1);$ $y.store(1, release);$ $(*)$ $\textbf{while}\ (y.load(acquire) \neq 1);$ $(*)$ $a = 2;$

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 $x = y = 0$.

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

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.

Release sequences too strong (relaxed writes)

Initially $x = y = 0$.

$a = 1;$ $x.\text{store}(1, \mathit{release});$ $x.\text{store}(3, \mathit{relaxed});$	$\mathit{while}(x.\text{load}(\mathit{acquire}) \neq 3);$ $a = 2;$

This program is not racy.

The acquire synchronizes with the release.

Release sequences too strong (relaxed writes)

Initially $x = y = 0$.

$a = 1;$	$x.\text{store}(2, \mathit{relaxed});$
$x.\text{store}(1, \mathit{release});$	$\mathit{while}(x.\text{load}(\mathit{acquire}) \neq 3);$
$x.\text{store}(3, \mathit{relaxed});$	$a = 2;$

But this one is racy according to C11.

The acquire no longer synchronizes with the release.

Conclusion

Relaxed program logics
are a tool for
understanding
weak memory models