Relaxed Separation Logic

Tutorial @ POPL'14

Viktor Vafeiadis

MPI-SWS

20 January 2014

Tutorial outline

Part I. Weak memory models

1. Introduction to relaxed concurrency
2. The C11 relaxed memory model

Part II. Relaxed program logics

3. Concurrent separation logic
4. Relaxed separation logic
5. RSL extensions (ongoing)

http://www.mpi-sws.org/~viktor/rsl/

Sequential consistency

Sequential consistency (SC):

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

Initially, $x = y = 0$.

$$\begin{array}{l} x := 1; \\ print(y); \end{array} \;\Big\|\; \begin{array}{l} y := 1; \\ print(x); \end{array}$$

In SC, this program can print 01, 10, or 11.

Sequential consistency (SC):

- Interleave each thread's atomic accesses.
- The standard model for concurrency.
- A
- F

Initia

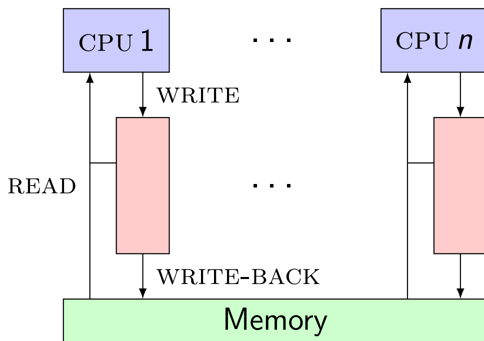
But SC is invalidated by:

- Hardware implementations
- Compiler optimisations

$$\begin{array}{l} x := 1; \\ print(y); \end{array} \quad \Bigg\| \quad \begin{array}{l} y := 1; \\ print(x); \end{array}$$

In SC, this program can print 01, 10, or 11.

Store buffering in x86-TSO



Initially, $x = y = 0$.

$$\begin{array}{l} x := 1; \\ print(y); \end{array} \;\Bigg\|\; \begin{array}{l} y := 1; \\ print(x); \end{array}$$

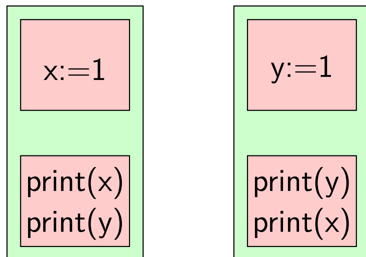
This program can also print 00.

IRIW: Not just reordering

Initially, $x = y = 0$.

$$x := 1 \;\Big\|\; y := 1 \;\Big\|\; \begin{array}{l} print(x); \\ print(y); \end{array} \;\Big\|\; \begin{array}{l} print(y); \\ print(x); \end{array}$$

Both threads can print 10.



Basic compiler optimisations break SC

Initially, $x = y = 0$.

$$x := 1;\; y := 1; \;\Big\|\; print(x);\; print(y);\; print(x);$$

The program can print 010.

Justification:

The compiler may perform CSE:

Load x into a temporary t

and print t, y, and t.

When should we care about relaxed memory?

All *sane* memory models satisfy the DRF property:

Theorem (DRF-property)

If $[\![\mathit{Prg}]\!]_{\mathrm{SC}}$ *contains no data races, then* $[\![\mathit{Prg}]\!]_{\mathrm{Relaxed}} = [\![\mathit{Prg}]\!]_{\mathrm{SC}}$.

- Program logics that disallow data races are trivially sound.
- What about *racy* 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

Primitives for explicit fences

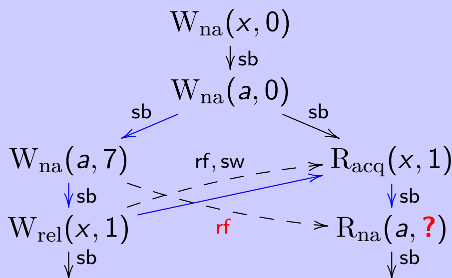
C11 executions

- Execution = set of events & a few relations:
  - sb: sequenced before
  - rf: reads-from map
  - mo: memory order per location
  - sc: seq.consistency order
  - sw [derived]: synchronized with
  - hb [derived]: happens before
- Axioms constraining the *consistent* executions.
- $\mathcal{C}(\!|\mathit{prog}|\!)$ = set of all consistent exec's.
- if all $\mathcal{C}(\!|\mathit{prog}|\!)$ race-free on ordinary accesses, $[\![\mathit{prog}]\!] = \mathcal{C}(\!|\mathit{prog}|\!)$; otherwise, $[\![\mathit{prog}]\!] =$"error"

Release-acquire synchronization: message passing in C11

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

$$\left(\begin{array}{l|l} a = 7; & \textbf{if } (x.\mathrm{load}(acq) \neq 0) \\ x.\mathrm{store}(1, release); & \quad \textbf{print}(a); \end{array}\right)$$



happens-before $\stackrel{\text{def}}{=}$ (sequenced-before $\cup$ sync-with)$^{+}$

sync-with$(a, b) \stackrel{\text{def}}{=}$ reads-from$(b) = a \wedge$ release$(a) \wedge$ acquire(b)

Rel-acq synchronization is weaker than SC

Example (SB)

Initially, $x = y = 0$.

$$\begin{array}{l} x.store(1, release); \\ t = y.load(acquire); \end{array} \;\Big\|\; \begin{array}{l} y.store(1, release); \\ t' = x.load(acquire); \end{array}$$

This program may produce $t = t' = 0$.

Example (IRIW)

Initially, $x = y = 0$.

$$\begin{array}{l} x.store \\ \quad (1, rel); \end{array} \Big\| \begin{array}{l} y.store \\ \quad (1, rel); \end{array} \Big\| \begin{array}{l} a{=}x.load(acq); \\ b{=}y.load(acq); \end{array} \Big\| \begin{array}{l} c{=}y.load(acq); \\ d{=}x.load(acq); \end{array}$$

May produce $a = c = 1 \wedge b = d = 0$.

Coherence

Example (Read-Read Coherence)

Initially, $x = 0$.

$$x.store(1, rel); \parallel x.store(2, rel); \parallel \begin{array}{l} a = x.load(acq); \\ b = x.load(acq); \end{array} \parallel \begin{array}{l} c = x.load(acq); \\ d = x.load(acq); \end{array}$$

Cannot get $a = d = 1 \wedge b = c = 2$.

- Plus similar WR, RW, WW coherence properties.
- Ensure SC behaviour for a single variable.
- Also guaranteed for relaxed atomics
  (the weakest kind of atomics in C11).

Part II

Relaxed Program Logics

- Concurrent separation logic
- Relaxed separation logic
- RSL extensions (ongoing)

Key concept of *ownership* :

- Resourceful reading of Hoare triples.

$$\{P\}\ C\ \{Q\}$$

- To access a normal 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:

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

Relaxed separation logic (simplified)

Joint work with Chinmay Narayan, published at OOPSLA'13

Ownership transfer by rel-acq synchronizations.

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

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

- Release write $\leadsto$ give away permissions.

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

- Acquire read $\leadsto$ gain permissions.

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ t = x.load(acq);\ \{\mathcal{Q}(t)\}$$

Let $\mathcal{Q}(v) \stackrel{\text{def}}{=} v = 0 \lor \&a \mapsto 7$.

$$
\begin{array}{c}
\{\text{true}\} \\
\textbf{atomic_int } x = 0;\ \textbf{int } a = 0; \\
\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x) * \mathbf{R}_{\mathcal{Q}}(x)\} \\
\left(\begin{array}{l||l}
\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x)\} & t = x.\text{load}(acq); \\
a = 7; & \{t = 0 \lor \&a \mapsto 7)\} \\
\{\&a \mapsto 7 * \mathbf{W}_{\mathcal{Q}}(x)\} & \textbf{if } (t \neq 0)\{\&a \mapsto 7\} \\
x.\text{store}(1, release); & \qquad \textbf{print}(a); \\
\{\text{true}\} & \qquad \{\text{true}\}
\end{array}\right) \\
\{\text{true}\}
\end{array}
$$

Multiple readers/writers

Write permissions can be duplicated:

$$\mathbf{W}_{\mathcal{Q}}(\ell) \iff \mathbf{W}_{\mathcal{Q}}(\ell) * \mathbf{W}_{\mathcal{Q}}(\ell)$$

Read permissions cannot, but may be split:

$$\mathbf{R}_{\mathcal{Q}_1 * \mathcal{Q}_2}(\ell) \iff \mathbf{R}_{\mathcal{Q}_1}(\ell) * \mathbf{R}_{\mathcal{Q}_2}(\ell)$$

$$\begin{array}{l||l||l} a = 7; & t = x.\mathrm{load}(acq); & t' = x.\mathrm{load}(acq); \\ b = 8; & \textbf{if } (t \neq 0) & \textbf{if } (t' \neq 0) \\ x.\mathrm{store}(1, rel); & \quad \textbf{print}(a); & \quad \textbf{print}(b); \end{array}$$

Basically, disallow ownership transfer.

- Relaxed reads:

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ x.\mathit{load}(\mathit{rlx})\ \{y.\ \mathcal{Q}(y) \not\equiv \mathsf{false}\}$$

- Relaxed writes:

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

Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ x.load(rlx)\ \{y.\ \mathcal{Q}(y) \not\equiv \mathsf{false}\}$$

- Relaxed writes:

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

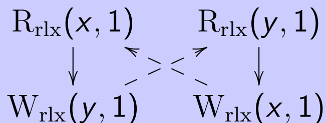
Unfortunately *not sound* because of a bug in the C11 memory model.

Initially $x = y = 0$.

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

The formal C11 model allows $x = y = 1$.

Justification:



Relaxed accesses don't synchronize

Dependency cycles in C11

Initially $x = y = 0$.

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

The formal C11 model allows $x = y = 1$.

What goes wrong:

Non-relational invariants are unsound.

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

The DRF-property does not hold.

Dependency cycles in C11

Initially $x = y = 0$.

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

The formal C11 model allows $x = y = 1$.

How to fix this:

Don't use relaxed writes

$\vee$

Require $acyclic(sb \cup rf)$.
(Disallow RW reodering.)

Compare and swap (CAS)

- New assertion form, $P := \ldots \mid \mathbf{R}^{\mathbf{U}}_{\mathcal{Q}}(x)$.
- Duplicable, $\mathbf{R}^{\mathbf{U}}_{\mathcal{Q}}(x) \iff \mathbf{R}^{\mathbf{U}}_{\mathcal{Q}}(x) * \mathbf{R}^{\mathbf{U}}_{\mathcal{Q}}(x)$.

$$\frac{\begin{array}{c} X \in \{rel, rlx\} \Rightarrow \mathcal{Q}(v) \equiv \mathsf{emp} \\ X \in \{acq, rlx\} \Rightarrow \mathcal{Q}(v') \equiv \mathsf{emp} \\ P * \mathcal{Q}(v) \Rightarrow \mathcal{Q}(v') * R[v/z] \\ \{P\}\ x.\mathsf{load}(Y)\ \{z.\ z \neq v \Rightarrow R\} \end{array}}{\{\mathbf{R}^{\mathbf{U}}_{\mathcal{Q}}(x) * \mathbf{W}_{\mathcal{Q}}(x) * P\}\ x.\mathsf{CAS}(v, v', X, Y)\ \{z.\ R\}}$$

$$\textbf{Let}\quad \mathcal{Q}_J(v) \stackrel{\text{def}}{=} (v = 0 \land \text{emp}) \lor (v = 1 \land J)$$
$$\mathit{Lock}(x, J) \stackrel{\text{def}}{=} \mathbf{W}_{\mathcal{Q}_J}(x) * \mathbf{R}^{\mathbf{U}}_{\mathcal{Q}_J}(x)$$

$\mathit{new\text{-}lock}() \stackrel{\text{def}}{=}$
- $\{J\}$
- $\mathit{res} = \textbf{alloc}(1)$
- $\{\mathit{Lock}(\mathit{res}, J)\}$

$\mathit{unlock}(x) \stackrel{\text{def}}{=}$
- $\{J * \mathit{Lock}(x, J)\}$
- $x.\text{store}(1, \mathit{rel})$
- $\{\mathit{Lock}(x, J)\}$

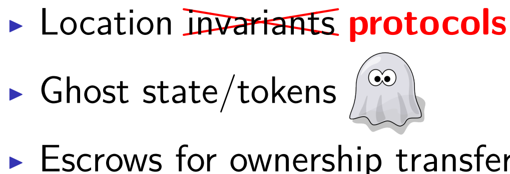
$\mathit{lock}(x) \stackrel{\text{def}}{=}$
- $\{\mathit{Lock}(x, J)\}$
- **repeat**
  - $\{\mathit{Lock}(x, J)\}$
  - $y = x.\text{CAS}(1, 0, \mathit{acq}, \mathit{rlx})$
  - $\left\{\mathit{Lock}(x, J) * \begin{pmatrix} y{=}0 \land \text{emp} \\ \lor\ y{=}1 \land J \end{pmatrix}\right\}$
- **until** $y \neq 0$
- $\{J * \mathit{Lock}(x, J)\}$

GPS: A better logic for release-acquire

Joint work with Aaron Turon and Derek Dreyer.

Three key features:

- Location ~~invariants~~ **protocols**
- Ghost state/tokens
- Escrows for ownership transfer



Example (Racy message passing)

Initially, $x = y = 0$.

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

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

Racy message passing in GPS

Protocol for x: $\textbf{A: } x = 0$ → $\textbf{B: } x = 1$

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

Acquire reads gain knowledge, not ownership.

$$\{x.st \geq \textbf{A} \land y.st \geq \textbf{C}\}$$
$$x.\text{store}(1, \mathit{rlx});$$
$$\{x.st \geq \textbf{B} \land y.st \geq \textbf{C}\}$$
$$y.\text{store}(1, \mathit{rel});$$
$$\{x.st \geq \textbf{B} \land y.st \geq \textbf{D}\}$$

∥

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

To gain ownership, we use ghost state & escrows.

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

Example (Message passing using escrows)

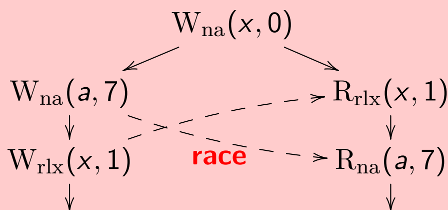
Invariant for x: $x = 0 \lor \textbf{Esc}(K, \&a \mapsto 7)$.

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

Incorrect message passing

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

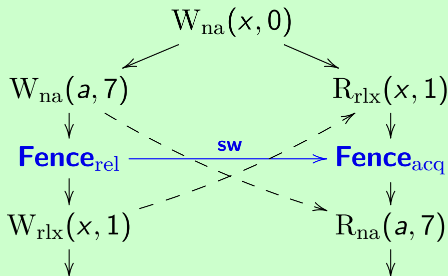
$$\left(\begin{array}{l} a = 7; \\ x.\mathsf{store}(1, rlx); \end{array} \middle\| \begin{array}{l} \textbf{if } (x.\mathsf{load}(rlx) \neq 0)\{ \\ \qquad \textbf{print}(a); \} \end{array}\right)$$



Message passing with C11 memory fences

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

$$\left(\begin{array}{l} a = 7; \\ \text{fence}(\mathit{release}); \\ x.\text{store}(1, \mathit{rlx}); \end{array} \middle\| \begin{array}{l} \textbf{if}\ (x.\text{load}(\mathit{rlx}) \neq 0)\{ \\ \qquad \text{fence}(\mathit{acq}); \\ \qquad \textbf{print}(a);\ \} \end{array}\right)$$



Reasoning about fences

Joint work with Marko Doko (in progress)

Let $\mathcal{Q}(v) \stackrel{\text{def}}{=} v = 0 \lor \&a \mapsto 7$.

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

Thread 1	Thread 2
$\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x)\}$	$t = x.\text{load}(rlx);$
$a = 7;$	$\{\nabla(t = 0 \lor \&a \mapsto 7)\}$
$\{\&a \mapsto 7 * \mathbf{W}_{\mathcal{Q}}(x)\}$	**if** $(t \neq 0)$
fence(*release*)	fence(*acq*);
$\{\triangle(\&a \mapsto 7) * \mathbf{W}_{\mathcal{Q}}(x)\}$	$\{\&a \mapsto 7\}$
$x.\text{store}(1, rlx);$	**print**(a); }
$\{\text{true}\}$	$\{\text{true}\}$

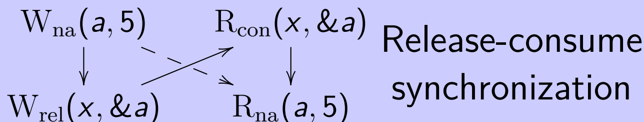
$$\{P\}\ \text{fence}(release)\ \{\triangle P\}$$
$$\{\nabla P\}\ \text{fence}(acq)\ \{P\}$$
$$\{\mathbf{R}_{\mathcal{Q}}(x)\}\ x.\text{load}(rlx)\ \{y.\ \nabla \mathcal{Q}(y)\}$$
$$\{\mathbf{W}_{\mathcal{Q}}(x) * \triangle \mathcal{Q}(v)\}\ x.\text{store}(v, rlx)\ \{\text{true}\}$$

Initially $a = x = 0$.

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

This program cannot crash nor print 0.

Justification:



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

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

This program cannot crash nor print 0.

Index the ∇ with program variable t.
t data dependence $\Longrightarrow$ locally open ∇_t.

Again, work in progress...

Conclusion

Take away:

Formal reasoning about C11 programs
is not so difficult after all.

We're not quite there yet; there's still a lot to do:

Liveness, refinement, tool support, . . .

Topics that were not covered:

- The soundness proof:

  Really interesting and fully mechanized in Coq.

- Found 4+1 bugs in the C11 model.

  Program logic as a debugging tool for WMM.