Program logics for relaxed consistency

UPMARC Summer School 2014

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

1st Lecture, 28 July 2014

Outline

Part I. Weak memory models

1. Intro to relaxed memory consistency
2. The C11 memory model

Part II. Program logics

3. Separation logic
4. Relaxed separation logic
5. GPS : ghosts & protocols
6. Advanced features

`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, $x = y = 0$.

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

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

Sequential consistency

Sequential consistency (SC):

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

Initia

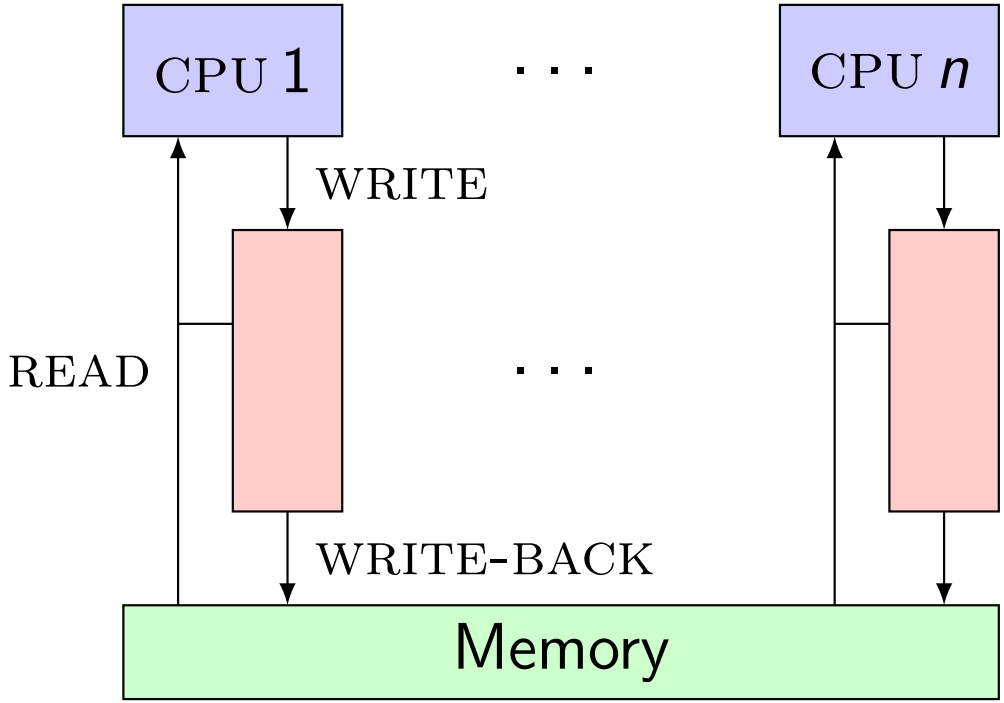
But SC is invalidated by:

- Hardware implementations
- Compiler optimisations

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

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

Store buffering in x86-TSO



Initially, $x = y = 0$.

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

This program can also print 00.

Basic compiler optimisations break SC / TSO

Initially, $x = y = 0$.

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

Can the program print 010?

Justification:

The compiler may perform CSE:

Load x into a temporary t

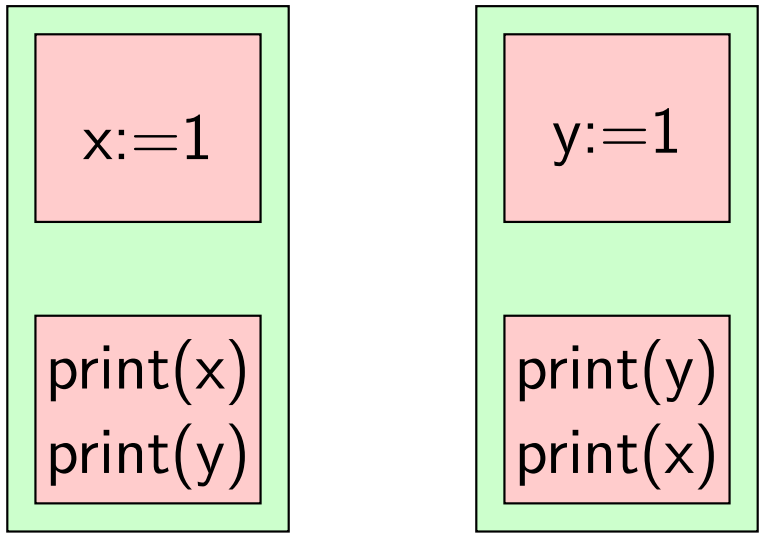
and print t, y, and t.

IRIW: Not just store buffering

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.



From IRIW to the store buffering example

Take the IRIW example:

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

Linearize twice (threads 1-3 and 2-4):

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

That's the store buffering example (with two extra print statements).

Coherence

Initially, $x = 0$.

$$\begin{array}{l} x = 1; \\ x = 2; \end{array} \Bigg\| \begin{array}{l} print(x); \\ print(x); \end{array}$$

Cannot print 10 nor 20 nor 21.

- Programs with one shared variable have SC semantics.
- Ensured by the cache coherence protocol.

Message passing

Initially, $x = y = 0$.

$$\begin{array}{l|l} x = 1; & \mathit{print}(y); \\ [\text{WW fence}] & [\text{RR fence}] \\ y = 1; & \mathit{print}(x); \end{array}$$

Cannot print 10.

- No fences needed on x86-TSO
- lwsync/isync on Power
- dmb/isync on ARM

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...

Which memory model?

Hardware or language models?

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

C/C++ or Java?

- JMM is broken [Ševčík & Aspinall, ECOOP'08]
- So, only C11 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

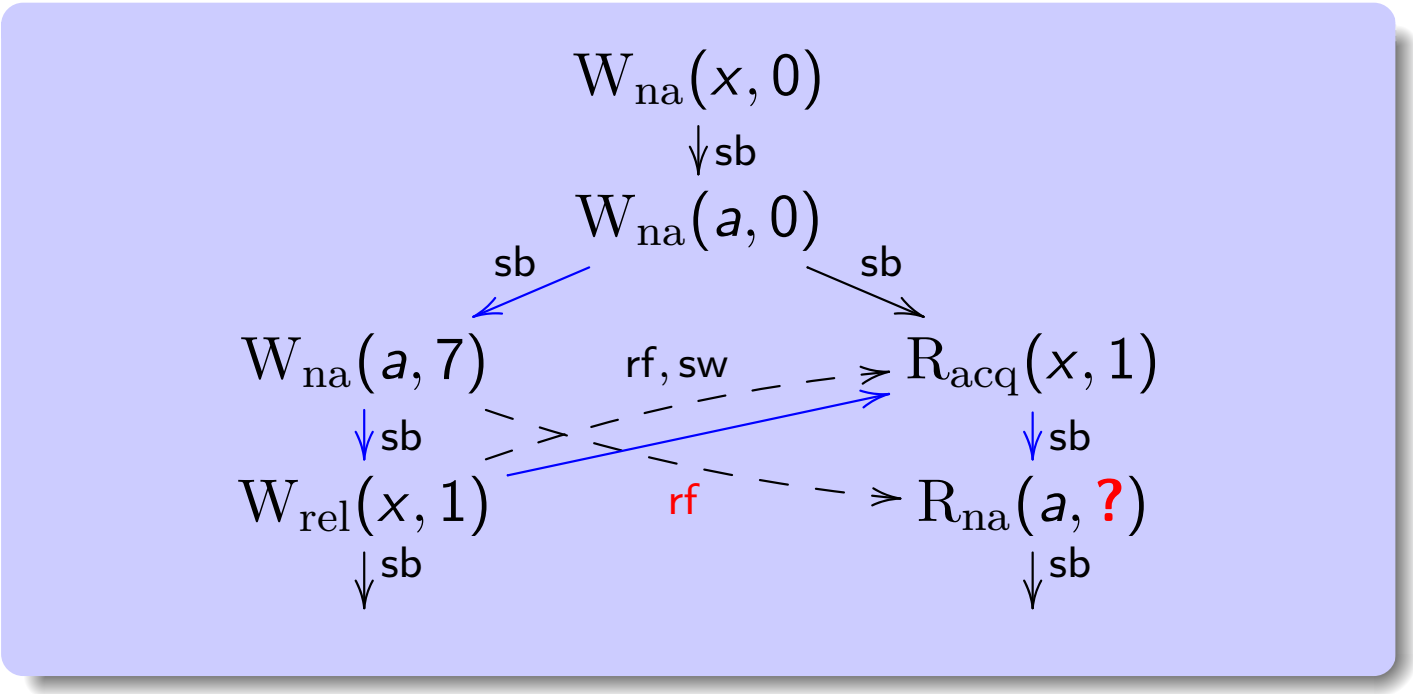
Primitives for explicit fences

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.
- $\mathcal{C}(\lVert prog \rVert)$ = set of all consistent exec's.
- if all $\mathcal{C}(\lVert prog \rVert)$ race-free on ordinary accesses, $[\![prog]\!] = \mathcal{C}(\lVert prog \rVert)$; otherwise, $[\![prog]\!] =$"error"

Release-acquire synchronization: message passing in C11

$$\textbf{atomic\_int}\ x = 0; \textbf{int}\ a = 0;$$
$$\left(\begin{array}{l} a = 7; \\ x.\mathsf{store}(1, \mathit{release}); \end{array} \middle\| \begin{array}{c} \textbf{if}\ (x.\mathsf{load}(\mathit{acq}) \neq 0) \\ \textbf{print}(a); \end{array} \right)$$



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, $x = y = 0$.

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

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

Example (IRIW)

Initially, $x = y = 0$.

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

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

Coherence

Example (Read-Read Coherence)

Initially, $x = 0$.

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

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

Today:

- Separation logic
- Relaxed separation logic

When should we care about relaxed memory?

All *sane* memory models satisfy the DRF property:

Theorem (DRF-property)

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

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

Separation logic assertions

Assertions describe the heap ($Loc \rightharpoonup Val$):

- emp: the empty heap
- $\ell \mapsto v$: a cell at address ℓ containing v

$$h \models \ell \mapsto v \iff h = \{\ell \mapsto v\}$$

- $P * Q$: separating conjunction

$$h \models P * Q \iff \exists h_1\, h_2.\ h = h_1 \uplus h_2 \land h_1 \models P \land h_2 \models Q$$

- $\land, \lor, \neg, \mathsf{true}, \mathsf{false}, \forall, \exists$: as usual

The separating conjunction

Some basic properties:

- $*$ is commutative & associative.
- $P * \mathsf{emp} \iff \mathsf{emp} * P \iff P$
- $\ell \mapsto v * \ell \mapsto v' \implies \mathsf{false}$

Useful for describing inductive data structures:

- $\mathit{list}(x) \stackrel{\text{def}}{=} (x = 0 \land \mathsf{emp}) \lor \exists y.\ x \mapsto y * \mathit{list}(y)$
- $\mathit{ls}(x, z) \stackrel{\text{def}}{=} (x = z \land \mathsf{emp}) \lor \exists y.\ x \mapsto y * \mathit{ls}(y, z)$
- $\mathit{tree}(x) \stackrel{\text{def}}{=} (x = 0 \land \mathsf{emp}) \lor \exists y, z.$
  $x \mapsto y * x{+}1 \mapsto z * \mathit{tree}(y) * \mathit{tree}(z)$

Separation logic

Key concept of *ownership* :

- Resourceful reading of Hoare triples

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}} \quad \text{(Par)}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \quad \text{(Frame)}$$

- Ensure memory safety & race-freedom

Separation logic rules for non-atomic accesses

- Allocation gives you permission to access x.

$$\{\mathsf{emp}\}\ x = \mathsf{alloc}();\ \{\exists v.\ x \mapsto v\}$$

- To access a normal location, you must own it:

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

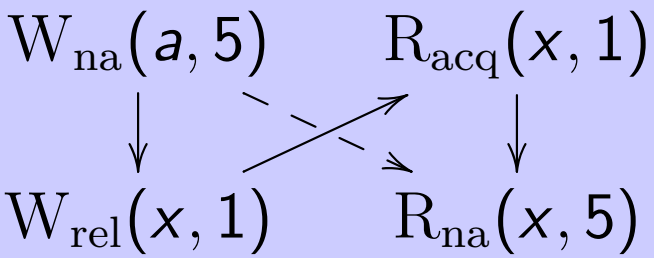
Release-acquire synchronization: message passing

Initially $a = x = 0$.

$$
\begin{array}{l|l}
a = 5; & \textbf{while } (x.\text{load}(acq) == 0); \\
x.\text{store}(release, 1); & \text{print}(a);
\end{array}
$$

This will always print 5.

Justification:



Release-acquire synchronization

Rules for release/acquire accesses

Relaxed separation logic [OOPSLA'13]

Ownership transfer by rel-acq synchronizations.

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

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

- Release write $\leadsto$ give away permissions.

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

- Acquire read $\leadsto$ gain permissions.

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

Message passing in RSL

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

$$\{\mathsf{true}\}$$

$$\textbf{atomic\_int}\ x = 0;\ \ \textbf{int}\ a = 0;$$

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

$$\left(\begin{array}{l|l}
\{\&a \mapsto 0 * \mathbf{W}_{\mathcal{Q}}(x)\} & \{\mathbf{R}_{\mathcal{Q}}(x)\} \\
a = 5; & \textbf{while}\ (x.\mathsf{load}(\mathit{acq}) == 0); \\
\{\&a \mapsto 5 * \mathbf{W}_{\mathcal{Q}}(x)\} & \{\&a \mapsto 5\} \\
x.\mathsf{store}(1, \mathit{release}); & \textbf{print}(a); \\
\{\mathsf{true}\} & \{\&a \mapsto 5\}
\end{array}\right)$$

$$\{\mathsf{true}\}$$

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:

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

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

Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

$$\left\{\mathbf{R}_{\mathcal{Q}}(x)\right\}\ t = x.load(rlx)\ \left\{\mathbf{R}_{\mathcal{Q}}(x) * (\mathcal{Q}(t) \not\equiv \text{false})\right\}$$

- Relaxed writes:

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

Relaxed accesses

Basically, disallow ownership transfer.

- Relaxed reads:

$$\left\{\mathbf{R}_{\mathcal{Q}}(x)\right\}\ t = x.load(rlx)\ \left\{\mathbf{R}_{\mathcal{Q}}(x) * (\mathcal{Q}(t) \not\equiv \text{false})\right\}$$

- Relaxed writes:

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

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

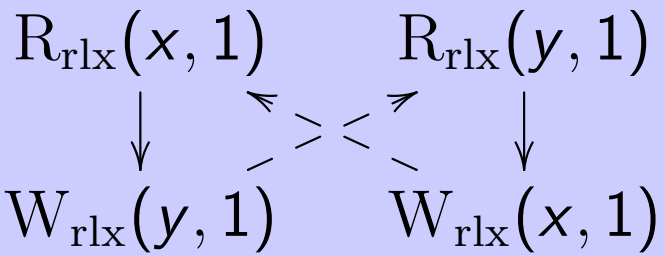
Dependency cycles in C11

Initially $x = y = 0$.

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

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

Justification:



Relaxed accesses don't synchronize

Dependency cycles in C11

Initially $x = y = 0$.

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

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.)

Conclusion

Topics covered today:

- The C11 memory model
- Separation logic
- Relaxed separation logic

Tomorrow:

- Compare and swap
- GPS
- Advanced C11 features