Reasoning over Relaxed Shared Memory Models: A Tutorial

Brijesh Dongol
University of Surrey, UK

with thanks to
Lara Bargmann, Mark Batty, Sadegh Dalvandi, Simon Doherty, Ori Lahav, Azalea Raad, Mikhail Semenyuk, Eleni Vafeiadi Bila, Heike Wehrheim, John Wickerson, Daniel Wright

Tutorial Plan

1. Background
   - Sequential consistency
   - Owicki-Gries reasoning
   - Isabelle/HOL
   - Examples: Message passing, Lamport buffer
2. Relaxed Memory
   - Motivation
   - RC11 Memory Model (Relaxed Accesses)
3. Verification
   - Owicki-Gries reasoning, revisited
   - Examples, revisited
   - Isabelle/HOL
4. Formal Model
5. Release/Acquire Accesses
   - Lamport buffer, revisited again
   - Message passing
   - Isabelle/HOL

Background

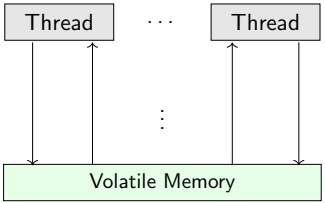
Sequential Consistency

"... the result of **any execution** is the same as if

- **the operations of all the processors** were executed **in some sequential order**, and
- **the operations of each individual processor appear** in this sequence **in the order specified by its program**."

— *[Lamport, 1979]*

Sequentially Consistent Architecture



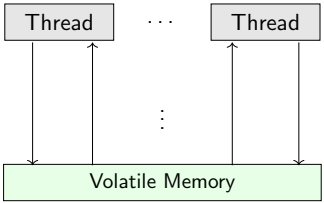
Concurrent execution $T_1 \| T_2 \| \ldots \| T_n$

$\equiv$

Interleaving of atomic statements in

$T_1, T_2, \ldots, T_n$

Sequentially Consistent Architecture



Concurrent execution $T_1 \| T_2 \| \ldots \| T_n$

$\equiv$

Interleaving of atomic statements in $T_1, T_2, \ldots, T_n$

Example

$$\begin{array}{c} \mathtt{I}\colon y = 0 \\ \hline \mathtt{a_1}\colon x := 1; \;\|\; \mathtt{b_1}\colon r := y; \\ \mathtt{a_2}\colon y := 1 \;\|\; \mathtt{b_2}\colon s := x \end{array}$$

Possible executions

- $\mathtt{I}; \mathtt{a_1}; \mathtt{a_2}; \mathtt{b_1}; \mathtt{b_2}$
- $\mathtt{I}; \mathtt{a_1}; \mathtt{b_1}; \mathtt{a_2}; \mathtt{b_2}$
- $\mathtt{I}; \mathtt{a_1}; \mathtt{b_1}; \mathtt{b_2}; \mathtt{a_2}$
- $+$ symmetric cases starting with $\mathtt{b_1}$

Sequentially Consistent Architecture

... makes a memory operation appear to execute atomically (instantaneously) with respect to other memory operations. — [Adve and Gharachorloo, 1996]

Sequentially Consistent Architecture

... makes a memory operation appear to execute atomically (instantaneously) with respect to other memory operations. — [Adve and Gharachorloo, 1996]

Sequentially Consistent Architecture

> *... makes a memory operation appear to execute atomically (instantaneously) with respect to other memory operations. — [Adve and Gharachorloo, 1996]*

Convenient state model: $\Sigma \mathrel{\widehat{=}} Loc \to Val$

Sequentially Consistent Architecture

> *... makes a memory operation appear to execute atomically (instantaneously) with respect to other memory operations. — [Adve and Gharachorloo, 1996]*

Convenient state model: $\Sigma \mathrel{\widehat{=}} Loc \to Val$

Example.

$$\begin{array}{c} \mathtt{I}\colon y = 0 \\ \hline \mathtt{a}_1\colon x := 1; \;\|\; \mathtt{b}_1\colon r := y; \\ \mathtt{a}_2\colon y := 1 \;\|\; \mathtt{b}_2\colon s := x \end{array}$$

Traces are of the form:

$$\begin{bmatrix} x \mapsto ?, \\ y \mapsto 0, \\ r \mapsto ?, \\ s \mapsto ? \end{bmatrix} \xrightarrow{x:=1} \begin{bmatrix} x \mapsto 1, \\ y \mapsto 0, \\ r \mapsto ?, \\ s \mapsto ? \end{bmatrix} \xrightarrow{r:=y} \begin{bmatrix} x \mapsto 1, \\ y \mapsto 0, \\ r \mapsto 0, \\ s \mapsto ? \end{bmatrix} \xrightarrow{y:=1} \begin{bmatrix} x \mapsto 1, \\ y \mapsto 1, \\ r \mapsto 0, \\ s \mapsto ? \end{bmatrix} \xrightarrow{s:=x} \begin{bmatrix} x \mapsto 1, \\ y \mapsto 1, \\ r \mapsto 0, \\ s \mapsto 1 \end{bmatrix}$$

Verification

- Dozens of techniques developed since the 1970s [de Roever et al., 2001]:
  - Owicki-Gries
  - Rely-guarantee
  - TLA
  - Action systems
  - Input/output automata
  - Concurrent separation logic
  - ...
- Let's focus on one: Owicki-Gries [Owicki, 1975]
- Extension of Hoare Logic with rules for "interference freedom"

Owicki-Gries proof outline

Example (Message passing). For *any* interleaving of threads, if thread b reads $y = 1$, then it must read $x = 1$.

$$
\begin{array}{c}
\{y = 0\} \\
\begin{array}{l|l}
\{true\} & \{y = 1 \Rightarrow x = 1\} \\
\quad x := 1; & \quad r := y; \\
\{x = 1\} & \{r = 1 \Rightarrow x = 1\} \\
\quad y := 1 & \quad s := x \\
\{true\} & \{r = 1 \Rightarrow s = 1\}
\end{array} \\
\{r = 1 \Rightarrow s = 1\}
\end{array}
$$

Owicki-Gries proof outline

Example (Message passing). For *any* interleaving of threads, if thread b reads $y = 1$, then it must read $x = 1$.

$$
\begin{array}{l}
\{y = 0\} \\
\{true\} \quad \| \quad \{y = 1 \Rightarrow x = 1\} \\
\quad x := 1; \quad \| \quad \quad r := y; \\
\{x = 1\} \quad \| \quad \{r = 1 \Rightarrow x = 1\} \\
\quad y := 1 \quad \| \quad \quad s := x \\
\{true\} \quad \| \quad \{r = 1 \Rightarrow s = 1\} \\
\{r = 1 \Rightarrow s = 1\}
\end{array}
$$

1. Initialisation

1. Initialisation proof:

$$y = 0 \;\Rightarrow\; true \wedge (y = 1 \Rightarrow x = 1)$$

Owicki-Gries proof outline

Example (Message passing). For *any* interleaving of threads, if thread b reads $y = 1$, then it must read $x = 1$.

$$
\begin{array}{l}
\{y = 0\} \\
\{true\} \quad \| \quad \{y = 1 \Rightarrow x = 1\} \\
\quad x := 1; \quad \| \quad \quad r := y; \\
\{x = 1\} \quad \| \quad \{r = 1 \Rightarrow x = 1\} \\
\quad y := 1 \quad \| \quad \quad s := x \\
\{true\} \quad \| \quad \{r = 1 \Rightarrow s = 1\} \\
\{r = 1 \Rightarrow s = 1\}
\end{array}
$$

1. Initialisation
2. Local correctness

2. Local correctness proofs:

$$\{true\}\ x := 1\ \{x = 1\} \qquad \{y = 1 \Rightarrow x = 1\}\ r := y\ \{r = 1 \Rightarrow x = 1\}$$

$$\{x = 1\}\ y := 1\ \{true\} \qquad \{r = 1 \Rightarrow x = 1\}\ s := x\ \{r = 1 \Rightarrow s = 1\}$$

Owicki-Gries proof outline

Example (Message passing). For *any* interleaving of threads, if thread b reads $y = 1$, then it must read $x = 1$.

$$
\begin{array}{c}
\{y = 0\} \\
\{true\} \quad \| \quad \{y = 1 \Rightarrow x = 1\} \\
x := 1; \quad \| \quad r := y; \\
\{x = 1\} \quad \| \quad \{r = 1 \Rightarrow x = 1\} \\
y := 1 \quad \| \quad s := x \\
\{true\} \quad \| \quad \{r = 1 \Rightarrow s = 1\} \\
\{r = 1 \Rightarrow s = 1\}
\end{array}
$$

1. Initialisation
2. Local correctness
3. Global correctness

3. Global correctness proofs:

For $P \in \{true, x = 1\}$,

$\{P \wedge (y = 1 \Rightarrow x = 1)\}\ r := y\ \{P\}$

$\{P \wedge (r = 1 \Rightarrow x = 1)\}\ s := x\ \{P\}$

For $Q \in \left\{ \begin{array}{ll} y = 1 \Rightarrow x = 1, & r = 1 \Rightarrow x = 1, \\ r = 1 \Rightarrow s = 1 & \end{array} \right\}$,

$\{Q \wedge true\}\ x := 1\ \{Q\}$

$\{Q \wedge x = 1\}\ y := 1\ \{Q\}$

Owicki-Gries proof outline

Example (Message passing). For *any* interleaving of threads, if thread b reads $y = 1$, then it must read $x = 1$.

$$
\begin{array}{l}
\{y = 0\} \\
\{true\} \quad \| \quad \{y = 1 \Rightarrow x = 1\} \\
\quad x := 1; \quad \| \quad \quad r := y; \\
\{x = 1\} \quad \| \quad \{r = 1 \Rightarrow x = 1\} \\
\quad y := 1 \quad \| \quad \quad s := x \\
\{true\} \quad \| \quad \{r = 1 \Rightarrow s = 1\} \\
\{r = 1 \Rightarrow s = 1\}
\end{array}
$$

1. Initialisation
2. Local correctness
3. Global correctness
4. Finalisation

4. Finalisation proof:

$$true \wedge (r = 1 \Rightarrow s = 1) \;\Rightarrow\; r = 1 \Rightarrow s = 1$$

Message Passing under SC

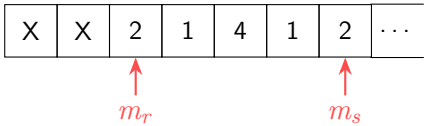
Isabelle/HOL Demo

Lamport Buffer

$$\{m_s = m_r = s = r = 0 \wedge b > 0\}$$

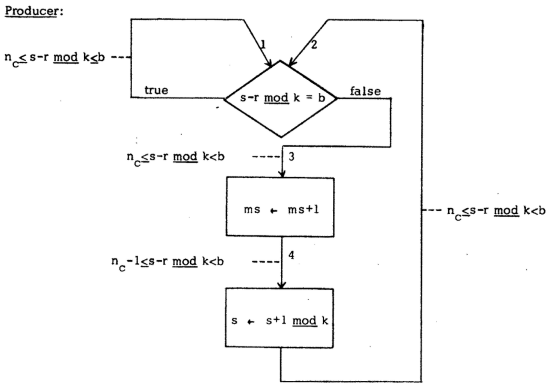
while *true*	**while** *true*
wait $s - r < b$ **end**;	**wait** $s - r > 0$ **end**;
$m_s := m_s + 1;$	$m_r := m_r + 1;$
$s := s + 1;$	$r := r + 1;$

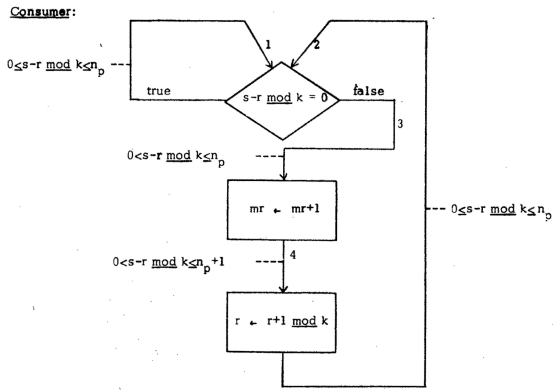
$$\{0 \leq m_s - m_r \leq b\}$$



- Synchronisation via shared variables s and r
- Auxiliary variables
  - m_s = #messages sent
  - m_r = #messages received
- Safety invariant $0 \leq m_s - m_r \leq b$

Lamport's Proof



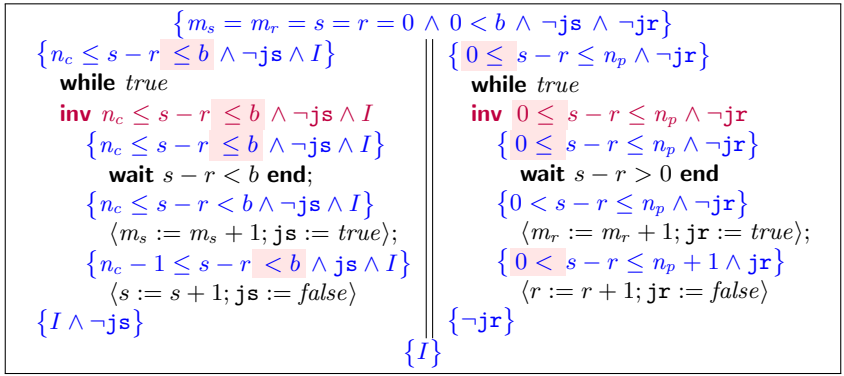


Let $n = m_s - m_r$

$$n_c = \begin{cases} n+1 & \text{if the consumer's token is on arc 4} \\ n & \text{otherwise} \end{cases}$$

$$n_p = \begin{cases} n-1 & \text{if the producer's token is on arc 4} \\ n & \text{otherwise} \end{cases}$$

Owicki-Gries Proof of Lamport's Buffer



- Define $I \;\widehat{=}\; 0 \leq n \leq b$
- Highlighted assertions can be omitted

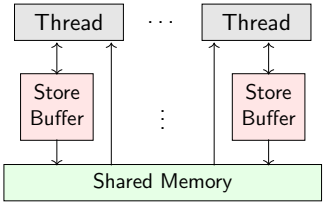
Lamport Buffer under SC

Isabelle/HOL Demo

Relaxed Memory

(Modern?) Multiprocessors and Languages

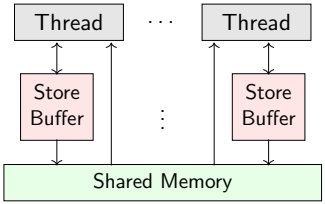
Hardware memory models



- Store buffer:
  Intel (FIFO)
  Arm, Power (Weakly ordered)
- Shared memory:
  Intel, Armv8/v9 (Multi-copy atomic)
  Armv7, Power (Not multi-copy atomic)

(Modern?) Multiprocessors and Languages

Hardware memory models



- Store buffer:
  Intel (FIFO)
  Arm, Power (Weakly ordered)
- Shared memory:
  Intel, Armv8/v9 (Multi-copy atomic)
  Armv7, Power (Not multi-copy atomic)

Language memory models

C11, C20, RC11, OCaml, Java, LLVM, WASM ...

Kernel memory models

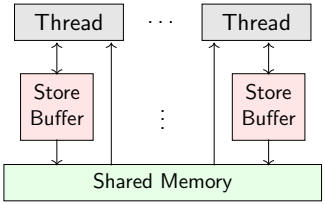
Linux, OpenBSD, ...

Heterogenous models

CUDA, PTX, RDMA, Px86, ...

(Modern?) Multiprocessors and Languages

Hardware memory models



- Store buffer:
  Intel (FIFO)
  Arm, Power (Weakly ordered)
- Shared memory:
  Intel, Armv8/v9 (Multi-copy atomic)
  Armv7, Power (Not multi-copy atomic)

Language memory models

C11, C20, RC11, OCaml, Java, LLVM, WASM ...

Kernel memory models

Linux, OpenBSD, ...

Heterogenous models

CUDA, PTX, RDMA, Px86, ...

Message Passing Under Relaxed Memory

$$
\begin{array}{c}
y = 0 \\
\hline
x := 1; \;\|\; r := y; \\
y := 1 \;\|\; s := x
\end{array}
$$

Is the final outcome $r = 1 \wedge s \neq 1$ allowed?

- Arm ✓
- Intel-TSO ✗
- C11 - it depends

Message Passing Under Relaxed Memory

$$
\begin{array}{c}
y = 0 \\
\hline
x := 1; \;\|\; r := y; \\
y := 1 \;\|\; s := x
\end{array}
$$

Is the final outcome $r = 1 \wedge s \neq 1$ allowed?

- Arm ✓
- Intel-TSO ✗
- C11 - it depends

Conclusion: Relaxed memory makes concurrency *much* more complicated

Message Passing Under Relaxed Memory

$$
\begin{array}{c}
y = 0 \\
\hline
x := 1; \;\|\; r := y; \\
y := 1 \;\|\; s := x
\end{array}
$$

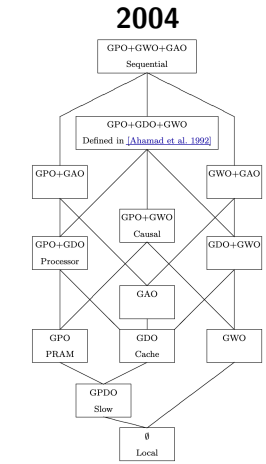
Is the final outcome $r = 1 \wedge s \neq 1$ allowed?

- Arm ✓
- Intel-TSO ✗
- C11 - it depends

Conclusion: Relaxed memory makes concurrency *much* more complicated

... but this is not a new phenomenon!

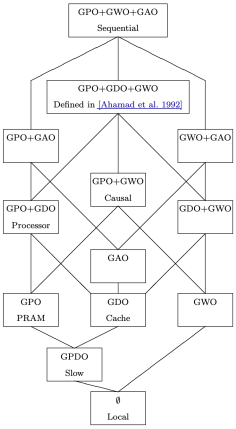
Relaxed Memory (Historically)



Memory consistency hierarchy
[Steinke and Nutt, 2004]

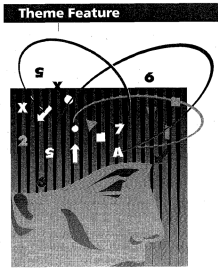
Relaxed Memory (Historically)

2004



Memory consistency hierarchy
[Steinke and Nutt, 2004]

1996



[Adve and Gharachorloo, 1996]

Relaxed Memory (Historically)

1993

Memory Consistency Models

David Mosberger

Department of Computer Science
University of Arizona
Tucson, AZ 85721
davidm@cs.arizona.edu

Abstract

This paper discusses memory consistency models and their influence on software in the context of parallel machines. In the first part we review previous work on memory consistency models. The second part discusses the issues that arise due to weakening memory consistency. We are especially interested in the influence that weakened consistency models have on language, compiler, and runtime system design. We conclude that tighter interaction between those parts and the memory system might improve performance considerably.

1 Introduction

Traditionally, memory consistency models were of interest only to computer architects designing parallel machines. The goal was to present a model as close as possible to the model exhibited by sequential machines. The model of choice was sequential consistency (SC). Sequential consistency guarantees that the result of any execution of n processors is the same as if the operations of all processors were executed in some sequential as the consistency model becomes weaker. That is, an architecture can employ a weaker memory model only if the software using it is prepared to deal with the new programming model. Consequently, memory consistency models are now of concern to operating system and language designers too.

We can also turn the coin around. A compiler normally considers memory accesses to be expensive and therefore tries to replace them by accesses to registers. In terms of a memory consistency model, this means that certain accesses suddenly are not observable any more. In effect, compilers implicitly generate weak memory consistency. This is possible because a compiler knows exactly (or estimates conservatively) the points where memory has to be consistent. For example, compilers typically write back register values before a function call, thus ensuring consistency. It is only natural to attempt to make this implicit weakening explicit in order to let the memory system take advantage too. In fact, it is anticipated that software could gain from a weak model to a much higher degree than hardware [GGH91] by enabling optimizations such as code scheduling or delaying updates that are not legal under SC.

Relaxed Memory (Historically)

1993

Memory Consistency Models

David Mosberger

Department of Computer Science
University of Arizona
Tucson, AZ 85721
davidm@cs.arizona.edu

Abstract

This paper discusses memory consistency models and their influence on software in the context of parallel machines. In the first part we review previous work on memory consistency models. The second part discusses the issues that arise due to weakening memory consistency. We are especially interested in the influence that weakened consistency models have on language, compiler, and runtime system design. We conclude that tighter interaction between those parts and the memory system might improve performance considerably.

1 Introduction

Traditionally, memory consistency models were of interest only to computer architects designing parallel machines. The goal was to present a model as close as possible to the model exhibited by sequential machines. The model of choice was sequential consistency (SC). Sequential consistency guarantees that the result of any execution of n processors is the same as if the operations of all processors were executed in some sequential as the consistency model becomes weaker. That is, an architecture can employ a weaker memory model only if the software using it is prepared to deal with the new programming model. Consequently, memory consistency models are now of concern to operating system and language designers too.

We can also turn the coin around. A compiler normally considers memory accesses to be expensive and therefore tries to replace them by accesses to registers. In terms of a memory consistency model, this means that certain accesses suddenly are not observable any more. In effect, compilers implicitly generate weak memory consistency. This is possible because a compiler knows exactly (or estimates conservatively) the points where memory has to be consistent. For example, compilers typically write back register values before a function call, thus ensuring consistency. It is only natural to attempt to make this implicit weakening explicit in order to let the memory system take advantage too. In fact, it is anticipated that software could gain from a weak model to a much higher degree than hardware [GGH91] by enabling optimizations such as code scheduling or delaying updates that are not legal under SC.

1988

Efficient and Correct Execution of Parallel Programs that Share Memory

DENNIS SHASHA
Courant Institute of Mathematical Sciences, New York University
and
MARC SNIR
IBM T. J. Watson Research Center

In this paper we consider an optimization problem that arises in the execution of parallel programs on shared-memory multiple-instruction-stream, multiple-data-stream (MIMD) computers. A program on such machines consists of many sequential program segments, each executed by a single processor. These segments interact as they access shared variables. Access to memory is asynchronous, and memory accesses are not necessarily executed in the order they were issued. An execution is correct if it is sequentially consistent: It should seem as if all the instructions were executed sequentially, in an order obtained by interleaving the instruction streams of the processors. Sequential consistency can be enforced by delaying each access to shared memory until the previous access of the same processor has terminated. For performance reasons, however, we want to allow several accesses by the same processor to proceed concurrently. Our analysis finds a minimal set of delays that enforces sequential consistency. The analysis extends to interprocessor synchronization constraints and to code where blocks of operations have to execute atomically. We use a conflict graph similar to that used to schedule transactions in distributed databases. Our graph incorporates the order on operations given by the program text, enabling us to do without locks even when database conflict graphs would suggest that locks are necessary. Our work has implications for the design of multiprocessors; it offers new compiler optimization techniques for parallel languages that support shared variables.

Relaxed Memory (Historically)

... **achieving sequential consistency may not be worth the price** *of slowing down the processors.*

1979

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Relaxed Memory (Historically)

1979

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

... **achieving sequential consistency may not be worth the price** *of slowing down the processors. In this case, one must be aware that* **conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs.**

Relaxed Memory (Historically)

1979

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

... **achieving sequential consistency may not be worth the price** *of slowing down the processors. In this case, one must be aware that* **conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs.** *Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and* **verifying their correctness becomes a monumental task.**

Relaxed Memory (Recently)

2010s/2020s explosion of formal models and results

- [Batty et al., 2011] presented the first formalisation of the C11 memory model
- [Alglave et al., 2014] defined the axiomatic CAT language for formalising consistency models
- [Lahav et al., 2017] discovered inconsistencies in the C11 specification, and proposed a fix called RC11

For this tutorial, we focus on **RC11** —

- Program verification focusses on language-level relaxed-memory
- Rely on compiler to map to hardware memory model

Repaired C11 (RC11)

Repairing C11

[Lahav et al., 2017] discovered three key problems with the C11 memory model

1. Compilation from C11 to PowerPC is unsound
2. Fences (used to enforce instruction order) are too weak
3. Model allows out-of-thin-air reads

Repairing C11

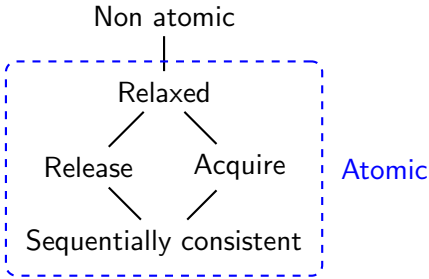
[Lahav et al., 2017] discovered three key problems with the C11 memory model

1. Compilation from C11 to PowerPC is unsound
2. Fences (used to enforce instruction order) are too weak
3. Model allows out-of-thin-air reads

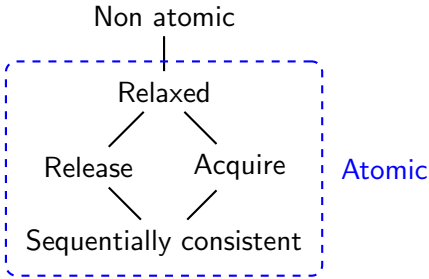
Proposed RC11 memory model as a fix

- Introduces an axiom that **maintains program order**
- Allows us to view weak memory programs as an interleaving of threads
- Relaxed behaviour induces **delayed propagation** of writes

RC11 Memory Accesses



RC11 Memory Accesses



We will focus on relaxed, release and acquire

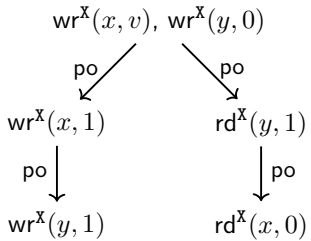
RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

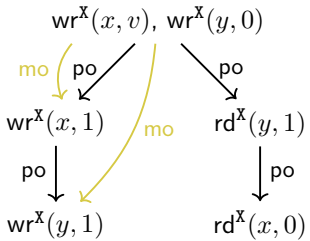
Axiomatic (aka declarative) semantics



RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

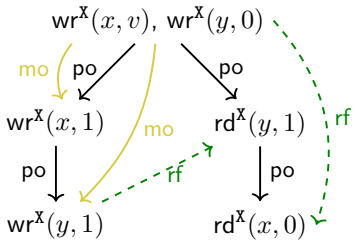
Axiomatic (aka declarative) semantics



RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

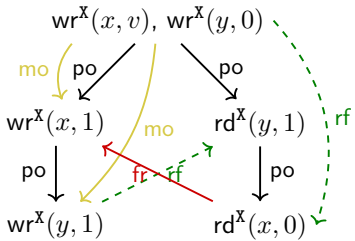
Axiomatic (aka declarative) semantics



RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

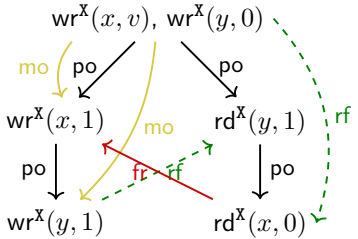
Axiomatic (aka declarative) semantics



RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

Axiomatic (aka declarative) semantics

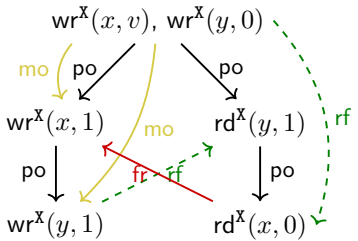


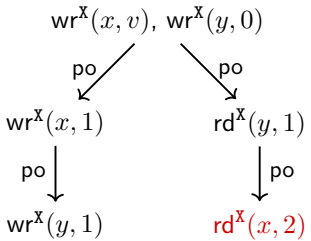
Allowed

RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\Big\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\Big\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

Axiomatic (aka declarative) semantics



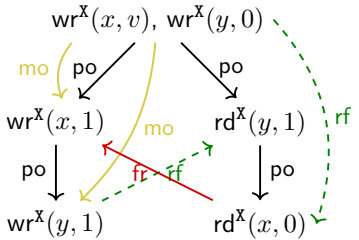


Allowed

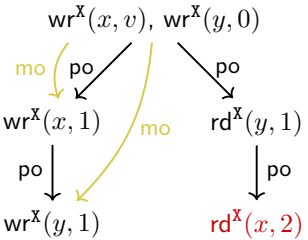
RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

Axiomatic (aka declarative) semantics



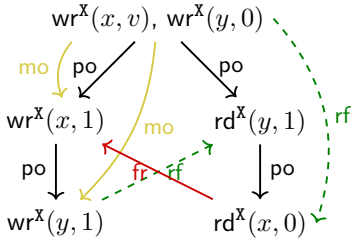
Allowed



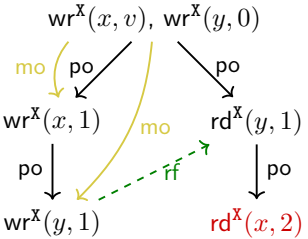
RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

Axiomatic (aka declarative) semantics



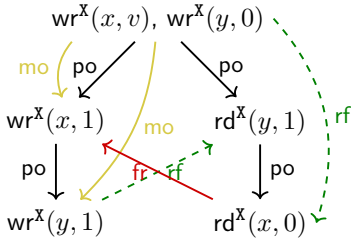
Allowed



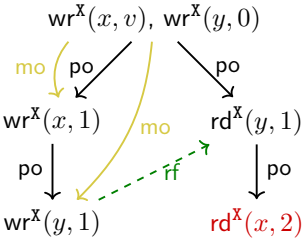
RC11 Models

$$
\begin{array}{c}
y = 0 \\
\hline
[x] :\overset{\mathtt{X}}{=} 1; \;\|\; r :\overset{\mathtt{X}}{=} [y]; \\
[y] :\overset{\mathtt{X}}{=} 1 \;\|\; s :\overset{\mathtt{X}}{=} [x]
\end{array}
$$

Axiomatic (aka declarative) semantics



Allowed



Disallowed

Axiomatic Models

- Often the "authoritative" formal model, particularly in hardware E.g., https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat

```
(*** External visibility requirement ***)
irreflexive ob as external

(*** Internal visibility requirements ***)
irreflexive [Exp & R]; ((po & same-loc) | rmw); [Exp & W];
            rfi; [Exp & R] as coRW1-Exp
irreflexive [Imp & Tag & R]; (po & same-loc); [Exp & Tag & W];
            rfi; [Imp & Tag & R] as coRW1-MTE
irreflexive [Exp & W]; (po & same-loc); [Exp & W];
            (ca & int); [Exp & W] as coWW-Exp
irreflexive [Exp & W]; (po & same-loc); [Exp & R];
            (ca & int); [Exp & W] as coWR-Exp
irreflexive [Exp & Tag & W]; (po & same-loc); [Imp & Tag & R];
            (ca & int) as coWR-MTE
```

Axiomatic Models

- Often the "authoritative" formal model, particularly in hardware E.g., https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat

```
(*** External visibility requirement ***)
irreflexive ob as external

(*** Internal visibility requirements ***)
irreflexive [Exp & R]; ((po & same-loc) | rmw); [Exp & W];
            rfi; [Exp & R] as coRW1-Exp
irreflexive [Imp & Tag & R]; (po & same-loc); [Exp & Tag & W];
            rfi; [Imp & Tag & R] as coRW1-MTE
irreflexive [Exp & W]; (po & same-loc); [Exp & W];
            (ca & int); [Exp & W] as coWW-Exp
irreflexive [Exp & W]; (po & same-loc); [Exp & R];
            (ca & int); [Exp & W] as coWR-Exp
irreflexive [Exp & Tag & W]; (po & same-loc); [Imp & Tag & R];
            (ca & int) as coWR-MTE
```

- ... but cannot directly be connected to traditional operational proof techniques

Axiomatic Models

- Often the "authoritative" formal model, particularly in hardware E.g.,
  https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat

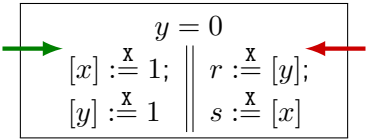
```
(*** External visibility requirement ***)
irreflexive ob as external

(*** Internal visibility requirements ***)
irreflexive [Exp & R]; ((po & same-loc) | rmw); [Exp & W];
            rfi; [Exp & R] as coRW1-Exp
irreflexive [Imp & Tag & R]; (po & same-loc); [Exp & Tag & W];
            rfi; [Imp & Tag & R] as coRW1-MTE
irreflexive [Exp & W]; (po & same-loc); [Exp & W];
            (ca & int); [Exp & W] as coWW-Exp
irreflexive [Exp & W]; (po & same-loc); [Exp & R];
            (ca & int); [Exp & W] as coWR-Exp
irreflexive [Exp & Tag & W]; (po & same-loc); [Imp & Tag & R];
            (ca & int) as coWR-MTE
```

- ... but cannot directly be connected to traditional operational proof techniques
- ... however every axiomatic model gives rise to 1-1 operational model

  — check the axioms of the memory model at every step

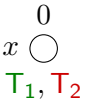
RC11 Models II

Example 1.



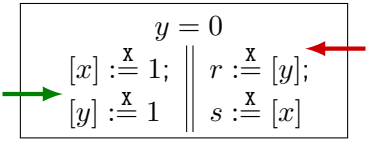
Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto ? \end{bmatrix}$$

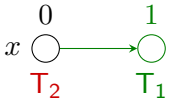


RC11 Models II

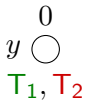
Example 1.



Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

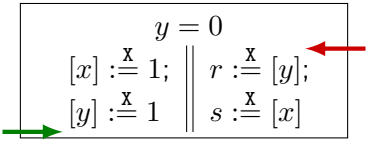


$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto ? \end{bmatrix}$$



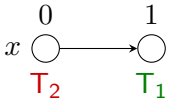
RC11 Models II

Example 1.



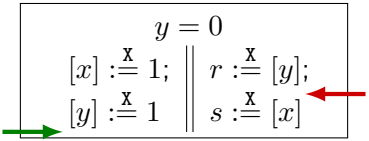
Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto ? \end{bmatrix}$$

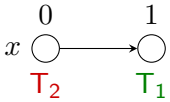


RC11 Models II

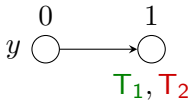
Example 1.



Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

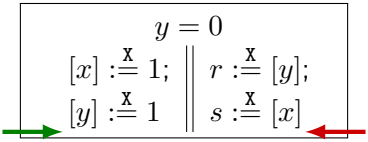


$$\begin{bmatrix} r \mapsto 1, \\ s \mapsto ? \end{bmatrix}$$



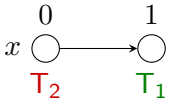
RC11 Models II

Example 1.



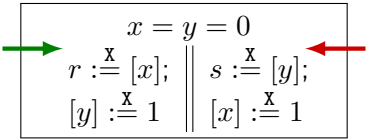
Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

$$\begin{bmatrix} r \mapsto 1, \\ s \mapsto 0 \end{bmatrix}$$



RC11 Models II

Example 2.



Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

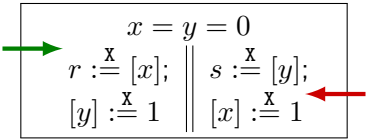
$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto ? \end{bmatrix}$$

$\mathsf{T}_1, \mathsf{T}_2$

$\mathsf{T}_1, \mathsf{T}_2$

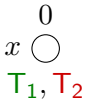
RC11 Models II

Example 2.



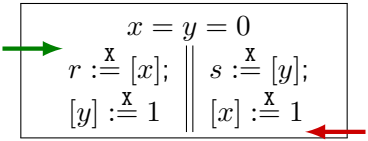
Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto 0 \end{bmatrix}$$

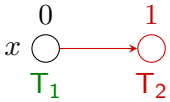


RC11 Models II

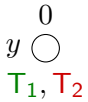
Example 2.



Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

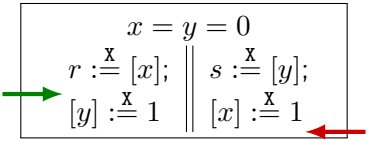


$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto 0 \end{bmatrix}$$

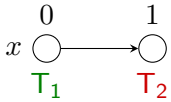


RC11 Models II

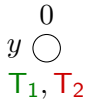
Example 2.



Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

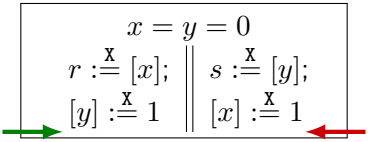


$$\begin{bmatrix} r \mapsto 0, \\ s \mapsto 0 \end{bmatrix}$$



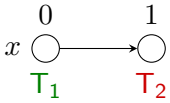
RC11 Models II

Example 2.



Operational (view-based) models [Kang et al., 2017, Kaiser et al., 2017]

$$\begin{bmatrix} r \mapsto 0, \\ s \mapsto 0 \end{bmatrix}$$



Verification

RC11: Towards a Reasoning Framework

- View model for RC11 supports **interleaved execution**
- Allows relaxed behaviours — reads may **return stale values**
- Difficulty comes from the state model
  - $Loc \to Val$ states only valid for thread-local variables
  - View-based states required to model shared variables

Key idea: develop high-level assertions directly over view states

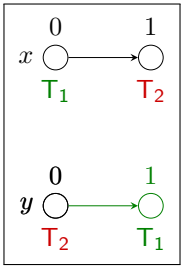
RC11: View assertions

Only a handful of assertions needed to handle most examples

- *Last value predicate* $x \langle\langle \tau$
  thread τ is viewing the last-written value for shared variable x
- *Values view function* $|\tau\rangle x$
  the set of values that thread τ can see for shared variable x
- *Definite value predicate* $x \stackrel{\tau}{=} \mathsf{xv}$
  $x \langle\langle \tau \ \wedge \ |\tau\rangle x = \{\mathsf{xv}\}$

View Assertions (Example)

Consider the state σ:



The following properties hold for σ:

$$\neg\, x\langle\langle \mathsf{T}_1 \wedge y\langle\langle \mathsf{T}_1 \qquad |\mathsf{T}_1\rangle x = \{0,1\} \qquad y \overset{\mathsf{T}_1}{=} 1$$

$$x\langle\langle \mathsf{T}_2 \wedge \neg\, y\langle\langle \mathsf{T}_2 \qquad |\mathsf{T}_2\rangle x = \{1\} \qquad x \overset{\mathsf{T}_2}{=} 1$$

Verifying Load Buffering

$$x = y = 0 \wedge s = r = 0$$

Thread a	**Thread** b
$\{y = 0 \wedge s = 0\}$	$\{x = 0 \wedge r = 0\}$
$r := x;$	$s := y;$
$\{y = 0 \wedge s = 0\}$	$\{x = 0 \wedge r = 0\}$
$y := 1$	$x := 1$
$\{r = 0 \vee s = 0\}$	$\{r = 0 \vee s = 0\}$

$$\{r = 0 \vee s = 0\}$$

$$x \overset{a}{=} 0 \wedge y \overset{b}{=} 0 \wedge s = r = 0$$

Thread a	**Thread** b
$\left\{y \overset{b}{=} 0 \wedge s = 0\right\}$	$\left\{x \overset{a}{=} 0 \wedge r = 0\right\}$
$r \overset{\mathtt{X}}{:=} [x];$	$s \overset{\mathtt{X}}{:=} [y];$
$\left\{y \overset{b}{=} 0 \wedge s = 0\right\}$	$\left\{x \overset{a}{=} 0 \wedge r = 0\right\}$
$[y] \overset{\mathtt{X}}{:=} 1$	$[x] \overset{\mathtt{X}}{:=} 1$
$\{r = 0 \vee s = 0\}$	$\{r = 0 \vee s = 0\}$

$$\{r = 0 \vee s = 0\}$$

Load Buffering in RC11

Isabelle/HOL Demo

Lamport Buffer

$$\left\{ s \overset{p}{=} \mathsf{sv} \wedge r \overset{c}{=} \mathsf{rv} \wedge V_{pc} \wedge b > 0 \wedge \mathsf{sv} - \mathsf{rv} = n \wedge \neg \mathtt{js} \wedge \neg \mathtt{jr} \wedge I \right\}$$

$$\left\{ \begin{array}{l} s \overset{p}{=} \mathsf{sv} \wedge V_p \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathsf{sv} - \mathsf{rv}) \end{array} \right\}$$

$$\mathtt{s}_p \overset{\mathtt{x}}{:=} s;$$

$$\left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge V_p \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathsf{rv}) \end{array} \right\}$$

do $\mathtt{r}_p \overset{\mathtt{x}}{:=} r;$

$$\left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge V_p \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathsf{rv} \wedge \mathtt{r}_p \leq \mathsf{rv}) \end{array} \right\}$$

until $\mathtt{s}_p - \mathtt{r}_p < b;$

$$\left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge (\exists \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv}) \wedge \neg \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathsf{rv} < b) \end{array} \right\}$$

$\langle m_s := m_s + 1; \mathtt{js} := \mathit{true} \rangle;$

$$\left\{ \begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge \mathtt{js} \wedge I \wedge \\ (\forall \mathsf{rv}.\ r \overset{c}{=} \mathsf{rv} \Rightarrow n_c - 1 \leq \mathsf{sv} - \mathsf{rv}) \end{array} \right\}$$

$\langle s \overset{\mathtt{x}}{:=} \mathtt{s}_p + 1, \mathtt{js} := \mathit{false} \rangle$

$$\left\{ s \overset{p}{=} \mathtt{s}_p + 1 \wedge \neg \mathtt{js} \wedge I \right\}$$

$\Big\|$

$$\left\{ \begin{array}{l} r \overset{c}{=} \mathsf{rv} \wedge V_c \wedge \neg \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathsf{sv} - \mathsf{rv} \leq n_p) \end{array} \right\}$$

$$\mathtt{r}_c \overset{\mathtt{x}}{:=} r;$$

$$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge V_c \wedge \neg \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathsf{sv} - \mathtt{r}_c \leq n_p) \end{array} \right\}$$

do $\mathtt{s}_c \overset{\mathtt{x}}{:=} s$

$$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge V_c \wedge \neg \mathtt{js} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathtt{s}_c \leq \mathsf{sv} \wedge \mathsf{sv} - \mathtt{r}_c \leq n_p) \end{array} \right\}$$

until $\mathtt{s}_c - \mathtt{r}_c > 0;$

$$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge \neg \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow 0 < \mathsf{sv} - \mathtt{r}_c \leq n_p) \end{array} \right\}$$

$\langle m_r := m_r + 1; \mathtt{jr} := \mathit{true} \rangle;$

$$\left\{ \begin{array}{l} r \overset{c}{=} \mathtt{r}_c \wedge \mathtt{jr} \wedge \\ (\forall \mathsf{sv}.\ s \overset{p}{=} \mathsf{sv} \Rightarrow \mathsf{sv} - \mathtt{r}_c \leq n_p + 1) \end{array} \right\}$$

$\langle r \overset{\mathtt{x}}{:=} \mathtt{r}_c + 1, \mathtt{jr} := \mathit{false} \rangle$

$$\left\{ s \overset{p}{=} \mathtt{s}_p + 1 \wedge \neg \mathtt{jr} \right\}$$

$$\{I\}$$

Lamport Buffer in RC11

Isabelle/HOL Demo

Lamport Buffer

$\{m_s = m_r = s = r = 0 \wedge 0 < b \wedge \neg\mathtt{js} \wedge \neg\mathtt{jr}\}$
$\{n_c \leq s - r \wedge \neg\mathtt{js} \wedge I\}$
while $true$
inv $n_c \leq s - r \wedge \neg\mathtt{js} \wedge I$
$\{n_c \leq s - r \wedge \neg\mathtt{js} \wedge I\}$
wait $s - r < b$ **end**;
$\{n_c \leq s - r < b \wedge \neg\mathtt{js} \wedge I\}$
$\langle m_s := m_s + 1; \mathtt{js} := true\rangle$;
$\{n_c - 1 \leq s - r \wedge \mathtt{js} \wedge I\}$
$\langle s := s + 1; \mathtt{js} := false\rangle$
$\{I \wedge \neg\mathtt{js}\}$
$\{I\}$

$\cdots$

$\{\ldots\}$
$\left\{\begin{array}{l} s \overset{p}{=} \mathtt{sv} \wedge V_p \wedge \neg\mathtt{js} \wedge I \wedge \\ (\forall \mathtt{rv}.\ r \overset{c}{=} \mathtt{rv} \Rightarrow n_c \leq \mathtt{sv} - \mathtt{rv}) \end{array}\right\}$
$\mathtt{s}_p \overset{\mathtt{X}}{:=} s$;
$\left\{\begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge V_p \wedge \neg\mathtt{js} \wedge I \wedge \\ (\forall \mathtt{rv}.\ r \overset{c}{=} \mathtt{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathtt{rv}) \end{array}\right\}$
do $\mathtt{r}_p \overset{\mathtt{X}}{:=} r$;
$\left\{\begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge V_p \wedge \neg\mathtt{js} \wedge I \wedge \\ (\forall \mathtt{rv}.\ r \overset{c}{=} \mathtt{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathtt{rv} \wedge \mathtt{r}_p \leq \mathtt{rv}) \end{array}\right\}$
until $\mathtt{s}_p - \mathtt{r}_p < b$;
$\left\{\begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge (\exists \mathtt{rv}.\ r \overset{c}{=} \mathtt{rv}) \wedge \neg\mathtt{js} \wedge I \wedge \\ (\forall \mathtt{rv}.\ r \overset{c}{=} \mathtt{rv} \Rightarrow n_c \leq \mathtt{s}_p - \mathtt{rv} < b) \end{array}\right\}$
$\langle m_s := m_s + 1; \mathtt{js} := true\rangle$;
$\left\{\begin{array}{l} s \overset{p}{=} \mathtt{s}_p \wedge \mathtt{js} \wedge I \wedge \\ (\forall \mathtt{rv}.\ r \overset{c}{=} \mathtt{rv} \Rightarrow n_c - 1 \leq \mathtt{sv} - \mathtt{rv}) \end{array}\right\}$
$\langle s \overset{\mathtt{X}}{:=} \mathtt{s}_p + 1, \mathtt{js} := false\rangle$
$\left\{s \overset{p}{=} \mathtt{s}_p + 1 \wedge \neg\mathtt{js} \wedge I\right\}$
$\{I\}$

$\cdots$

Lamport Buffer in RC11

Isabelle/HOL Demo

Formal Model

Syntax

$$Exp ::= \mathtt{v} \mid \mathtt{r} \mid \ominus e \mid e_1 \oplus e_2 \qquad Aux ::= \mathtt{v} \mid \mathtt{r} \mid \mathtt{a} \mid \ominus f \mid f_1 \oplus f_2$$

$$Asgn ::= \mathtt{r} := e \mid \mathtt{r} :\overset{\mathtt{M}}{=} x \mid x :\overset{\mathtt{M}}{=} e$$

$$Com ::= a \mid \langle a; \overline{\mathtt{a}} := \overline{f} \rangle \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{do } c \textbf{ until } b$$

$$Prog ::= \lambda\tau \in Tid.\ c$$

Operational Semantics (Uninterpreted)

$$\textsc{Rd}\quad \frac{\gamma \xrightarrow{\langle x, \mathsf{read}(v), \mathtt{M}\rangle}_{\tau} \gamma'}{\langle \sigma, \gamma, \mathtt{r} \overset{\mathtt{M}}{:=} x\rangle \rightarrow_{\tau} \langle \sigma[\mathtt{r} \mapsto v], \gamma', \bot\rangle}$$

$$\textsc{Wr}\quad \frac{\gamma \xrightarrow{\langle x, \mathsf{write}([\![e]\!]_{\sigma}), \mathtt{M}\rangle}_{\tau} \gamma'}{\langle \sigma, \gamma, x \overset{\mathtt{M}}{:=} e\rangle \rightarrow_{\tau} \langle \sigma, \gamma', \bot\rangle}$$

$$\textsc{Prog}\quad \frac{\langle \sigma, \gamma, \Pi(\tau)\rangle \rightarrow_{\tau} \langle \sigma', \gamma', c'\rangle}{\langle \sigma, \gamma, \Pi\rangle \rightarrow \langle \sigma', \gamma', \Pi[\tau \mapsto c']\rangle}$$

RC11 State

- Use encoding by [Castañeda et al., 2024]

- Only need two components $\langle m, \mathcal{V} \rangle$
  - $m =$ set of timestamped messages (writes)
  - $\mathcal{V} =$ view of each thread

- Each message $\mu = \langle x, \mathtt{v}, t, V, \mathtt{M} \rangle$ where:
  - $x \in Var$ is the variable of the message
  - $\mathtt{v} \in Val$ is the written value
  - $t \in \mathbb{Q}$ is the timestamp
  - $V \in \text{View} \mathrel{\widehat{=}} Var \to \mathbb{Q}$ is the view of the message (ignored for now)
  - $\mathtt{M} \in \{\mathtt{X}, \mathtt{R}\}$ is the memory ordering

- Thread view: $\mathcal{V} \in Tid \to \text{View}$

Isabelle/HOL encoding

```
record ('Var, 'Val) Msg =
    var  :: 'Var
    val  :: 'Val
    ts   :: TS
    view :: "'Var View"
    mord :: Mord

record ('Tid, 'Var, 'Val) C11State =
    msgs :: "('Var, 'Val) Msg set"
    TView :: "'Tid ⇒ 'Var View"
```

RC11 Relaxed Reads / Writes

READ-RLX

$$\frac{e = \langle x, \mathsf{read}(v), \mathtt{X}\rangle \qquad \langle x, v, t, \_, \_\rangle \in m \qquad \mathcal{V}(\tau)(x) \leq t \qquad V' = \mathcal{V}(\tau)[x \mapsto t]}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

WRITE-RLX-REL

$$\frac{e = \langle x, \mathsf{write}(v), \mathtt{M}\rangle \qquad V' = \mathcal{V}(\tau)[x \mapsto t] \qquad \mathcal{V}(\tau)(x) < t \qquad t \notin \mathsf{ts}(m_{|x}) \qquad \mu = \langle x, v, t, V', \mathtt{M}\rangle}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m \cup \{\mu\}, \mathcal{V}[\tau \mapsto V']\rangle}$$

- READ-RLX reads from the message with timestamp t
- WRITE-RLX-REL introduces a new write message with *fresh* timestamp t

```
definition "readRlxTrans τ x μ σ ≡
  let V' = (TView σ τ)(x := ts μ) in
          σ ;; updateTView τ V'"

definition "readRlx τ x v σ σ' ≡
  ∃ μ ∈ (obs τ σ)|x .
     val μ = v ∧ σ' = readRlxTrans τ x μ σ"
```

```
definition "writeTransExp b τ x v t σ ≡
   let V' = (TView σ τ)(x := t);
     μ = (| var=x, val=v, ts=t,
            view=V', mord=b |)
   in σ ;; updateMsgs μ ;; updateTView τ V'"


definition "writeTrans b τ x v σ σ' ≡
  ∃ t. validFreshTS σ τ x t ∧
       σ' = writeTransExp b τ x v t σ"
```

Formalising View Assertions

- *Last value predicate*: Thread τ is viewing the last-written value for x

$$x\langle\langle\tau \quad \widehat{=} \quad \lambda\langle m, \mathcal{V}\rangle.\ \mathcal{V}(\tau)(x) = \max(\mathsf{ts}(m_{|x}))$$

```
definition lastView :: "..." ("_⟪_ _" ...)
  where "lastView x τ σ ≡ ∀ ts ∈ tss ((msgs σ)|x). ts ≤ TView σ τ x "
```

- *Values view function*: The set of values that thread τ can see for x

$$\begin{aligned} \mathsf{obs}(\tau) &\ \widehat{=}\ \lambda\langle m, \mathcal{V}\rangle.\ \{\mu \in m \mid \mathsf{ts}(\mu) \geq \mathcal{V}(\tau)(\mathsf{var}(\mu))\} \\ |\tau\rangle x &\ \widehat{=}\ \lambda\sigma.\ \mathsf{val}(\mathsf{obs}(\tau)(\sigma)_{|x}) \end{aligned}$$

```
definition valsView :: ... ("|_⟩_ _" ...)
  where "valsView τ x σ ≡ vals ((obs τ σ)|x)"
```

- *Definite value predicate*

$$x \overset{\tau}{=} \mathsf{xv} \quad \widehat{=} \quad x\langle\langle\tau \ \wedge\ |\tau\rangle x = \{\mathsf{xv}\}$$

Hoare Triples

Definition

Given assertions $P, Q : \Sigma \times \Gamma \to \mathbb{B}$, thread $\tau \in Tid$, command $c \in Com$ and program $\Pi \in Prog$:

$$\{P\}\tau \rhd c\{Q\} \mathrel{\widehat{=}} \forall \sigma, \gamma.\ P(\langle \sigma, \gamma \rangle) \wedge \langle \sigma, \gamma, c \rangle \to_\tau^* \langle \sigma', \gamma', \bot \rangle \Rightarrow Q(\langle \sigma', \gamma' \rangle)$$

$$\{P\}\Pi\{Q\} \mathrel{\widehat{=}} \forall \sigma, \gamma.\ P(\langle \sigma, \gamma \rangle) \wedge \langle \sigma, \gamma, \Pi \rangle \to^* \langle \sigma', \gamma', \lambda\tau.\ \bot \rangle \Rightarrow Q(\langle \sigma', \gamma' \rangle)$$

RC11 Axioms

Examples:

RD-STABLE

$$\frac{}{\left\{x \overset{\tau}{=} \mathsf{xv}\right\}\tau' \rhd \mathtt{r} :\overset{\mathtt{M}}{=} y\left\{x \overset{\tau}{=} \mathsf{xv}\right\}}$$

WR-LATEST

$$\frac{}{\left\{x\langle\!\langle\tau\right\}\tau \rhd x :\overset{\mathtt{M}}{=} e\left\{x \overset{\tau}{=} e\right\}}$$

Example soundness proof for WR-LATEST:

```
lemma wrAtLast_stable_sameTh: "x⟪τ σ ⇒ writeStep τ y v σ σ' ⇒ x⟪τ σ'"
  by (auto simp add: unfold_defs, auto)

lemma lastView_Wr: "x⟪τ σ ⇒ writeStep τ x v σ σ' ⇒ |τ⟩x σ' = {v}"
  apply (auto simp add: unfold_defs, auto)
  apply ((metis Msg.select_convs(2))+)[2]
  by (metis Msg.select_convs(1,2,3) dual_order.refl)+

lemma DVal_Update_Wr:
  "x⟪τ σ ⇒ writeStep τ x v σ σ' ⇒ x =^τ v σ'"
  by (meson defVal_def lastView_Wr wrAtLast_stable_sameTh)
```

Release-Acquire

Lamport Buffer

while $true$	**while** $true$
$\mathtt{s}_p :\overset{\mathtt{X}}{=} s;$	$\mathtt{r}_c :\overset{\mathtt{X}}{=} r;$
do $\mathtt{r}_p :\overset{\mathtt{X}}{=} r;$	**do** $\mathtt{s}_c :\overset{\mathtt{X}}{=} s$
until $\mathtt{s}_p - \mathtt{r}_p < b;$	**until** $\mathtt{s}_c - \mathtt{r}_c > 0;$
$\langle m_s := m_s + 1; \mathtt{js} := true\rangle;$	$\langle m_r := m_r + 1; \mathtt{jr} := true\rangle;$
$\langle s :\overset{\mathtt{X}}{=} \mathtt{s}_p + 1, \mathtt{js} := false\rangle$	$\langle r :\overset{\mathtt{X}}{=} \mathtt{r}_c + 1, \mathtt{jr} := false\rangle$

- Program guarantees the invariant $I = 0 \leq m_s - m_r \leq b$ under RC11
- Question: What is the problem?
- Answer: Messages sent by the producer may not arrive at the consumer in time
- This problem is due to weak memory effects

Message Passing

Relaxed accesses

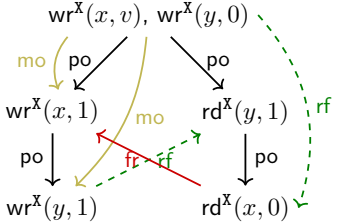
$$y = 0$$

$$\begin{array}{l|l} [x] :\overset{\mathtt{X}}{=} 1; & r :\overset{\mathtt{X}}{=} [y]; \\ [y] :\overset{\mathtt{X}}{=} 1 & s :\overset{\mathtt{X}}{=} [x] \end{array}$$

Message Passing

Relaxed accesses

$$y = 0$$
$$[x] :\overset{\mathtt{X}}{=} 1; \;\Big\|\; r :\overset{\mathtt{X}}{=} [y];$$
$$[y] :\overset{\mathtt{X}}{=} 1 \;\Big\|\; s :\overset{\mathtt{X}}{=} [x]$$



Allowed

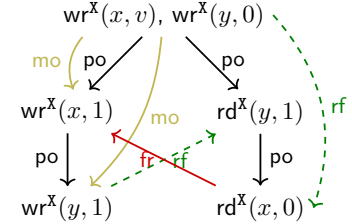
Release-acquire

$$y = 0$$
$$[x] :\overset{\mathtt{X}}{=} 1; \;\Big\|\; r :\overset{\mathtt{A}}{=} [y];$$
$$[y] :\overset{\mathtt{R}}{=} 1 \;\Big\|\; s :\overset{\mathtt{X}}{=} [x]$$

Message Passing

Relaxed accesses

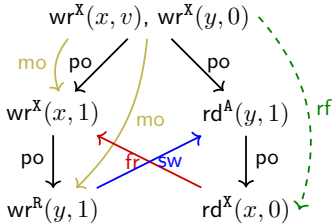
$$y = 0$$
$$[x] :\overset{\mathtt{X}}{=} 1; \;\Big\|\; r :\overset{\mathtt{X}}{=} [y];$$
$$[y] :\overset{\mathtt{X}}{=} 1 \;\Big\|\; s :\overset{\mathtt{X}}{=} [x]$$



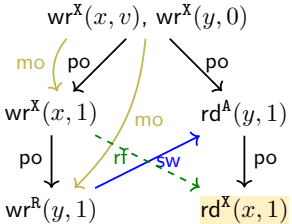
Allowed

Release-acquire

$$y = 0$$
$$[x] :\overset{\mathtt{X}}{=} 1; \;\Big\|\; r :\overset{\mathtt{A}}{=} [y];$$
$$[y] :\overset{\mathtt{R}}{=} 1 \;\Big\|\; s :\overset{\mathtt{X}}{=} [x]$$



Disallowed



Allowed

RA Synchronisation (Operationally)

Recall. Write rule models both relaxed and releasing writes

$$\frac{\begin{array}{c}\text{WRITE-RLX-REL}\\ e = \langle x, \mathsf{write}(v), \mathtt{M}\rangle \qquad V' = \mathcal{V}(\tau)[x \mapsto t]\\ \mathcal{V}(\tau)(x) < t \qquad t \notin \mathsf{ts}(m_{|x})\\ \mu = \langle x, v, t, V', \mathtt{M}\rangle\end{array}}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m \cup \{\mu\}, \mathcal{V}[\tau \mapsto V']\rangle}$$

Key ideas.

- Each message records the *view of the executing thread* at the time of writing
- The view indicates the thread's *happens-before knowledge* (at the time of writing)
- Later if RA synchronisation occurs,

  view transfer $\Leftrightarrow$ happens-before transfer

RA Synchronisation

Write transition: Recall $\mathtt{M} \in \{\mathtt{X}, \mathtt{R}\}$

$$\text{WRITE-RLX-REL}\quad \frac{e = \langle x, \mathsf{write}(v), \mathtt{M}\rangle \qquad V' = \mathcal{V}(\tau)[x \mapsto t] \qquad \mathcal{V}(\tau)(x) < t \qquad t \notin \mathsf{ts}(m_{|x}) \qquad \mu = \langle x, v, t, V', \mathtt{M}\rangle}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m \cup \{\mu\}, \mathcal{V}[\tau \mapsto V']\rangle}$$

Acquiring reads:

$$\text{ACQ-RF-RLX}\quad \frac{e = \langle x, \mathsf{read}(v), \mathtt{A}\rangle \qquad \langle x, v, t, \_, \mathtt{X}\rangle \in m \qquad \mathcal{V}(\tau)(x) \leq t \qquad V' = \mathcal{V}(\tau)[x \mapsto t]}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

$$\text{ACQ-RF-REL}\quad \frac{e = \langle x, \mathsf{read}(v), \mathtt{A}\rangle \qquad \langle x, v, t, V, \mathtt{R}\rangle \in m \qquad \mathcal{V}(\tau)(x) \leq t \qquad V' = \mathcal{V}(\tau) \sqcup V}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

where $V_1 \sqcup V_2 \;\widehat{=}\; \lambda x \in \mathit{Var}.\, \max\{V_1(x), V_2(x)\}$

RA in Isabelle

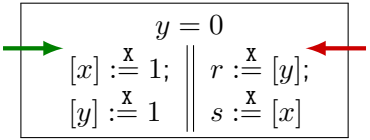
$$\text{ACQ-RF-RLX}\quad \frac{e = \langle x, \mathsf{read}(v), \mathtt{A}\rangle \qquad \langle x, v, t, \_, \mathtt{X}\rangle \in m \qquad \mathcal{V}(\tau)(x) \leq t \qquad V' = \mathcal{V}(\tau)[x \mapsto t]}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

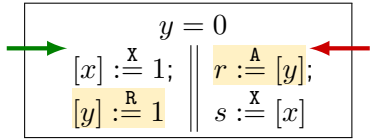
$$\text{ACQ-RF-REL}\quad \frac{e = \langle x, \mathsf{read}(v), \mathtt{A}\rangle \qquad \langle x, v, t, V, \mathtt{R}\rangle \in m \qquad \mathcal{V}(\tau)(x) \leq t \qquad V' = \mathcal{V}(\tau) \sqcup V}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

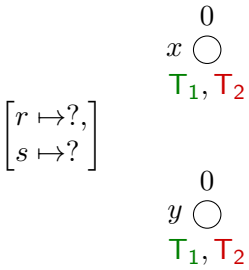
where $V_1 \sqcup V_2 \;\widehat{=}\; \lambda x \in Var.\, \max\{V_1(x), V_2(x)\}$

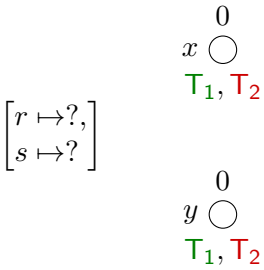
```
definition maxView:: "... w" ("_ ⊔ _" [100, 100]) where
"maxView V1 V2 ≡ λ x. max (V1 x) (V2 x)"

definition
"readAcqTrans τ x μ σ ≡
let V' = if mord μ = rlx then (TView σ τ)(x := ts μ) else (view μ) ⊔ (TView σ τ) in
σ ;; updateTView τ V'"
```

Recap: Relaxed Message-Passing Execution

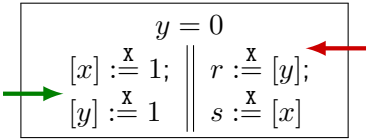


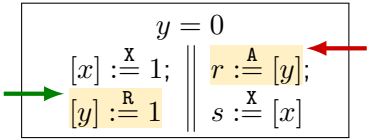


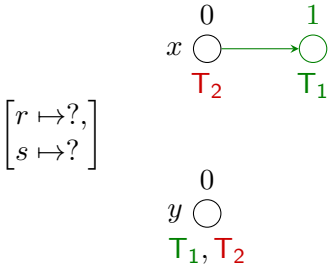


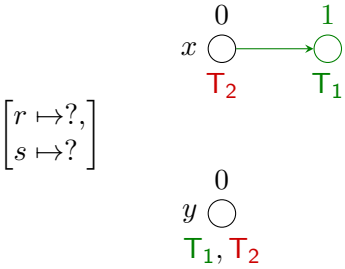


Recap: Relaxed Message-Passing Execution

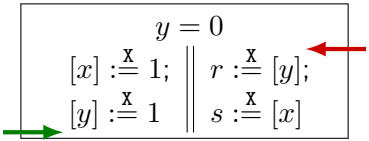


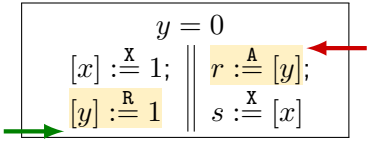


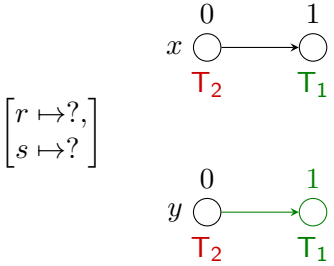


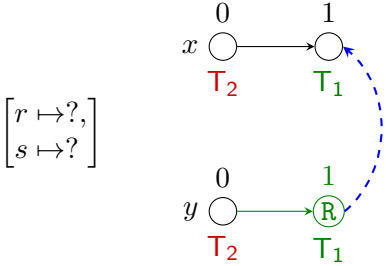


Recap: Relaxed Message-Passing Execution

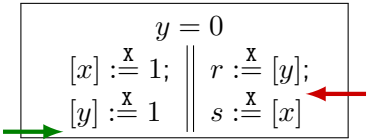


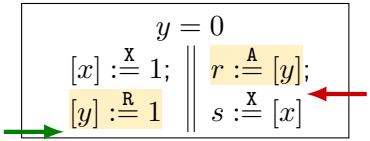


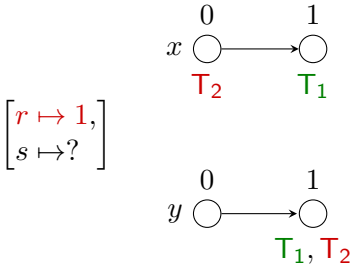


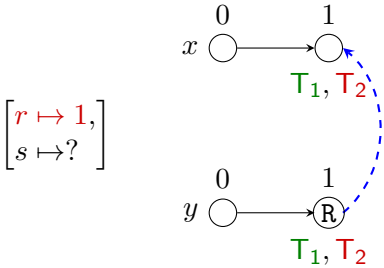


Recap: Relaxed Message-Passing Execution

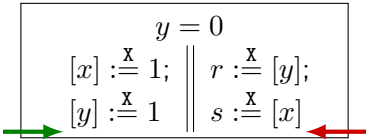


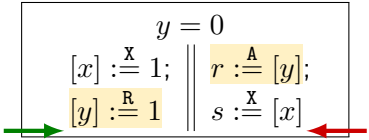


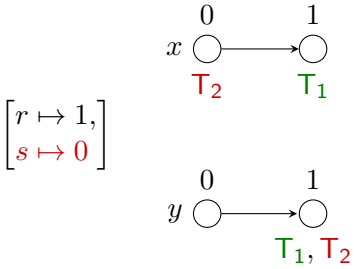


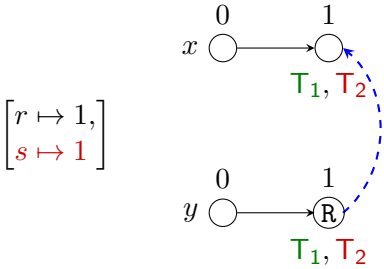


Recap: Relaxed Message-Passing Execution





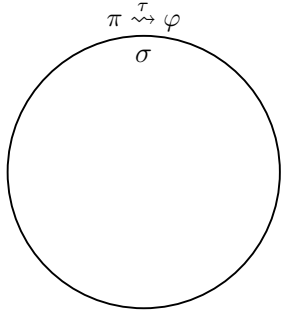




View-Transfer Assertion

Use dynamic **modal** predicate $\pi \overset{\tau}{\rightsquigarrow} \varphi$

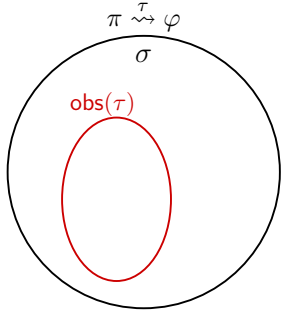
- π is predicate over messsages (a read filter)
- all messages observable to τ satisfying π are releasing
- performing an acquiring read when the read trigger holds guarantees φ after the read



View-Transfer Assertion

Use dynamic **modal** predicate $\pi \overset{\tau}{\rightsquigarrow} \varphi$

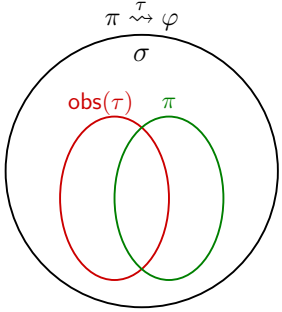
- π is predicate over messsages (a read filter)
- all messages observable to τ satisfying π are releasing
- performing an acquiring read when the read trigger holds guarantees φ after the read



View-Transfer Assertion

Use dynamic **modal** predicate $\pi \overset{\tau}{\rightsquigarrow} \varphi$

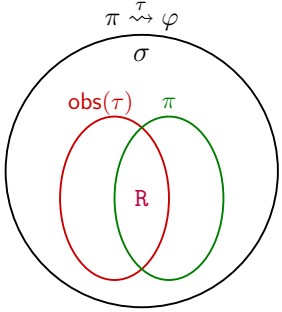
- π is predicate over messsages (a read filter)
- all messages observable to τ satisfying π are releasing
- performing an acquiring read when the read trigger holds guarantees φ after the read



View-Transfer Assertion

Use dynamic **modal** predicate $\pi \overset{\tau}{\rightsquigarrow} \varphi$

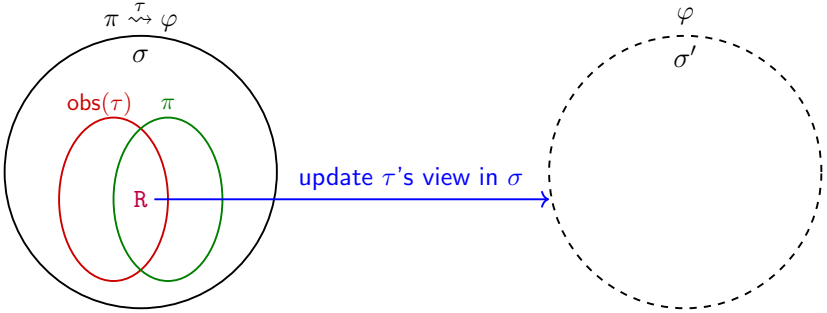
- π is predicate over messsages (a read filter)
- all messages observable to τ satisfying π are releasing
- performing an acquiring read when the read trigger holds guarantees φ after the read



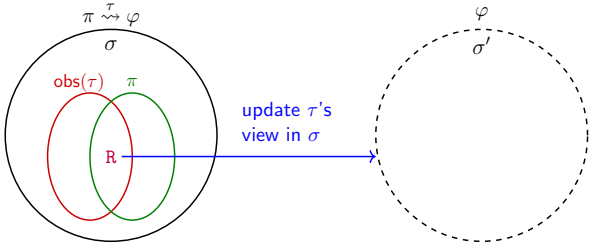
View-Transfer Assertion

Use dynamic **modal** predicate $\pi \overset{\tau}{\rightsquigarrow} \varphi$

- π is predicate over messsages (a read filter)
- all messages observable to τ satisfying π are releasing
- performing an acquiring read when the read trigger holds guarantees φ after the read



Formalising the View-Transfer Assertion



ACQ-RF-REL

$$\frac{e = \langle x, \mathsf{read}(v), \mathtt{A}\rangle \qquad \langle x, v, t, V, \mathtt{R}\rangle \in m \qquad \mathcal{V}(\tau)(x) \leq t \qquad V' = \mathcal{V}(\tau) \sqcup V}{\langle m, \mathcal{V}\rangle \xrightarrow{e}_{\tau} \langle m, \mathcal{V}[\tau \mapsto V']\rangle}$$

Construct the intersection:

$$\mathsf{filt}(\tau, \pi) \mathrel{\widehat{=}} \lambda\alpha.\ \{\mu \in \mathsf{obs}(\tau)(\alpha) \mid \pi(\mu)\}$$

Define modal view transfer:

$$\pi \overset{\tau}{\rightsquigarrow} \varphi \mathrel{\widehat{=}} \lambda\alpha.\ \textbf{let}\ \alpha = \langle\sigma, \langle m, \mathcal{V}\rangle\rangle\ \textbf{in}$$
$$\forall \mu \in \mathsf{filt}(\tau, \pi)(\alpha).\ \mathsf{ord}(\mu) = \mathtt{R} \wedge \varphi(\langle\sigma, \langle m, \mathcal{V}[\tau \mapsto \mathcal{V}(\tau) \sqcup \mathsf{view}(\mu)]\rangle\rangle)$$

Formalising the View-Transfer Assertion

Special case:

$$(x = \mathsf{xv}) \overset{\tau}{\rightsquigarrow} \varphi \quad \widehat{=} \quad (\lambda\mu.\ \mathsf{var}(\mu) = x \wedge \mathsf{val}(\mu) = \mathsf{xv}) \overset{\tau}{\rightsquigarrow} \varphi$$

Whenever τ acquire-reads xv for x, φ is guaranteed in the post state

Using the special case of view-transfer is easy

ACQ-VIEW-TRANSFER

$$\frac{}{\left\{x = \mathsf{xv} \overset{\tau}{\rightsquigarrow} \varphi\right\}\tau \rhd \mathtt{r} :\overset{\mathtt{A}}{=} x\{\mathtt{r} = \mathsf{xv} \Rightarrow \varphi\}}$$

```
lemma
"(x = xv) ⇝τ φ σ
  ⇒ σ τ ▷ r :=A [x] σ'
  ⇒ r = xv → φ σ'"
by (auto simp add: unfold_defs)
```

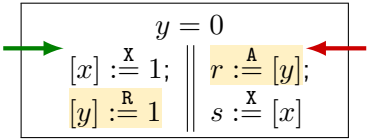
Establishing view-transfer is not

REL-DEF-VIEW-SET

$$\frac{\forall i \in S.\ x \neq y_i}{\left\{\mathsf{xv} \notin |\tau'\rangle x \wedge (\forall i \in S. y_i \overset{\tau}{=} \mathsf{yv}_i)\right\}\tau \rhd x :\overset{\mathtt{R}}{=} \mathsf{xv}\ \left\{x = \mathsf{xv} \overset{\tau'}{\rightsquigarrow} (\forall i \in S. y_i \overset{\tau'}{=} \mathsf{yv}_i)\right\}}$$

```
lemma
"wfs σ ⇒ x ≠ y ⇒ v ∉ |τ'⟩x σ ⇒ y =τ u σ
⇒ σ τ ▷ [x] :=R v σ'
⇒ (x = v) ⇝τ' (λ σ. y =τ' u σ) σ'"
proof
(approx 30 lines)
```

View-Transfer in Action

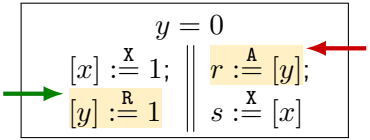


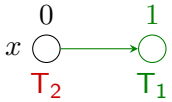
$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto ? \end{bmatrix}$$

$$\begin{matrix} 0 \\ x \bigcirc \\ \mathsf{T}_1, \mathsf{T}_2 \end{matrix}$$

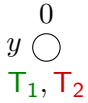
$$\begin{matrix} 0 \\ y \bigcirc \\ \mathsf{T}_1, \mathsf{T}_2 \end{matrix}$$

View-Transfer in Action

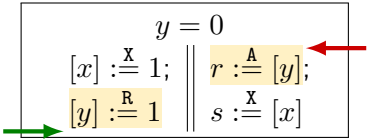




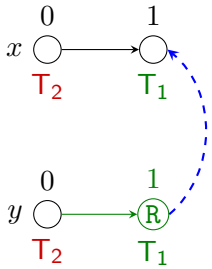
$$\begin{bmatrix} r \mapsto ?, \\ s \mapsto ? \end{bmatrix}$$



View-Transfer in Action



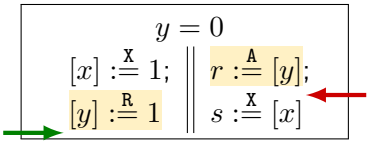
$\begin{bmatrix} r \mapsto ?, \\ s \mapsto ? \end{bmatrix}$



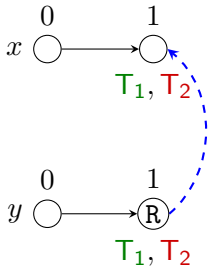
T_1 establishes

$y = 1 \overset{\mathsf{T}_2}{\rightsquigarrow} (x \overset{\mathsf{T}_2}{=} 1)$

View-Transfer in Action



$\begin{bmatrix} r \mapsto 1, \\ s \mapsto ? \end{bmatrix}$



T_2 uses

$y = 1 \overset{\mathsf{T}_2}{\rightsquigarrow} (x \overset{\mathsf{T}_2}{=} 1)$ to establish

$x \overset{\mathsf{T}_2}{=} 1$

View-Transfer in Action

$$\{x \langle\!\langle \mathsf{T}_1 \ \wedge \ 1 \notin |\mathsf{T}_2\rangle y\}$$

Thread T_1	Thread T_2
$\{1 \notin \|\mathsf{T}_2\rangle y \ \wedge \ x \langle\!\langle \mathsf{T}_1\}$	$\{y = 1 \overset{\mathsf{T}_2}{\rightsquigarrow} (x \overset{\mathsf{T}_2}{=} 1)\}$
$[x] :\overset{\mathtt{X}}{=} 1;$	$r :\overset{\mathtt{A}}{=} [y];$
$\{1 \notin \|\mathsf{T}_2\rangle y \ \wedge \ x \overset{\mathsf{T}_1}{=} 1\}$	$\{r = 1 \Rightarrow x \overset{\mathsf{T}_2}{=} 1\}$
$[y] :\overset{\mathtt{R}}{=} 1$	$s :\overset{\mathtt{X}}{=} [x]$
$\{true\}$	$\{r = 1 \Rightarrow s = 1\}$

$$\{r = 1 \Rightarrow s = 1\}$$

View-Transfer in Action

Isabelle/HOL Demo

Questions?

1. Background
   - Sequential consistency
   - Owicki-Gries reasoning
   - Isabelle/HOL
   - Examples: Message passing, Lamport buffer
2. Relaxed Memory
   - Motivation
   - RC11 Memory Model (Relaxed Accesses)
3. Verification
   - Owicki-Gries reasoning, revisited
   - Examples, revisited
   - Isabelle/HOL
4. Formal Model
5. Release/Acquire Accesses
   - Lamport buffer, revisited again
   - Message passing
   - Isabelle/HOL

References I


Adve, S. V. and Gharachorloo, K. (1996).
Shared memory consistency models: A tutorial.
IEEE Computer, 29(12):66–76.


Alglave, J., Maranget, L., and Tautschnig, M. (2014).
Herding cats: Modelling, simulation, testing, and data mining for weak memory.
ACM Trans. Program. Lang. Syst., 36(2):7:1–7:74.


Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. (2011).
Mathematizing C++ concurrency.
In Ball, T. and Sagiv, M., editors, *POPL*, pages 55–66. ACM.


Castañeda, A., Chockler, G., Dongol, B., and Lahav, O. (2024).
What Cannot Be Implemented on Weak Memory?
In Alistarh, D., editor, *DISC*, volume 319 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:22, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.


de Roever, W. P., de Boer, F. S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., and Zwiers, J. (2001).
Concurrency Verification: Introduction to Compositional and Noncompositional Methods, volume 54 of *Cambridge Tracts in Theoretical Computer Science*.
Cambridge University Press.

References II


Kaiser, J., Dang, H., Dreyer, D., Lahav, O., and Vafeiadis, V. (2017).
Strong logic for weak memory: Reasoning about release-acquire consistency in iris.
In *ECOOP*, pages 17:1–17:29.


Kang, J., Hur, C., Lahav, O., Vafeiadis, V., and Dreyer, D. (2017).
A promising semantics for relaxed-memory concurrency.
In *POPL*, pages 175–189. ACM.


Lahav, O., Vafeiadis, V., Kang, J., Hur, C., and Dreyer, D. (2017).
Repairing sequential consistency in C/C++11.
In Cohen, A. and Vechev, M. T., editors, *PLDI*, pages 618–632. ACM.


Lamport, L. (1979).
How to make a multiprocessor computer that correctly executes multiprocess programs.
IEEE Trans. Computers, 28(9):690–691.


Owicki, S. S. (1975).
Axiomatic Proof Techniques for Parallel Programs.
Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York.


Steinke, R. C. and Nutt, G. J. (2004).
A unified theory of shared memory consistency.
J. ACM, 51(5):800–849.