Taming Release-Acquire Consistency

Ori Lahav Nick Giannarakis Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

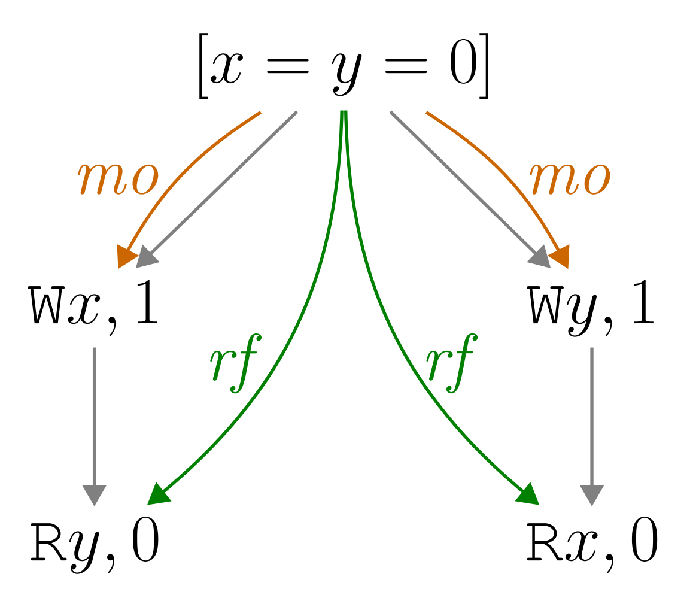
Max Planck Institute for Software Systems

C11's release/acquire declarative memory model

Store buffering

$x = y = 0$	
$x := 1;$ `print` y	$y := 1;$ `print` x

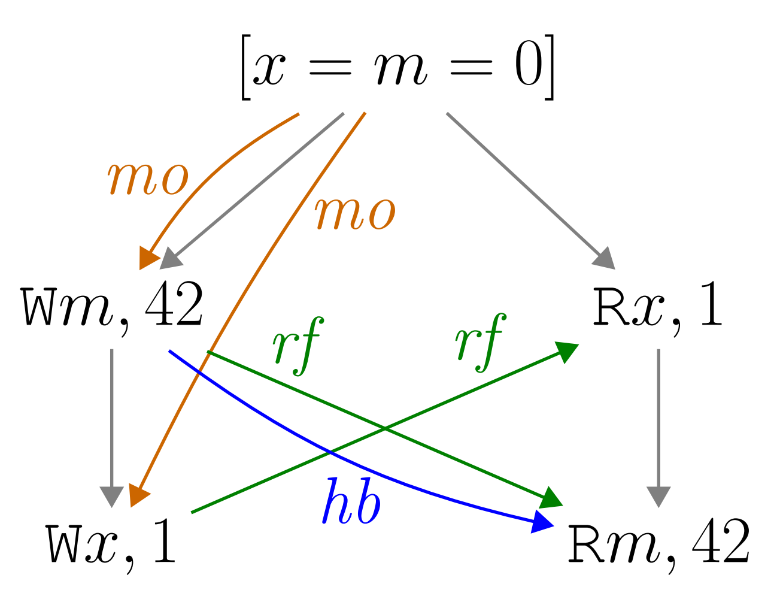
both threads may print 0



Message passing

$x = 0$	
$m := 42;$ $x := 1$	`while` $x = 0$ `skip;` `print` m

only 42 may be printed



RA model: C11 model where all reads are acquire, all writes are release, and all atomic updates are acquire/release

Good balance between performance and programmability:

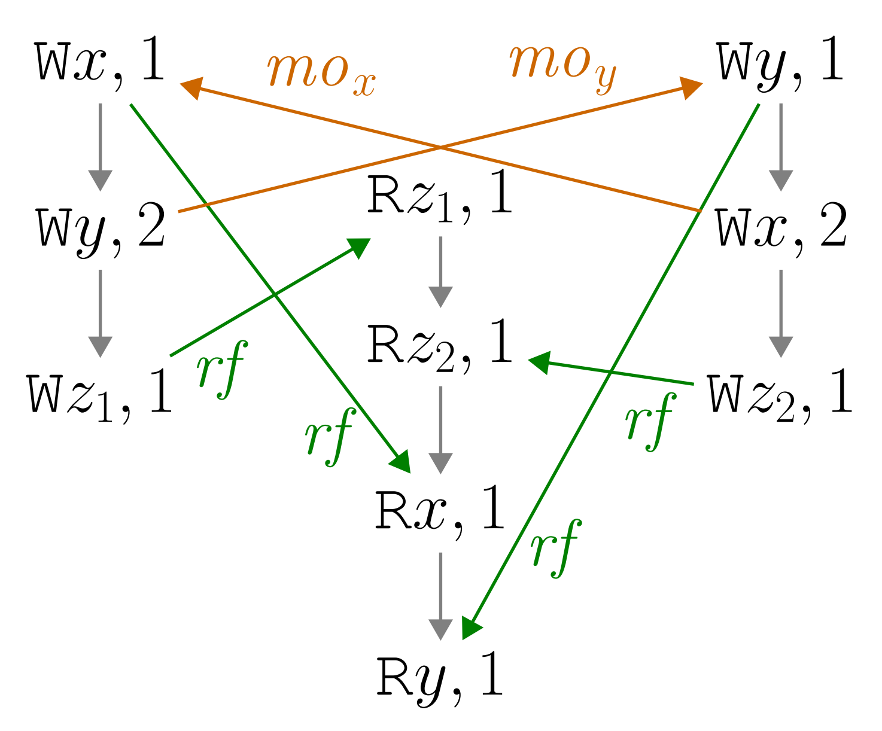
- Supports intended hardware/compiler optimizations:
  - Elimination of redundant adjacent accesses
  - Store-load reordering: $\mathtt{W}x \to \mathtt{R}y \rightsquigarrow \mathtt{R}y \to \mathtt{W}x$ (unlike SC)
- DRF theorem:
  No data races under SC ensures no weak behaviors
- Monotonicity:
  Adding synchronization does not introduce behaviors (unlike TSO)
- Verified compilation schemes for x86-TSO and Power
- Program logics: RSL, GPS, OGRA

Strong release/acquire

Problem: Some behaviors allowed by RA are never observed.

$x := 1;$ $y := 2;$ $z_1 := 1$	`print` z_1; `print` z_2; `print` x; `print` y	$y := 1;$ $x := 2;$ $z_2 := 1$

C11 allows printing $1, 1, 1, 1$.



Proposed solution: Rule out $hb \cup mo$ cycles.

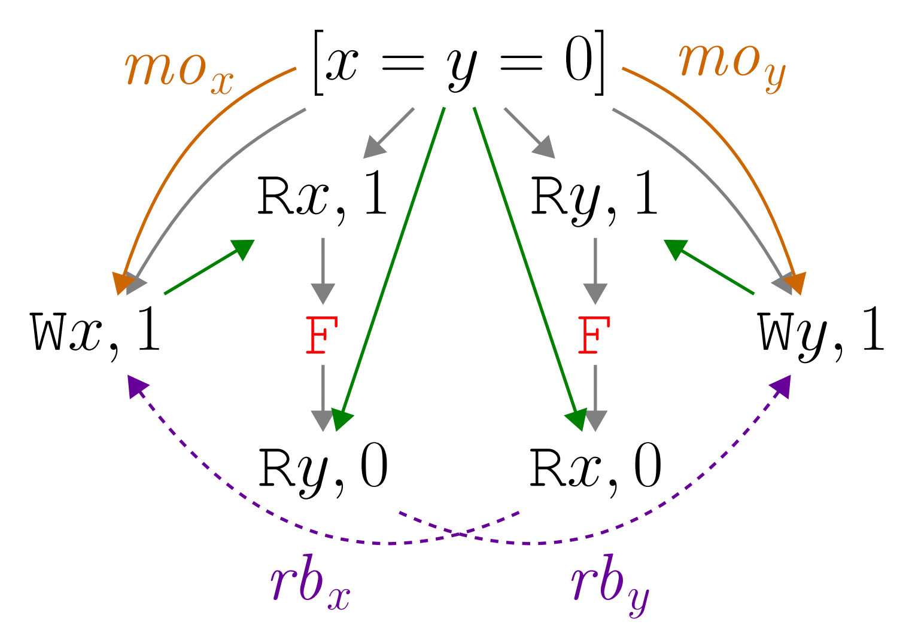
- No additional cost:
  - Compilation schemes are not affected.
  - Same compiler optimizations are sound.
- No better deal for Power:
  Power model restricted to RA accesses = strong RA

SC-fences

Problem: SC-fences are overly weak.

$x = y = 0$			
$x := 1$	`print` x; `fence()`; `print` y	`print` y; `fence()`; `print` x	$y := 1$

C11 allows both threads to print $1, 0$.



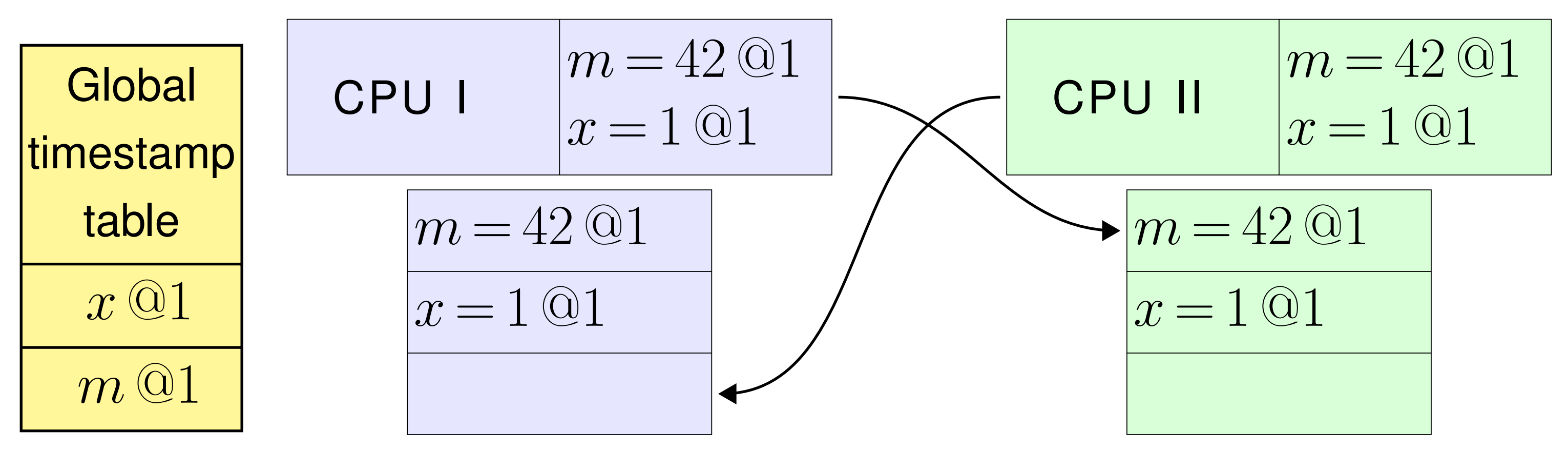
Proposed solution: Model SC-fences as atomic updates of a distinguished fence location.

- RA semantics enforces all fence events to be ordered by hb.
- Compilation schemes are not affected.
- Adding fences to guarantee SC:
  - Between every two racy accesses
  - Between racy writes and racy reads for *client-server programs*

Operational semantics

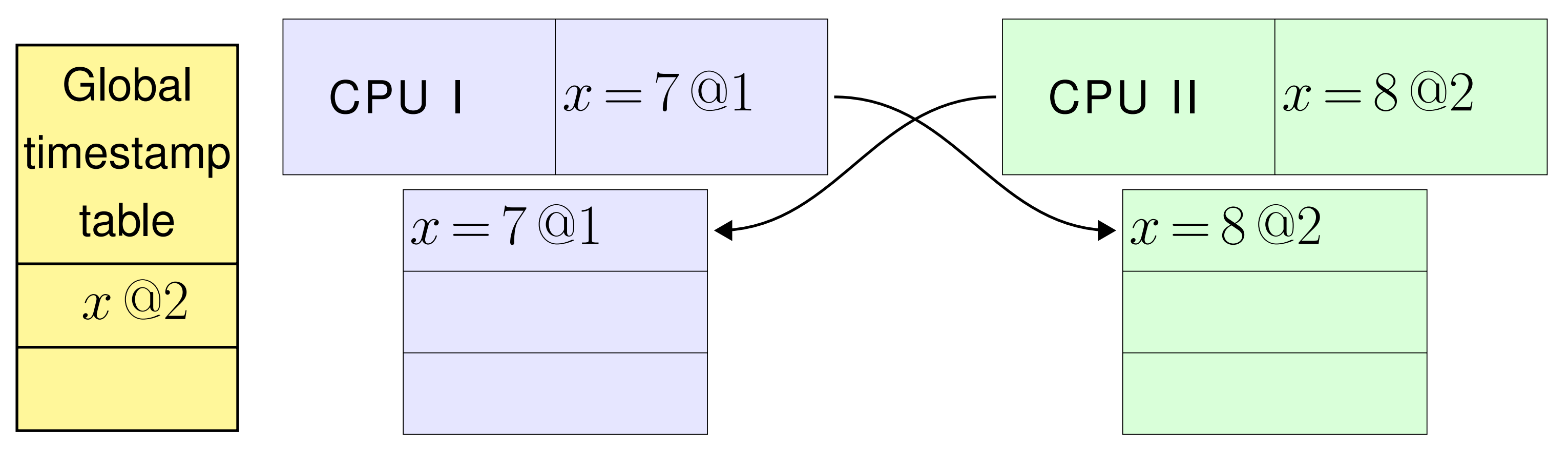
- Based on point-to-point communication.
- Each processor has a local memory and an outgoing message buffer.
- Processors non-deterministically choose between performing their own commands and processing incoming messages.
- Messages are processed in the order they were issued.
- Processing a message updates the local memory and adds the message to the outgoing buffer.
- Global timestamps are used to ensure coherence properties:
  - Every write obtains a new timestamp, larger than all previous ones.
  - When processing a message, the local memory is updated only if the message's timestamp is larger than the stored one.

$m = x = 0$	
$m := 42;$ $x := 1;$ ▶ ...	`while` $x = 0$ `skip;` ▶ `print` m



$x := 7;$ ▶ `print` x	$x := 8;$ ▶ `print` x

If the first thread prints 8,
the second thread cannot print 7


Global timestamp table
x @2
CPU I
x = 7 @1
x = 7 @1
CPU II
x = 8 @2
x = 8 @2