PURDUE UNIVERSITY

Principles of Concurrency

Week 7

Hardware Memory Models

Memory Models

- A memory model is a characterization of how memory governs the visibility of writes
  - *When should the effect of a write become visible to a subsequent read?*
- In the simplest case, writes become visible the instant they are performed
  - Sequential consistency
- In practice, however, processor microarchitectures and programming language definitions often provide weaker guarantees
  - Relaxed (or Weak) Memory

Store Buffering

The behavior of Intel and AMD processors admits the following execution

SB

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

Can this behavior be realized in an SC execution?

How might this behavior be captured operationally?

Store Buffering

Architectural design choices can confound intuitive expectations of higher-level properties

- e.g., causality

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[x] MOV EBX←[y]	MOV [y]←2 MOV [x]←2
Allowed Final State: Proc 0:EAX=1 ∧ Proc 0:EBX=0 ∧ [x]=1	

This execution observable on Intel multicore processors.
Does it preserve causal ordering on memory accesses?

Store Buffering

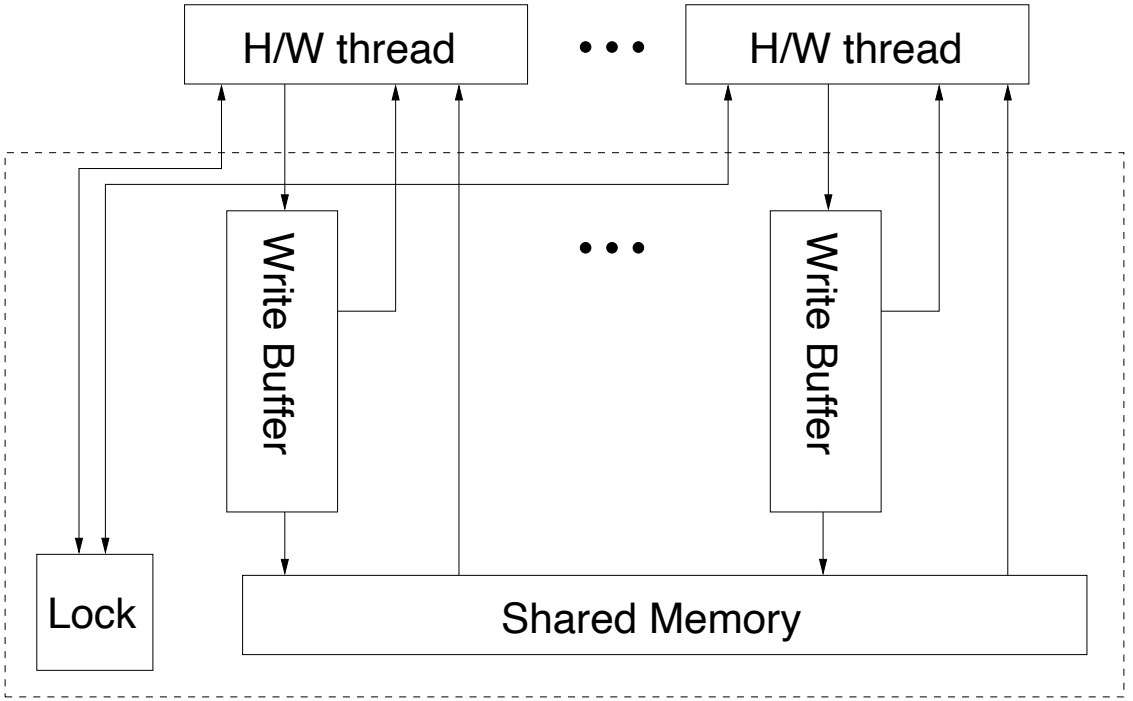
On the other hand, some behaviors make stronger assumptions on processor behavior

IRIW

Proc 0	Proc 1	Proc 2	Proc 3
MOV [x]←1	MOV [y]←1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Forbidden Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0			

What kind of semantics should we ascribe to how processors and memory interact with one another that forbids this example, but allows the earlier one?

x86-TSO



- Each processor manages its own local store buffer
- Other than this buffer, all processors share the same view of memory
- No other aspect of the microarchitecture (caching, speculation, etc.) are visible to the programmer

Semantics

1. A thread can write into its buffer at any time;
2. A thread can read from the newest write to an address from its store buffer, if there is one and it is not blocked by another LOCKed instruction in progress;
3. A thread can read from memory if there are no writes to that address in its store buffer and it is not blocked by another LOCKed instruction in progress;
4. A thread can silently dequeue the oldest write from its store buffer and place the value in memory whenever it is not blocked by another LOCKed instruction in progress;
5. A thread can execute an MFENCE whenever its store buffer is empty (otherwise, it must use the previous transition);
6. A thread can begin a LOCKd instruction if no other is in progress; and a thread can end a LOCKed instruction if its store buffer is empty.

Total Store Order: All threads see writes to memory in the same order

Rules and Litmus Tests

Reordering

Proc 0	Proc 1
MOV [x]←1 MOV [y]←1	MOV EAX←[y] MOV EBX←[x]
Forbidden Final State: Proc 1:EAX=1 ∧ Proc 1:EBX=0	

(a) No store-store reordering

Proc 0	Proc 1
MOV EAX←[x] MOV [y]←1	MOV EBX←[y] MOV [x]←1
Forbidden Final State: Proc 0:EAX=1 ∧ Proc 1:EBX=1	

(b) No load-store reordering

SB

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

(c) Store-Load reordering on indepedent locations allowed

Proc 0
MOV [x]←1 MOV EAX←[x]
Required Final State: Proc 0:EAX=1

(d) No store-load reordering on same location

Rules and Litmus Tests

Consistency and Visibility

Proc 0	Proc 1	Proc 2
MOV [x]←1	MOV EAX←[x] MOV [y]←1	MOV EBX←[y] MOV ECX←[x]
Forbidden Final State: Proc 1:EAX=1 ∧ Proc 2:EBX=1 ∧ Proc 2:ECX=0		

(e) Transitive visibility of stores

IRIW

Proc 0	Proc 1	Proc 2	Proc 3
MOV [x]←1	MOV [y]←1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Forbidden Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0			

(f) Stores consistent across all processors (No IRIW)

Locks

Proc 0	Proc 1
XCHG [x]←EAX MOV EBX←[y]	XCHG [y]←ECX MOV EDX←[x]
Initial state: Proc 0:EAX=1 ∧ Proc 1:ECX=1 (elsewhere 0)	
Forbidden Final State: Proc 0:EBX=0 ∧ Proc 1:EDX=0	

(g) Loads never reordered with locks

Proc 0	Proc 1
XCHG [x]←EAX MOV [y]←1	MOV EBX←[y] MOV ECX←[x]
Initial state: Proc 0:EAX=1 (elsewhere 0)	
Forbidden Final State: Proc 1:EBX=1 ∧ Proc 1:ECX=0	

(h) Stores never reordered with locks

Data Races

The "fundamental" property of a relaxed memory model:

> DRF programs executing in a relaxed memory setting are SC

- Whenever there is a visible temporal dependency among operations, the memory model ensures that the dependency is globally respected
  - system always maintains a consistent view of their ordering
- Any actions that could potentially induce an inconsistency must have a temporal dependency

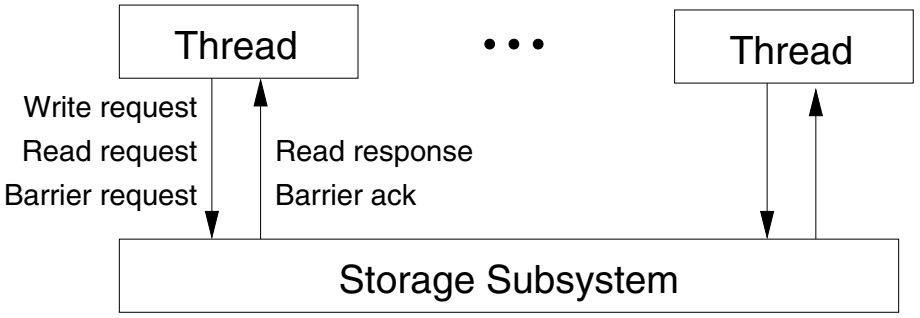
Triangular Data Races

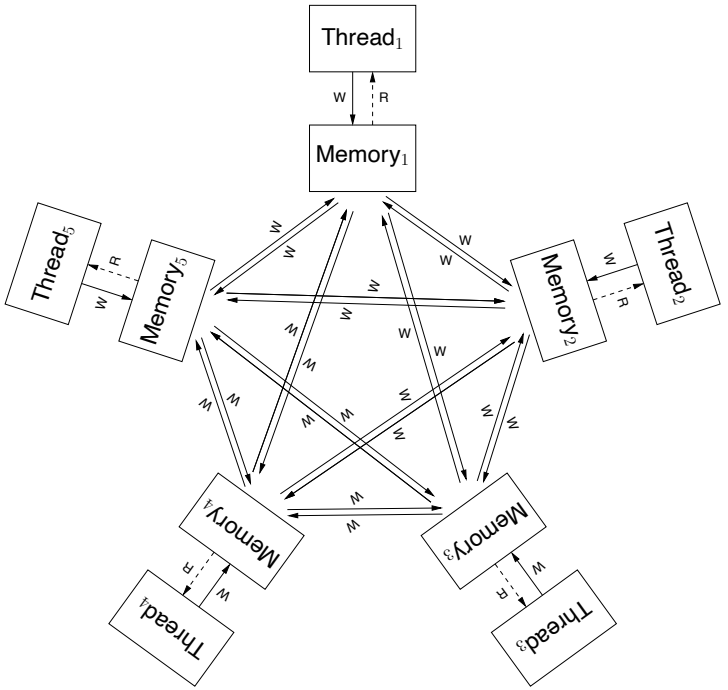
- We can enforce SC behavior in a TSO machine by flushing every write after it's written to a processor's local buffer
  - store buffer is ignored, preventing behaviors expressible as reorderings that are not possible in any SC execution of the original program
- A triangular data race occurs if there is a data race between a read R_x^{T1} and write W_x^{T2} and there is a preceding write to a potentially different variable by T1, W_y^{T1} that is not separated from the read by a lock or fence
  - Intuition: the preceding write may be in T1's store buffer and can thus induce non-SC behaviors
  - Theorem: a program that is triangular race-free is sequentially consistent

The IBM Power Memory Model

Highly relaxed, significantly more behaviors than possible under TSO

- Hardware threads can each perform reads and writes out-of-order, or even speculatively
- Arbitrary local reordering is allowed
- Does not support multi-copy atomicity: a write issued by a processor is not guaranteed to be visible to all other threads at the same time

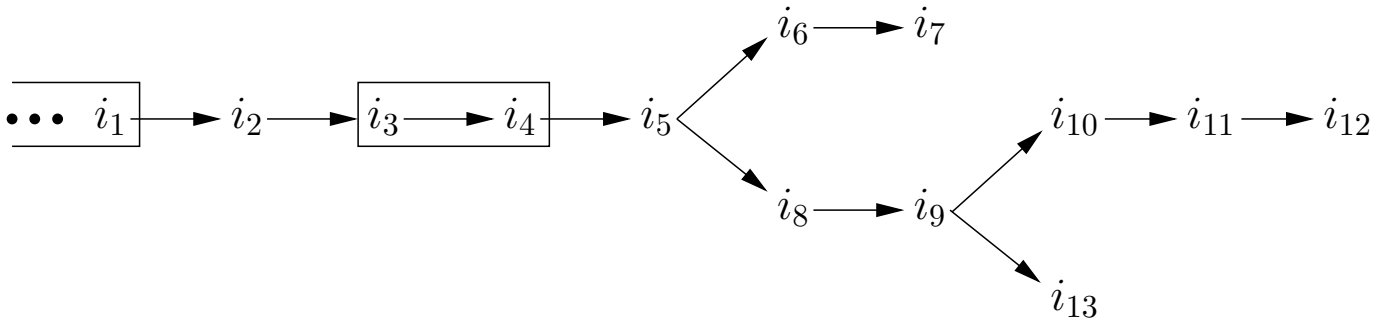




Material from: Understanding POWER Multiprocessors, Sarkar et. al (PLDI'11)

Operational Model

Each thread, at each step in time, maintains a tree of committed and in-flight instruction instances



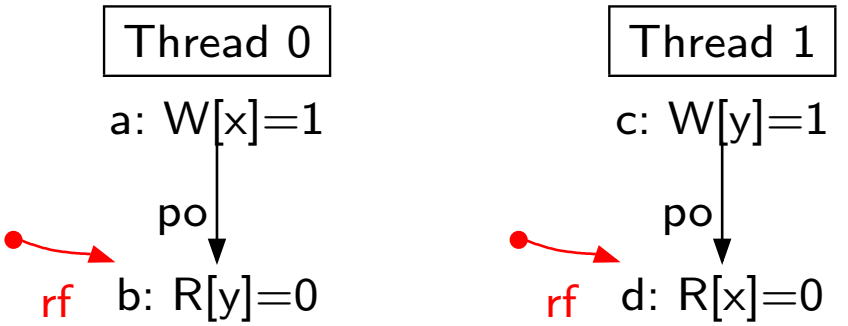
Instruction i5 and i9 are branches for which the thread has multiple possible successors

When a branch is committed, all alternative paths are discarded

Actions become committed when the relevant address and value are determined ("satisfied" for reads, "committed" for writes)

Test Execution Diagrams

Thread 0	Thread 1
`x=1` `r1=y`	`y=1` `r2=x`
Initial shared state: `x=0` and `y=0`	
Allowed final state: `r1=0` and `r2=0`	



Relations:

- `po`: **program order**
- `rf`: **reads-from**

https://www.cl.cam.ac.uk/~pes20/ppcmem/

Message-Passing

MP-loop Pseudocode

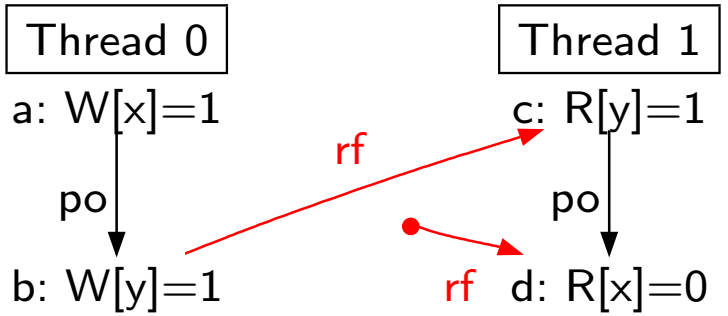
Thread 0	Thread 1

x=1 // write data
y=1 // write flag
	while (y==0) {} // busy-wait for flag
	r2=x // read data

Initial state: x=0 ∧ y=0	
Forbidden?: Thread 1 register r2 = 0	

MP Pseudocode

Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: x=0 ∧ y=0	
Forbidden?: 1:r1=1 ∧ 1:r2=0	



Allowed because (a) writes by Thread 0 are to distinct addresses and can be committed out-of-order; (b) reads performed by Thread 1 can be satisfied out-of-order

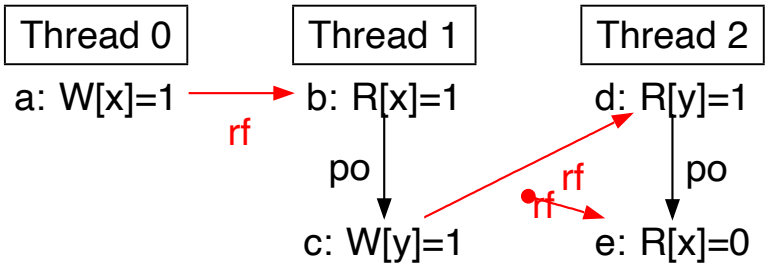
Is this outcome possible under SC?

Iterated Message-Passing

WRC

Pseudocode

Thread 0	Thread 1	Thread 2
x=1	r1=x y=1	r2=y r3=x
Initial state: x=0 ∧ y=0		
Allowed: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0		



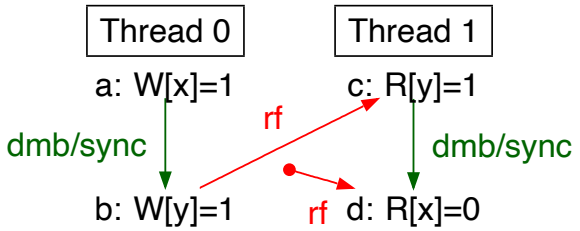
Test WRC: Allowed

Thread 1 reads and writes to different addresses and can be thus reordered

MP+dmb/syncs

Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y dmb/sync r2=x
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=1 ∧ 1:r2=0	



Test MP+dmbs/syncs: Forbidden

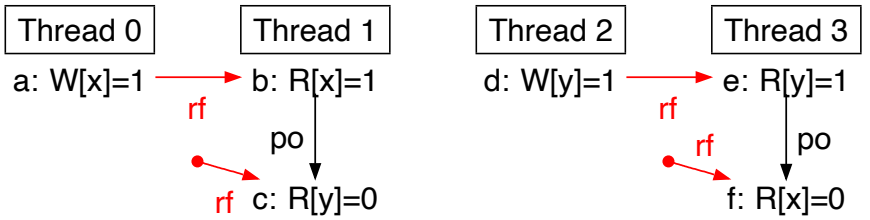
Memory fences (barriers) enforce ordering. Called "sync" on Power and "dmb" on ARM maintains local ordering and fixes propagation order to other threads

IRIW and Coherence Ordering

IRIW

Pseudocode

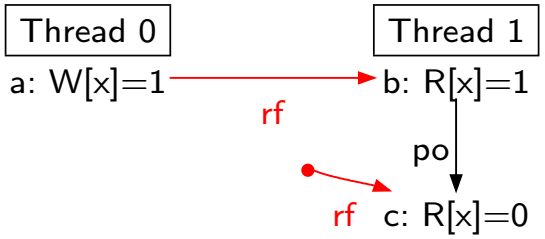
Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=y	y=1	r3=y r4=x
Initial state: x=0 ∧ y=0			
Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0			



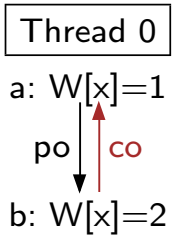
Test IRIW: Allowed

Writes can be propagated to different threads in different orders

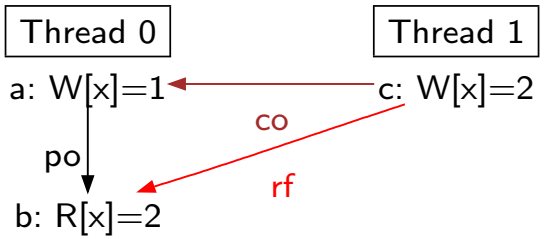
Coherence constraints



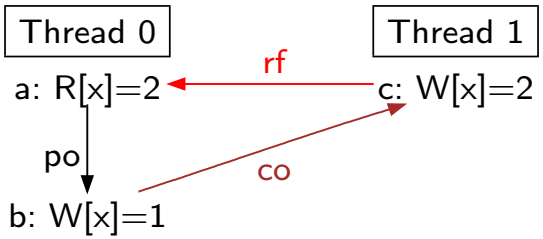
Test CoRR1 : Forbidden



Test CoWW : Forbidden



Test CoWR : Forbidden



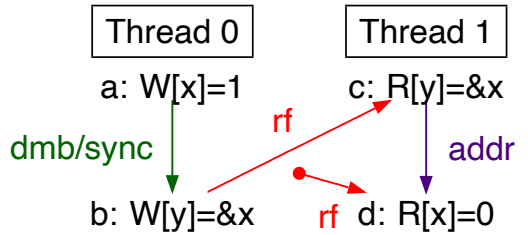
Test CoRW : Forbidden

Dependency Ordering

- Address dependency: value loaded by a read is used to compute the address used in a subsequent read or write

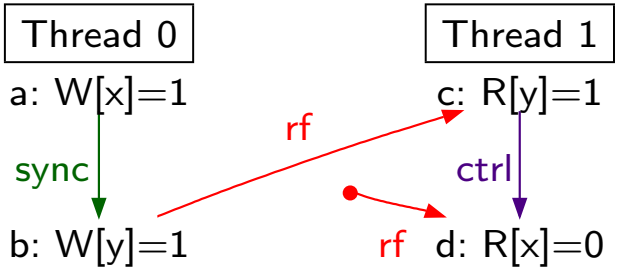
MP+dmb/sync+addr′ Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=&x	r1=y r2=*r1
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=&x ∧ 1:r2=0	



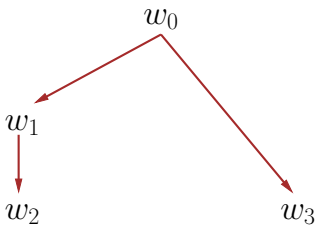
Test MP+dmb/sync+addr': Forbidden

- Control dependency: value loaded by a read is used to compute the value of a conditional that is program-order-before another read or write



Test MP+sync+ctrl : Allowed

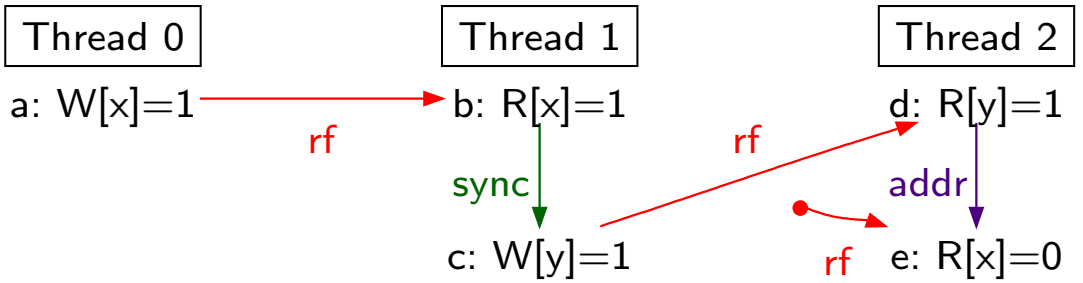
Coherence



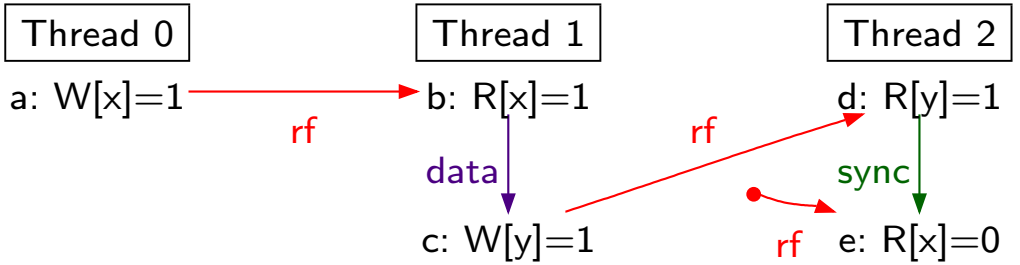
Given a read request by a thread t, what write could be sent in response? (Assume the only write known to have propagated to t is w1)

- it cannot be sent w_0, as w_0 is coherence-before the w_1 write that (because it is in the writes-propagated list) it might have read from;
- it could re-read from w_1, leaving the coherence constraint unchanged;
- it could be sent w_2, again leaving the coherence constraint unchanged, in which case w_2 must be appended to the events propagated to *tid*; or
- it could be sent w_3, again appending this to the events propagated to *tid*, which moreover entails committing to w_3 being coherence-after w_1, as in the coherence constraint on the right above. Note that this still leaves the relative order of w_2 and w_3 unconstrained, so another thread could be sent w_2 then w_3 or (in a different run) the other way around (or indeed just one, or neither).

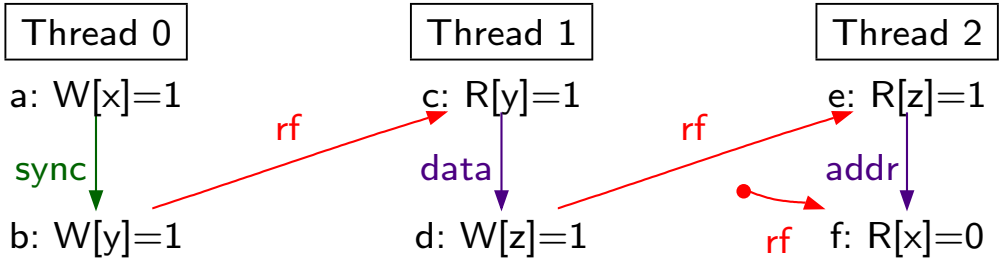
Cumulativity



Test WRC+sync+addr : Forbidden



Test WRC+data+sync : Allowed



Test ISA2+sync+data+addr : Forbidden