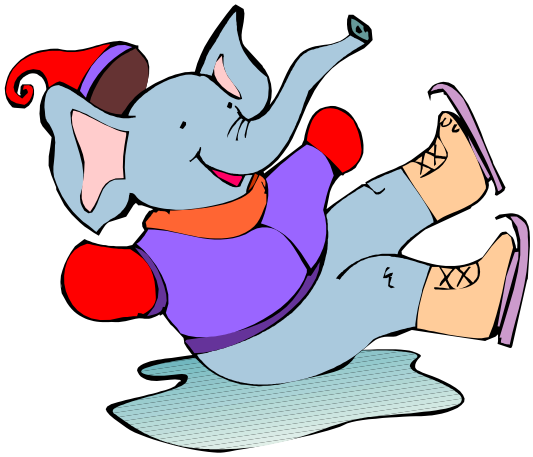
Relaxed Memory Models



1

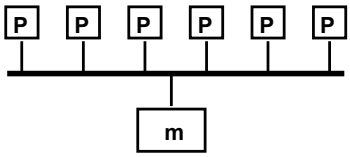
Episode III in our multiprocessing miniseries.

Relaxed memory models.

What I really wanted here was an elephant with sunglasses relaxing

On a beach, but I couldn’t find one.

Sequential Consistency: A Memory Model

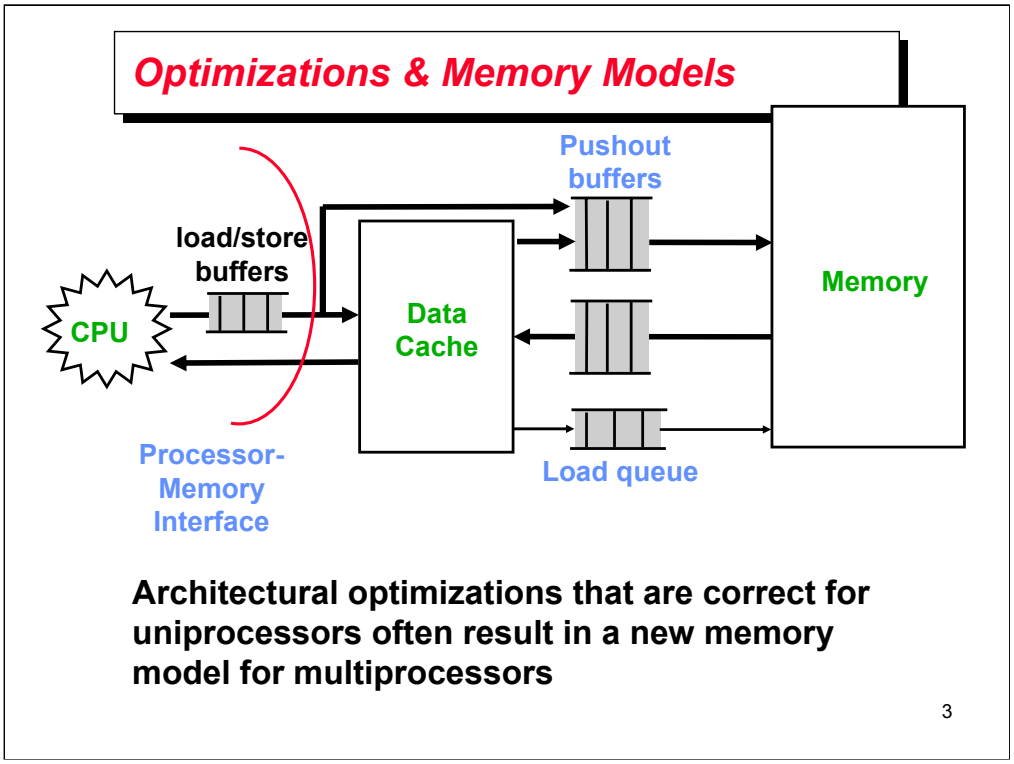


Sequential Consistency =
arbitrary *order-preserving interleaving*
of memory references of sequential programs

SC is easy to understand but architects and compiler writers want to violate it for performance

2

Mark Hill written a paper which essentially says “why break your back for 20%”. Actually people are out there breaking their backs for 1% in architecture these days.



This means that we are relaxing the ordering or relaxing atomicity.

Relaxed Models

- **What orderings among reads and writes performed by a *single* processor are preserved by the model?**
  - **R → R, R → W, W → W, W → R**

dependence if they are to the same address

- **If there is a dependence, then program semantics demand that operations be ordered**
- **If there is *no* dependence, the memory consistency model determines what orders must be preserved**
  - **Relaxed model may allow an operation executed later to complete first**

Memory Fences & Weak Memory Models

Processors with *relaxed or weak memory models* need *memory fence* instructions to force serialization of memory accesses

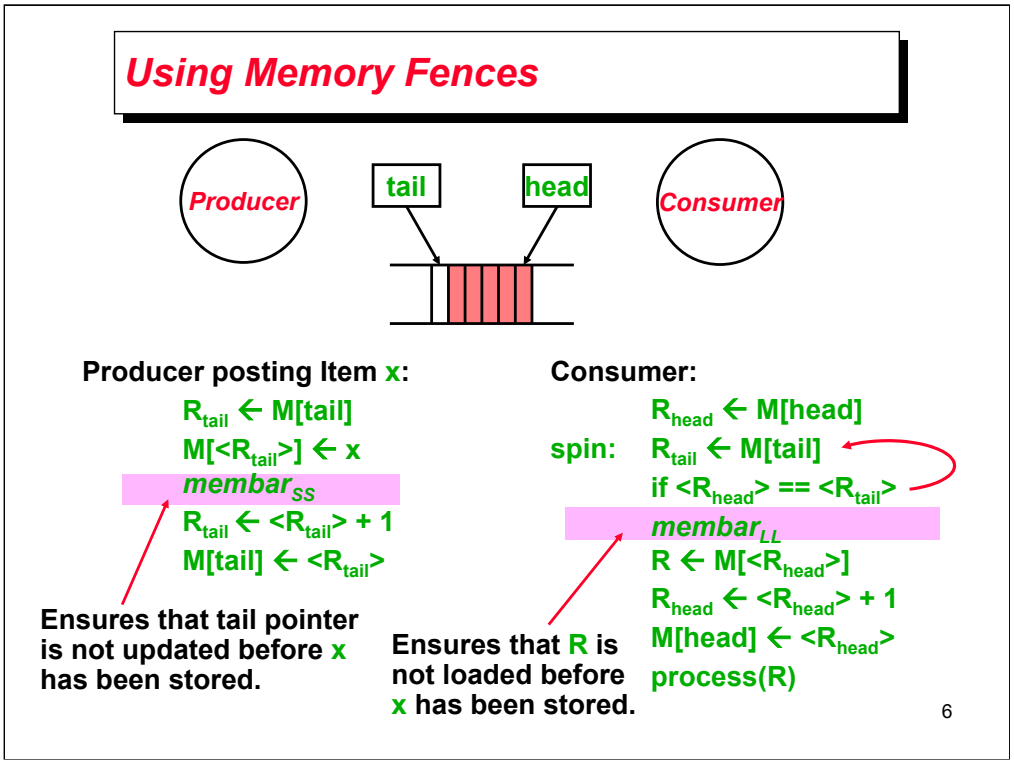
- **In SC there is an implicit fence at each memory operation**

Processors with relaxed memory models:

Sparc V8 (TSO, PSO): Membar
PowerPC (WO): Sync, EIEIO

Memory fences are expensive operations, however, one pays for serialization only when it is required

5



Ensures that tail pointer is not updated before X has been stored.

Ensures that R is not loaded before x has been stored.

Data-Race Free Programs
(a.k.a. Properly Synchronized Programs)

Process 1

```
...
Acquire(mutex);
  < critical section >
Release(mutex);
```

Process 2

```
...
Acquire(mutex);
  < critical section >
Release(mutex);
```

Synchronization variables (e.g., mutex) are separate from data variables

Accesses to writable shared data variables are protected in critical regions

⇒ no data races except for locks

(Formal definition is elusive)

In general, it cannot be proven if a program is data-race free.

7

Nondeterminator.

Fences in Data-Race Free Programs

Process 1

```
...
Acquire(mutex);
membar;
    < critical section >
membar;
Release(mutex);
```

Process 2

```
...
Acquire(mutex);
membar;
    < critical section >
membar;
Release(mutex);
```

Relaxed memory models allow reordering of instructions by the compiler or the processor as long as the reordering is not done across a fence

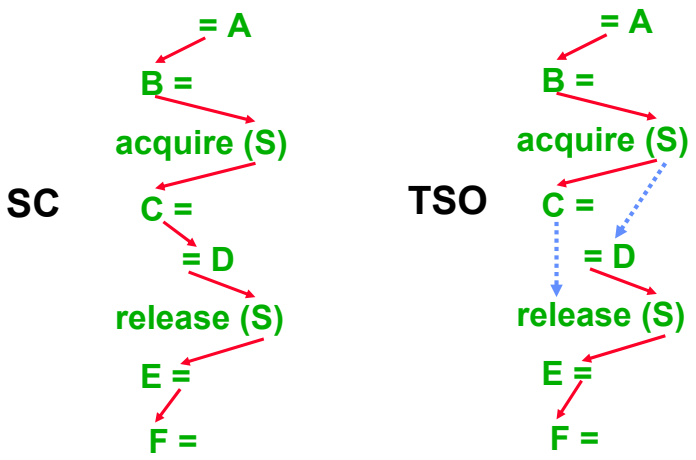
What about speculation and prefetching?

8

Processor should not speculate or prefetch across fences.

Total Store Order (TSO)
IBM370, DECVAX

- **Eliminates the order W(a) → R(b) a ≠ b**
- ***Advantage?***



9

Allows the buffering of writes with bypassing by reads, which occurs whenever the

processor allows a read to proceed before it guarantees that

an earlier write by the processor has been seen by all the other processors.

TSO vs. SC

Initially x = old, y = old

Processor P_1	**Processor P_2**
x = new;	**y = new;**
y_copy = y;	**x_copy = x;**

Under SC what values can* x_copy *and* y_copy *get ?

Under TSO what values can* x_copy *and* y_copy *get ?

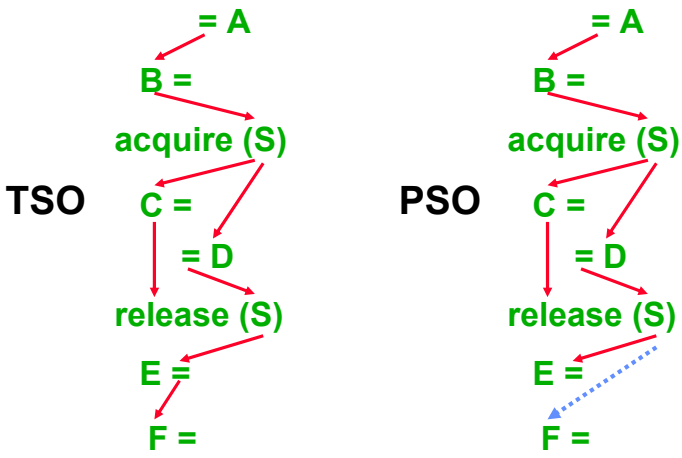
10

TSO both can get old values.

SC at least one has to get the value of new.

Partial Store Ordering (PSO)
SPARC

- **Also eliminates the order W(a) → W(b) a ≠ b**
- ***Advantage?***

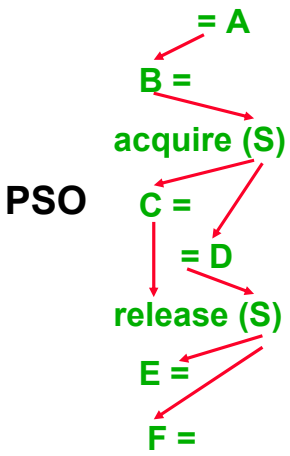


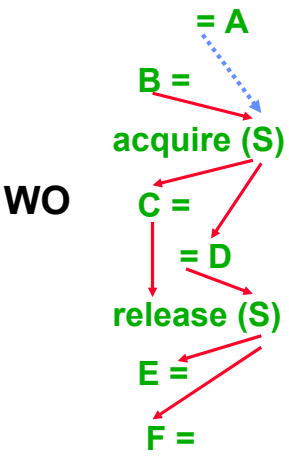
11

Allows pipelining or overlapping of write operations, rather than Forcing one operation to complete before another.

Weak Ordering
POWERPC

- **Also eliminates the orders R(a) → R(b) a ≠ b and R(a) → W(b) a ≠ b**
- **Need non-blocking reads to exploit relaxation**



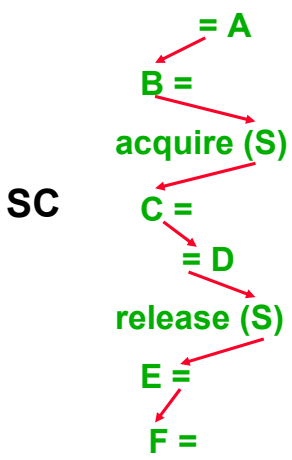


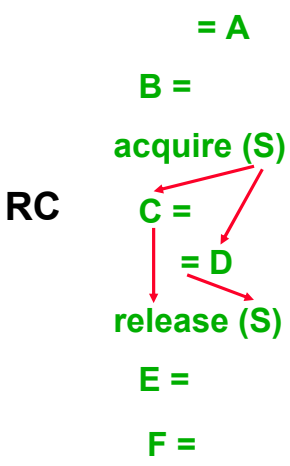
12

Non-blocking reads, doesn't help too much.

Release Consistency
Alpha, MIPS

- **Read/write that precedes acquire need not complete before acquire, and read/write that follows a release need not wait for the release**





13

Weakest of the memory models used these days.

Release Consistency Example

Initially data = old

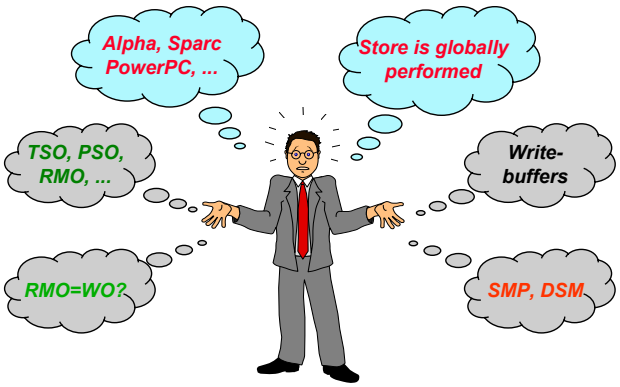
Processor P_1	**Processor P_2**
data = new;	**while(flag != SET) { }**
flag = SET;	**data_copy = data;**

How do we ensure that* data_copy *is always set to* new *?

14

Membar Store Store Membar Load Load

Weaker Memory Models



- **Hard to understand and remember**
- **Unstable - *Modèle de l'année***

15

Weak ordering. Interaction with cache coherence. Weak atomicity.

Why SC is not the right model for compilers

- **Intermediate representation is intrinsically a partial order (data-flow graph)**

  ⇒ ***expose*** **scope for instruction reordering to the underlying architecture**

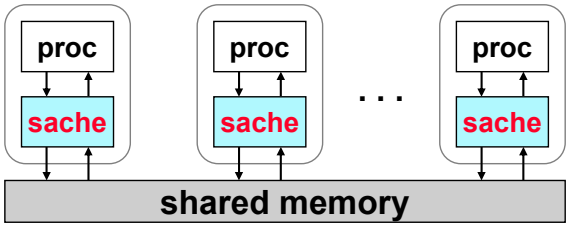
- **Load/Store atomicity forces compilers to over-specify requirements for completion of operations**

  ⇒ ***expose*** **cache coherence actions**

16

The CRF Model

X. Shen, Arvind, L. Rudolph (1999)



Exposes

- ***data caching via semantic caches***

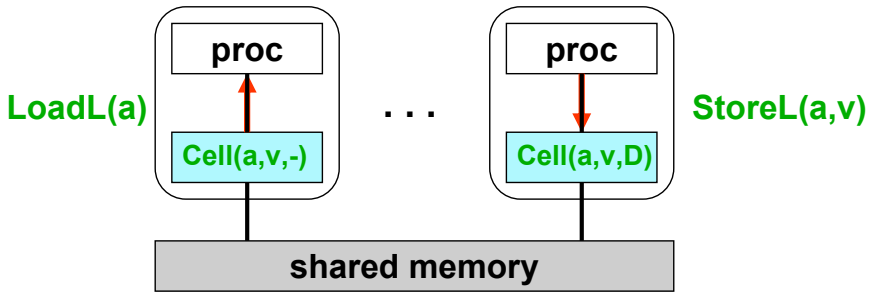
  Store(a,v) ≡ StoreL(a,v); Commit(a)

  Load(a) ≡ Reconcile(a); LoadL(a)

- ***instruction reordering*** **(controllable via Fence)**

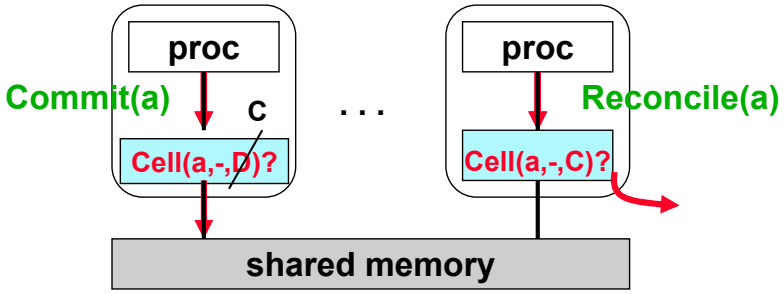
17

CRF: Load Local & Store Local



- LoadL reads from the sache if the address is cached
- StoreL writes into the sache and sets the state to Dirty

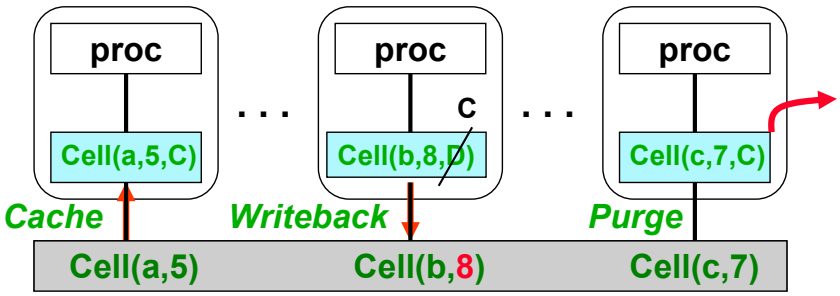
CRF: Commit and Reconcile



- **Commit completes if the address is not cached in the Dirty state**
- **Reconcile completes if the address is not cached in Clean**

19

CRF: Background Operations



- Cache (retrieve) a copy of an uncached address from memory
- Writeback a Dirty copy to memory and set its state Clean
- Purge a Clean copy

CRF: Fences

Instructions can be reordered except for

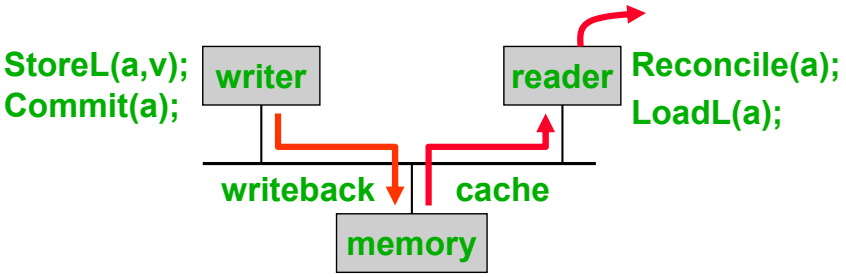
- **Data dependence**
- **StoreL(a,v); Commit(a);**
- **Reconcile(a); LoadL(a);**

Reconcile(a_1);
LoadL(a_1);
Fence$_{rr}$ (a_1, a_2)
Reconcile(a_2);
LoadL(a_2);

Fence$_{wr}$; Fence$_{rw}$; Fence$_{ww}$;

21

Producer-Consumer Synchronization

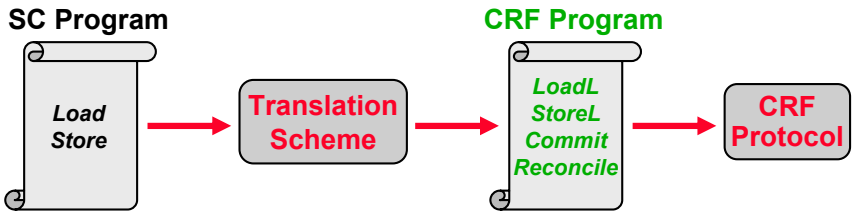


- **Break down the synchronization equally between the producer and consumer**
- **Semantically, memory behaves as the *rendezvous* between processors**

  ⇒ no operation involves more than one sache

22

CRF: A Universal Memory Model



A CRF protocol is automatically a protocol for any memory model whose programs can be translated into CRF programs

23

Translating SC into CRF

Initially a = 0, flag = 0

Processor 1		*Processor 2*
Store(a,10);	L:	r_1 = Load(flag);
Store(flag,1);		Jz(r_1,L);
		r_2 = Load(a);

Translating SC into CRF

Processor 1		*Processor 2*
Store(a,10);	L:	r_1 = Load(flag);
$Fence_{ww}$(a, flag);		Jz(r_1,L);
Store(flag,1);		$Fence_{rr}$(flag, a);
		r_2 = Load(a);

Weak ordering

25

Translating SC into CRF

Processor 1	*Processor 2*
StoreL(a,10);	L: Reconcile(flag);
Commit(a);	r_1 = LoadL(flag);
$Fence_{ww}$(a, flag);	Jz(r_1,L);
StoreL(flag,1);	$Fence_{rr}$(flag, a);
Commit(flag);	Reconcile(a);
	r_2 = LoadL(a);