Beyond Sequential Consistency: Relaxed Memory Models

Joel Emer
Computer Science and Artificial Intelligence Lab
M.I.T.

http://www.csg.csail.mit.edu/6.823

CC and False Sharing

Performance Issue - 1

state	blk addr	data0	data1	...	dataN

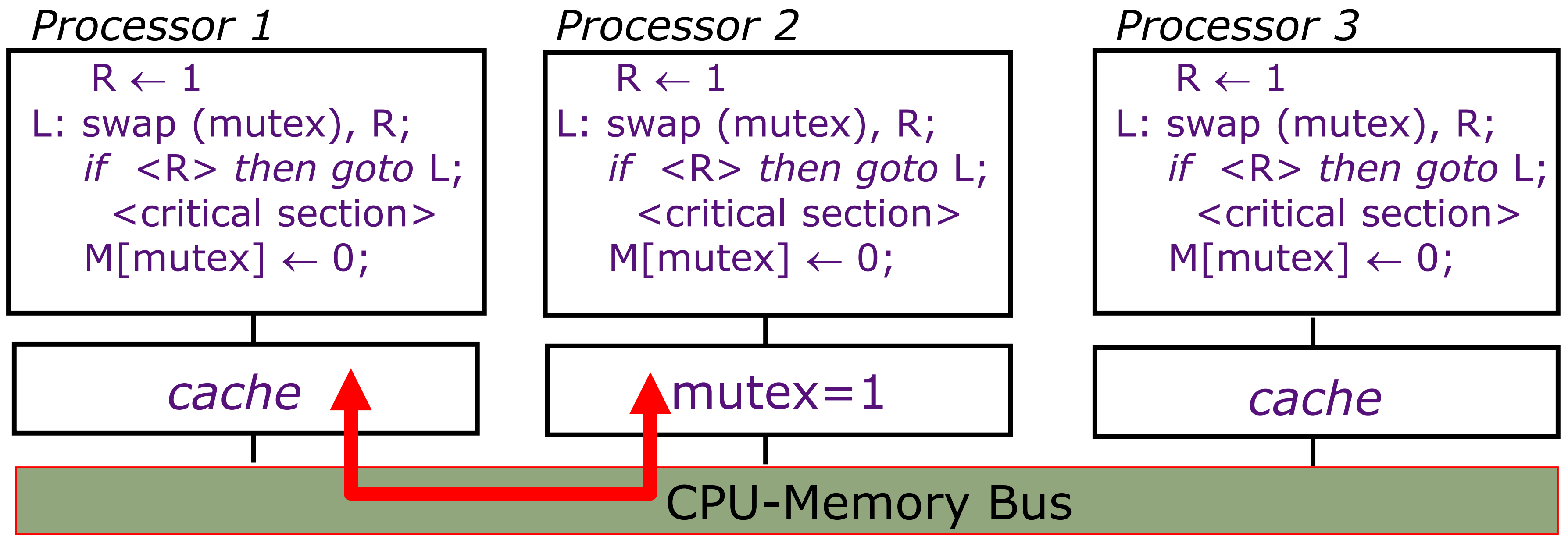
A cache block contains more than one word and cache-coherence is done at the block-level and not word-level

Suppose P_1 writes $word_i$ and P_2 writes $word_k$ and both words have the same block address.

What can happen? The block may be invalidated (ping pong) many times unnecessarily because the addresses are in same block.

CC and Synchronization

Performance Issue - 2



Cache-coherence protocols will cause mutex to *ping-pong* between P1's and P2's caches.

Ping-ponging can be reduced by first reading the mutex location *(non-atomically)* and executing a swap only if it is found to be zero.

CC and Bus Occupancy

Performance Issue - 3

In general, an atomic *read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors

In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation

$\Rightarrow$ expensive for simple buses

$\Rightarrow$ *very expensive* for split-transaction buses

modern processors use

load-reserve

store-conditional

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional (a), R:
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0

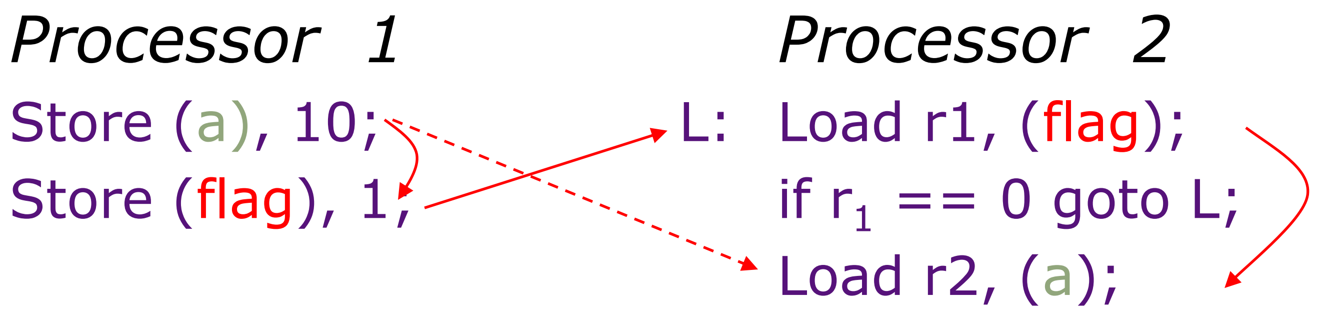
- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Performance:
Load-reserve & Store-conditional

The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses
- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform stores each time

Sequential Consistency



```
   Processor 1                  Processor 2
   Store (a), 10;           L:  Load r1, (flag);
   Store (flag), 1;             if r1 == 0 goto L;
                                Load r2, (a);
```

initially flag = 0

- In-order instruction execution
- Atomic loads and stores

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

Memory Model Issues

Architectural optimizations that are correct for uniprocessors, often violate sequential consistency and result in a new memory model for multiprocessors

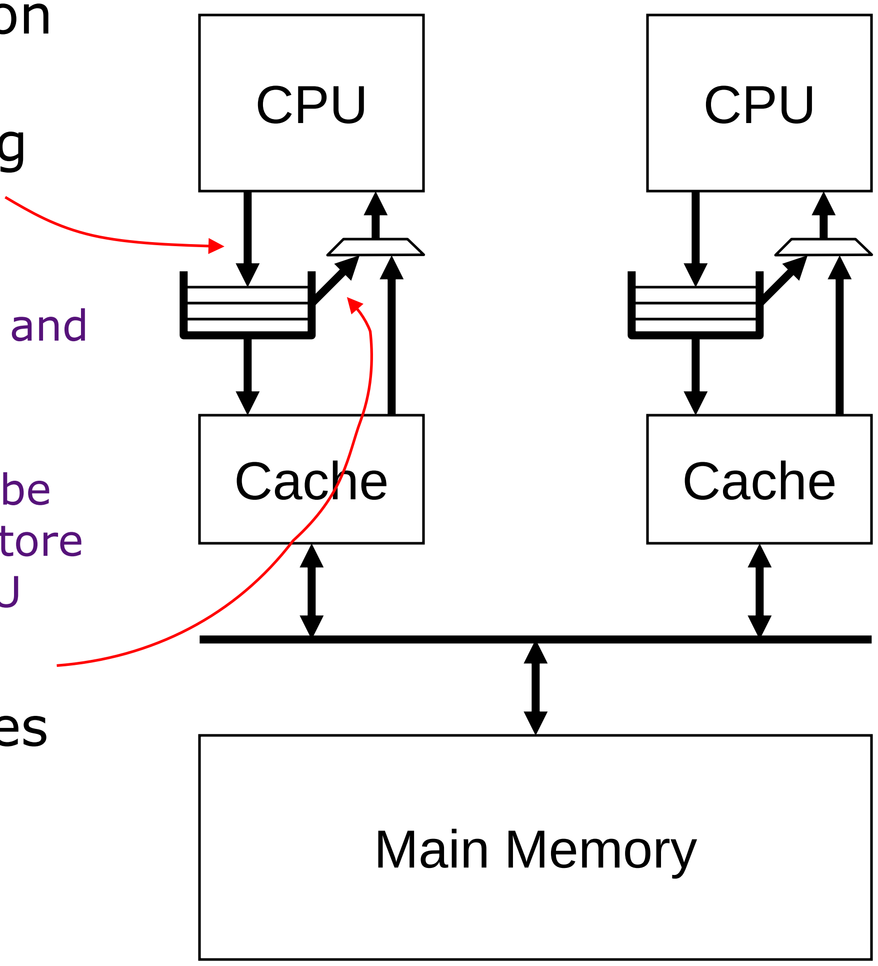
Consistency Models

- Sequential Consistency
  - All reads and write in order

- Relaxed Consistency (one or more of the following)
  - Loads may be reordered after loads
    - e.g., PA-RISC, Power, Alpha
  - Loads may be reordered after stores
    - e.g., PA-RISC, Power, Alpha
  - Stores may be reordered after stores
    - e.g., PA-RISC, Power, Alpha, PSO
  - Stores may be reordered after loads
    - e.g., PA-RISC, Power, Alpha, PSO, TSO
  - Other more esoteric characteristics
    - e.g., Alpha

Committed Store Buffers

- CPU can continue execution while earlier committed stores are still propagating through memory system
  - Processor can commit other instructions (including loads and stores) while first store is committing to memory
  - Committed store buffer can be combined with speculative store buffer in an out-of-order CPU
- Local loads can bypass values from buffered stores to same address



Example 1: Store Buffers

Process 1	*Process 2*
Store ($flag_1$),1;	Store ($flag_2$),1;
Load r_1, ($flag_2$);	Load r_2, ($flag_1$);

Question: Is it possible that $r_1=0$ and $r_2=0$?

- *Sequential consistency: No*

- *Suppose Loads can go ahead of Stores waiting in the store buffer: Yes !*

Total Store Order (TSO):
Sun SPARC, IBM 370

Initially, all memory locations contain zeros

Example 2: Store-Load Bypassing

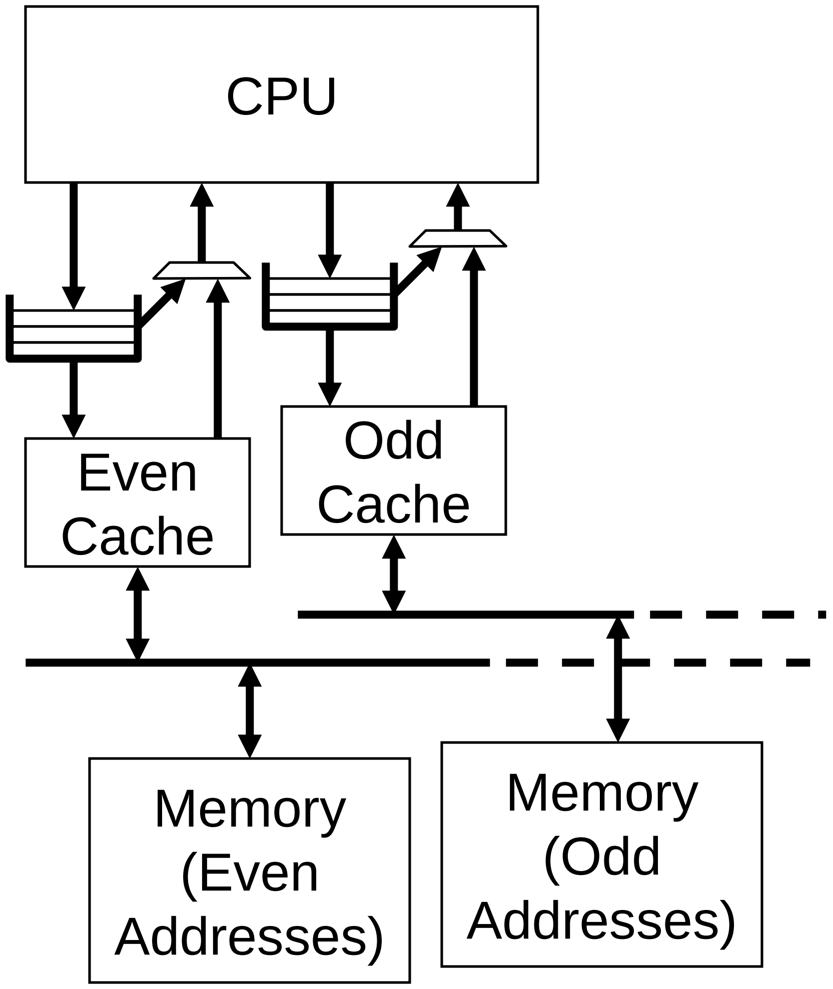
Process 1	*Process 2*
Store $(flag_1)$, 1;	Store $(flag_2)$, 1;
Load r_3, $(flag_1)$;	Load r_4, $(flag_2)$;
Load r_1, $(flag_2)$;	Load r_2, $(flag_1)$;

Question: Do extra Loads have any effect?

- *Sequential consistency: No*
- *Suppose Store-Load bypassing is permitted in the store buffer*
  - No effect in Sparc's TSO model, still not SC
  - In IBM 370, a load cannot return a written value until it is visible to other processors => implicity adds a memory fence, looks like SC

Interleaved Memory System

- Achieve greater throughput by spreading memory addresses across two or more parallel memory subsystems
  - In snooping system, can have two or more snoops in progress at same time (e.g., Sun UE10K system has four interleaved snooping busses)
  - Greater bandwidth from main memory system as two memory modules can be accessed in parallel



Example 3: Non-FIFO Store buffers

Process 1	*Process 2*
Store (a), 1;	Load r_1, (flag);
Store (flag), 1;	Load r_2, (a);

Question: Is it possible that r_1=1 but r_2=0?

- *Sequential consistency: No*
- *With non-FIFO store buffers: Yes*

Sparc's PSO memory model

Example 4: Non-Blocking Caches

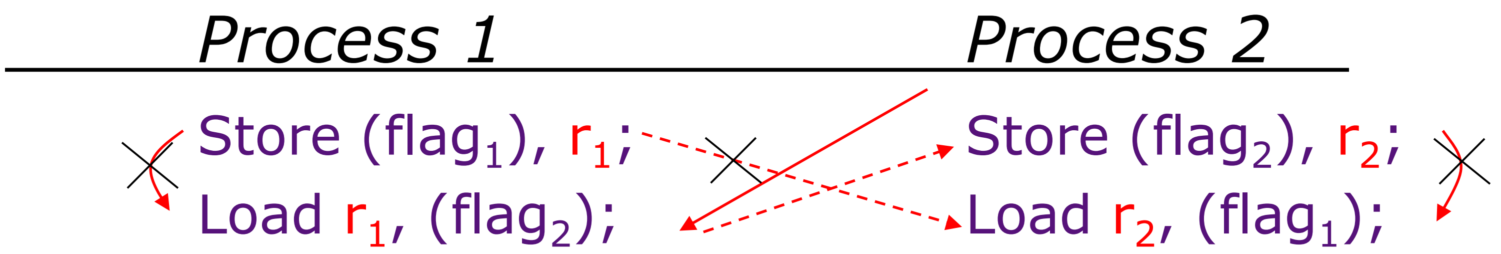
Process 1	*Process 2*
Store (a), 1;	Load r_1, (flag);
Store (flag), 1;	Load r_2, (a);

Question: Is it possible that r_1=1 but r_2=0?

- *Sequential consistency: No*
- *Assuming stores are ordered: Yes because Loads can be reordered*

Alpha, Sparc's RMO, PowerPC's WO

Example 5: Register Renaming



Process 1

Store (flag$_1$), r_1;
Load r_1, (flag$_2$);

Process 2

Store (flag$_2$), r_2;
Load r_2, (flag$_1$);

Register renaming will eliminate this edge

Initially both r_1 and r_2 contain 1.

Question: Is it possible that $r_1=0$ but $r_2=0$?

- *Sequential consistency: No*
- *Register renaming: Yes because it removes anti-dependencies*

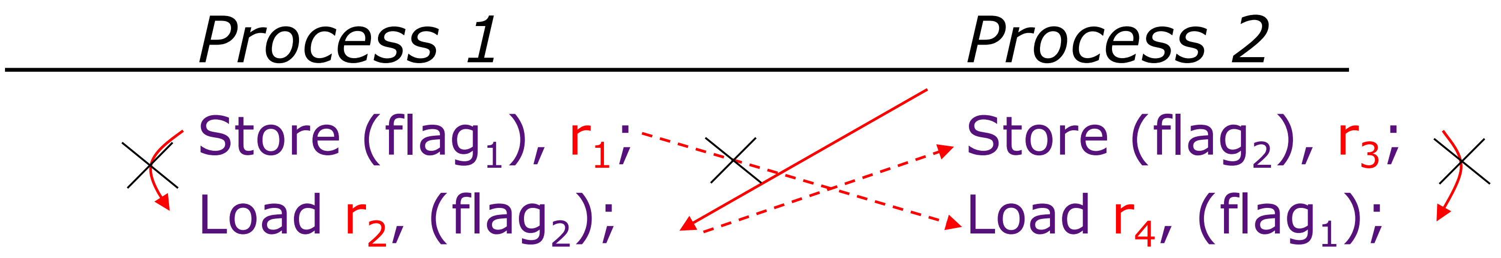
Example 6: Speculative Execution

Process 1		*Process 2*
Store (a), 1;	L:	Load r_1, (flag);
Store (flag), 1;		if r_1 == 0 goto L;
		Load r_2, (a);

Question: Is it possible that r_1=1 but r_2=0?

- *Sequential consistency: No*
- *With speculative loads: Yes even if the stores are ordered*

Example 7: Address Speculation



Process 1	*Process 2*
Store ($flag_1$), r_1;	Store ($flag_2$), r_3;
Load r_2, ($flag_2$);	Load r_4, ($flag_1$);

Address speculation will eliminate this edge

Initially both r_1 and r_3 contain 1.

Question: Is it possible that r_2=0 but r_4=0?

- *Sequential consistency: No*
- *Address speculation: Yes because it removes the dependencies between the stores and loads*

Flag$_1$ and flag$_2$ are registers pointing at memory locations

Example 8: Store Atomicity

Process 1	*Process 2*	*Process 3*	*Process 4*
Store (a),1;	Store (a),2;	Load r_1, (a);	Load r_3, (a);
		Load r_2, (a);	Load r_4, (a);

Question: Is it possible that $r_1=1$ and $r_2=2$ but $r_3=2$ and $r_4=1$?

- *Sequential consistency: No*
- *Even if Loads on a processor are ordered, the different ordering of stores can be observed if the Store operation is not atomic.*

Example 9: Causality

Process 1	*Process 2*	*Process 3*
Store $(flag_1)$,1;	Load r_1, $(flag_1)$;	Load r_2, $(flag_2)$;
	Store $(flag_2)$,1;	Load r_3, $(flag_1)$;

Question: Is it possible that r_1=1 and r_2=1 but r_3=0 ?

- *Sequential consistency: No*
- *With load/load reordering: Yes*

  Alpha

Weaker Memory Models & Memory Fence Instructions

- Architectures with weaker memory models provide memory fence instructions to prevent otherwise permitted reorderings of loads and stores

Store (a_1), r2;
Fence_{wr}
Load r1, (a_2);

The Load and Store can be reordered if a_1 =/= a_2. Insertion of Fence_{wr} will disallow this reordering

Similarly: Fence_{rr}; Fence_{rw}; Fence_{ww};

SUN's Sparc: MEMBAR;
MEMBARRR; MEMBARRW; MEMBARWR; MEMBARWW
PowerPC: Sync; EIEIO

Enforcing Ordering using Fences

Processor 1	*Processor 2*
Store (a),10;	L: Load r_1, (flag);
Store (flag),1;	if r_1 == 0 goto L;
	Load r_2, (a);

Processor 1	*Processor 2*
Store (a),10;	L: Load r_1, (flag);
Fence_{ww};	if r_1 == 0 goto L;
Store (flag),1;	Fence_{rr};
	Load r_2, (a);

Weak ordering

Weaker (Relaxed) Memory Models

Alpha, Sparc
PowerPC, …
Store is globally
performed
TSO, PSO,
RMO, …
Write-
buffers
RMO=WO?
SMP, DSM

- Hard to understand and remember
- Unstable - *Modèle de l'année*
- Abandon weaker memory models in favor of implementing SC.

Implementing SC

1. The memory operations of each individual processor appear to all processors in the order the requests are made to the memory.

   – *Provided by cache coherence*, which ensures that all processors observe the same order of loads and stores to an address

2. Any execution is the same as if the operations of all the processors were executed in some sequential order

   – Provided by enforcing a dependence between each memory operation and the following one.

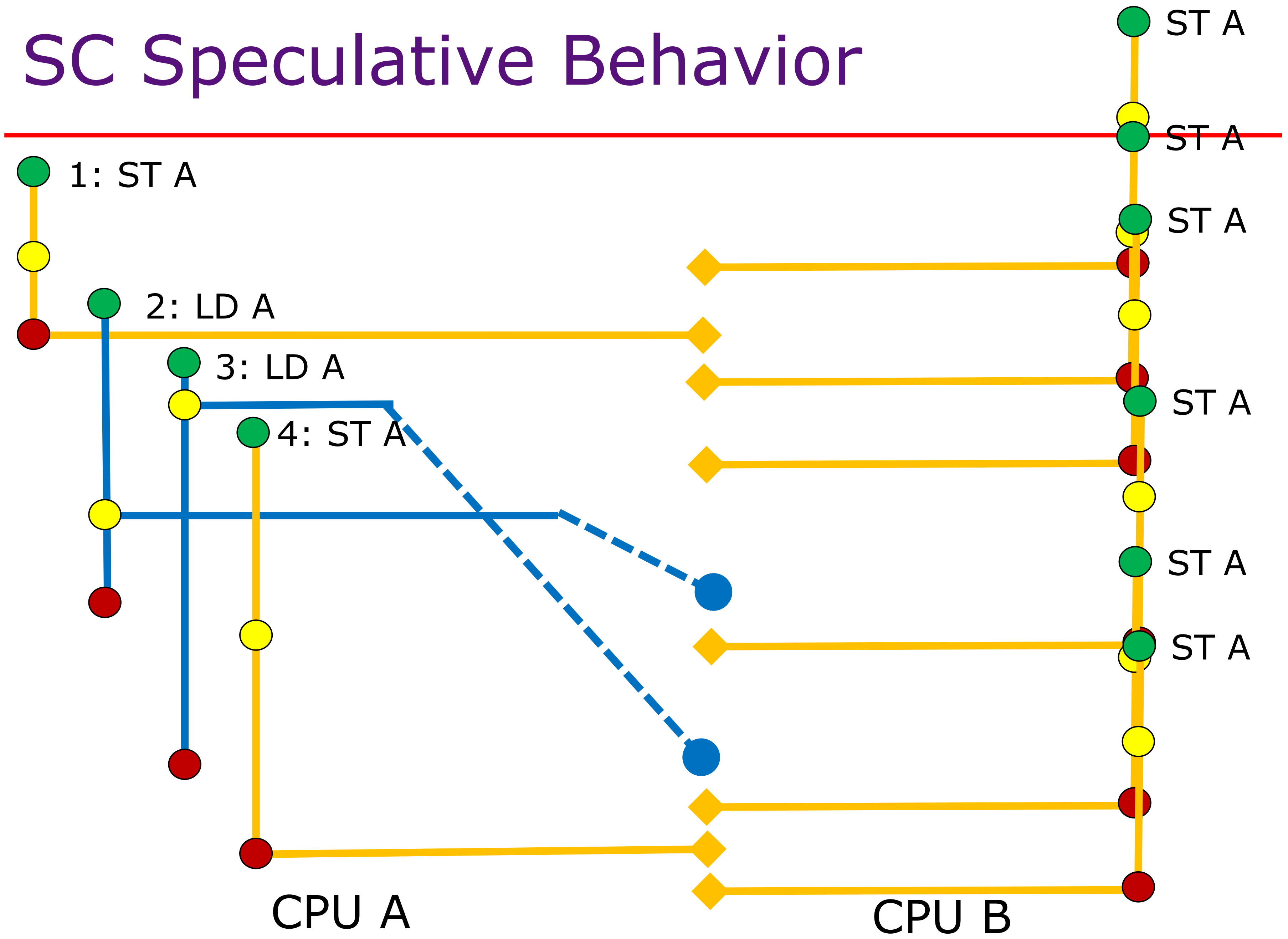
SC Data Dependence

- *Stall*
  - *Use in-order execution with blocking cache*
    - *Cache coherence plus allowing a processor to have only one request in flight at a time will provide SC*
- *Change architecture ⇒ Relaxed memory models*
  - *Use OOO and non-blocking caches*
    - *Cache coherence and allowing multiple requests (different addresses) concurrently gives high performance, then add fence operations to force ordering when needed*
- *Speculate…*

Sequential Consistency Speculation

- Local load-store ordering uses standard OOO mechanism
- Globally non-speculative stores
  - Stores execute at commit -> stores are in-order!
- Globally speculative loads
  - **Guess** at issue that the memory location used by a load will not change between issue and commit of the instruction
    - this is equivalent to loads happening in-order at commit
  - **Check** at commit by remembering all loads addresses starting at issue and watching for writes to that location.
  - **Data Management** for rollback relies on the basic out-of-order speculative data management used for uni-processor rollback and instruction re-execution.

SC Speculative Behavior



Properly Synchronized Programs

- Very few programmers do programming that relies on SC; instead higher-level synchronization primitives are used
  - locks, semaphores, monitors, atomic transactions
- A "properly synchronized program" is one where each shared writable variable is protected (say, by a lock) so that there is no race in updating the variable.
  - There is still race to get the lock
  - There is no way to check if a program is properly synchronized
- For properly synchronized programs, instruction reordering does not matter as long as updated values are committed before leaving a locked region.

Release Consistency

- Only care about inter-processor memory ordering at thread synchronization points, not in between
- Can treat all synchronization instructions as the only ordering points

```
…
Acquire(lock) // All following loads get most recent written values
… Read and write shared data ..
Release(lock) // All preceding writes are globally visible before
              // lock is freed.
…
```

Takeaway

- SC is too low level a programming model. High-level programming should be based on critical sections & locks, atomic transactions, monitors, ...
- High-level parallel programming should be oblivious of memory model issues.
  - Programmer should not be affected by changes in the memory model
- ISA definition for Load, Store, Memory Fence, synchronization instructions should
  - Be precise
  - Permit maximum flexibility in hardware implementation
  - Permit efficient implementation of high-level parallel constructs.

ONLINE SUBJECT EVALUATIONS

Now open at:

http://web.mit.edu/subjectevaluation

- You have until Monday, May 19 at 9 AM
- Please evaluate all subjects in your list
- Don’t forget your TAs
- Write comments

Your feedback is read and valued!

One more to go!

Thanks for listening

- Quiz 4 is on Wednesday May 14th