CS533: Memory Consistency Models

Josep Torrellas

University of Illinois in Urbana-Champaign

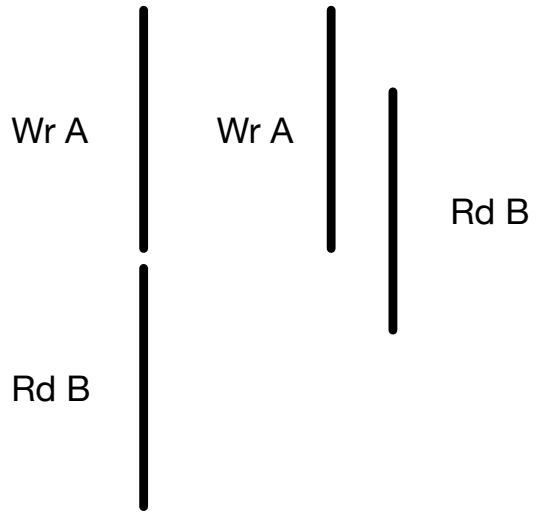
February 12, 2015

Outline

- K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors.
  In *International Symposium on Computer Architecture*, 1990
- K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models.
  In *International Conference on Parallel Processing*, 1991
- S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial.
  In *DEC WRL Research Report*, 1995

Hiding Memory Latency

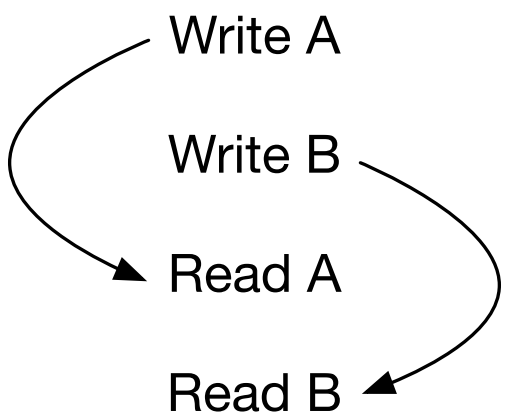
- Overlap memory accesses with other accesses and with computation:



- Simple in uniprocessors
- Can affect correctness in MPs
- Memory Model:
  - Specifies the odering constraints among accesses
  - Specifies the value(s) that a read can return

Uniprocessor Memory Model

- Memory accesses atomic and in program order



- Not necessary to maintain sequential order for correctness
  - Hardware: buffering, pipelining
  - register allocation, code motion
- Simple for programmers
- Allows for high performance

Shared Memory-Multiprocessors

Order between accesses to different locations becomes important

```
P1
A = 1;
Flag = 1;
```

```
P2

while (Flag != 1) {};
.. = A;
```

Unsafe reordering can happen: accesses issued in order may be observed out of order (even without caches):

- Flag is in the local memory module of P2

Shared-Memory Multiprocessors

Multiple copies of the same location

```
P1
A = 1;
```

```
P2
wait (A == 1);
B = 1;
```

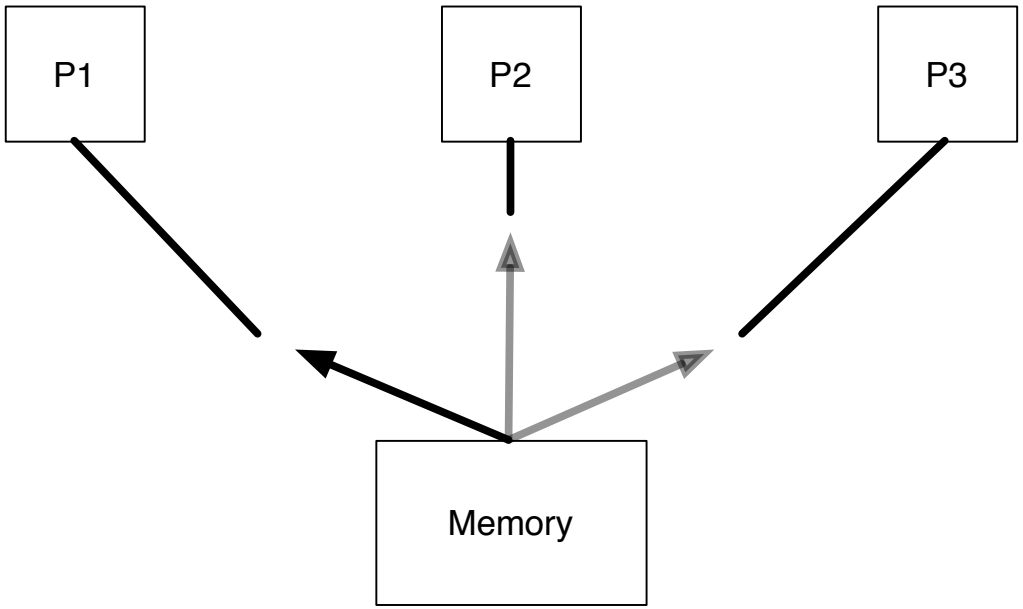
```
P3

wait (B == 1);
.. = A;
```

P3 had $A = B = 0$ in its cache, invalidations for `B` have arrived before the invalidations for `A`. P3 reads 0

Sequential Consistency

Formalized by Lamport

- "Execution of parallel programs appear as some interleaving of the parallel processes on a sequential machine"



Intuitive orders assumed by programmers are typically maintained

Example

Initially: all vars are 0

```
P1
A = 1
Flag = 1
```

```
P2

x = Flag
y = A
```

Possible (x,y) = (0,0),(0,1),(1,1)
Impossible (x,y) = (1,0)

How We Will Proceed

- Focus on the instructions issued by a processor, and put ordering constraints among them
  - when a load is seen by others
  - when a store is seen by others
- Put constraints on the writes to a memory location
  - Atomic
- Define sufficient conditions so that a particular memory consistency model is supported
- Note that accesses issued by a processor to the **same** variable cannot be reordered

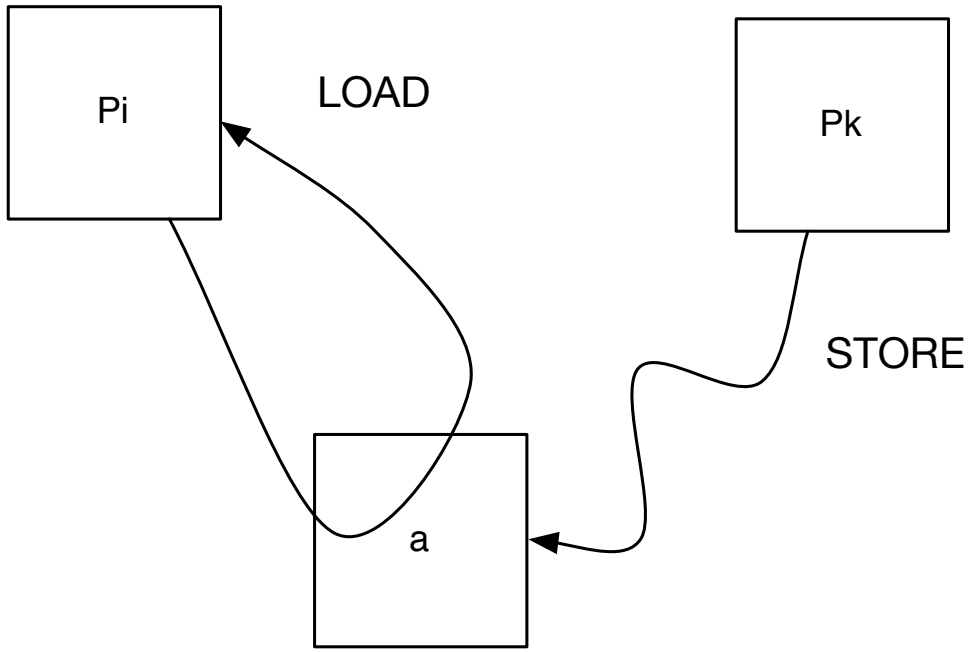
```
  P1
Wr A
Rd X
```

vs

```
  P1
Wr X
Rd X
```

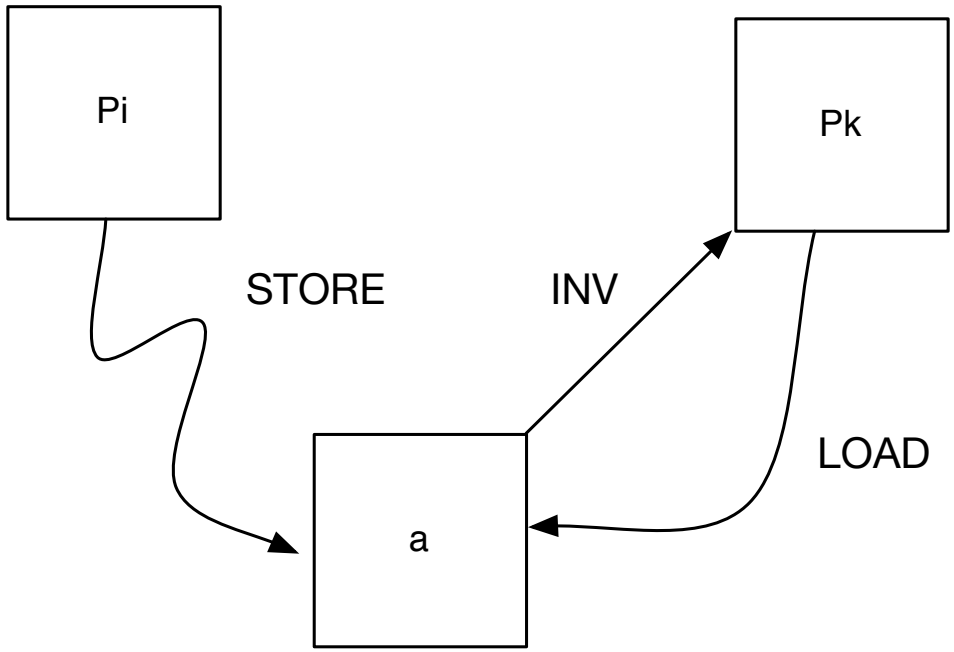
Performing

LOAD by P_i is performed w.r.t. P_k when a STORE by P_k cannot affect the value returns by the LOAD



Performing

STORE by P_i is performed w.r.t. P_k when a LOAD by P_k returns the value defined by that STORE



Sequential Consistency

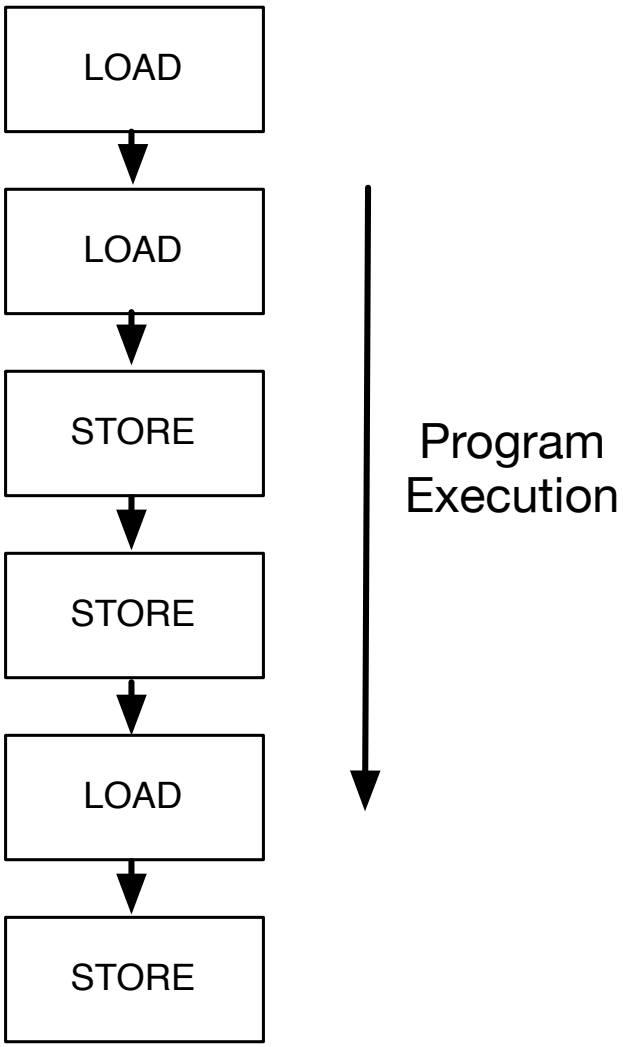
Sufficient conditions for satisfying Sequential Consistency and other models can be formulated so that . . .

. . . Process needs to keep track of requests initiated by itself

Sequential Consistency

- Before a LOAD is allowed to perform w.r.t. any processor, all previous LOAD/STORE accesses must be performed w.r.t. everyone
- Before a STORE is allowed to perform w.r.t. any processor, all previous LOAD/STORE accesses must be performed w.r.t. everyone
- **Note** Globally performed

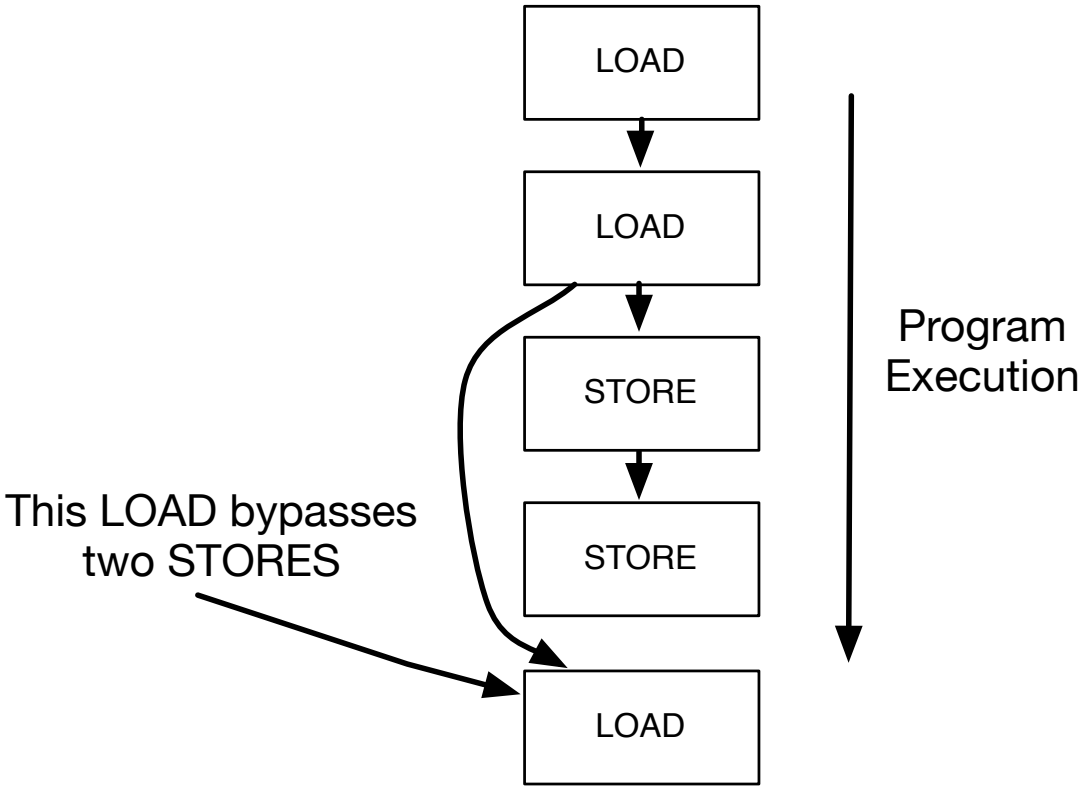
Sequential Consistency



Processor Consistency

Main idea: LOADS are allowed to bypass STORES

- . . . Honoring, of course, local dependences



Processor Consistency

- Before a LOAD is allowed to perform w.r.t. any processor, all previous **LOAD/~~STORE~~** accesses must be performed w.r.t. everyone
- Before a STORE is allowed to perform w.r.t. any processor, all previous **LOAD/STORE** accesses must be performed w.r.t. everyone

Weak Consistency

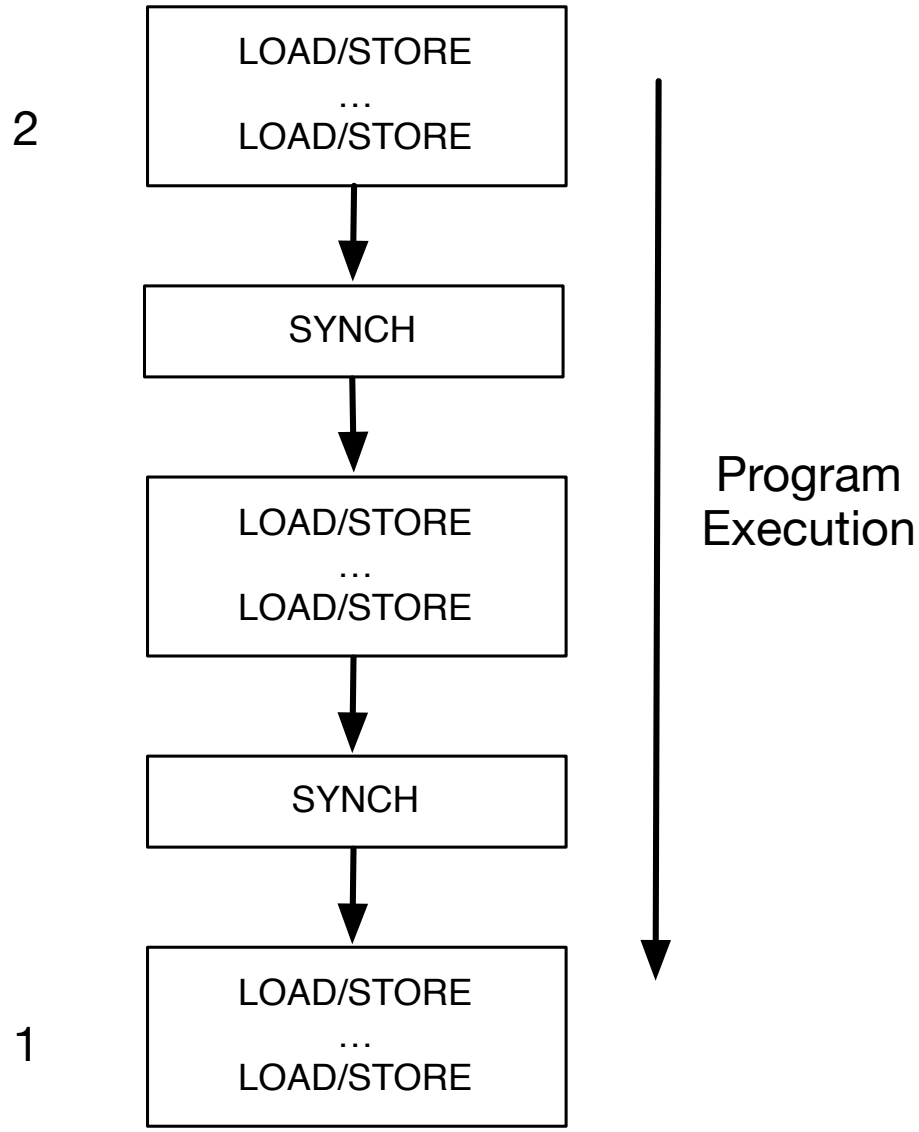
Suppose we are in a critical section

Then, we can have several accesses pipelined because programmer has made sure that:

- No other process can rely on that data structure being consistent until the critical section is exited

Advantage: Higher performance (more overlap)
Disadvantage: Need to distinguish between ordinary LOAD/STORES and SYNCH

Weak Consistency



Weak Consistency

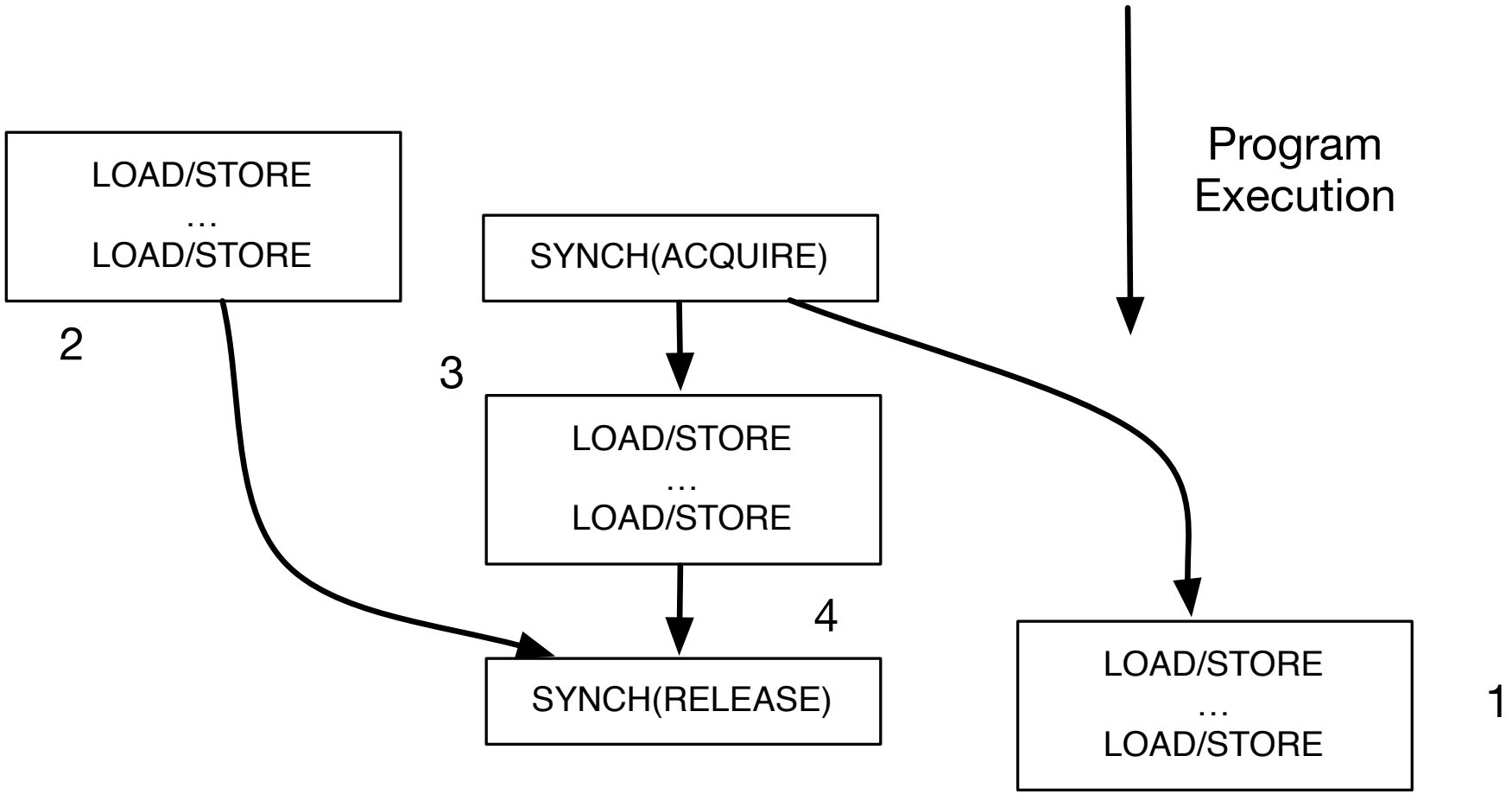
1. Before an ordinary LOAD/STORE is allowed to perform w.r.t. any processor, all previous SYNCH accesses must be performed w.r.t. everyone
2. Before a SYNCH access is allowed to perform w.r.t. any processor, all previous ordinary LOAD/STORE accesses must be performed w.r.t. everyone

- SYNCH accesses are sequentially consistent w.r.t. one another

Release Consistency

- Distinguish between:
  - SYNCH acquires: e.g. LOCK
  - SYNCH releases: e.g. UNLOCK
- LOAD/STORE following a RELEASE do not have to be delayed for the RELEASE to complete
- An ACQUIRE needs not to be delayed for previous LOAD/STORES to complete
- Accesses in the critical section do not wait or delay LOAD/STORES outside the critical section

Release Consistency



Release Consistency

Advantages: Higher performance
Disadvantages: Need to additionally distinguish between ACQUIRE/RELEASE

Release Consistency

1. Before an ordinary LOAD/STORE is allowed to perform w.r.t. any processor, all previous ~~**SYNCH**~~**ACQUIRE** accesses must be performed w.r.t. everyone
2. Before a ~~**SYNCH**~~**RELEASE** access is allowed to perform w.r.t. any processor, all previous ordinary LOAD/STORE accesses must be performed w.r.t. everyone

- ACQUIRE/RELEASE access are processor consistent w.r.t. one another

How to enforce these stalls

- With Fence instructions
- Different types of fences present in current processors
- Check manuals of processors to see which types of fences are supported

Further Readings

- V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ilp processors.
  In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996
- Dubois, Annavaram, Stenstrom course textbook
- Culler and Singh course textbook
- Processors have their own memory consistency models: e.g. SUN's PSO, TSO

Performance Gains from Relaxed Models

Gains both in hardware and compiler

Gains in hardware: Come from latency hiding

- Overlap several memory operations: RDs and WRs
  - Need a lock up free cache (of course): multiple misses serviced at a time
  - Puts extra pressure on the buffers (read and write buffers):
    - have more transactions pending at a time
    - These transactions need to keep record until fully performed

Performance Gains in HW (II)

Evaluation on Superscalar processors:

- Allow multiple outstanding reads: Unlock more potential for relaxed models
- But the computation is also smaller because of ILP
- Relative performance gains of relaxation under ILP can be bigger or smaller than under simple processor

Performance Gains in SW

- Common compiler optimizations require:
  - Change the order of memory operations
  - Eliminate operations
- Examples
  - Register allocating a flag that is used to synchronize

    ```
    While( flag ==0);
    ```

  - Code motion or register allocation across synchronization

    Lock L
    Read A
    Write B
    Unlock L
    Lock L
    Read A
    Read B
    Unlock L
- Sequential consistency disallows reordering of shared accesses

Performance Gains in SW

- More advanced optimizations such as loop transformation and blocking
- Relaxed models allow compilers to do more re-arrangements

Summary

- Release consistency model
  - Reasonable abstraction for programmer
  - Performance gains in SW and HW
- Relaxed models are universal in current multiprocessors
- Different manufacturers have different models
- Programmers prefer sequential consistency
- If there are no data races: all models become equivalent to SC